Title

Communication-Avoiding and Memory-Constrained Sparse Matrix-Matrix Multiplication at Extreme Scale

Authors

Hussain, Md Taufique
Selvitopi, Oguz
Buluç, Aydin
et al.

Publication Date

2020-10-16

Peer reviewed

Communication-Avoiding and Memory-Constrained Sparse Matrix-Matrix Multiplication at Extreme Scale

Md Taufique Hussain*, Oguz Selvitopi†, Aydin Buluç‡ and Ariful Azad§
* Indiana University, Bloomington, IN (mth@indiana.edu)
† Lawrence Berkeley National Laboratory, Berkeley, CA (roselvitopi@lbl.gov)
‡ Lawrence Berkeley National Laboratory, Berkeley, CA (abuluc@lbl.gov)
§ Indiana University, Bloomington, IN (azad@iu.edu)

***Abstract*—Sparse matrix-matrix multiplication (SpGEMM) is a widely used kernel in various graph, scientific computing and machine learning algorithms. In this paper, we consider SpGEMMs performed on hundreds of thousands of processors generating trillions of nonzeros in the output matrix. Distributed SpGEMM at this extreme scale faces two key challenges: (1) high communication cost and (2) inadequate memory to generate the output. We address these challenges with an integrated communication-avoiding and memory-constrained SpGEMM algorithm that scales to 262,144 cores (more than 1 million hardware threads) and can multiply sparse matrices of any size as long as inputs and a fraction of output fit in the aggregated memory. As we go from 16,384 cores to 262,144 cores on a Cray XC40 supercomputer, the new SpGEMM algorithm runs 10x faster when multiplying large-scale protein-similarity matrices.**

I. Introduction

Multiplication of two sparse matrices (SpGEMM) is a common operation in numerous computing fields including data analytics [1, 2], graph processing [3]–[5], bioinformatics [6, 7], machine learning [8], computational chemistry [9], and scientific computing [10]. Most use cases of SpGEMM have low arithmetic intensity, resulting in poor scaling to large concurrencies. Despite these inherent challenges, the need to scale SpGEMM on large distributed-memory clusters for solving important science problems generated a fruitful line of research. As a result, the last decade saw the development of increasingly sophisticated distributed-memory parallel SpGEMM algorithms that employ communication-avoiding techniques, perform communication overlapping, and partitioning the matrices to minimize communication [11, 12].

Different from other matrix multiplication instances such as dense-dense and sparse-dense, SpGEMM has four unique features that make distributed SpGEMM harder to develop. (1) The output of SpGEMM often has more nonzeros than the inputs combined. In some applications, the output of SpGEMM may not even fit in the aggregate memory of large supercomputers. (2) the number of nonzeros of the output matrix is not known a-priori for SpGEMM. Hence, a distributed symbolic step is needed to estimate the memory required for the multiplication. (3) SpGEMM typically has low arithmetic intensity. As a result, local computations on each node becomes computationally expensive. (4) At extreme scale, inter-process communication becomes the performance bottleneck. Since the last two problems arise in all distributed SpGEMM, they are well studied in the literature [11]–[13]. By contrast, the first two problems are relatively new to the community as SpGEMM performed with emerging biological and social networks has started to overrun the memory limit of supercomputers.

This paper presents a distributed-memory algorithm and its high-performance implementation that harmoniously address all four challenges. (1) When the memory requirement of SpGEMM exceeds the aggregate memory, we multiply in b batches, where a batch computes a subset of columns of the output. (2) The required number of batches depends on the aggregate memory, input matrices, and the process grid used to distribute the matrix. We developed a distributed symbolic algorithm that efficiently identifies the memory requirements and the optimal number of batches needed. (3) We integrate the batching technique within an existing communication-avoiding framework [13] that multiplies matrices on a 3D process grid. (4) We observe that in-node SpGEMM and merging routines needed by distributed SpGEMM do not need nonzeros sorted within columns of input and output matrices. In light of this observation, we developed new sort-free SpGEMM and merging algorithms to make local computations significantly faster than current state-of-the-art approaches. Thus, this paper presents novel solutions to four challenges in large-scale SpGEMM, and it does so within the established communication-avoiding algorithms framework.

The proposed memory-constrained SpGEMM algorithm can be directly used with applications that do not access the whole output matrix all at once. For example, consider finding the Jaccard similarity between two datasets, a problem that is successfully formulated as multiplication of a sparse matrix with its transpose [14]. BELLA sequence overlapper [7] and PASTIS many-to-many protein sequence aligner [15] use the SpGEMM operation in a similar way. Since the subsequent analysis only needs to access subsets of the output $\mathbf{AA}^\mathsf{T}$, we can form it in batches, and discard each batch. In hypergraph partitioning using the effective multi-level paradigm, successively coarser graphs are generated. Prior to coarsening, one typically finds the number of shared hyperedges between all pairs of vertices in order to run a matching algorithm that identifies pairs of vertices to coarsen. This process is called heavy-connectivity matching [16] or inner-product matching [17], and can be formulated as another SpGEMM of type $\mathbf{AA}^\mathsf{T}$. Due

to memory limitations and the higher density of the product, this SpGEMM is done in batches in distributed-memory multilevel partitioners such as Zoltan [18]. Finally, HipMCL [19] is a distributed-memory scalable version of the Markov clustering algorithm. HipMCL iterations involve matrix squaring followed by various pruning steps to reduce the memory footprint and increase convergence. In both HipMCL and hypergraph coarsening, columns of the output can be formed in batches, and immediately used for pruning or matching without requiring the whole output.

In these scenarios, the higher-level application can form subsets of the output matrix in batches, perform the required computation on it, and discard some or all of its nonzeros before moving into the next batch. The solutions provided in the literature for all these examples are ad-hoc algorithms. The community lacks a thorough understanding of theoretical and practical performance that can be expected from a memory-constrained SpGEMM that operates in batches. This results in late discoveries of performance and scalability problems in large-scale runs. Our work fills that crucial gap.

Our integrated communication-avoiding (CA) and memory-constrained SpGEMM algorithm is generally applicable even when forming the full output is feasible. In that case, our algorithm will choose to form the output all at once if that minimizes communication. Our algorithm strong scales naturally well because the increase in the available memory allow our integrated algorithm to either decrease the number of passes over the input, or increase replication in exchange of reduced communication, or do both.

Overall, the faster local computation, reduced communication and batching techniques delivered a massively parallel algorithm that scales to millions of threads on a Cray XC40 supercomputer. Using batching, our SpGEMM can multiply massive matrices using 0.5 PB memory whereas previous SpGEMMs would have required 2.2 PB memory, thereby could not solve the problem at all.

The main contributions of the paper are summarized below.

1) We develop a new distributed-memory SpGEMM algorithm that multiplies matrices in batches when the required memory to generate the output exceeds the available memory. This algorithm makes it possible to multiply sparse matrices of any size as long as inputs and a fraction of output fit in the memory.
2) We adapt existing CA techniques in a memory-constrained setting. We develop new algorithms to reduce the overhead of CA algorithms due to batching.
3) We develop lower and upper bounds on the number of batches and present a symbolic algorithm to compute the exact number of batches needed.
4) We demonstrate that our SpGEMM algorithm scales to 4096 nodes (more than 1 million hardware threads) on a Cray XC40 supercomputer. The utility of the SpGEMM algorithm is demonstrated with large-scale protein similarity networks where the multiplication requires up to 300 trillion floating point operations and 2.2 PB memory.

II. Background and Notations

A. Notations

Given two sparse matrices $\mathbf{A}\in\mathbb{R}^{m\times k}$ and $\mathbf{B}\in\mathbb{R}^{k\times n}$, SpGEMM multiplies $\mathbf{A}$ and $\mathbf{B}$ and computes another potentially sparse matrix $\mathbf{C}\in\mathbb{R}^{m\times n}$. The operation can be also performed on an arbitrary semiring $\mathbb{S}$ instead of the field of real numbers $\mathbb{R}$ and our algorithms are applicable to that case as well since we do not utilize Strassen-like algorithms. In our analysis, we consider n-by-n matrices (that is, $m{=}k{=}n$) for simplicity. Given a matrix $\mathbf{A}$, *nrows*($\mathbf{A}$) and *ncols*($\mathbf{A}$) denote the number of rows and columns in $\mathbf{A}$. *nnz*($\mathbf{A}$) denotes the number of nonzeros in $\mathbf{A}$. flops denotes the number of multiplications needed to compute $\mathbf{AB}$. The *compression factor* (*cf*) is the ratio of flops to *nnz*($\mathbf{C}$): *cf*=flops/*nnz*($\mathbf{C}$). Since at least one multiplication is needed for an output nonzero, *cf*$\geq$1.

B. Problem Description

In many data analytics applications that use SpGEMM, the output matrix $\mathbf{C}$ is often too large to store all at once in aggregate memory. Fortunately, the subsequent analysis rarely requires access to the whole $\mathbf{C}$, so it can be formed in *batches*. Our paper is about this *memory-constrained SpGEMM* formulation where the available aggregate memory $M = nnz(\mathbf{C})/k$ where $k > 1$. We also assume that the inputs fit into aggregate memory, meaning $M > nnz(\mathbf{A}) + nnz(\mathbf{B})$. Together, these two conditions imply $nnz(\mathbf{C}) > k(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ for $k > 1$. Note that the actual memory requirement of distributed SpGEMM can be much larger than $O(nnz(\mathbf{C}))$ because of the need to store unmerged results in each process. Our algorithm also covers the case where the final output fits in the memory, but the intermediate results exceed the available memory.

C. Related work

The algorithms developed for parallel SpGEMM stem from how the matrices and the relevant computations are distributed across parallel computing units. Most of the shared-memory parallel SpGEMM algorithms rely on Gustavson's algorithm [20], which yields a high level of parallelism as the rows or the columns of the output matrix can be computed independent of each other. In the column variant, a number of columns of $\mathbf{A}$ need to be scaled and accumulated in order to compute the output column $\mathbf{C}(:, j)$, i.e.,

$$\mathbf{C}(:, j) = \sum_{i:\mathbf{B}(i,j)\neq 0} \mathbf{A}(:, i)\mathbf{B}(i, j).$$

The algorithms usually differ in the data structure they use for accumulating values. Among them are the sparse accumulators [21, 22], heaps [13, 23], hash tables [24, 25], and merge sorts [26, 27]. The accumulator choice has important implications on the performance of the parallel algorithm and often depends on sparsity pattern of the multiplied matrices, compression factor, and parallel architecture.

The studies regarding parallelization of SpGEMM on many- and multi-core systems often focus on optimization aspects related to accumulators, efficient data access, and load balancing.

Algorithm 1 An overview of 2D sparse SUMMA algorithm.

