UC Berkeley

UC Berkeley Previously Published Works

Title

Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication

Permalink

https://escholarship.org/uc/item/6rx4x01r

ISBN

978-1-5090-2140-6

Authors

Koanantakool, Penporn

Azad, Ariful

Buluc, Aydin

et al.

Publication Date

2016-05-01

DOI

10.1109/ipdps.2016.117

Peer reviewed

eScholarship.org Powered by the California Digital Library

University of California

Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication

Penporn Koanantakool[†,‡], Ariful Azad[†], Aydın Buluç[†],
Dmitriy Morozov[†], Sang-Yun Oh[†,*], Leonid Oliker[†], Katherine Yelick[†,‡]
[†]*Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, USA*
[‡]*EECS Department, University of California, Berkeley, USA*
[*]*Department of Statistics and Applied Probability, University of California, Santa Barbara, USA*

***Abstract*—Multiplication of a sparse matrix with a dense matrix is a building block of an increasing number of applications in many areas such as machine learning and graph algorithms. However, most previous work on parallel matrix multiplication considered only both dense or both sparse matrix operands. This paper analyzes the communication lower bounds and compares the communication costs of various classic parallel algorithms in the context of sparse-dense matrix-matrix multiplication. We also present new communication-avoiding algorithms based on a 1D decomposition, called 1.5D, which — while suboptimal in dense-dense and sparse-sparse cases — outperform the 2D and 3D variants both theoretically and in practice for sparse-dense multiplication. Our analysis separates one-time costs from per iteration costs in an iterative machine learning context. Experiments demonstrate speedups up to 100x over a baseline 3D SUMMA implementation and show parallel scaling over 10 thousand cores.**

I. Introduction and Prior Work

Computing the product of a sparse matrix with a dense matrix is an understudied primitive in numerical linear algebra. In this paper, we focus on the case where both matrices have similar dimensions, in contrast to the well-studied case in iterative methods where the dense matrix is tall and skinny [1].

Sparse-dense matrix-matrix multiplication, or SpDM^3 in short, has applications in diverse domains. Examples include the all-pairs shortest-paths problem [2] in graph analytics, non-negative matrix factorization [3] for dimensionality reduction, a novel formulation of the restriction operation [4] in Algebraic Multigrid, quantum Monte Carlo simulations for large chemical systems [5], interior-point methods for semidefinite programming [6], and the siting problem in terrain modeling [7].

Another application in statistical/machine learning is sparse inverse covariance selection (ICS) and related problems [8], [9], [10], [11], [12]. ICS estimation is used for data analysis. Its computational burden can be as much as cubic in the number of dimensions [10], [11], [12] per iteration and, hence, easily becomes intractable as dimensionality increases beyond a few thousand. To improve scalability, a divide-and-conquer type approach has been proposed on a shared memory architecture [13]; however, we know of none that take advantage of parallel distributed memory computing environments. More recently, an algorithm suitable for massively parallel distributed memory computing environments has been suggested [14] for computing the CONCORD estimator [12]. The running time of this ICS estimation algorithm, CONCORD-ISTA, is dominated by the solution of two SpDM^3 problems at every iteration. Hence, a fast parallel SpDM^3 would significantly increase CONCORD-ISTA's scalability and improve its running time. CONCORD-ISTA and additional examples of computational algorithms that can benefit from SpDM^3 are further discussed in Section VI.

There has been relatively little work on the SpDM^3 problem. Bader and Heinecke [15] presented cache-oblivious algorithms based on space filling curves, together with their high-performance shared memory implementations. Greiner and Jacob [16] presented I/O-efficient serial algorithms and related lower bounds. Ortega et al. [17] provided an efficient GPU implementation. In terms of multi-node parallelism, the literature is even sparser. Pietracaprina et al. [18] gave lower bounds on the number of rounds it takes to compute the sparse matrix product in MapReduce. We are unaware of any existing multi-node implementations.

Communication-avoiding algorithms aim to reformulate linear algebra operations to minimize the communication costs [19], [20]. In the case of dense-dense matrix multiplication, both 2D and 3D algorithms are optimal, given their memory footprint [21]. 3D algorithms, however, further minimize the cost of communication relative to 2D algorithms, at the expense of more memory usage [22]. The literature uses both the 3D and the 2.5D names when referring to a non-perfect cube where the third dimension (the replication dimension) is shorter than the first two dimensions. In this paper, we stick to the 3D naming. Sparse-sparse matrix multiplication is more complicated due to different sparsity patterns. Lower bounds are only known for Erdős-Rényi matrices, for which optimal 3D algorithms have also been proposed [23]. The efficient implementation of the 3D sparse-sparse matrix multiplication algorithm on distributed-memory architectures has been done only recently [24].

We make several contributions in this paper. We first provide communication lower bounds for parallel SpDM^3 algorithms. We then introduce efficient parallel algorithms, together with rigorous analysis of their communication costs. We also provide performance results on up to ten thousands of cores using sparse matrices with both uniform and skewed nonzero distributions. Finally, we analyze the SpDM^3 problem within an iterative algorithm.

II. Preliminaries

This paper focuses on the parallel matrix-matrix multiplication $C = A \times B$. A is a sparse, $m \times \ell$ matrix, B is a dense, $\ell \times n$ matrix, and C is an $m \times n$ matrix which is usually dense, depending on the sparsity pattern of A and the size of ℓ. For theoretical analysis and lower bounds, we assume that the nonzeroes in A are uniformly distributed, as in the Erdős-Rényi model [25], and that there are d nonzeroes per row. For experimental analysis, we also use more realistic sparsity patterns. Note that all analyses can be easily extended to the reverse case where A is dense and B is sparse.

A. Notation

We consider a distributed, homogeneous parallel system with p processors connected through the network. Let P be the array of all processors. P can be one-, two-, or three-dimensional, depending on the arrangement of the algorithm, and is indexed with tuples of same dimensionality. The : operator returns a vector of all indices in the dimension it is used in, e.g., $P(i,:)$ refers to all processors in row i of the processor grid. Our indexing is zero-based.

Let $X^{h\times w}$ denote a partitioning of a matrix X into an $h \times w$ grid of equal size submatrices. The same tuple indexing applies here, for example, $X^{h\times w}(i,:) = [X^{h\times w}(i,0)|...|X^{h\times w}(i,w-1)]$. We will refer to matrix elements as a corresponding lowercase letter with subscript. x_{ij} means the element in row i and column j of matrix X.

Let $\mathrm{nnz}(\cdot)$ denote the number of nonzeroes of a matrix. We use this notation for dense matrices as well because it allows us to simplify notation and compare asymptotic results for matrices with different sparsity and aspect ratio. Throughout this paper, $\mathrm{nnz}(A) = md$, $\mathrm{nnz}(B) = \ell n$, and $\mathrm{nnz}(C) = mn$.

B. Computation

The iteration space for the problem can be seen as an $m \times n \times \ell$ cuboid where each voxel (i, j, k) represents the computation c_{ij} += $a_{ik} \cdot b_{kj}$. All p processors partition the cuboid and compute the voxels in their subsets. Figure 1 shows the computational cuboid, an example voxel subset V, and its projections onto the A, B, and C planes.

C. Communication

We are interested in per-processor communication costs along the critical path. There are two types of communication cost: latency (cost to send a message) and bandwidth (cost per word). The communication time can be modeled as, $S\alpha + W\beta$, where S is the number of messages, α is the latency cost, W is the number of words, and β is the reciprocal bandwidth. We assume α and β are fixed in our analysis, so it is sufficient to measure S and W to capture the communication trend. In addition, we assume that work must be performed on more than one processor so the communication costs cannot be trivially zero.

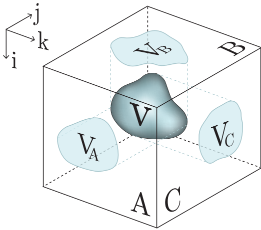


Fig. 1: The matrix multiplication computational cuboid.

There are 4 communication operations used in this paper,

- **Shift:** Shifting w words by distance d means sending w words to the dth neighbor in one direction and replacing it with w words from the dth neighbor in the opposite direction. The communication costs are $S = O(1)$ and $W = O(w)$.
- **Broadcast:** Broadcasting w words sends w words to all p processors in a specified communication group. We assume a broadcast tree implementation [26] which has communication costs, $S = O(\log p)$ and $W = O(w \log p)$.
- **Reduce:** Reducing w words sums a word from each of p processors in a specified communication group for w words. It has the same costs as broadcast, $S = O(\log p)$ and $W = O(w \log p)$.
- **Gather:** Gathering w words from each of p processors in a specified communication group creates wp words on the gather root. It has communication costs $S = O(\log p)$ and $W = O(wp \log p)$.

We assume that each processor starts out with original data divided equally to p parts and that replicating them incurs communication.