Input and Output: Input matrices $\mathbf{A}$ and $\mathbf{B}$ and output matrix $\mathbf{C}$ are distributed on a 2D process grid P_{2D}. $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$, and $\tilde{\mathbf{C}}$ denote local submatrices in the process considered.

1: **procedure** SUMMA2D($\mathbf{A}, \mathbf{B}, P_{2D}$)
2: **for** all processes in $P_{2D}(i,j)$ **in parallel do**
3: stages $\leftarrow$ number of process columns in P_{2D}
4: **for** $s \leftarrow 1$ to stages **do** ▷ SUMMA stages
5: $\tilde{\mathbf{A}}_{\text{recv}} \leftarrow$ BCAST($\tilde{\mathbf{A}}, P_{2D}(i,s), P_{2D}(i,:)$)
6: $\tilde{\mathbf{B}}_{\text{recv}} \leftarrow$ BCAST($\tilde{\mathbf{B}}, P_{2D}(s,j), P_{2D}(:,j)$)
7: $\tilde{\mathbf{C}}[s] \leftarrow$ LOCALMULTIPLY($\tilde{\mathbf{A}}_{\text{recv}}$, $\tilde{\mathbf{B}}_{\text{recv}}$)
8: $\tilde{\mathbf{C}} \leftarrow$ MERGE($\tilde{\mathbf{C}}$[1..stages])
9: **return** $\mathbf{C}$

As the many-core architectures necessitate a fine-grain load balancing for good performance, several works on GPUs [23, 24, 26, 28] aim to achieve that goal. The works for multi-core architectures [22, 25] often strive for improving the poor cache behavior with various techniques such as blocking or using appropriate accumulators according to *cf.*

The distributed memory algorithms for parallel SpGEMM can be categorized according to the data distribution method they use. The algorithms that rely on one-dimensional (1D) distribution partition all matrices across the entire row or column dimension. Although 1D distribution [29] suffers from poor communication scalability, the communication costs can be reduced by pre-processing with graph/hypergraph partitioning models [11] that exploit the sparsity pattern of the multiplied matrices in order to reduce communicated data. This pre-processing, however, can be prohibitive if SpGEMM is not utilized in a repeated context as the pre-processing stage often does not scale well. CombBLAS [5] uses Sparse SUMMA algorithm [30, 31] tailored for 2D distribution of matrices. In 2D distribution, the matrices are partitioned into rectangular blocks and a 2D process grid is associated with the distribution. The 2D distribution has better communication characteristics compared to 1D distribution. In the 3D variant of the Sparse SUMMA algorithm, each sub-matrix is further divided into layers. This approach exhibits better scalability at larger node counts [13, 32], where the multiplied instances become more likely to be latency-bound. We examine these algorithms in detail in Sections III-A and III-B, respectively, as they form the basis of our work. Cannon's algorithm [33], which uses a 2D distribution of matrices, has also been used in parallel SpGEMM [9].

III. 2D AND 3D SPARSE SUMMA ALGORITHMS

Our algorithmic framework is based on the 2D SUMMA algorithm [30, 31], and our communication-avoiding technique relies on the 3D SUMMA algorithm developed in prior work [13]. In this section, we revisit 2D and 3D sparse SUMMA algorithms by putting them in a common framwork upon which the new memory-constrained algorithm will be built in the next section.

A. 2D Sparse SUMMA

Data distribution. The original sparse SUMMA algorithm [30] works on a 2D $p_r \times p_c$ process grid P_{2D}. In this paper, we only consider square process grid with $p_r = p_c$. $P_{2D}(i,j)$ denotes the process in ith row and jth column in the 2D grid. $P_{2D}(i,:)$ denotes all processes in the ith row and $P_{2D}(:,j)$ denotes all processes in the jth column.

The algorithm. The SUMMA2D function in Algorithm 1 describes the 2D multiplication algorithm that operates on matrices distributed on P_{2D}. SUMMA2D proceeds in p_c stages. At stage s, each process $P_{2D}(i,j)$ participates in two broadcast operations in its process row and process column. In the ith process row $P_{2D}(i,:)$, the process $P_{2D}(i,s)$ broadcasts its local submatrix $\tilde{\mathbf{A}}$ to all processes in $P_{2D}(i,:)$ (line 5, Alg. 1). Similarly, in the jth process column $P_{2D}(:,j)$, the process $P_{2D}(s,j)$ broadcasts its local submatrix $\tilde{\mathbf{B}}$ to all processes in $P_{2D}(;,j)$ (line 6, Alg. 1). Each process stores the received data in $\tilde{\mathbf{A}}_{\text{recv}}$ and $\tilde{\mathbf{B}}_{\text{recv}}$. In line 7 of Alg. 1, we locally multiply the received matrices and generate a low-rank version of the output. To facilitate merging at the end of all SUMMA stages, partial result from each stage is stored in an array. For example, at stage s, each process multiplies received input submatrices $\tilde{\mathbf{A}}_{\text{recv}}$ and $\tilde{\mathbf{B}}_{\text{recv}}$ and stores the result at $\tilde{\mathbf{C}}[s]$. Finally, at the end of all SUMMA stages, each process merges p_c pieces of partial results and creates its local piece $\tilde{\mathbf{C}}$ of the result (line 8, Alg. 1). When SUMMA2D returns, all local pieces $\tilde{\mathbf{C}}$ form the output matrix $\mathbf{C}$ distributed on P_{2D}.

Major steps. Each stage of SUMMA2D has three major steps: (a) *A-Broadcast*: broadcasting parts of $\mathbf{A}$ along the process row; (b) *B-Broadcast*: broadcasting parts of $\mathbf{B}$ along the process column; and (c) *Local-Multiply*: performing multithreaded local multiplication. After all SUMMA stages, we perform another step *Merge* (will be called *Layer-Merge* in the 3D algorithm) that merges partial results. Here, merging means adding multiplied values with the same row and column indices. While one can incrementally merge partial results after local multiplications, it is computationally more expensive in the worst case [34]. Hence, in this paper, we consider merging after completing all stages as shown in line 8 of Alg. 1.

SUMMA2D performs reasonably well on a few hundred processes. However, as the number of processes increases, communication (broadcasting $\mathbf{A}$ and $\mathbf{B}$) becomes the performance bottleneck of SUMMA2D [30]. 3D Sparse SUMMA [13] was developed to reduce the communication cost of SpGEMM.

B. 3D Sparse SUMMA

Processes grid. Here, each matrix is distributed in a 3D $p_r \times p_c \times l$ process grid P_{3D}. All processes $P_{3D}(:,:,k)$ with a fixed value k in the third dimension form the kth *layer* of P_{3D}, where $1 \leq k \leq l$. In this context, $P_{3D}(:,:,k)$ is similar to a 2D process grid discussed in Sec. III-A. In this paper, we only consider square process grid in each layer. Hence, with p processes divided into l layers, the shape of P_{3D} is $\sqrt{p/l} \times \sqrt{p/l} \times l$. Here, $P_{3D}(i,j,k)$ denotes the process in the ith row and jth column in the kth layer in P_{3D}. All processes

Algorithm 2 3D sparse SUMMA.

Input and Output: Input matrices $\mathbf{A}$ and $\mathbf{B}$ and output matrix $\mathbf{C}$ are distributed in a 3D process grid P_{3D}. $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$, and $\tilde{\mathbf{C}}$ denote local submatrices in the current process. $\mathbf{A}^{(k)}$ denotes the 2D submatrix in the kth layer of P_{3D}.

1: **procedure** SUMMA3D($\mathbf{A}, \mathbf{B}, P_{3D}$)
2: **for** all processes in $P_{3D}(i,j,k)$ **in parallel do**
3: $\mathbf{D}^{(k)} \leftarrow$ SUMMA2D($\mathbf{A}^{(k)}$, $\mathbf{B}^{(k)}$, $P_{3D}(:,:,k)$)
4: $\tilde{\mathbf{D}}^{(k)}[1..l] \leftarrow$ ColSplit($\tilde{\mathbf{D}}^{(k)}, l$)
5: $\tilde{\mathbf{C}}^{(k)}[1..l] \leftarrow$ AllToAll ($\tilde{\mathbf{D}}^{(k)}[1..l]$, $P_{3D}(i,j,:)$)
6: $\tilde{\mathbf{C}}^{(k)} \leftarrow$ Merge($\tilde{\mathbf{C}}^{(k)}[1..l]$)
7: **return** $\mathbf{C}$

TABLE I: Symbols used in this paper

Symbol	Meaning
p	total number of processes
l	number of layers in a 3D grid
b	number of batches
P_{3D}	a $\sqrt{p/l} \times \sqrt{p/l} \times l$ process grid
$\mathbf{A}$	first input matrix (distributed)
$\mathbf{B}$	second input matrix (distributed)
$\mathbf{C}$	output matrix (distributed)
$\mathbf{A}^{(k)}$	Part of $\mathbf{A}$ in the kth layer (similarly $\mathbf{B}^{(k)}$ and $\mathbf{C}^{(k)}$)
$\mathbf{D}^{(k)}$	A low-rank version of output matrix in the kth layer
$\tilde{\mathbf{A}}$	parts of $\mathbf{A}$ stored in the current process (similarly $\tilde{\mathbf{B}}$, $\tilde{\mathbf{C}}$)
n	total number of rows/columns in $\mathbf{A}, \mathbf{B}, \mathbf{C}$
M	total available memory in all processes
r	number of bytes needed to store a nonzero (on average)

in $P_{3D}(i,j,:)$ form a *fiber* that consists of processes with the same row and column indices from different layers.

Data distribution. Suppose $\mathbf{A}$ is an $n \times n$ matrix. After distributing $\mathbf{A}$ in a 3D grid, let $\mathbf{A}^{(k)} \in \mathbb{R}^{n \times (n/l)}$ be the submatrix of $\mathbf{A}$ distributed in the kth layer. Fig. 1(c) shows that each layer gets slices of $\mathbf{A}$ that respect the 2D process boundary. After 3D distribution, each local piece $\tilde{\mathbf{A}}$ is an $\left(n/\sqrt{p/l}\right) \times \left(n/\sqrt{pl}\right)$ submatrix. Hence, $nrows(\tilde{\mathbf{A}}) = l \cdot ncols(\tilde{\mathbf{A}})$ for all local submatrices $\tilde{\mathbf{A}}$. Similarly, we split $\mathbf{B}$ along the rows, and each local piece $\tilde{\mathbf{B}}$ is an $\left(n/\sqrt{pl}\right) \times \left(n/\sqrt{p/l}\right)$ submatrix (Fig. 1(f)). As l increases, $\tilde{\mathbf{A}}$ becomes tall and skinny and $\tilde{\mathbf{B}}$ becomes short and fat. Fig. 1 shows an example with a 3D grid where $p = 8$ processes are organized in $l = 2$ layers, and each layer is a 2×2 grid. Since $\mathbf{A}$ and $\mathbf{B}$ are distributed differently in P_{3D}, we chose to distribute $\mathbf{C}$ similar to $\mathbf{A}$.