III. Communication Lower Bounds

For simplicity, we assume that all matrices are square with length n in this section. Let M be the size in words of the fast memory each of our p processing elements has. Let F be the total number of FLOPS to multiply A and B, $F = O(dn^2)$. The general lower bounds for communication along the critical path from [27],

$$S = \Omega\left(\frac{F}{p\sqrt{M^3}}\right), \quad W = \Omega\left(\frac{F}{p\sqrt{M}}\right), \tag{1}$$

trivially apply here. To relate M to our problem parameters, we assume that M can fit at most c copies of all three matrices, i.e., $M = O(n^2c/p)$. Plugging in F and M into Equation (1) gives us the lower bounds,

$$S_{\text{compute}} = \Omega\left(\frac{d\sqrt{p}}{nc^{3/2}}\right), \quad W_{\text{compute}} = \Omega\left(\frac{dn}{\sqrt{pc}}\right).$$

However, these bounds does not consider the data movements to make the n^2c/p data available. This replication costs are usually omitted in the dense-dense or sparse-sparse matrix multiplication analysis because they are of lower order than the main S and W costs. In our case, we have to consider the cost for W since $d \ll n$. For practical reasons, we assume each processor starts with only n^2/p words. The lower bounds of collecting n^2/p words from $c-1$ other processors are

$$S_{\text{collect}} = \Omega(\log c), \quad W_{\text{collect}} = \Omega\left(\frac{n^2}{p} \cdot \frac{c-1}{c}\right).$$

Therefore, the communication lower bounds are

$$S = \Omega\left(\frac{d\sqrt{p}}{nc^{3/2}}\right), \quad W = \Omega\left(\frac{dn}{\sqrt{pc}} + \frac{n^2}{p}\right). \tag{2}$$

To simplify the analysis, we enforced the same replication factor on all matrices. Allowing different replication factors for each matrix would change the lower bounds, which is the subject of future work.

IV. Algorithms

Here we discuss existing parallel algorithms and present a few new variants. The algorithms are categorized based on how they partition the computational cuboid [23]. The constants on the leading order terms can be compared across all analyzed algorithms, so we are going to abuse the big-O notation by keeping the constants inside.

A. 1D algorithms

1D algorithms partition only one dimension of the cuboid, i.e., slicing the cuboid into planes. They logically arrange processors into a ring (1D torus) topology and partition matrices along a single dimension. Only one matrix needs to be passed around. We only analyze communicating A since it is asymptotically cheaper than communicating B or C.

1D blocked column: Processor $P(j)$ has $A^{1\times p}(:,j)$ and $B^{1\times p}(:,j)$, and is responsible for computing

$$C^{1\times p}(:,j) = \sum_{k=0}^{p-1} A^{1\times p}(:,k)B^{p\times p}(k,j).$$

See Figure 2b (with $c=1$) for illustration. For p rounds, each processor multiplies A by a corresponding $\ell/p \times n/p$ submatrix of B, adds the result to C, and shifts A cyclically by one. Each round a processor sends one message of size $\mathrm{nnz}(A)/p$ words. Therefore, the communication costs are,

$$S = O(p), \quad W = O(\mathrm{nnz}(A)).$$

1D blocked inner product: $P(j)$ owns $A^{p\times 1}(j,:)$ and $B^{1\times p}(:,j)$, and must compute $C^{1\times p}(:,j)$ by calculating $C^{p\times p}(i,j) = A^{p\times 1}(i,:)B^{1\times p}(:,j)$ for all $0 \le i < p$. See Figure 2d (with $c=1$) for illustration. For p rounds, each processor calculates a corresponding $n/p \times n/p$ submatrix of C using the locally available A and B, and shifts A cyclically by one. Each round a processor sends one message of size $\mathrm{nnz}(A)/p$. so the communication costs are the same as the 1D blocked column variant,

$$S = O(p), \quad W = O(\mathrm{nnz}(A)).$$

1D blocked row and **1D blocked outer product** require passing dense matrices B or C around, so they are omitted.

B. 2D algorithms

2D algorithms split two dimensions of the computational cuboid, e.g., pencils of length ℓ. They logically arrange processors into a 2D grid of size $p_m \times p_n$. There are many variants including Cannon's algorithm [28] and SUMMA [29]. Since both algorithms have similar costs, we will only discuss the stationary-C SUMMA algorithm because it is more generalizable and more widely used. We assume $p_m = p_n = \sqrt{p}$ for simplicity.

2D SUMMA calculates b outer products, where b is a blocking factor. Here we assume $b = \sqrt{p}$. Processor $P(i,j)$ owns $A^{\sqrt{p}\times\sqrt{p}}(i,j)$ and $B^{\sqrt{p}\times\sqrt{p}}(i,j)$ and computes

$$C^{\sqrt{p}\times\sqrt{p}}(i,j) = \sum_{k=0}^{\sqrt{p}-1} A^{\sqrt{p}\times\sqrt{p}}(i,k)B^{\sqrt{p}\times\sqrt{p}}(k,j).$$

See Figure 2a (with $c=1$) for illustration. In the kth round, $P(i,k)$ broadcasts its A to $P(i,:)$ and $P(k,j)$ broadcasts its B to $P(:,j)$. All processors then do the multiplication and accumulate the product in their local C's. There are $\sqrt{p}$ broadcasts of A ($\log\sqrt{p}$ messages and $\mathrm{nnz}(A)/p$ words each), and $\sqrt{p}$ broadcasts of B ($\log\sqrt{p}$ messages and $\mathrm{nnz}(B)/p$ words each). Therefore, 1

$$S{=}O(2\sqrt{p}\log\sqrt{p}), W{=}O\left(\frac{\mathrm{nnz}(A){+}\mathrm{nnz}(B)}{\sqrt{p}}\log\sqrt{p}\right).$$

C. 3D algorithms

3D algorithms partition all 3 dimensions of the computational cuboid, e.g., into subcubes, length-$n/2$ pencils, etc.

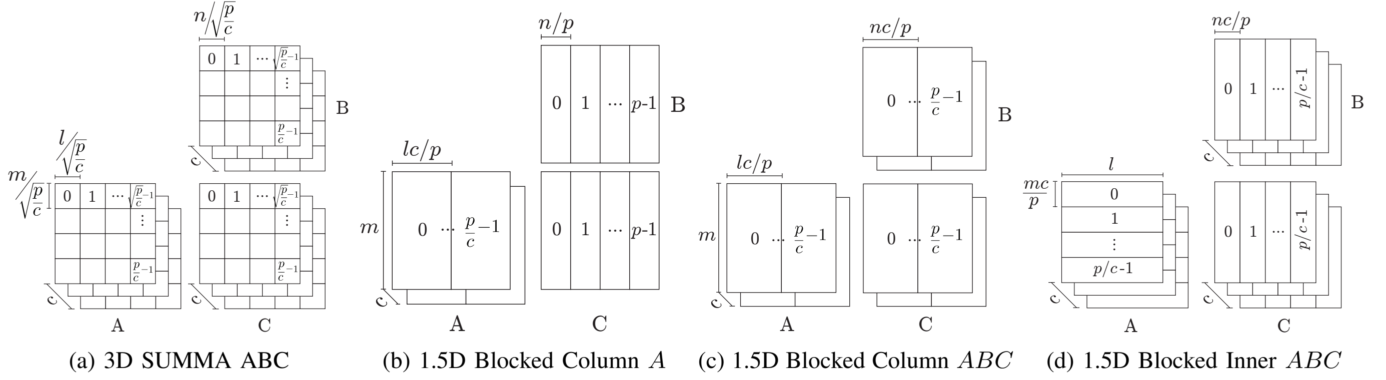
3D SUMMA algorithms [30], [22] (sometimes called *2.5D*) utilize replication to avoid communication. It logically arranges processors into a 3D $p_m \times p_n \times c$ mesh. In essence, it is c layers of 2D SUMMA algorithm with $p_m \times p_n$ processor grids, except that each layer only computes one-cth of the outer products. We assume $p_m = p_n = \sqrt{p/c}$ for simplicity, as illustrated in Figure 2a.

Converting from a 2D layout to a 3D layout requires preprocessing. We call this *replication* since it also makes c processors hold the same blocks of each matrix. This can be done by exchanging blocks within a group of c processors and on each processor concatenating into larger blocks. The algorithm arranges the processors in a way that $A^{\sqrt{p/c}\times\sqrt{p/c}}(i,j)$ and $B^{\sqrt{p/c}\times\sqrt{p/c}}(i,j)$ are on $P(i,j,:)$. These $P(i,j,:)$ cooperate as a team on computing $C^{\sqrt{p/c}\times\sqrt{p/c}}(i,j)$. Each layer is responsible for calculating $\sqrt{p}/c^{3/2} = q$ outer products. $P(i,j,\ell)$ calculates

$$C^{\sqrt{\frac{p}{c}}\times\sqrt{\frac{p}{c}}}(i,j){=}\sum_{k=0}^{q-1} A^{\sqrt{\frac{p}{c}}\times\sqrt{\frac{p}{c}}}(i,\ell q{+}k)B^{\sqrt{\frac{p}{c}}\times\sqrt{\frac{p}{c}}}(\ell q{+}k,j).$$

In the kth round, $P(i,\ell q{+}k,\ell)$ broadcasts its A to $P(i,:,\ell)$ and $P(\ell q+k,j,\ell)$ broadcasts its B to $P(:,j,\ell)$, then each processor computes the product. After q rounds, $P(i,j,:)$ do a sum reduction on C to get the final result. This costs q broadcasts of A ($\log\sqrt{p/c}$ messages and $\mathrm{nnz}(A)c/p\,\log\sqrt{p/c}$ words each), q broadcasts of B ($\log\sqrt{p/c}$ messages and $\mathrm{nnz}(B)c/p\,\log\sqrt{p/c}$ words each), and one reduction of C ($\log c$ messages and $\mathrm{nnz}(C)c/p\,\log c$ words).