The algorithm. The SUMMA3D function in Algorithm 2 describes the 3D multiplication algorithm that operates on matrices distributed on P_{3D}. Initially, SUMMA3D proceeds independently at each layer $P_{3D}(:,:,k)$ by calling SUMMA2D with 2D input matrices in that layer (line 3, Alg. 2). At each layer, SUMMA2D multiplies $\mathbf{A}^{(k)} \in \mathbb{R}^{n \times \frac{n}{l}}$ and $\mathbf{B}^{(k)} \in \mathbb{R}^{\frac{n}{l} \times n}$ to produce $\mathbf{D}^{(k)} \in \mathbb{R}^{n \times n}$. Here, $\mathbf{D}^{(k)}$ denotes an intermediate low-rank output matrix that needs to be merged across layers to form the final product. Next, each process splits $\tilde{\mathbf{D}}^{(k)}$, the local submatrix of $\mathbf{D}^{(k)}$, into l parts $\tilde{\mathbf{D}}^{(k)}[1..l]$ by splitting $\tilde{\mathbf{D}}^{(k)}$ along the column (line 4, Alg. 2). Then, each process performs an AllToAll operation on every fiber $P_{3D}(i,j,:)$ (line 5, Alg. 2). Finally, each process merges all the copies received from the AllToall operation in previous step to form the final local copy of the result matrix (line 6, Alg. 2). The last two steps are called AllToAll-Fiber and Merge-Fiber.

IV. New algorithms for memory-constrained 3D SpGEMM

A. The case for batching

Memory requirement of 3D Sparse SUMMA. If r bytes are needed to store a nonzero, the memory requirement to perform SpGEMM is a least $r(nnz(\mathbf{A})+nnz(\mathbf{B})+nnz(\mathbf{C}))$ bytes. However, a distributed algorithm requires much more memory in practice. The SUMMA3D function from Algorithm 2 on a $\sqrt{p/l} \times \sqrt{p/l} \times l$ process grid stores unmerged matrices $\mathbf{D}^{(k)}$ at layer k. Thus, we have the following inequality:

$$\text{flops} \geq \sum_{k=1}^{l} nnz(\mathbf{D}^{(k)}) \geq nnz(\mathbf{C}). \tag{1}$$

Here, the Merge-Layer operation will generate $\mathbf{D}^{(k)}$ at every layer, and the Merge-Fiber will produce $\mathbf{C}$. In the worst case, we may need to store flops nonzeros when no merging happens inside Local-Multiply in SUMMA2D. Let mem($\mathbf{C}$) denote the memory requirement for the SUMMA3D function. Then, $\text{mem}(\mathbf{C}) = r \sum_{k=1}^{l} nnz(\mathbf{D}^{(k)})$.

For example, consider the Metaclust50 matrix in Table V. If we need r =24 bytes to store a nonzero (16 bytes for row and column indices and 8 bytes for the value), we need 24TB memory to store the final output. However, the actual memory requirement could be up to 92 Trillion$*24 = 2208$TB (2.2PB) as squaring this matrix requires 92 trillion flops. Cori supercomputer used in our experiment has 1.09PB aggregated memory. Hence, we need algorithmic innovations to multiply matrices at this extreme scale.

Batched 3D Sparse SUMMA. When the memory requirement of SUMMA3D exceeds the available memory, we multiply matrices in b batches where each batch computes n/b columns of $\mathbf{C}$. Hence, in a batch, we multiply $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times \frac{n}{b}}$ to generate $\mathbf{C} \in \mathbb{R}^{n \times \frac{n}{b}}$. We chose to form batches of $\mathbf{C}$ column-by-column so that an application can take an informed decision based on the entire columns of $\mathbf{C}$. Our motivation comes from graph analytics where we need the entire result for a batch of vertices before an application decides how to process the result and move to the next batch. For example, Markov clustering [19] keeps top-k entries in each column from the resultant matrix. Hence, we did not consider partitioning C into square tiles despite the fact that it may reduce communication even further.

A symbolic step to determine the required number of batches. If M denotes the aggregated memory in p processes, then an SpGEMM algorithm needs at least b phases as follows:

$$b \geq \left\lceil \frac{\text{mem}(\mathbf{C})}{M - r(nnz(\mathbf{A}) + nnz(\mathbf{B}))} \right\rceil. \tag{2}$$

In practice, b cannot be determined analytically because mem($\mathbf{C}$) is not known in advance. b also depends on the layout of the the process grid P_{3D} and the load balancing factor in distributing the matrices.

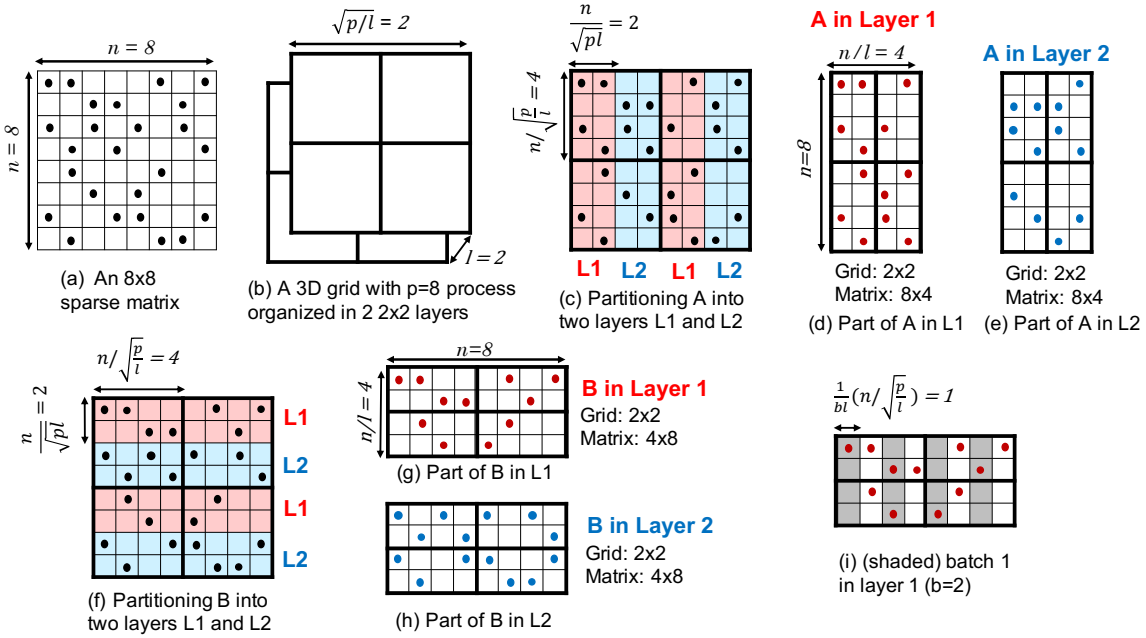

Fig. 1: Distributing an input matrix shown in (a) into a $2 \times 2 \times 2$ 3D process grid shown in (b). Here, we use the same matrix for both $\mathbf{A}$ and $\mathbf{B}$. Thick borders denotes process boundaries in each 2×2 layer. (c) $\mathbf{A}$ is partitioned along the column to create layers. Red slices form layer 1 and blue slices form layer 2. (d, e) Rectangular submatrices of $\mathbf{A}$ in layer 1 and 2. (f) $\mathbf{B}$ is partitioned along the row to create layers. Red slices form layer 1 and blue slices form layer 2. (g, h) Rectangular submatrices of $\mathbf{B}$ in layer 1 and 2. (i) Assuming $b = 2$, the first batch in layer 1 of $\mathbf{B}$ is shown by shaded regions.

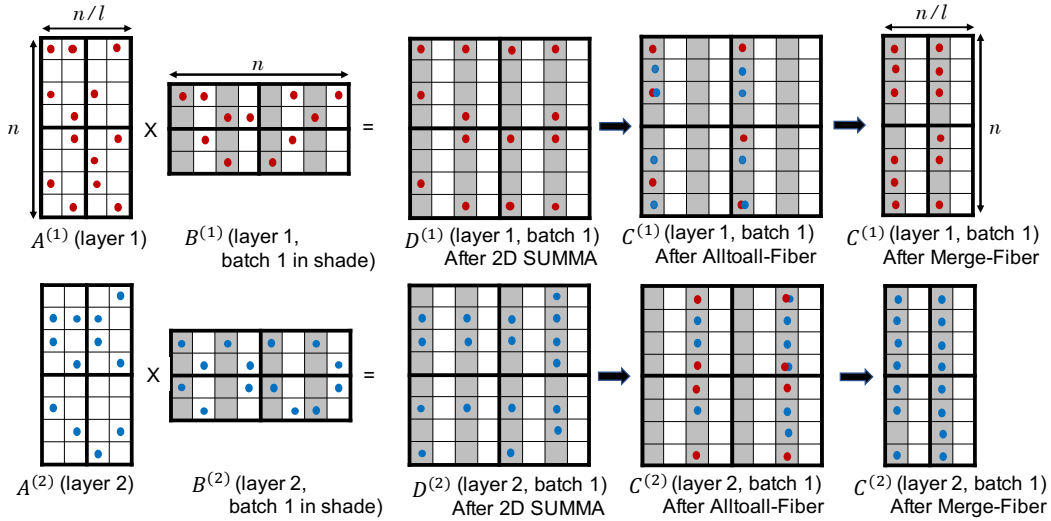

Fig. 2: Assuming $b = 2$, major steps of the first batch of Alg. 4 are shown when multiplying the input matrices shown in Fig. 1. Columns involved in this batch are shown by shaded regions. Red dots represent nonzeros in layer 1 and blue dots represent nonzeros in layer 2. After performing 2D multiplication in each layer, we obtain $\mathbf{D}^{(1)}$ and $\mathbf{D}^{(2)}$ at layer 1 and 2. After AllToall-Fiber both layers have some red and some blue dots as each process exchanges along the fiber some part of its result obtained from SUMMA2D. Overlapping red and blue dots indicate that result of 2D SUMMA in each layer had nonzeros at those positions. These overlapped entries are merged during Merge-Fiber step to have final result of the first batch. Similarly, we can compute the result of the second batch using unshaded parts of $\mathbf{B}$.

We developed the SYMBOLIC3D function (Algorithm 3) that estimates the number of phases from input matrices and the process grid P_{3D}. Similar to the SUMMA3D function, SYMBOLIC3D operates in several stages within each layer (line 5-8 in Algorithm 3). At each stage, local submatrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$ are broadcast along process rows and columns, respectively. After $P_{3D}(i, j, k)$ receives submatrices of $\mathbf{A}$ and $\mathbf{B}$ from other processes, it performs a symbolic multiplication locally using LOCALSYMBOLIC that computes the number of nonzeros in the output. At the end of all SUMMA stages, we find the maximum nnz for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$ stored at any process (lines 9-11 in Algorithm 3). Finally, line 12 computes

Algorithm 3 Symbolic step to determine b.