As for the replication cost, we will model it as $P(i,j,0)$ gathering all matrices from $P(i,j,:)$ then broadcasting the concatenated matrices back to them. Replicating A takes one gather ($\log c$ messages and $\mathrm{nnz}(A)c/p\log c$ words) and one broadcast ($\log c$ messages and $\mathrm{nnz}(A)c/p\log c$ words). Replicating B takes one gather ($\log c$ messages and



(a) 3D SUMMA ABC (b) 1.5D Blocked Column A (c) 1.5D Blocked Column ABC (d) 1.5D Blocked Inner ABC

Fig. 2: Illustrating processor mesh layouts for 1.5D and 3D algorithms. Matrix names at the end indicate that they are being replicated. (Substituting $c = 1$ gives corresponding 1D and 2D algorithms.)

$\text{nnz}(B)c/p \log c$ words) and one broadcast ($\log c$ messages and $\text{nnz}(B)c/p \log c$ words). Let W_r represent the bandwidth costs of replication and reduction combined,

$$W_r = O\left(\frac{2\,\text{nnz}(A) + 2\,\text{nnz}(B) + \text{nnz}(C)}{p} c \log c\right).$$

The total communication costs are,

$$S = O\left(2\frac{\sqrt{p}}{c^{3/2}} \log\sqrt{\frac{p}{c}} + 5\log c\right),$$
$$W = O\left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{pc}} \log\sqrt{\frac{p}{c}} + W_r\right).$$

D. 1.5D algorithms

2D and 3D algorithms communicate at least one dense matrix while 1D algorithms can limit the movement to just the sparse matrix A. This section applies replication to 1D algorithms to avoid more communication. The first algorithm simply increases the size of matrix A that is stored locally. The latter two algorithms are similar to the communication-avoiding N-body algorithms [31], [32] which also operate on a ring topology. We will still use c for replication factor.

1.5D blocked column replicating A (ColA): We start by replicating just the sparse matrix A c times, $1 \le c \le p$. Processor $P(j)$ has $A^{1\times p/c}(:, j)$ and $B^{1\times p}(:, j)$, and must compute

$$C^{1\times p} = \sum_{k=0}^{p/c-1} A^{1\times p/c}(:, k)B^{p/c\times p}(k, j).$$

See Figure 2b for illustration. For p/c rounds, each processor computes the product and shifts A among processors in the same layer by one. Latency cost is improved by a factor of c, but the total bandwidth cost stays the same, since the message size is also increased by c. Replication takes $2\log c$ messages and $2\,\text{nnz}(A)c/p \log c$ words. Therefore, the total costs are,

$$S = O(2\log c + p/c),$$
$$W = O\left(\frac{2\,\text{nnz}(A)c}{p} \log c + \text{nnz}(A)\right)$$

1.5D blocked column replicating all matrices (ColABC): Next, we investigate paying an extra cost of replicating the dense matrix B in an attempt to reduce more shifting costs asymptotically. This algorithm groups p processors into a $p/c \times c$ grid. See Figure 2c for illustration. $P(j, :)$ have $A^{1\times p/c}(:, j)$ and $B^{1\times p/c}(:, j)$, and work as a team to compute

$$C^{1\times p/c}(:, j) = \sum_{k=0}^{p/c-1} A^{1\times p/c}(:, k)B^{p/c\times p/c}(k, j).$$

All c team members split these p/c summation terms equally. $P(j, \ell)$ computes $p/c^2 = q$ terms,

$$C^{1\times p/c}(:, j) = \sum_{k=\ell q}^{(\ell+1)q-1} A^{1\times p/c}(:, k)B^{p/c\times p/c}(k, j).$$

This computation pattern can be done by first shifting A in the same layer by distance ℓq (to jump to the starting point), then all processors can alternate between multiplication and shifting by one as usual for q rounds, and reduce C at the end. Figure 3a shows an example with $p = 8$ and $c = 2$.

Replication and reduction cost the same as 3D SUMMA algorithm's. The matrix A is shifted p/c^2 times so it takes p/c^2 messages and $p/c^2 \cdot \text{nnz}(A)c/p = \text{nnz}(A)/c$ words. Total communication costs are,

$$S = O\left(5\log c + \frac{p}{c^2}\right), \quad W = O\left(\frac{\text{nnz}(A)}{c} + W_r\right).$$

1.5D blocked inner product replicating all matrices (InnerABC): Next, we apply replication to the inner product algorithm.[1] This algorithm also groups p processors into $p/c \times c$ grid, except this time $P(j, :)$ have $A^{p/c\times 1}(j, :)$ and $B^{1\times p/c}(:, j)$, and compute $C^{1\times p/c}(:, j)$ together as a team. See Figure 2d for illustration.

There are p/c blocked inner products to do and each team member does $p/c^2 = q$ of them. $P(j, \ell)$ computes

$$C^{p/c\times p/c}(k, j) = A^{p/c\times 1}(k, :)B^{1\times p/c}(:, j)$$

for $\ell q \le k < (\ell + 1)q$.

[1] *1.5D blocked inner product replicating A* has the same communication costs and output layout as *1.5D blocked column replicating A* and is omitted.

Algorithms	#messages $= S$			#words $= W$		
	Replication	Propagation	Collection	Replication	Propagation	Collection
1.5D Col A	$2\log c$	$\frac{p}{c}$	-	$2\frac{\text{nnz}(A)}{p}c\log c$	$\text{nnz}(A)$	-
1.5D Col ABC	$4\log c$	$\frac{p}{c^2}$	$\log c$	$2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$	$\frac{\text{nnz}(A)}{c}$	$\frac{\text{nnz}(C)}{p}c\log c$
1.5D Inner ABC	$4\log c$	$\frac{p}{c^2}$	$\log c$	$2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$	$\frac{\text{nnz}(A)}{c}$	$\frac{\text{nnz}(C)}{p}c\log c$
3D SUMMA ABC	$4\log c$	$2\frac{\sqrt{p}}{c^{3/2}}\log\sqrt{\frac{p}{c}}$	$\log c$	$2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$	$\frac{\text{nnz}(A)+\text{nnz}(B)}{\sqrt{pc}}\log\sqrt{\frac{p}{c}}$	$\frac{\text{nnz}(C)}{p}c\log c$

TABLE I: Algorithm communication costs. Uppercase letters at the end of algorithm names indicate the matrices being replicated.

$P(j, \ell)$ initially shifts A by ℓq to start at the required offset then alternates between multiplication and shifting by one for q rounds. Finally, the algorithm gathers the final matrix C to $P(j, 0)$ on the first layer. Figure 3b shows an example with $p = 8$ and $c = 2$.

Shifting A costs p/c^2 messages and $\text{nnz}(A)/c$ words. Gathering C costs the same as reduction asymptotically so the total communication costs are,

$$S = O\left(5\log c + \frac{p}{c^2}\right), \quad W = O\left(\frac{nnz(A)}{c} + W_r\right).$$

E. Comparison

We are interested in comparing our 1.5D algorithms, ColA, ColABC, and InnerABC with the classic 3D SUMMA algorithm which will be called SummaABC from now on, with *ABC* indicating that it replicates all three matrices. Table I summarizes all communication costs of all replicating algorithms. The costs of 1D and 2D algorithms can be obtained by substituting $c = 1$ into 1.5D and 3D algorithms, respectively. None of the presented algorithms obtained the communication lower bounds, although SummaABC has quite similar costs.

Communication consists of three phases, replication, propagation, and collection. Replication is the gathering of neighboring matrices and the broadcasting of the concatenated matrix. Propagation is the communication within the multiplication steps to get the necessary blocks for each local multiplication. It corresponds to the shiftings of A in 1D and 1.5D algorithms, and the broadcastings of A and B in 2D and 3D algorithms. Collection refers to reduction or gathering of C at the end after all multiplications are done.

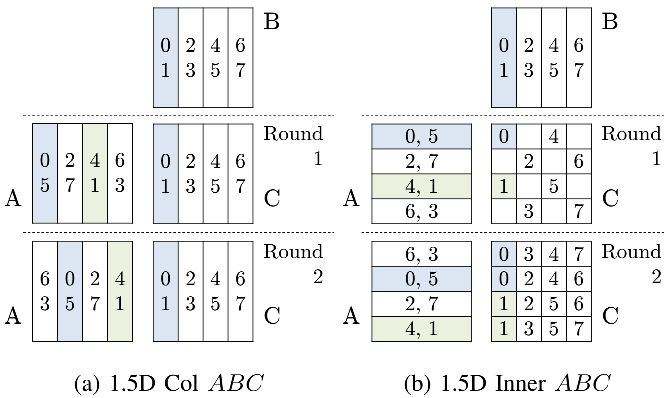


Fig. 3: Example computations of *1.5D blocked column ABC* and *1.5D blocked inner product ABC* on 8 processors. Numbers in the grid are processor ranks.

Even though ColABC and InnerABC have the same asymptotic costs, InnerABC uses gather in the collection phase which can be significantly faster than ColABC's reduction in practice. They also store matrices in different layouts, which can affect local computation rates. The storage format for C is key to performance in some context as well. We will discuss this in Section VI.

There are limits to the effective replication factors for each algorithm. For ColA, $c = p$ corresponds to replicating the whole matrix A, therefore $c \leq p$ is the limit. ColABC and InnerABC have $c \leq \sqrt{p}$, since when $c = \sqrt{p}$, each processor layer only computes one round of local matrix multiplication – any larger c's would leave some layer idle. The same reasoning applies to SummaABC algorithm whose upper limit is $c \leq \sqrt[3]{p}$ in which case each layer only computes one outer product.

Latency costs are not dependent on matrix inputs, but purely on the number of processors p and the replication factor c. Out of all three phases, the propagation cost grows fastest with p and is the dominating cost. The best latency cost for propagation is $O(1)$ and is attainable by all algorithms with their highest effective c's. However, higher c's mean more memory requirement and increased bandwidth costs for replication and possibly collection. For a fixed c, SummaABC achieves lowest latency costs.

Bandwidth cost is the number of words sent. It can be computed from latency cost (number of messages sent) and message size (number of words sent in each message). Most analysis in prior work for dense-dense or sparse-sparse case considers the message sizes for matrices A and B to be the same. Thus, SummaABC also minimizes bandwidth costs altogether and is the best algorithm overall in their cases. This assumption does not hold in our case, and one can utilize less bandwidth by moving A more and moving B less often than SummaABC does, at the expense of higher latency costs. For example, ColA only moves the sparse matrix A around. It has the lowest overall asymptotic bandwidth cost, but also the highest latency cost. ColABC and InnerABC opt to replicate the dense matrix B to achieve asymptotically lower propagation latency and bandwidth than ColA. They also have to move the dense matrix C in the collection phase.

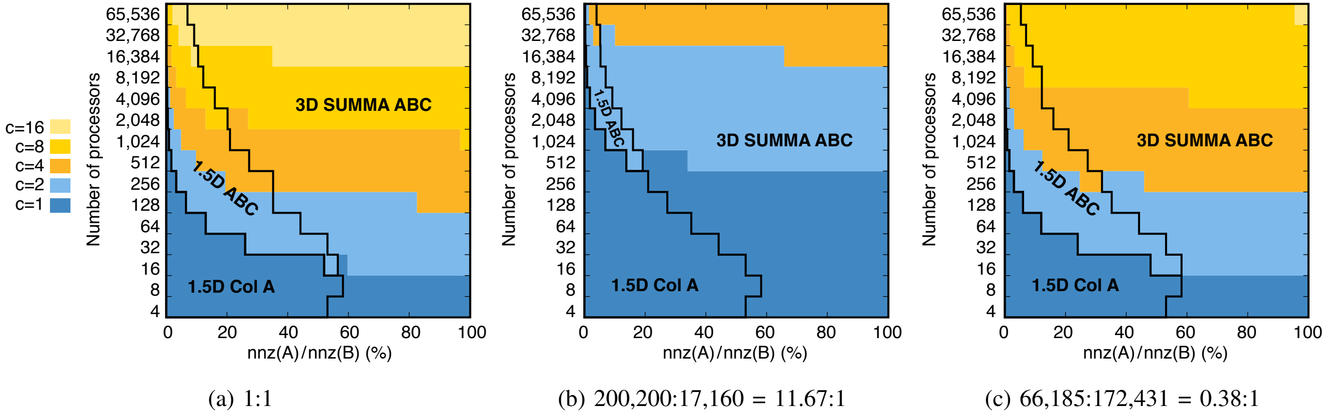


Fig. 4: Illustrating areas that each algorithm has theoretically lowest overall bandwidth cost. X-axis is the ratio of $\mathrm{nnz}(A)$ versus $\mathrm{nnz}(B)$. Y-axis is the number of processors. There are three subgraphs for three different $\mathrm{nnz}(C) : \mathrm{nnz}(B)$ ratios. *1.5D ABC* stands for both Col ABC and InnerABC. The area for *1.5D ABC* includes the area for *1.5D Col A*. Best replication factors for each data point are shown in colors. General observation is that ColA is best for sparser matrices or lower concurrency while SummaABC is the opposite. *1.5D ABC* algorithms help improve scalability of ColA.

In other words, they send $(2\,\mathrm{nnz}(B)+\mathrm{nnz}(C))/p\cdot c\log c$ more words to reduce the propagation bandwidth from $\mathrm{nnz}(A)$ to $\mathrm{nnz}(A)/c$. In general, $\mathrm{nnz}(A) \ll \mathrm{nnz}(B), \mathrm{nnz}(C)$ so p has to be considerably large for this trade-off to pay off. When ColABC and InnerABC are not replicating, they have equal overall bandwidth costs to ColA. SummaABC moves the dense matrix B in every phase so it is unlikely to beat any of the 1.5D algorithms in terms of bandwidth when A is very sparse. It will become preferable again when $\mathrm{nnz}(A)$ becomes closer to $\mathrm{nnz}(B)$, decreasing the message-size imbalance, or when the number of processors grows large (since it minimizes latency).

It is best to obtain hardware parameters to determine this latency-bandwidth trade-off. However, it would be great to see the big picture of where each algorithm is most suitable for without being specific to any particular machine. We found that the bandwidth costs are more prominent in our experiments, so we focus our analysis on just them for simplicity. Dividing the bandwidth costs in Table I with $\mathrm{nnz}(B)$ and representing nonzero ratios $\mathrm{nnz}(A)/\mathrm{nnz}(B) = f$ and $\mathrm{nnz}(C)/\mathrm{nnz}(B) = g$ eliminate one variable off the table. Knowing g, we can plot a graph with p and f as axes and search for the best algorithm over all possible c's at each point. We picked three different $\mathrm{nnz}(C) : \mathrm{nnz}(B)$ ratios (g), 1:1 in Figure 4a, 11.67:1 in Figure 4b, and 0.38:1 in Figure 4c. For an SpDM^3 problem, $\mathrm{nnz}(C) : \mathrm{nnz}(B) \approx m : \ell$ and can be interpreted as the *tallness* of matrix A. For example, 1:1 means square A's, 11.67:1 applies to tall A's, and 0.38:1 refers to rather fat A's. We draw black lines to separate between algorithms and use colors to show the best replication factors. The best replication factor for ColA is always 1 because it does not reduce bandwidth with increasing c. The area that ColA wins is a subset of the area that ColABC and InnerABC win. The graphs confirm the intuition from earlier analysis that ColA is most suitable with very sparse matrices or small scale runs. ColABC and InnerABC can help improve scalability to some level, but eventually SummaABC wins as we move towards larger concurrency or denser matrices.

Since this analysis is based on just $\mathrm{nnz}(A)$, $\mathrm{nnz}(B)$, and $\mathrm{nnz}(C)$, it is trivially applicable to sparse-sparse matrix-matrix multiplication (of different sparsities and/or sizes) or even dense-dense matrix-matrix multiplication (of different sizes).

V. Performance Results

We implemented all four algorithms listed in Table I using C++ and MPI. A is stored in zero-based indexing Compressed Sparse Row (CSR) format;[2] B and C are stored in row-major format, except where noted. We used the multi-threaded Intel® Math Kernel Library (MKL) for local sparse-dense matrix-matrix multiplication (*mkl_dcsrmm*). We ran our experiments on Edison, a Cray XC30 machine at the National Energy Research Scientific Computing Center (NERSC). Edison has a Cray Aries interconnect with a Dragonfly topology and consists of 5,576 compute nodes, each with 2 sockets of 12-core Intel Ivy Bridge processors running at 2.4GHz and with 64 GB memory. We used Intel's C++ compiler (icpc) version 15.0.1, Intel MKL version 11.2.1, and Cray MPICH version 7.3.1. All benchmarks are run with 2 MPI processes per node and 12-way multi-threaded MKL operation per process. We did not utilize Intel's Hyper-Threading Technology nor Turbo Boost Technology to avoid high performance variance.

A. Trends in Communication Costs

Figure 5 shows the cost breakdown of all algorithms running on $3,072$ processors (256 MPI processes). A is an Erdős-Rényi matrix with n=$65,536$ and 41 nonzeroes per row (0.0625% nonzeroes). The first two bars on the left belongs to SummaABC where all three matrices are replicated 1 (i.e., not at all) and 4 times, respectively. The next group is the ColA algorithm in which A is partitioned into block columns and replicated with the factors (c) shown above the algorithm's name. The last two groups are ColABC and InnerABC with

[2] The CSC (Compressed Sparse Column) format would scale better in terms of storage for the blocked column algorithms, but we found MKL's multiplication routine for the CSC format (*mkl_dcscmm*) significantly slower than the CSR's (*mkl_dcsrmm*), so we used CSR format in all implementations.