Input: $\mathbf{A}$ and $\mathbf{B}$ are distributed on a 3D process grid P_{3D}. M denotes the total available memory in bytes and r denotes the number of bytes needed to store each nonzero.

1: **procedure** SYMBOLIC3D($\mathbf{A}, \mathbf{B}, P_{3D}, M$)
2: **for** all processes in $P_{3D}(i,j,k)$ **in parallel do**
3: $nnz[i,j,k] \leftarrow 0$ ▷ per-process nnz
4: stages $\leftarrow$ number of process columns in P_{3D}
5: **for** $s \leftarrow 1$ to stages **do** ▷ SUMMA stages
6: $\tilde{\mathbf{A}}_{\text{recv}} \leftarrow$ BCAST($\tilde{\mathbf{A}}, P_{3D}(i,s,k), P_{3D}(i,:,k)$)
7: $\tilde{\mathbf{B}}_{\text{recv}} \leftarrow$ BCAST($\tilde{\mathbf{B}}, P_{3D}(s,j,k), P_{3D}(:,j,k)$)
8: $nnz[i,j,k]$ += LOCALSYMBOLIC($\tilde{\mathbf{A}}_{\text{recv}}$, $\tilde{\mathbf{B}}_{\text{recv}}$)
9: maxnnzC $\leftarrow$ ALLREDUCEMAX($nnz[i,j,k], P_{3D}$)
10: maxnnzA $\leftarrow$ ALLREDUCEMAX($nnz(\tilde{\mathbf{A}}), P_{3D}$)
11: maxnnzB $\leftarrow$ ALLREDUCEMAX($nnz(\tilde{\mathbf{B}}), P_{3D}$)
12: $b \leftarrow \frac{r \times \text{maxnnzC}}{M/p - r(\text{maxnnzA+maxnnzB})}$
13: **return** b

Algorithm 4 3D memory-constrained sparse SUMMA with batching.

Input and Output: Input matrices $\mathbf{A}$ and $\mathbf{B}$ and output matrix $\mathbf{C}$ are distributed in a 3D process grid P_{3D}. $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$, and $\tilde{\mathbf{C}}$ denote local submatrices in the current process. b is the number of batches. $\mathbf{B}[batch]$ denotes a 3D matrix consisting of local batch $\tilde{\mathbf{B}}[batch]$ from all processes.

1: **procedure** BATCHEDSUMMA3D($\mathbf{A}, \mathbf{B}, P_{3D}$)
2: $b \leftarrow$ SYMBOLIC3D($\mathbf{A}, \mathbf{B}, P_{3D}$)
3: **for** all processes in $P_{3D}(i,j,k)$ **in parallel do**
4: $\tilde{\mathbf{B}}[1..b] \leftarrow$ ColSplit($\tilde{\mathbf{B}}, b$)
5: **for** $batch \leftarrow 1$ to b **do**
6: $\mathbf{C}[batch] \leftarrow$ SUMMA3D($\mathbf{A}$, $\mathbf{B}[batch], P_{3D}$)
7: $\tilde{\mathbf{C}} \leftarrow$ ColConcat($\tilde{\mathbf{C}}[1..b]$)
8: **return** $\mathbf{C}$

b from per-process available memory M/p and other memory requirements. Note that the SYMBOLIC3D function considers the maximum unmerged nonzeros stored by a process so that no process exhausts its available memory. This choice makes our algorithm robust to different sparsity patterns. Hence, in comparison to perfectly-balanced computation, SYMBOLIC3D will estimate more batches for load-imbalanced cases.

Unlike SUMMA3D, SYMBOLIC3D has lightweight computations because LOCALSYMBOLIC can be computed much faster than LOCALMULTIPLY. By contrast, SYMBOLIC3D still needs expensive broadcasts as was needed by SUMMA3D. Consequently, the communication-avoiding scheme used in SYMBOLIC3D has more significant impact on its performance.

B. The batched SUMMA3D algorithm

Data distribution. We perform batching on top of 3D distribution discussed in Fig. 1. Since a batch computes n/b columns of $\mathbf{C}$, batching does not change the distribution of $\mathbf{A}$ shown in Fig. 1. Globally, a batch should have n/b columns of $\mathbf{B}$ and $\mathbf{C}$ across all processes. This can be achieved by simply splitting all local submatrices $\tilde{\mathbf{B}}$ along the column and making b pieces. Let $\tilde{\mathbf{B}}[i]$ be the part of local $\mathbf{B}$ in the ith batch. Since $\tilde{\mathbf{B}}$ has $n/\sqrt{p/l}$ columns in 3D distribution, $\tilde{\mathbf{B}}[i]$ has $n/(b\sqrt{p/l})$ columns. However, a block column partitioning of $\tilde{\mathbf{B}}$ may create load imbalance in the Merge-Fiber operation (to be explained next). Hence, we use block-cyclic partitioning where each block has $n/(bl\sqrt{p/l})$ columns. A collection of l such blocks gives us a batch of desired shape as shown in Fig. 1(i).

The algorithm. The BATCHEDSUMMA3D function in Algorithm 4 multiplies matrices distributed on P_{3D} and generates the result matrix batch by batch. At first, each process splits $\tilde{\mathbf{B}}$, its local copy of $\mathbf{B}$, column-wise into b batches $\tilde{\mathbf{B}}[1..b]$ (line 4, Alg. 4). The exact block cycling splitting is described in the previous paragraph and in Fig. 1(i). It effectively creates as many pieces of $\mathbf{B}$ as the number of batches b. Then for each $batch$, the algorithm calls SUMMA3D to multiply $\mathbf{A}$ and $\mathbf{B}[batch]$ to get the output matrix $\mathbf{C}[batch]$ (line 6, Alg. 4). At the end of all batches, each process has b pieces of $\tilde{\mathbf{C}}[1..b]$. Here, submatrices in $\tilde{\mathbf{C}}[1..b]$ have non-overlapping columns. Hence, each process concatenates $\tilde{\mathbf{C}}[1..b]$ along the column to form its own piece $\tilde{\mathbf{C}}$ of the final result (line 7, Alg. 4). The overall result matrix $\mathbf{C}$ is a collection of these local matrices distributed on P_{3D}. Note that we showed the formation of $\mathbf{C}$ for completeness of the algorithm. In practice, the output $\mathbf{C}[batch]$ from each batch is pruned or saved to disk by the application. Fig. 2 shows an execution of Algorithm 4. Note that when $nnz(\mathbf{A}) \gg nnz(\mathbf{B})$, column-wise batching can be expensive. However, if inputs are square matrices, we can easily use row-by-row batching on $\mathbf{B}$ using the same algorithm.

Major steps. BATCHEDSUMMA3D uses SUMMA3D in each batch and SUMMA3D uses SUMMA2D in each layer. Hence, all major steps in SUMMA3D and SUMMA2D are also executed in BATCHEDSUMMA3D. Additionally, BATCHEDSUMMA3D uses Alg. 3 to estimate the number of batches needed for an SpGEMM. Therefore, BATCHEDSUMMA3D has seven major steps: (1) Symbolic multiplication to estimate b (once; involve communication and computation), (2) A-Broadcast (once per SUMMA2D stage: along a process row on every layer), (3) B-Broadcast (once per SUMMA2D stage: along a process column on every layer), (4) Local-Multiply (once per SUMMA2D stage: local computation), (5) Merge-Layer (once per SUMMA2D stage: local computation), (6) AllToAll-Fiber (once per batch: communicate in a fiber) (7) Merge-Fiber (once per batch: local computation).

C. Communication complexity

Among seven major steps in BATCHEDSUMMA3D, four steps (Symbolic, A-Broadcast, B-Broadcast, and AllToAll-Fiber) involves communication. When $b > 1$, BATCHED-SUMMA3D increases the number of times matrix $\mathbf{A}$ needs to be re-communicated, making the impact of communication-avoidance even more significant for batched SpGEMM. Even

though we relied on the communication-avoiding technique developed in a prior work [13], batching makes communication steps more fine-grained and irregular. To analyze the communication complexity we used the $\alpha - \beta$ model, where α is the latency constant corresponding to the fixed cost of communicating a message regardless of its size, and β is the inverse bandwidth corresponding to the cost of transmitting one word of data. Consequently, communicating a message of n words takes $\alpha + \beta n$ time.

Table II shows the bandwidth and latency costs of different steps of BATCHEDSUMMA3D. Communication costs of the symbolic step are similar to A-Broadcast and B-Broadcast, except the fact that the communication cost of the symbolic step does not rely on b. Here, the bandwidth bound for AllToAll-Fiber is rather loose because there is already significant compression of intermediate products expected within (a) local SpGEMM calls, and (b) within each SUMMA layer. As the number of layers l increases, the effect of this compression diminishes. It is tighter to use $\sum_{k=1}^{l} nnz(\mathbf{D}^{(k)})/p$ in lieu of flops/p, which is a function that grows slowly with l.

Table II shows that all communication steps are performed at least b times. Consequently, BATCHEDSUMMA3D has higher latency overheads relative to SUMMA3D. Since, $\mathbf{A}$ is communicated b times, the bandwidth cost of A-Broadcast could dominate the overall communication cost, especially for a large value of b. Fortunately, we can increase the number of layers to reduce the overhead of re-communicating $\mathbf{A}$.

D. Faster local computations using hash tables

We developed new algorithms for local multiplication and merging within each process with an aim to utilize special structures due to layering and batching. One particular optimization is in the sortedness of results after Local-Multiply, Merge-Layer, and Merge-Fiber. Since the final output $\mathbf{C}$ is obtained after Merge-Fiber, we keep the output of Merge-Fiber sorted within each column. However, the outputs of Local-Multiply and Merge-Layer do not need to be sorted within each column because they will eventually be sorted after Merge-Fiber. To facilitate unsorted outputs in Local-Multiply and Merge-Layer, we used hash-based local SpGEMM and merging algorithms. By contrast, heap-based algorithms used in prior work [13] always keep results sorted. These modifications can make local computations more than $5\times$ faster for many matrices as we will demonstrate in Fig. 15.

Sort-free local SpGEMM. Previous 3D SUMMA algorithm [13] used a multithreaded heap-based local SpGEMM algorithm in each process. However, the heap algorithm was later replaced by a hybrid algorithm that used either a hash table or a heap to form the ith column depending on the compression ratio of the column [25]. After forming the column, that hybrid algorithm sorted the column in an ascending order of row indices. With an aim to keep matrices unsorted, we use a hash-based SpGEMM in our Local-Multiply routine (line 7 of Algo. 1) because the hash SpGEMM does not require sorted matrices as inputs. Additionally, we refrain from sorting columns once they are formed. Thus, our local multiplication uses an "unsorted-hash" algorithm. In practice, the unsorted-hash algorithm can be 30%-50% faster than the hybrid algorithm.