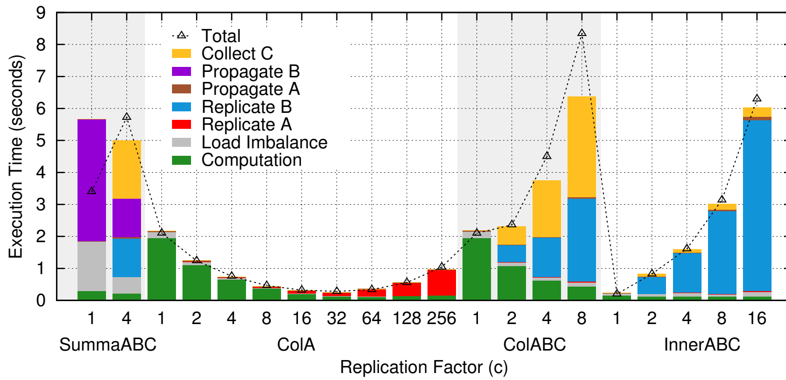


Fig. 5: Cost breakdown for multiplying a square Erdős-Rényi matrix of size $n = 65,536$ with 41 nonzeroes per row with a dense matrix of same size on $p = 3,072$ cores of Cray XC30 (256 MPI processes). All three 1.5D algorithms outperform SummaABC with ColA having the lowest communication time and also the best total running time. Only SummaABC moves the dense B matrix, so it is the slowest as expected.

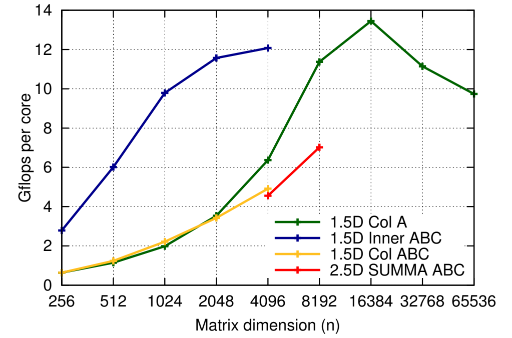


Fig. 6: Illustrating varying efficiency of single-node MKL matrix-multiplication for different matrix shapes used in Figure 5. n refers to different matrix-multiplication shapes for each algorithm: $n \times n \times n$ for SummaABC, $65,536 \times n \times 256$ for ColA, $65,536 \times n \times n$ for ColABC, and $n \times 65,536 \times n$ for InnerABC. For these problems, n=4096c for SummaABC and n=256c for the rest.

similar replication factors (c) shown in each label. All costs in the stacked bars are average costs over all processors.

The computation times in green are unequal even though all algorithms do the same amount of work. This is because the local MKL matrix-multiplication routine has varying efficiency for different shapes of input matrices. Figure 6 shows MKL performance for all of the relevant shapes and explains the variability in computation time in our algorithms. In general, MKL performs better on larger matrices, since they have higher computational intensity, although there is a dropoff in one case. To be precise, SummaABC performs $4\text{K} \times 4\text{K} \times 4\text{K}$ local matrix multiplications when $c = 1$, and $8\text{K} \times 8\text{K} \times 8\text{K}$ when $c = 4$. ColA does $64\text{K} \times 256c \times 256$ local matrix multiplications for $1 \leq c \leq 256$. ColABC does $64\text{K} \times 256c \times 256c$ local matrix multiplications for $1 \leq c \leq 16$. Finally, InnerABC does $256c \times 64\text{K} \times 256c$ local matrix multiplications for $1 \leq c \leq 16$. We exclude the bar of ColABC at $c = 16$ from Figure 6 because it is too tall.

Sometimes we found nontrivial variability across processors within an algorithm, even though all performing the same number of local multiplications on the same shape of matrices. We believe this is due to differences in the nonzero pattern of A and also from cache effects. To separate idle time due to load imbalance from useful computation or communication, there is an extra barrier after the computation phase for these time breakdown graphs. The barrier time can therefore be substantial and is shown in gray on top of the computation time. The total height of the stacked bars is the average total runtime of the run with barrier. We also show the maximum total runtime across all processors from similar runs without barriers in black dotted line.

For any of the algorithms with $c > 1$ for A or B, the time to replicate those matrices is shown in bright red and blue, respectively. Replication times increase linearly with c as predicted, although barrier costs decrease with c since the set of processors involved in a barrier is smaller.

In each step within the multiplication algorithm, the local matrices are broadcast or shifted right after local matrix multiplication. The time to propagate A (shift or broadcast), and propagate B (broadcast) are in brown and purple, respectively. ColA reduces S by c but does not reduce W, and its shift time stays the same but with moderate variance, which could be because it sends many small messages. Both ColABC and InnerABC reduce S by c^2 and W by c so we expect between two to four times reduction in *shift* or *broadcast* time as we double c. This trend might not be apparent in the graph since the duration is very short and there might be some overheads introduced. The SummaABC algorithm reduces S by $c^{3/2}$ and W by $\sqrt{c}$ and it does show a decrease in communication time by a factor of between $\sqrt{4} = 2$ to $4^{3/2} = 8$ as c is quadrupled in the graph. Since the factor seems closer to 2, it means that bandwidth is more prominent than latency on Edison, for our problem.

ColABC and SummaABC require a reduction of matrix C while InnerABC gathers C at the end. All of these are shown in yellow as collection cost. Even though gather asymptotically costs the same as reduce, in practice it can cost much less, because each processor only sends a message of size ranging from n^2/p to n^2c/p, depending on its position in the gather tree, instead of all n^2c/p in reduction. ColA has the best communication and also overall cost. At $c = 32$, it is 12.06 times faster than SummaABC, whose communication time is the worst because it also propagates the dense matrix.

B. Scalability

Figure 7 shows strong scaling performance on 384 to 12,288 cores for $65,536 \times 65,536$ Erdős-Rényi matrices with 1% nonzeroes for A. All our non-cost-breakdown graphs were run without barriers. For each algorithm at each number of cores, we report the best speedup over all available replication factors (c), so the graphs are not expected to be smooth or monotonic. Since the problem cannot fit into one node, we timed the multiplication on 2 nodes (48 cores) with the same Hybrid MPI configuration, and excluded communication time for a

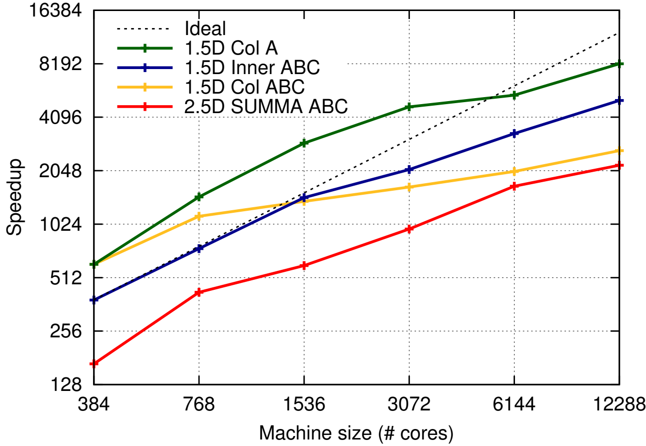


Fig. 7: Strong scaling of Erdős-Rényi matrices of size $n = 65,536$ with 1% nonzeroes on 384 to 12,288 cores of Cray XC30. Each point of each algorithm represents the best running time across all available replication factors (c). Ideal speedups are calculated from an estimated serial time from a 48-core run. ColA has the best overall speedup.

Fig.	Number of processors					
	384	768	1,536	3,072	6,144	12,288
7	2,①,1,1	1,1,1,②	2,1,1,⑧	1,2,1,⑯	2,2,2,㉜	4,1,4,⑧
8a	2,1,①,1	1,1,①,2	2,1,①,4	1,1,1,⑧	2,2,2,⑯	4,2,2,㉜
8b	2,1,1,②	1,1,1,②	2,1,1,④	1,1,1,⑧	2,2,2,⑯	4,2,2,㉜
9a	2,1,1,②	1,1,1,⑧	2,1,1,⑧	1,2,1,⑯	2,2,2,⑧	-
10a	-	1,1,①,64	2,1,①,64	1,2,①,64	2,2,①,64	-
11a	2,1,①,32	1,1,①,32	2,1,①,32	1,1,①,32	2,1,1,㉜	-

TABLE II: Showing the best replication factors (c) for each strong/weak scaling graph in the paper. Each cell lists the replication factors of the algorithms in the following order: SummaABC, ColABC, InnerABC, and ColA. The winning algorithm in each cell is circled. A dash means we did not run an experiment for that configuration.

baseline. We estimated the serial running time by multiplying this measured time by 48. The black dotted line indicates the ideal speedup.

The superlinear speedup of ColA and ColABC at the beginning was because of significantly faster computation time (again due to different MKL performance with different matrix sizes, see the trends in Figure 6). They both fell to sublinear speedup at larger scale where the edge in computation time was gone and the increasing communication time dominated. ColABC was also faster than InnerABC because of the faster computation time. SummaABC was outperformed by ColA by factors from $3.25\times$ to $4.89\times$, but it still has decent scalability.

We report the best replication factors in Table II. This experiment maps to a line in Figure 4a at $\text{nnz}(A)/\text{nnz}(B) = 1\%$ from 32 to 1,024 MPI processes. The figure predicts any of the 1.5D algorithms could win with $c = 1$ (or any c in case of ColA), which is true because InnerABC wins at 384 cores with $c = 1$, then ColABC wins the rest with $c > 1$. We also see larger replication factors as the number of cores increases, consistent with the trend in Figure 4. All algorithms have best c's greater than 1 on 6,144 cores onwards.

C. Non-uniform Distribution

Next, we experiment with matrices with non-uniform nonzero distribution. A Graph500 matrix A is generated with RMAT parameters $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$ [33]. Using these parameters, RMAT is known to create a matrix with skewed distribution (approximating a power-law distribution if some noise is added [34]) of nonzero row and column counts. We deviated from the average edge factor (nonzero row/column count) suggested by the Graph500 benchmark, in order to stay consistent with the density of Erdős-Rényi matrices we used. We also modified our 1.5D algorithms to partition work based on equal number of nonzeroes (using greedy algorithm) instead of number of rows or columns to mitigate the expected load imbalance.

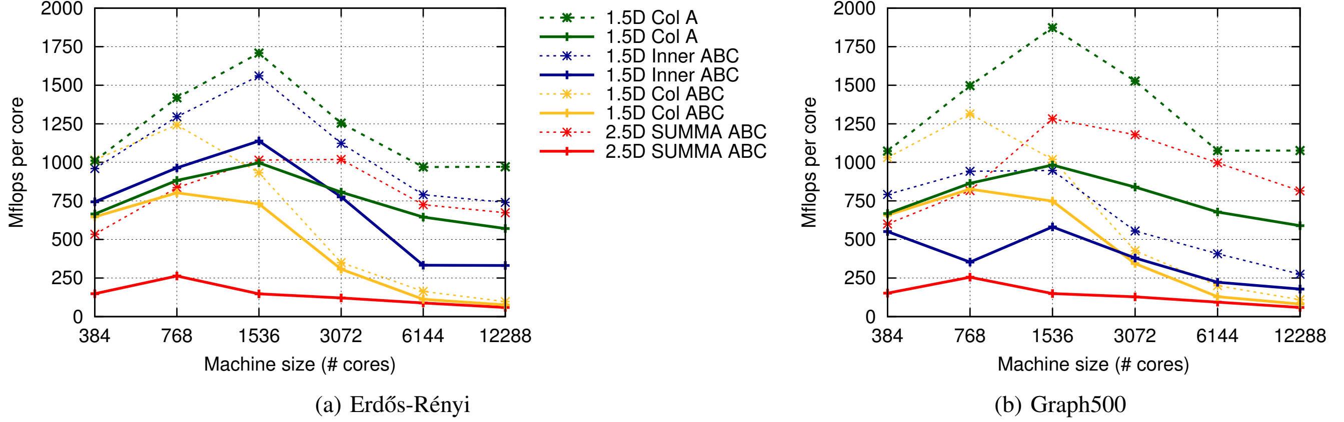
Figure 8 compares weak scaling performance of Erdős-Rényi versus Graph500 matrices. We fix the number of nonzeroes per row to $d = 164$ and vary the number of cores from 384 to $12,288$ cores. We start with $4,096 \times 4,096$ matrices (4% nonzeroes) on 384 cores and double the matrix size as we double the number of cores, ending with $131,072 \times 131,072$ (0.125% nonzeroes) matrices on $12,288$ cores. Some points from Graph500 have performance than Erdős-Rényi because of faster computation time. This might be due to the different structures of nonzeroes. Dotted lines show the weak scaling of the actual computation times of each algorithm. The data for InnerABC in Figure 8b are collected with one-based indexing version of multithreaded *mkl_dcsrmm* because the zero-based multithreaded version did not return when called with some local matrices specific to InnerABC's partitioning. Its performance in Figure 8b is lower than expected because the one-based indexing version has slower computation time.

We still observe the same performance trend for all algorithms in the Graph500 results without any significant load imbalance. ColA has highest speedup over SummaABC at $12,288$ cores, with $9.64\times$ speedup for Erdős-Rényi matrix and $9.94\times$ speedup for Graph500 matrix.

D. Real-world Matrices

Our final experiments test on three real-world matrices of different shapes from the University of Florida Sparse Matrix Collection [35]. Each of these sparse matrices (A) is multiplied by a generated dense matrix B of the same size as A^T.

Mouse gene network from V. Belcastro (mouse_gene.mtx) is a square, symmetric matrix of size $45,101 \times 45,101$ with $28,967,291$ nonzeroes (degree $d = 642.28$, 1.424% nonzeroes). Figure 9a and Figure 9b show the strong-scaling and cost-breakdown graphs. See Figure 4a at $\text{nnz}(A)/\text{nnz}(B) = 1.42\%$ for its bandwidth plot. The data for InnerABC at 3,072 and 6,144 cores are collected with one-based indexing multithreaded *mkl_dcsrmm* again due to the same zero-based indexing multithreaded MKL issue mentioned in Section V-C. These points are significantly slower than the points from 384 to 1,534 cores partially because they use slower routine. We get similar results to past experiments, except this time we see noticeably more load imbalance despite the greedy partitioning. ColA still performs best, followed by InnerABC,



(a) Erdős-Rényi

(b) Graph500

Fig. 8: Weak Scaling of square Erdős-Rényi and Graph500 matrices with fixed $d = 164$ nonzeroes per row on $p = 384, 768, 1536, 3072, 6144,$ and 12288 cores of Cray XC30 with corresponding matrix sizes of $n = 4096, 8192, 16384, 32768, 65536,$ and 131072 (4%, 2%, 1%, 0.5%, 0.25%, and 0.125% nonzeroes), respectively. Flops rates from only the computation time are shown in dotted lines and explain the performance jump of the algorithms around 768 and 1,536 cores. The skewed distribution for this particular Graph500 matrix does not introduce substantial load imbalance, yielding similar performance to Erdős-Rényi's. It also increases the computation efficiency in some cases.

ColABC, and SummaABC. The maximum speedup is $5.27\times$ from ColA over SummaABC on 384 cores.

Simplicial complexes from V. Welker (shar_te2-b2.mtx) is a tall matrix with dimensionality $200,200 \times 17,160$ with $600,600$ nonzeroes (degree $d = 3$, 0.0175% nonzeroes). The strong scaling and cost breakdown graphs are shown in Figure 10a and Figure 10b. The corresponding bandwidth plot is shown in Figure 4b. We observed mild load imbalance. The computation time for ColA and ColABC is higher than others because their local matrix shapes are tall and skinny. InnerABC has local matrices with better aspect ratio so it performs best in this scenario. The reduction time for ColABC and SummaABC is also high because the resulting matrix C is fairly large. The message sizes are very skewed: $\text{nnz}(C) \gg \text{nnz}(B) \gg \text{nnz}(A)$. The highest speedup is $38.24\times$ at 1,536 cores, between InnerABC and SummaABC.

Stochastic linear programming problem from C. Meszaros (stormg2-125.mtx) is a fat matrix of size $66,185 \times 172,431$ with $433,256$ nonzeroes (degree $d = 6.55$, 0.0038%). Figure 11a and Figure 11b show the strong scaling and cost breakdown graphs. According to the bandwidth plot in Figure 4c, any of the 1.5D algorithms could win. Again, InnerABC wins with $c = 1$ because of faster computation time. We also observed load imbalance. The largest speedup is $99.55\times$, between InnerABC and SummaABC at 1,536 cores.

VI. Iterative Multiplication

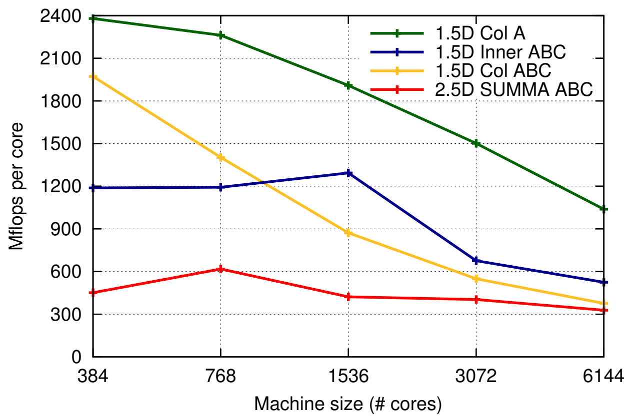
Many algorithms in statistical/machine learning are iterative: first an initial solution is chosen, then iterative updates are applied until some convergence criteria is met. Hence, if the updates involve SpDM^3 evaluations, total computational time spent on these evaluations can add up quickly. Also, required number of iterations are not known apriori, and may vary drastically depending on many factors. In case of CONCORD-ISTA, for example, number of iterations can depend on penalty parameter, (numerical) rank of the input data, choice of step size, etc [14]. In this section, we illustrate various methods that can potentially benefit from using SpDM^3.