TABLE II: Communication complexity for different steps of BATCHEDSUMMA3D.

	A-Bcast	B-Bcast	AllToAll-Fiber
Per process data	$nnz(\mathbf{A})/p$	$nnz(\mathbf{B})/(bp)$	$\text{flops}/(bp)$
Comm. size	$\sqrt{p/l}$	$\sqrt{p/l}$	l
Latency cost	$\alpha \lg p/l$	$\alpha \lg p/l$	αl
Bandwidth cost	$\beta(nnz(\mathbf{A})/p)$	$\beta(nnz(\mathbf{B})/(bp))$	$\beta(\text{flops}/(bp))$
How many times	$b(\sqrt{p/l})$	$b(\sqrt{p/l})$	b
Tatal latency	$\alpha b(\sqrt{p/l} \lg p/l)$	$\alpha b(\sqrt{p/l} \lg p/l)$	αbl
Total bandwidth	$\beta b(nnz(\mathbf{A})/\sqrt{pl})$	$\beta(nnz(\mathbf{B})/\sqrt{pl})$	$\beta(\text{flops}/p)$

TABLE III: Computational complexity for different steps of BATCHEDSUMMA3D.

	Local-Multiply	Merge-Layer	Merge Fiber
Complexity	$\text{flops}/(bp\sqrt{p/l})$	$\text{flops}/(bp) \lg p/l$	$\text{flops}/(bp) \lg l$
How many times	$b(\sqrt{p/l})$	b	b
Total	flops/p	$\text{flops}/p \lg p/l$	$\text{flops}/p \lg l$

TABLE IV: Overview of the evaluation platform.

	Cori-KNL	**Cori-Haswell**
Processor	Intel Xeon Phi 7250	Intel Xeon E5-2698
Cores/node	68	32
Clock	1.4 GHz	2.3 GHz
Hyper-threads/core	4	2
Memory/node	112GB	128GB
Total nodes	9,668	2,388
Total memory	1.09 PB	298.5 TB
Interconnect	Cray Aries with Dragonfly topology	
Compiler	Intel icpc Compiler 19.0.3 with -O3 option	

Sort-free hash merging algorithms. Previous 2D SUMMA [30] and 3D SUMMA [13] algorithms used a heap-based merging algorithm in Merge-Layer and Merge-Fiber routines. Observing the benefit of hash SpGEMM algorithms [25], we develop a new hash-based merging algorithm for Merge-Layer and Merge-Fiber steps. Given a collection of l matrices, the hash-merge algorithm forms the ith column of the merged output from the ith columns of all input matrices. The merging is done using a hash table that can work with unsorted input and outputs an unsorted output column. This new "unsorted-hash-merge" algorithm can be an order of magnitude faster than previous heap-merge algorithm.

V. RESULTS

A. Evaluation platforms

We evaluate the performance of our algorithm on NERSC Cori system. We used two types of compute nodes on Cori as described in Table. IV. We used MPI+OpenMP hybrid parallelization. All of our experiments used 16 and 6 threads per MPI process on Cori-KNL and Cori-Haswell, respectively. Only one thread in every process makes MPI calls.

TABLE V: Statistics about test matrices used in our experiments. $\mathbf{C}=\mathbf{A}\mathbf{A}^T$ for Rice-kmers and Metaclust20m. For all other cases, $\mathbf{C}=\mathbf{A}\mathbf{A}$. M, B and T denote million, billion and trillion, respectively.

Matrix ($\mathbf{A}$)	rows	columns	$nnz(\mathbf{A})$	$nnz(\mathbf{C})$	flops
Eukarya	3M	3M	360M	2B	134B
Rice-kmers	5M	2B	4.5B	6B	12.4B
Metaclust20m	20M	244M	2B	312B	347B
Isolates-small	35M	35M	17B	248B	42T
Friendster	66M	66M	3.6B	1T	1.4T
Isolates	70M	70M	68B	984B	301T
Metaclust50	282M	282M	37B	1T	92T

B. Test problems

Table V describes several large-scale matrices used in our experiments. Eukarya, Isolates-small, and Isolates are protein-similarity networks generated from isolate genomes in the IMG database [35]. These matrices are publicly available [19]. Metaclust50 stores similarities of proteins in Metaclust50 (https://metaclust.mmseqs.com/) dataset which contains predicted genes from metagenomes and metatranscriptomes of assembled contigs from IMG/M and NCBI. Friendster represents an online gaming network available in the SuiteSparse Matrix Collection [36]. The rows of the Rice-kmers matrix represent the PacBio sequences for the Oryza sativa rice, and its columns represent a subset of the k-mers (short nucleotide subsequences of length k) that are used by the sequence overlapping software BELLA [7]. The Metaclust20m matrix is originally used in distributed protein similarity search [15] and contains a subset of sequences for Metaclust50. The rows and columns of this matrix respectively represent protein sequences and k-mers. The $\mathbf{A}\mathbf{A}^\mathsf{T}$ operation involving this matrix produces candidates for batch pairwise sequence alignment.

Even though there are many applications of SpGEMM, we primarily focus on problems where $nnz(\mathbf{C}) > (nnz(\mathbf{A}) + nnz(\mathbf{B}))$ because for these problems, batching could be needed. For example, square of a sparse matrix often requires significantly more memory than input matrices [37]. We selected matrices in Table V considering the following three applications. (a) Markov clustering [19, 38] iteratively squares a protein-similarity matrix to discover protein clusters. We capture this application by extensively studying the computation of $\mathbf{A}\mathbf{A}$. (b) Clustering coefficients of nodes in a social network are computed by counting triangles. High-performance triangle counting algorithms rely on the multiplication of the lower triangular and upper triangular parts of the adjacency matrix [3]. $\mathbf{A}\mathbf{A}$ captures the computation needed in triangle counting algorithms. (c) By computing $\mathbf{A}\mathbf{A}^\mathsf{T}$, the sequence overlapper BELLA discovers all (if any) the shared k-mers between all pairs of sequences. By using the sparse matrix as an index data structure in lieu of a hash table, SpGEMM computes this seemingly all-pairs computation without incurring quadratic cost in the number of reads.

C. Impact on an end-to-end protein clustering application

We plugged the newly developed SpGEMM algorithm in HipMCL [19], a distributed-memory implementation of the Markov clustering algorithm. HipMCL clusters a protein-similarity network by iteratively squaring the input matrix and then applying column-wise pruning to keep the output sparse. For large networks in Table V, Cori does not have enough memory to store $\mathbf{A}^2$. When we employ BATCHEDSUMMA3D inside HipMCL, we form $\mathbf{A}^2$ batch-by-batch, apply various pruning strategies on the current batch and then proceed to the next batch. Hence, the presented algorithm satisfies the need of HipMCL perfectly well.

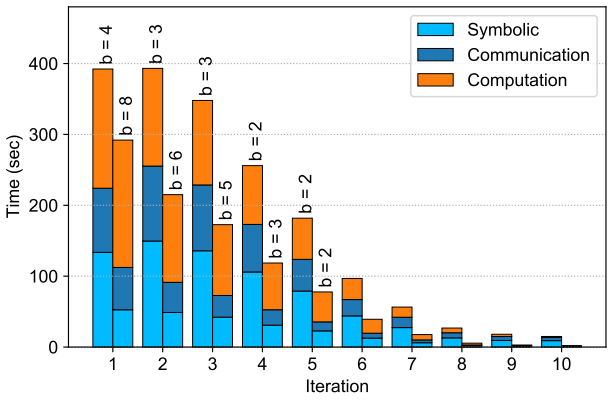

Fig. 3: Run times of the first 10 iterations of HipMCL when clustering the Isolates-small graph on 65,536 cores of Cori. The left bar of each group represents 1-layer setting and right bar represents 16-layer setting. Number of batches b is calculated using the symbolic step. $b = 1$ is not shown. HipMCL needed 66 iterations to cluster proteins of Isolates-small graph. Overall, it was $1.88\times$ faster with 16 layers than with 1 layer.

Fig. 3 show that the runtime of the first 10 iterations of HipMCL when BATCHEDSUMMA3D is used with 1 and 16 layers. In the first few iterations, HipMCL performs expensive multiplications that needs multiple batches on 1024 nodes. Even though the 16-layer setting needs more batches, the benefit of communication avoidance makes most iterations $2\times$ or more faster than 2D SpGEMM. Overall, BATCHEDSUMMA3D makes HipMCL more than $1.88\times$ faster than a batched 2D algorithm. More importantly, *HipMCL cannot even cluster Isolates-small on 1024 nodes of Cori if batching is not used*. Hence, the presented algorithm makes large-scale protein clustering possible, and at the same time, it makes HipMCL significantly faster than previous algorithms.

D. Evaluating the impact of number of layers and batches

Given the use of matrix squaring in HipMCL, we use the computation of $\mathbf{A}^2$ to show various feature of our algorithm. At first, we study the impact of l and b on different steps of BATCHEDSUMMA3D. Even though a suitable value of b depends on the available memory and is determined by the symbolic step, we vary b in this experiment to capture different amount of memory available in different distributed systems. Fig. 4 shows the results with two matrices and two different number of cores. We summarize key observations from Fig. 4.

A-Broadcast time: *With a fixed l, A-Broadcast time increases almost linearly with b.* This is expected as $\mathbf{A}$ is broadcast b times in total (once in every batch). *With a*

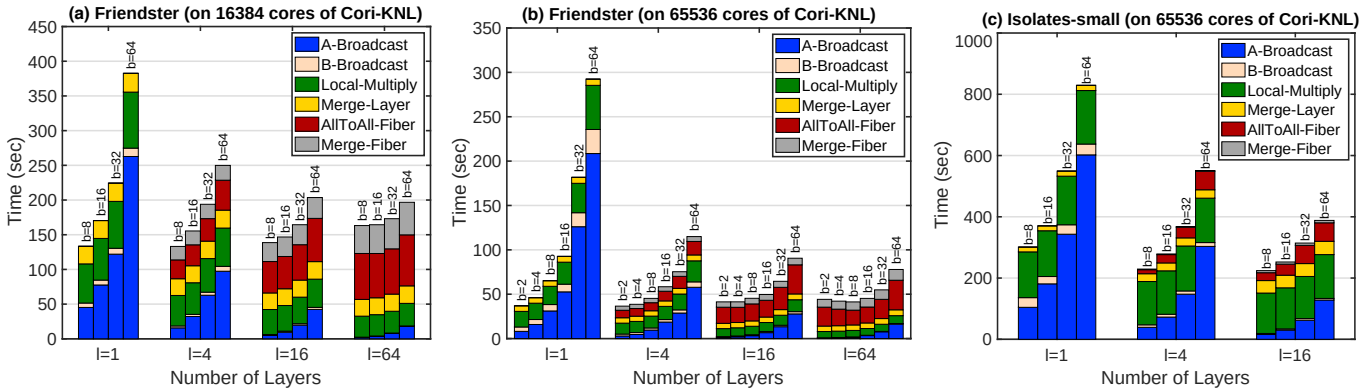

Fig. 4: Evaluating the impact of the number of layers (denoted by l) and batches (denoted by b) on different steps of BATCHEDSUMMA3D. In each experiment, a fixed number of nodes and 16 threads per process are used. Runtimes shown in a bar denote the total time needed for all batches shown at the top of the bar. (a) Squaring Friendster on 16,384 cores (256 nodes and 1024 processes); (b) Squaring Friendster on 65,536 cores (1024 nodes and 4096 processes); (c) Squaring Isolates-small on 65,536 cores (1024 nodes and 4096 processes).