Let X denote a matrix of dimension $r \times n$, where r-rows are independent observations of an n-dimensional random vector. Such matrix can represent data from various scientific disciplines including neuroscience, biology, and even social sciences. For example, X may be fMRI scan data collected for r time periods over n voxels [36], expressions of n genes from r individuals [37], or voting patterns [37]. In many high dimensional datasets, dimensions of matrix X is such that $r \ll n$, which we will assume is the case here.

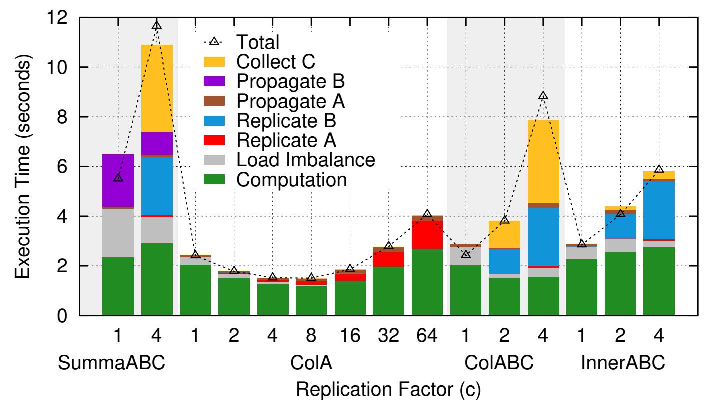
Suppose $B = X^T X$ so that A, B and C are all of size $n \times n$. Suppose there are s iterations, each of which consists of one global matrix multiplication ($C = AB$), The replication costs (red and blue bars in Figure 5) is only paid once. The propagation costs (brown and purple) and computation costs (green and gray) recur every iteration. CONCORD-ISTA uses element-wise soft-thresholding operator which depends on the total magnitude of c_{ij}, so a per-iteration reduction is needed for ColABC and SummaABC. ColA and InnerABC store an entire element c_{ij} on a single layer so they do not need to pay collection costs in each iteration. Assuming ColABC and InnerABC must pay collection costs every iteration, the total bandwidth costs are

$$\begin{aligned}
W_{\text{ColA}} &= 2\frac{\text{nnz}(A)}{p} c \log c + \text{nnz}(A) s, \\
W_{\text{ColABC}} &= 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \left(\frac{\text{nnz}(A)}{c} + \frac{\text{nnz}(C)}{p} c \log c \right) s, \\
W_{\text{InnerABC}} &= 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \left(\frac{\text{nnz}(A)}{c} \right) s + \frac{\text{nnz}(C)}{p} c \log c, \\
W_{\text{SUMMA}} &= 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \\
&\quad \left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{pc}} \log \sqrt{\frac{p}{c}} + \frac{\text{nnz}(C)}{p} c \log c \right) s.
\end{aligned}$$

If s is large enough then both the replication and collection terms of InnerABC with dense matrices can be amortized,

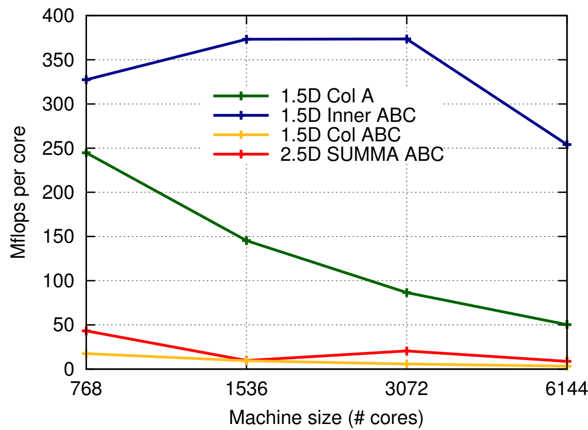


(a) Strong scaling

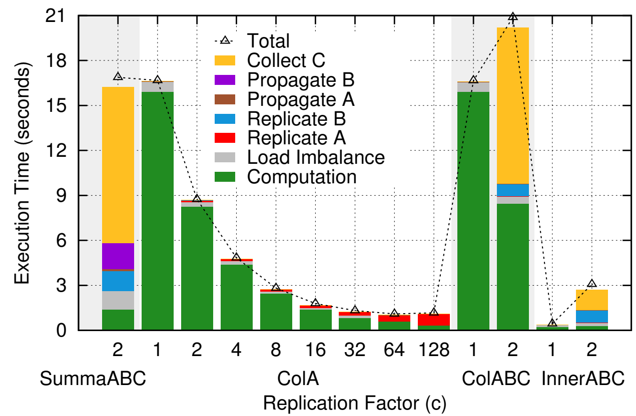


(b) Cost-breakdown graph on 768 cores

Fig. 9: Strong scaling and cost-breakdown results from multiplying the Mouse gene network matrix (*mouse_gene.mtx*) ($45,101 \times 45,101$, 1.42% nonzeroes) with a dense $45,101 \times 45,101$ matrix. ColA performs the best again as this configuration is well in its bandwidth-winning region (a vertical line at 1.42% in Figure 4a from 32 to 512 MPI processes).

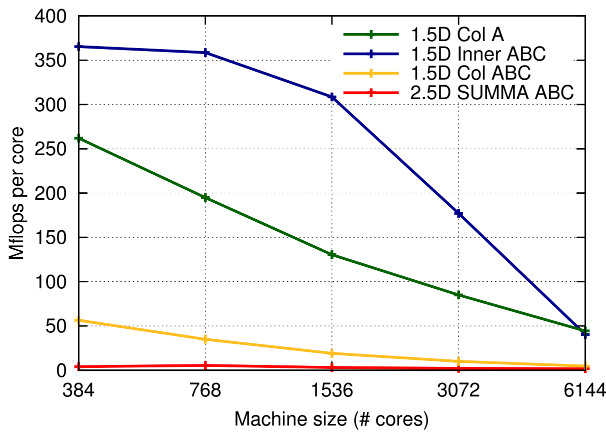


(a) Strong scaling

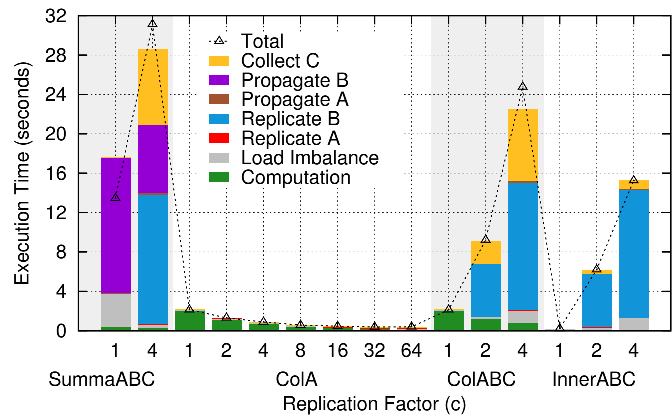


(b) Cost-breakdown graph on 1,536 cores

Fig. 10: Strong scaling and cost-breakdown results from multiplying the Simplicial complexes matrix (*shar_te2-b2.mtx*) ($200,200 \times 17,160$, 0.0175% nonzeroes) with a dense $17,160 \times 200,200$ matrix. InnerABC wins over ColA because ColA does tall-skinny local matrix multiplications which is significantly slower than InnerABC's fatter local matrices.



(a) Strong scaling



(b) Cost-breakdown graph on 768 cores

Fig. 11: Strong scaling and cost-breakdown results from multiplying the Stochastic linear programming problem matrix (*stormg2-215.mtx*) ($66,185 \times 172,431$, 0.0038% nonzeroes) with a dense $172,431 \times 66,185$ matrix.

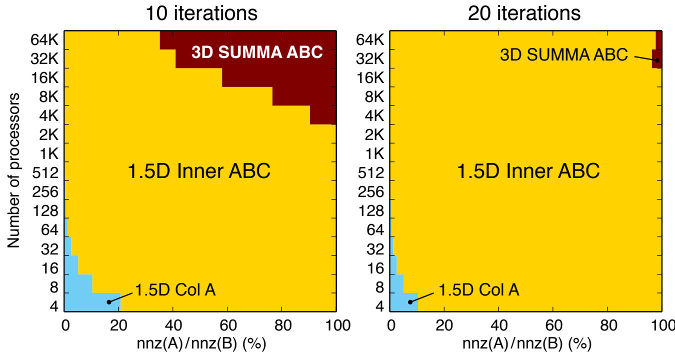


Fig. 12: Predicting the areas that each algorithm would have the lowest overall bandwidth cost after 10 and 20 iterations. X-axis is the ratio of $\mathrm{nnz}(A)$ versus $\mathrm{nnz}(B)$. Y-axis is the number of processors.

making InnerABC more desirable, since it does not move any dense matrix in propagation, does not need to collect every iteration, and it has potentially better propagation cost than ColA. Figure 12 illustrates the area each algorithm would have the lowest bandwidth cost for 10 and 20 iterations of multiplication. The area where InnerABC is best intuitively increases with the number of iterations.

In addition to CONCORD-ISTA, there are other iterative learning algorithms that compute one or more SpDM^3 at every iteration [6], [3]. Although it is difficult to gauge the exact extent of computational impact SpDM^3 would have on these algorithms, it would be reasonable that these algorithms would be able to solve higher dimensional problems than in shorter amount of wall-clock time than possible on a single processor.