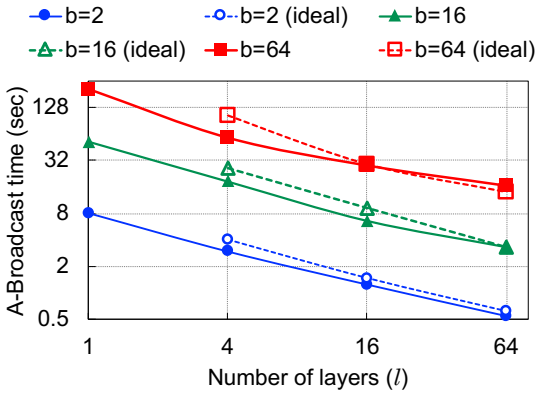

Fig. 5: With a fixed b, A-Broadcast time decreases when l increases. Here, we multiply Friendster on 65,536 cores (1024 nodes) with different batches (come from Fig. 4(b)). Solid lines denote observed A-Broadcast time and dashed lines denote expected A-Broadcast time that decreases by a factor of 2 as we increase l by a factor of 4.

fixed b, when l increases, A-Broadcast time decreases at a rate proportional to $\sqrt{l}$. Fig. 5 shows this observation for different batches when multiplying Friendster on 65,536 cores. In Fig. 5, solid lines (observed A-Broadcast time) closely follow dashed line (expected based on a factor of $\sqrt{l}$ decrease). To explain this, consider a $\sqrt{p/l} \times \sqrt{p/l} \times l$ process grid with p processes. If we increase the number of layers by a factor of 4, the grid becomes $\frac{1}{2}\sqrt{p/l} \times \frac{1}{2}\sqrt{p/l} \times 4l$. Hence, the number of processes in each process row within a layer is reduced by a factor of 2. Since A-Broadcast is performed within individual process rows of each layer, the communicator size for A-Broadcast would be decreased by a factor 2. Thus, the A-Broadcast time decreases at a rate proportional to $\sqrt{l}$.

B-Broadcast time: *With a fixed l, the B-Broadcast time does not change with b.* This is expected because the total data volume associated with B-Broadcast remains the same with batching. Hence, the bandwidth cost for B-Broadcast does not depend on b. However, the latency term increases linearly as we increase b. Since small latency-bound matrices may not need batching, B-Broadcast time does not rely on b as observed in Fig. 4. *With a fixed b, B-Broadcast time decreases when we increase l.* This observation is consistent with the corresponding observation in A-Broadcast.

Local-Multiply time: *With a fixed l, the Local-Multiply time does not change significantly with b.* As we increase b, per layer result matrix is computed in increasing number of batches. It does not change the complexity or data access patterns of local multiplication. However, if b is significantly large, the repeated cost of thread creation, memory allocation, and cache accesses may increase the Local-Multiply time as observed with $b = 64$ in Fig. 4. *With a fixed b, the Local-Multiply time decreases with the increase of l.* Suppose in a Local-Multiply with $l = 1$, we multiply an $\hat{n} \times \hat{k}$ matrix $\hat{\mathbf{A}}$ with a $\hat{k} \times \hat{n}$ matrix $\hat{\mathbf{B}}$ and generate an $\hat{n} \times \hat{n}$ matrix $\hat{\mathbf{D}}$. With l layers, each layer multiplies an $\hat{n} \times \frac{\hat{k}}{l}$ matrix $\hat{\mathbf{A}}$ with a $\frac{\hat{k}}{l} \times \hat{n}$ matrix $\hat{\mathbf{B}}$ and generate an $\hat{n} \times \hat{n}$ matrix $\hat{\mathbf{D}}$. Therefore, as we increase l, Local-Multiply generates a lower-rank version of the final output. As a result, with the increase of l, the complexity of local multiplication decreases. This effect is more significant for sparser matrices, where the result of local multiplication becomes hyper-sparse with many layers. For example with $b = 64$, when we go from $l = 1$ to $l = 16$, Local-Multiply decreases by a factor of $3.6\times$ for Friendster (Fig. 4(b)) and by a factor of $1.2\times$ for Isolates-small (Fig. 4(c)).

AllToAll-Fiber time: *With a fixed l, the AllToAll-Fiber time does not change significantly with b.* For a fixed l, AllToAll-Fiber is performed among l processes on each fiber in b batches. If AllToAll-Fiber in each batch is bandwidth-bound, its runtime does not change with b as is observed in Fig. 4. *With a fixed b, the AllToAll-Fiber time increases as we increase l.* As explain before, increasing l creates lower rank output in each layer, needing more data to be communicated across layer for merging. Furthermore, increasing l also increases the communicator size of AllToAll-Fiber. Hence, the AllToAll-Fiber time increases with the increase of l.

Merge-Fiber time: *With a fixed l, the Merge-Fiber time does not change significantly with b.* As with AllToAll-Fiber, the Merge-Fiber time is not influenced by b. *With a fixed b, the Merge-Fiber time increases as we increase l.* As explain in the previous paragraph, increasing l requires more data to

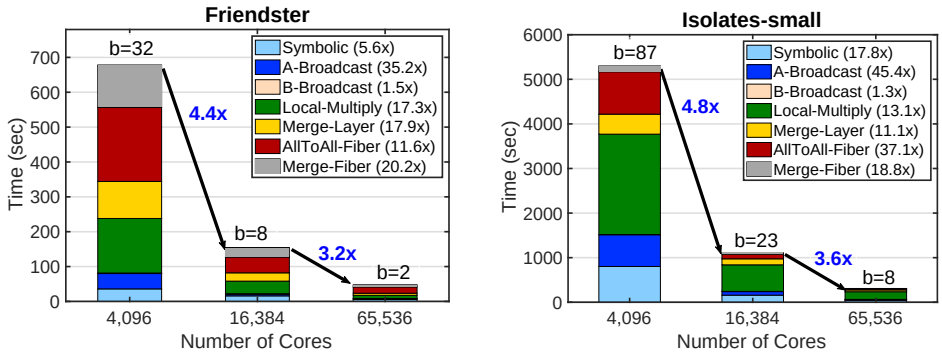

Fig. 6: Strong scaling when squaring Friendster and Isolates-small. The scaling experiments were conducted from 64 nodes (4,096 cores) to 1024 nodes (65,536 cores) on Cori-KNL. Number of layers is set to 16 and number of batches is shown on top of each bar. Total speedups from one bar to the next bar are shown by arrowheads. Numbers in captions denote the overall speedup of each step in batched-SUMMA3D when we go from 4,096 to 65,536 cores (a 16× increase in core counts).

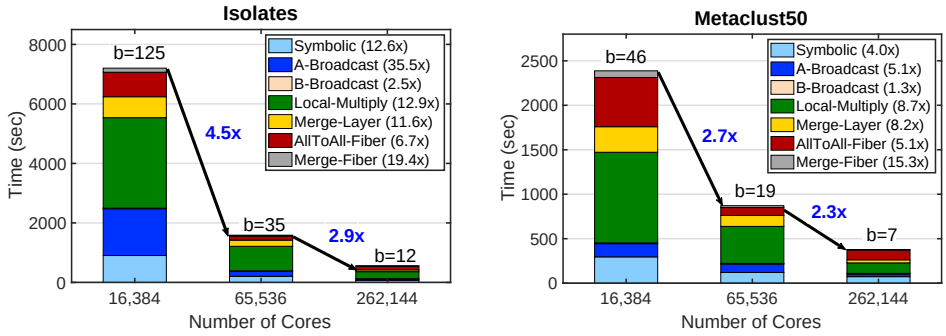

Fig. 7: Strong scaling when squaring two biggest matrices in our test suite. The scaling experiments were conducted from 256 nodes (16,384 cores) to 4096 nodes (262,144 cores) on Cori-KNL. Number of layers is set to 16 and number of batches is shown on top of each bar. Total speedups from one bar to the next bar are shown by arrowheads. Numbers in captions denote the overall speedup of each step when we go from 16,384 to 262,144 cores (a 16× increase in core counts).

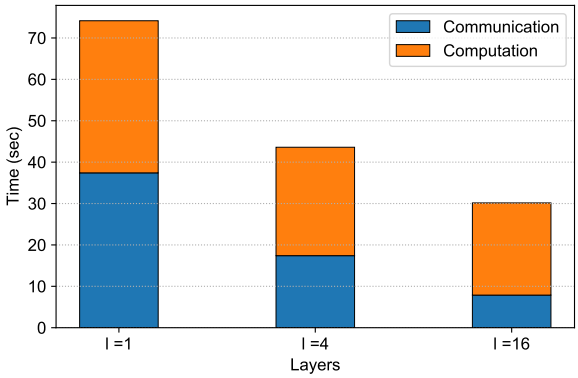

Fig. 8: Comparing computation and communication time in the symbolic step for the Isolates-small graph on 65,536 Cori KNL cores.

be merged in the Merge-Fiber step.

Symbolic time: The symbolic step used to determine b also uses a communication-avoiding algorithm as discussed in Algo. 3. Fig. 8 shows that the symbolic step becomes significantly faster as we increase l. The communication time in the symbolic step becomes more than $4\times$ faster when we use 16 layers, which results in more than $2\times$ speedup of the total symbolic runtime. Here, the communication-avoiding algorithm has a higher impact than the actual multiplication because Algo. 3 needs lighter local computation.

TABLE VI: The impact of l and b on different steps of BATCHED-SUMMA3D. $\leftrightarrow$: no change, $\uparrow$: increase, $\downarrow$: decrease. The Local-Multiply time may slightly increase with b.

l	b	A-Bcast	B-Bcast	Local-Multiply	Merge-Layer	Merge Fiber	AllToAll-Fiber
$\leftrightarrow$	$\uparrow$	$\uparrow$	$\leftrightarrow$	$\nearrow$	$\leftrightarrow$	$\leftrightarrow$	$\leftrightarrow$
$\uparrow$	$\leftrightarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\leftrightarrow$	$\uparrow$	$\uparrow$

Selecting the number of layers and batches. Table VI summarizes the overall impacts of l and b on different steps of BATCHEDSUMMA3D. In the rest of our experiments, we set b to the smallest possible value so that the result in a batch fits in the avilable memory. Generally, the A-Broadcast and B-Broadcast time continue to decrease as we increase the number of layers as seen in Fig. 4. However, All2All-Fiber and Merge-Fiber time increase as we increase l. Therefore, selecting the optimum number of layers is challenging as it depends on the tradeoff between broadcasts and fiber reduction/merge costs. In the rest of our experiments, we set $l = 16$ as it usually gives the best result as can be seen in Fig. 4.

E. Strong scaling results

Fig. 6 and Fig. 7 show the strong scaling of batched-SUMMA3D for four different matrices. We summarize the strong scaling results with the following key findings.

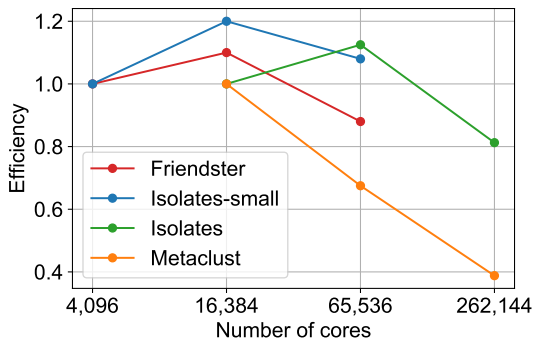

Fig. 9: Parallel efficiency of BATCHEDSUMMA3D.

BATCHEDSUMMA3D scales remarkably well at extreme scale. As we increase core counts by $16\times$, our algorithm scales remarkably well for all four matrices: Friendster ($14\times$), Isolates-small ($17.3\times$), Isolates ($13\times$), and Metaclust50 ($6.3\times$). For bigger matrices BATCHEDSUMMA3D scales to 4096 nodes (262,144 cores) and possibly beyond. This scalability is due to two factors: (a) all computations (Local-Multiply, Merge-Layer, and Merge-Fiber) scale almost linearly as shown in the captions of the scalability figures, and (b) dominant communication costs (A-Broadcast and AllToAll-Fiber) also scale well, thanks to the communication-avoiding algorithm. In fact, A-Broadcast can scale super-linearly (e.g., $45.4\times$ reduction for the Isolate-small matrix in Fig. 6) because of the reduced number of batches needed at higher concurrency. The Symbolic step also scales well. Only B-Broadcast does not scale as well as other steps possibly because of high latency overhead in broadcasting small pieces of $\mathbf{B}$. However, B-Broadcast constitutes about 1% of the total runtime for all matrices in Fig. 6 and Fig. 7. Hence, the B-Broadcast time does not significantly impact the scalability of BATCHEDSUMMA3D.

Super linear speedup is attainable with more aggregated memory at high node counts. As we increase the number of nodes by $4\times$, the aggregated memory also increases by a factor of 4. As a result, b decreases by at least a factor of 2 in all cases in Fig. 6 and Fig. 7. Fewer batches at higher concurrency, can super-linearly reduce the A-Broadcast time, resulting in possible super-linear speedups. For example, for Isolates in Fig. 7, the total runtime decreases by $4.5\times$ when we go from 256 nodes (16,384 cores) to 1024 nodes (65,536 cores) because b decreases from 125 to 35. The super-linear speedup is especially observed at low concurrencies because of the dramatic reduction of the number of batches. Even though b decreases as we increase the number of nodes, their relationship is not straightforward as they are related via the intermediate per-layer expanded matrix $\mathbf{D}^{(k)}$ that in turn depends on per-layer compression factor as well as the overall compression factor. For example, in Fig. 7, when we go from 65K cores to 262K cores, the number of batches is decreased by less than 3x even though the memory increases by 4x.

Parallel efficiency. We compute the parallel efficiency by using $\frac{P1}{P2}\frac{T(P1)}{T(P2)}$ where $T(P)$ denotes the runtime with P processes, and $P2{>}P1$. Fig. 9 shows the parallel efficiency of four large matrices. We observe that the efficiency remains close to 1 for three out of four large matrices. Here, super-linear speedups resulted in an efficiency greater than one. For Metaclust, the efficiency drops to 0.4 on 262K cores. Since Metaclust is sparser than the other big matrix Isolates, the communication cost for Metaclust starts to dominate quickly. For example, on 4096 nodes, the communication takes 48% and 36% of the total runtime for Metaclust and Isolates, respectively. As communication does not scale as well as computation, we observe a drop in parallel efficiency for sparser matrices like Metaclust.

TABLE VII: Overview of local computation improvements when multiplying Isolates-small on 65,536 Cori KNL cores.

Layers	Local-Multiply		Merge-Layer		Merge-Fiber	
	Previous	Now	Previous	Now	Previous	Now
1	144s	148s	258s	16.5s	-	-
4	149s	135s	349s	26.2s	16.7s	2.2s
16	172s	130s	443s	39.9s	74.7s	7.3s

F. The impact of faster computational kernels

As mentioned in Sec. IV-D, we developed new hash-based merging algorithms that replaced a heap-based merging algorithms used in previous work [13]. As hash-based SpGEMM and merging algorithm can work with unsorted matrices, we keep all intermediate results unsorted to reduce the computational complexity. These two modifications (hash-based merging and unsorted matrices) made local multiplication and merging significantly faster as shown in Table VII.

We observe that the unsorted-hash SpGEMM algorithm used in the Local-Multiply step of BATCHEDSUMMA3D can be up to 30% faster than the previous hybrid SpGEMM algorithm when 16 layers are used. The performance benefit of unsorted local multiplication is more significant with more layers. This is because the number of nonzeros in intermediate matrices $\mathbf{D}^k$ increases as we increase l, which requires sorting with a larger volume of data. When $l{=}1$, the previous hybrid algorithm can run faster than the unsorted-hash algorithm because the hybrid algorithm can also use a heap-based algorithm that is often faster when the compression ratio of a column is small [25].

Table VII shows a dramatic improvement when we use the new unsorted-heap-merging algorithm instead of the previous heap-merging algorithm. On 16 layers, both Merge-Layer and Merge-Fiber time reduces by an order of magnitude. Consequently, the total computation of BATCHEDSUMMA3D is made at least $8\times$ faster than previous state-of-the-art SUMMA 3D implementation [13].

G. The scalability when computing $\mathbf{AA}^\mathsf{T}$

We compute $\mathbf{AA}^\mathsf{T}$ only for Metaclust20m and Rice-kmers matrices considering their application in sequence overlapper BELLA [7] in computing the shared k-mers between all pairs of sequences.

Fig. 10 shows the performance of BATCHEDSUMMA3D when computing $\mathbf{AA}^\mathsf{T}$ for Metaclust20m. On 64 nodes (4K

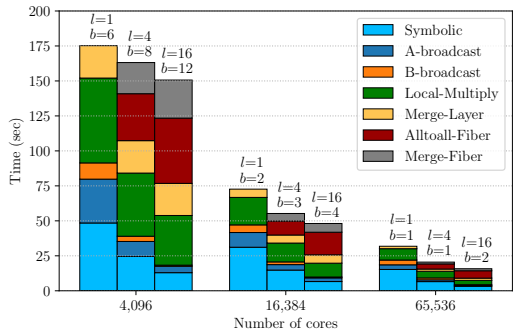

Fig. 10: Computing $\mathbf{AA}^\mathsf{T}$ with the Metaclust20m matrix on Cori.

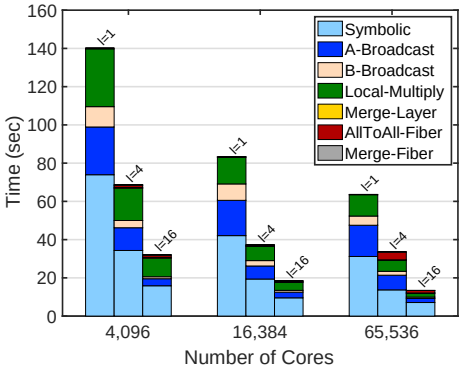

Fig. 11: Scalability of computing $\mathbf{AA}^\mathsf{T}$ with the Rice-kmers matrix on Cori-KNL.

cores), BATCHEDSUMMA3D with 16 layers is slightly faster than with 1 layer. This is because the 16-layer instance needed 12 phases whereas the 1-layer instance needed just 6 phases. Hence, on 64 nodes, the benefit of communication-avoidance is overshadowed by the need to broadcast $\mathbf{A}$ $2\times$ more time. On 1024 nodes (65K cores), BATCHEDSUMMA3D with 16 layers is about $2\times$ faster than with 1 layer even though the 1-layer case does not need any batching. The latter case highlights the significant performance benefit of BATCHEDSUMMA3D at high concurrencies with or without batching.

Table V shows that $nnz(\mathbf{AA}^\mathsf{T}) \approx nnz(\mathbf{A})$ for the Rice-kmers matrix. Hence, batching is often not required to compute $\mathbf{AA}^\mathsf{T}$ with Rice-kmers. In this case, BATCHEDSUMMA3D computes $\mathbf{AA}^\mathsf{T}$ with b=1, while the communication-avoiding algorithm reduces the communication costs. Fig. 11 shows that $\mathbf{AA}^\mathsf{T}$ computation on the Rice-kmers matrix is dominated by communication when only one layer is used (recall that the Symbolic step also performs A-Broadcast and B-Broadcast). This is expected since Rice-kmers has just 2 nonzeros per column on average. As SpGEMM is dominated by communication, using more layers reduces the runtime significantly, as expected. For example, on 65,536 cores (1024 nodes) of Cori-KNL, $\mathbf{AA}^\mathsf{T}$ can be computed $6\times$ faster if we use 16 layers instead of 1 layer. This experiment demonstrates that BATCHEDSUMMA3D helps any SpGEMM run faster at extreme scale with our without batching.

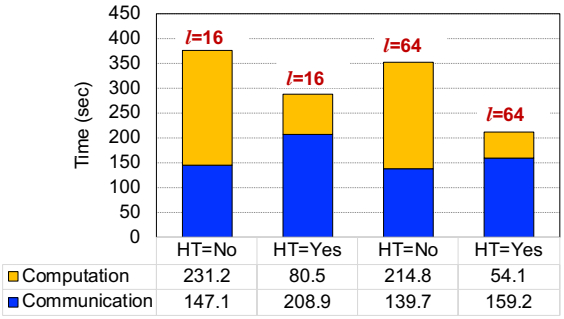

Fig. 12: The impact of using hyperthreads when squaring Metaclust50 on 4096 nodes on Cori-KNL. HT=Yes means 4 hardware threads per core are used. At this setting with 4096 nodes, we use 256 threads per node totaling 1,048,576 threads. HT=No means one thread per core is used, totaling 262,144 threads. As before, we used 16 threads per process. The number of layers used in the 3D grid is shown at the top of each bar. For this matrix, hyperthreading reduces the computation time, but increases communication time. Overall, hyperthreading decreases the total runtime of this SpGEMM.