VII. Conclusions

We presented four variations on parallel sparse-dense matrix-matrix multiplication (SpDM^3), all based on a traditional $O(n^3)$ algorithm, but using different approaches to replicating data and partitioning work to minimize communication costs. One of these is the 3D SUMMA algorithm and the other three represent new parallelization strategies specific to a setting involving sparse matrices. We derived communication lower bounds for the problem, then presented an analysis of new and existing algorithms, and compared their costs both theoretically and experimentally on over 10 thousand cores. The problem was motivated by iterative algorithms in machine learning, and both our experiments and cost analysis break the running time into parts to show how the algorithms would compare in such a setting — some parts are one-time costs and others occur at each iteration.

Our analysis shows that no single algorithm is optimal for all settings, but that the choice depends on sparsity, matrix size, available memory, and machine size. We show that when the theoretical analysis shows a difference in cost, it is a good predictor of which algorithm to use. The theory is quite general, and uses the number of matrix entries (nonzeroes) of each matrix, independent of whether the matrix is dense or sparse. Thus, while our experiment focus on the dense-sparse case, the algorithms are relevant to other settings in which one of the two matrices is much larger or denser than another. We give guidelines on how to choose between algorithms in terms of graphs indicating what area each algorithm would have the lowest bandwidth cost. The four algorithms each have benefits for some cases:

- **SummaABC** (previously known) is best with relatively dense matrices or very large processor counts. Because it moves all matrices during multiplication, it is suboptimal when one is significantly smaller or sparser.
- **ColA** is better with sparser A matrices or smaller scale parallelism. (An analogous algorithm that replicates B would work for the dense-sparse case.)
- **ColABC** and **InnerABC** generally work in ColA's range and also the intermediate range between ColA and SummaABC in both matrix density and processor count. They have equivalent theoretical communication costs, but InnerABC is faster in practice, sometimes substantially so.

Since sparse matrices rarely have more than a few percent of nonzeroes, the majority of SpDM^3 will be in ColA's area, which means the best algorithm could be ColA with any c, non-replicating ColABC, or non-replicating InnerABC. Our experimental results matched this trend. We observed up to $100\times$ speedup over SummaABC. Replicating ColABC or InnerABC will likely be more beneficial in iterative multiplication rather than in single multiplication.

Our models correctly predict the trends of all communication costs, and generally predict the faster algorithms and parameter settings, but they do not consider computation cost. In practice, MKL library performance varies when matrix shapes are different, with the usual observation that larger matrices and low aspect ratio matrices run at a higher machine efficiency. This is not accounted for in our theory, but the low communication algorithms also tend to have larger local matrices, so it adds to the benefit. This omission in the model does lead to substantial mis-predictions of computation time that sometimes are a deciding factor in which algorithm wins, for example, it often is a tie-breaker between ColA, ColABC at $c = 1$, and InnerABC at $c = 1$. ColA and ColABC gets faster computation time when they replicate, but ColABC also replicates the dense matrix and has to pay a much higher cost to get to the same computation efficiency as ColA. This and the fact that it uses reduction is why we often found it inferior to both ColA and InnerABC. Because of this, InnerA (blocked inner product replicating only A) might be worth investigating as well. A future analysis should take this unequal computation time into account.

In addition to the synthetic Erdős-Rényi, Graph500, and a few real-world matrices tested here, future work would involve a larger set of matrices from real machine learning problems. We are interested to see how the 1.5D algorithms would perform compared to SUMMA in such setting and expect that some type of graph partitioning may prove important. We are also under the process of implementing the CONCORD-ISTA [14] iterative machine learning algorithm with our SpDM^3 algorithms.

ACKNOWLEDGMENTS

The *1.5D Inner ABC* algorithm originated from unpublished work in a different context with Evangelos Georganas, Edgar Solomonik, Yili Zheng, and Samuel Williams.

This work was supported in part by the Applied Mathematics and Computer Science Programs of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory, including the XStack program and through use of computing resources at the National Energy Research Scientific Computing Center. Koanantakool was also supported by the Fulbright scholarship and by DOE contract DE-SC0008700 at UC Berkeley. Oh was supported by Laboratory Directed Research and Development (LDRD) at LBNL.

REFERENCES

[1] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *IPDPS*. IEEE, 2014, pp. 1213–1222.

[2] A. Tiskin, "All-pairs shortest paths computation in the BSP model," in *ICALP*, 2001, pp. 178–189.

[3] J. Kim and H. Park, "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011.

[4] M. McCourt, B. Smith, and H. Zhang, "Sparse matrix-matrix products executed through coloring," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 90–109, 2015.

[5] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby, "Quantum Monte Carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond," *Journal of computational chemistry*, vol. 34, no. 11, pp. 938–951, 2013.

[6] K. Fujisawa, M. Kojima, and K. Nakata, "Exploiting sparsity in primal-dual interior-point methods for semidefinite programming," *Mathematical Programming*, vol. 79, no. 1-3, pp. 235–253, 1997.

[7] G. C. Pena, S. V. Magalhães, M. V. Andrade, W. R. Franklin, C. R. Ferreira, and W. Li, "An efficient GPU multiple-observer siting method based on sparse-matrix multiplication," in *ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. ACM, 2014, pp. 54–63.

[8] A. P. Dempster, "Covariance selection," *Biometrics*, vol. 28, no. 1, pp. pp. 157–175, 1972.

[9] M. Yuan and Y. Lin, "Model selection and estimation in the Gaussian graphical model," *Biometrika*, vol. 94, no. 1, pp. 19–35, 2007.

[10] J. Friedman, T. Hastie, and R. Tibshirani, "Sparse inverse covariance estimation with the graphical lasso," *Biostatistics*, vol. 9, no. 3, pp. 432–441, 2008.

[11] J. Peng, P. Wang, N. Zhou, and J. Zhu, "Partial correlation estimation by joint sparse regression models," *Journal of the American Statistical Association*, vol. 104, no. 486, pp. 735–746, 2009.

[12] K. Khare, S.-Y. Oh, and B. Rajaratnam, "A convex pseudolikelihood framework for high dimensional partial correlation estimation with convergence guarantees," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 77, no. 4, pp. 803–825, 2015.

[13] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, P. K. Ravikumar, and R. Poldrack, "Big & quic: Sparse inverse covariance estimation for a million variables," in *NIPS*, 2013, pp. 3165–3173.

[14] S. Oh, O. Dalal, K. Khare, and B. Rajaratnam, "Optimization methods for sparse pseudo-likelihood graphical model selection," in *NIPS 27*, 2014, pp. 667–675.

[15] M. Bader and A. Heinecke, "Cache oblivious dense and sparse matrix multiplication based on Peano curves," in *Proceedings of the PARA*, vol. 8, 2008.

[16] G. Greiner and R. Jacob, "The I/O complexity of sparse matrix dense matrix multiplication," in *LATIN 2010: Theoretical Informatics*. Springer, 2010, pp. 143–156.

[17] G. Ortega, F. Vázquez, I. García, and E. M. Garzón, "Fastspmm: An efficient library for sparse matrix matrix product on GPUs," *The Computer Journal*, vol. 57, no. 7, pp. 968–979, 2014.

[18] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal, "Space-round tradeoffs for MapReduce computations," in *ICS*. ACM, 2012, pp. 235–244.

[19] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[20] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *IPDPS*. IEEE, 2008.

[21] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[22] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par*, 2011, vol. 6853, pp. 90–109.

[23] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *SPAA*. ACM, 2013, pp. 222–231.

[24] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *arXiv preprint arXiv:1510.00844*, Oct 2015.

[25] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae*, vol. 6, no. 1, pp. 290–297, 1959.

[26] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[27] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[28] A. Aggarwal, A. K. Chandra, and M. Snir, "Communication complexity of PRAMs," *Theoretical Computer Science*, vol. 71, no. 1, pp. 3 – 28, 1990.

[29] R. van de Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurr. Comput. : Pract. Exper.*, vol. 9, no. 4, pp. 255–274, 1997.

[30] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM J. Res. Dev.*, vol. 39, no. 5, pp. 575–582, Sep. 1995.

[31] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick, "A communication-optimal n-body algorithm for direct interactions," in *IPDPS*, 2013, pp. 1075–1084.

[32] P. Koanantakool and K. Yelick, "A computation- and communication-optimal parallel direct 3-body algorithm," in *ACM/IEEE SC'14*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 363–374.

[33] "Graph500 benchmark," www.graph500.org.

[34] C. Seshadhri, A. Pinar, and T. G. Kolda, "An in-depth study of stochastic Kronecker graphs," in *ICDM*. IEEE, 2011, pp. 587–596.

[35] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[36] I. Cribben, T. Wager, and M. Lindquist, "Detecting functional connectivity change points for single-subject fMRI data," *Frontiers in Computational Neuroscience*, vol. 7, no. 143, 2013.

[37] O. Banerjee, L. El Ghaoui, and A. d'Aspremont, "Model selection through sparse maximum likelihood estimation for multivariate Gaussian or binary data," *J. Mach. Learn. Res.*, vol. 9, Jun. 2008.