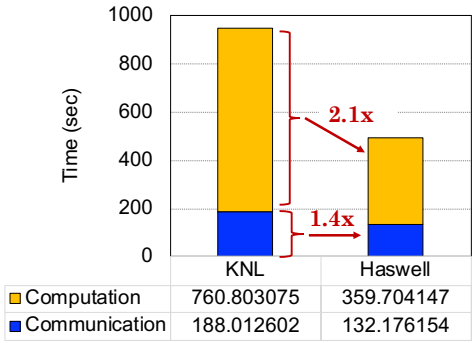

Fig. 13: Squaring Isolates-small on 256 nodes of Cori-KNL and Cori-Haswell. While the communication network remains the same, Cori-Haswell uses 32 fast cores per node. We use 6 cores/process on Haswell and 16 cores/process on KNL. With 16 layers and 23 batches, both experiments use the same process grid. Arrowheads show that computation and communication becomes $2.1\times$ and $1.4\times$ faster on Haswell.

H. Impact of hyper-threading at extreme scale

As with most modern manycore processors, each KNL processor has four hardware threads. In our prior experiments, we did not use hyperthreading because it can increase the communication time significantly due to larger process grids. Here, we study the impact of hyperthreads when squaring Metaclust50 on 4096 nodes on Cori-KNL. Fig. 12 shows that hyperthreading can help SpGEMM run faster by reducing the computation time significantly even though the communication time may increase. This gives us an unprecedented scalability to more than one million threads, which will help scale many graph and machine learning applications to upcoming exascale systems. The impact of hyperthreading is more significant when the total runtime is dominated by computation (for example when $l = 64$ in Fig. 12). That means, hyperthreading may not help when SpGEMM becomes communication-bound.

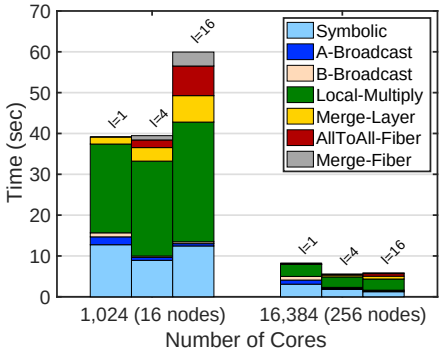

Fig. 14: Squaring Eukarya, the smallest matrix in our test suite, on Cori-KNL. SpGEMM needs two batches when $l = 16$ on 16 nodes. For all other cases, $b = 1$. SUMMA3D is not useful when communication cost is insignificant on 16 nodes. However, SUMMA3D with 4 layers is still useful on 256 nodes even though batching is not needed for this small matrix.

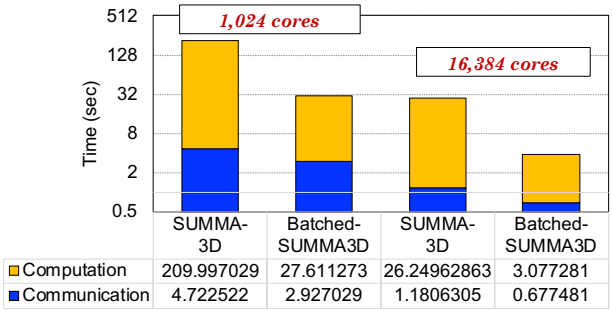

Fig. 15: Comparing BATCHEDSUMMA3D developed in this paper with the SUMMA3D algorithm presented in [13] in squaring the Eukarya matrix with 4 layers without batching.

I. Impact of using faster processors

If we employ faster processors while using the same communication network, communication could quickly become the bottleneck. Hence, faster in-node computations are expected to make BATCHEDSUMMA3D more even more beneficial. We investigate this hypothesis by running BATCHEDSUMMA3D on 256 nodes of Cori-KNL and Cori-Haswell and show the result in Fig. 13. We observe that computation is about $2.1\times$ faster on Haswell. Even with the same communication network, communication on Cori-Haswell is $1.4\times$ faster possibly because of faster data possessing around MPI calls. Since communication does not scale as well as computations on Haswell, communication takes an increased fraction of the total time in comparison to KNL. We expect even better benefits of our algorithm on GPU-based clusters because of the availability of faster in-node computations.

J. Applicability with small matrices at low concurrency

As observed in previous experiments, the BATCHEDSUMMA3D algorithm is extremely effective when multiplying large-scale matrices on thousands of nodes. Fig. 14 demonstrates that BATCHEDSUMMA3D can reduce the A-Broadcast time even on 16 nodes. However, the reduced communication time has little impact on the total runtime when communication does not dominate the runtime. On 256 node, BATCHEDSUMMA3D runs faster with 4 layers. However, using 16 layers on 256 nodes does not reduce the runtime any further as AllToAll-Fiber starts to become the communication bottleneck. Hence, BATCHEDSUMMA3D is useful even on few hundred nodes if l is set to a small value.

K. A direct comparison with 3D Sparse SUMMA

Fig. 15 compares BATCHEDSUMMA3D with SUMMA3D presented in previous work [13]. We downloaded the SUMMA3D code from the CombBLAS library. Note that the previous SUMMA3D algorithm simply fails when the memory requirement exceeds avaiable memory. Here, we square the Eukarya matrix where batching is not needed on 16 nodes and 256 nodes of Cori KNL using 4 layers and 16 threads per process. The computation in BATCHEDSUMMA3D is more than $8\times$ faster than the previous work, while the communication is also slightly faster. We relied on new hash-based multiplication and merging algorithms (see Sec IV-A), which made the computation much faster.

VI. CONCLUSIONS

This paper presents a robust SpGEMM algorithm that can multiply matrices in batches even when the output matrix exceeds the available memory of large supercomputers. Additionally, the presented algorithm reduces the communication so that SpGEMM can scale to the limit of modern supercomputers. These two techniques together eliminate two fundamental barriers – memory and communication – in large-scale sparse data analysis.

Our result is unexpectedly positive because communication-avoiding (CA) matrix multiplication algorithms trade increased memory for reduced communication. The conventional wisdom suggests that CA algorithms would be detrimental in this already memory-constrained regime. However, 3D algorithms offset the increased broadcast costs associated with batching required in the memory constrained setting, creating a previously unexplored synergy.

Our algorithm will boost many applications in genomics, scientific computing, and social network analysis where SpGEMM has emerged as a key computational kernel. For example, Yelick et al. [39] regarded SpGEMM as a parallelism motif of genomic data analysis with applications in alignment, profiling, clustering and assembly for both single genomes and metagenomes. With the size of genomic data growing exponentially, extreme-scale SpGEMM presented in this paper will enable rapid scientific discoveries in these applications.

ACKNOWLEDGMENTS

This work is supported in part by the Advanced Scientific Computing Research (ASCR) Program of the Department of Energy Office of Science under contract No. DE-AC02-05CH11231, and in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This work used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231.

REFERENCES

[1] G. He, H. Feng, C. Li, and H. Chen, "Parallel simrank computation on large graphs with iterative aggregation," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 543–552.

[2] F. Jamour, I. Abdelaziz, Y. Chen, and P. Kalnis, "Matrix algebra framework for portable, scalable and efficient query engines for rdf graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[3] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 804–811.

[4] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[5] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011.

[6] C. Jain, H. Zhang, A. Dilthey, and S. Aluru, "Validating Paired-End Read Alignments in Sequence Graphs," in *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, vol. 143, 2019, pp. 17:1–17:13.

[7] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "BELLA: Berkeley efficient long-read to long-read aligner and overlapper," *bioRxiv*, p. 464420, 2018.

[8] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[9] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, pp. 47–58, 2014.

[10] 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.

[11] K. Akbudak, O. Selvitopi, and C. Aykanat, "Partitioning models for scaling parallel sparse matrix-matrix multiplication," *ACM Transactions on Parallel Computing*, vol. 4, no. 3, pp. 13:1–13:34, 2018.

[12] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, pp. 1–34, 2016.

[13] 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," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[14] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," in *IEEE IPDPS*, 2020.

[15] O. Selvitopi, S. Ekanayake, G. Guidi, G. Pavlopoulos, A. Azad, and A. Buluç, "Distributed many-to-many protein sequence alignment using sparse matrices," in *Proceedings of the 2020 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'20, 2020.

[16] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

[17] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[18] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[19] A. Azad, A. Buluç, G. A. Pavlopoulos, N. C. Kyrpides, and C. A. Ouzounis, "HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, vol. 46, no. 6, pp. e33–e33, 01 2018.

[20] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software*, vol. 4, no. 3, pp. 250–269, Sep. 1978.

[21] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[22] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *ISC*. Springer, 2015, pp. 48–57.

[23] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *IEEE IPDPS*, 2014, pp. 370–381.

[24] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *IEEE IPDPS Workshops*, 2017, pp. 693–702.

[25] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Comput.*, vol. 90, 2019.

[26] F. Gremse, K. Küpper, and U. Naumann, "Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C429–C449, 2018.

[27] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth-optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *SPAA*, 2020, pp. 293–303.

[28] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU," in *ICPP*, 2017, pp. 101–110.

[29] K. L. Nusbaum, "Optimizing Tpetra's sparse matrix-matrix multiplication routine," SAND2011-6036, Sandia National Laboratories, Tech. Rep., 2011.

[30] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[31] R. A. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," Austin, TX, USA, Tech. Rep., 1995.

[32] A. Lazzaro, J. VandeVondele, J. Hutter, and O. Schütt, "Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5 d algorithm and one-sided MPI," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2017, pp. 1–9.

[33] L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State University, Bozman, MT, 1969.

[34] O. Selvitopi, M. T. Hussain, A. Azad, and A. Buluç, "Optimizing high performance Markov clustering for pre-exascale architectures," in *IEEE IPDPS*, 2020.

[35] I.-M. A. Chen, V. M. Markowitz, K. Chu, K. Palaniappan, E. Szeto, M. Pillay, A. Ratner, J. Huang, E. Andersen, M. Huntemann *et al.*, "IMG/M: integrated genome and metagenome comparative data analysis system," *Nucleic Acids Research*, p. gkw929, 2016.

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

[37] G. Ballard, A. Buluc, 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.

[38] S. van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, Utrecht University, 2000.

[39] K. Yelick, A. Buluç, M. Awan, A. Azad, B. Brock, R. Egan, S. Ekanayake, M. Ellis, E. Georganas, G. Guidi *et al.*, "The parallelism motifs of genomic data analysis," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190394, 2020.