Exploring the Design Space of Distributed Parallel Sparse Matrix-Multiple Vector Multiplication

Hua Huang and Edmond Chow

***Abstract*—We consider the distributed memory parallel multiplication of a sparse matrix by a dense matrix (SpMM). The dense matrix is often a collection of dense vectors. Standard implementations will multiply the sparse matrix by multiple dense vectors at the same time, to exploit the computational efficiencies therein. But such approaches generally utilize the same sparse matrix partitioning as if multiplying by a single vector. This article explores the design space of parallelizing SpMM and shows that a coarser grain partitioning of the matrix combined with a column-wise partitioning of the block of vectors can often require less communication volume and achieve higher SpMM performance. An algorithm is presented that chooses a process grid geometry for a given number of processes to optimize the performance of parallel SpMM. The algorithm can augment existing graph partitioners by utilizing the additional concurrency available when multiplying by multiple dense vectors to further reduce communication.**

***Index Terms*—SpMM, SpMV, distributed-memory matrix multiplication, communication optimization.**

I. Introduction

Linear algebra operations with sparse matrices are fundamental computational kernels in scientific computing, Big Data analysis, and artificial intelligence. While the sparsity of matrices greatly reduces computational costs, harnessing this sparsity poses a challenge. Therefore, accelerating sparse matrix linear algebra operations is crucial and extensively researched.

In sparse matrix linear algebra operations, sparse-dense matrix-matrix multiplication (SpMM) is an important building block. SpMM is used in block iterative solvers [1], [2], [3], dynamical simulations [4], non-negative matrix factorization (NNMF) [5], [6], graph neural network (GNN) training [7], [8], [9], [10], and deep neural network (DNN) training [11], [12]. SpMM calculations exhibit substantial parallelism, underscoring the need for efficient utilization of parallel resources to reduce computation time. Moreover, communication costs have been for a long time relatively more expensive than computation on both shared-memory and distributed-memory platforms. Reducing communication costs is the key to obtaining high performance in SpMM and other parallel algorithms. This work specifically concentrates on reducing *communication* costs in *distributed-memory* parallel SpMM.

Received 2 January 2024; revised 22 August 2024; accepted 25 August 2024. Date of publication 30 August 2024; date of current version 16 September 2024. This work was supported by the U.S. Department of Energy, Office of Science under Grant DE-SC0023445. Recommended for acceptance by Y. Zhang. *(Corresponding author: Edmond Chow.)*

The authors are with the School of Computational Science, Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: huangh223@gatech.edu; echow@cc.gatech.edu).

Digital Object Identifier 10.1109/TPDS.2024.3452478

A natural approach to parallelize SpMM is to treat it as sparse-dense matrix-vector multiplication (SpMV) with multiple input vectors and using the same parallelization as SpMV. This SpMV-based parallel SpMM approach can benefit from well-studied parallel SpMV algorithms. Methods such as (hyper)graph partitioning [13], [14], [15], [16] and other algorithms [17], [18] have been proposed for partitioning sparse matrices to reduce the communication costs of SpMV. A parallel SpMM implementation can directly reuse these sparse matrix partitionings and replace the local SpMV calculation with a SpMM routine to achieve good performance. However, SpMM introduces additional parallelism and allows partitioning the dense input vectors, enabling further reduction in communication costs and improvement in parallel performance.

Fig. 1 shows an example of two SpMM parallelization schemes: partitioning only the sparse matrix like in SpMV and partitioning both the sparse matrix and the input vectors. In the case of a relatively large number of input vectors, the second scheme can require a smaller communication size. Further, these two parallelization schemes are not the only possibilities for four processes. Other schemes may further reduce the SpMM communication size. This example raises a fundamental question: *when and how should the parallelism of multiple input vectors be harnessed to reduce the communication costs of parallel SpMM?* To the best of our knowledge, no existing research has considered this issue.

In response to this inquiry, this paper makes the following contributions.

- In Section II, we first analyze the vast design space of parallelizing SpMM, compare various parallelization schemes, and position existing parallel SpMM algorithms within the design space.
- Section III formulates the communication cost models for different parallelization schemes and provides illustrative examples for these cost models.
- In Section IV, we propose an algorithm to optimize the process grid geometry, aiming to reduce communication costs with multiple levels of parallelism.
- Finally, in Section V, we present theoretical analysis and numerical experiment results to demonstrate the efficacy of our new algorithm in reducing SpMM communication costs and achieving superior parallel performance compared to existing algorithms.

In this paper, we assume that a process is not limited by memory. Such limitations can be incorporated in practice but are omitted here for simplicity. We also assume that the number

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

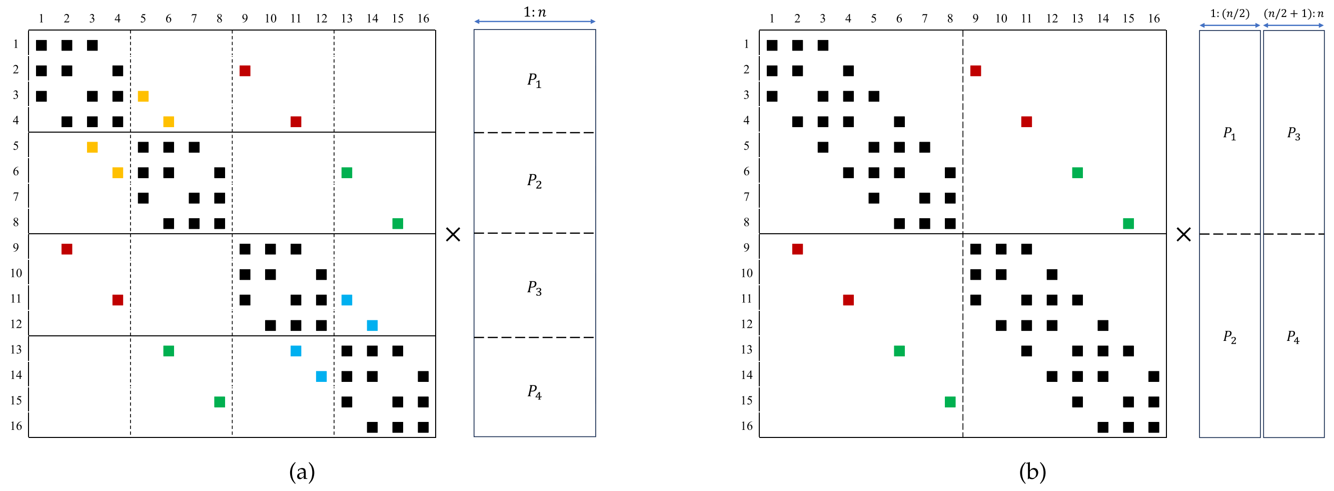


Fig. 1. Two SpMM parallelization schemes for a 16×16 Laplacian matrix A and 4 processes: (a) parallelize over the rows of A only and (b) parallelize over both the rows of A and the columns of B. Scheme (b) needs to replicate A once but needs to replicate fewer B matrix rows (required by non-zeros in off-diagonal blocks, marked with colors).

of vectors being multiplied simultaneously is large enough so that there are no additional efficiencies, e.g., from vectorization, gained by simultaneously multiplying more vectors. (When going from 1 to 4 vectors, there are great performance benefits of SpMM (time per vector), but not from, say 100 to 400 vectors, since the multiplication by 100 vectors is already highly vectorized.) In this regime, communication performance is much more important than vectorization and cache performance.

II. The Design Space of Parallelizing SpMM

Consider a general matrix-matrix multiplication (MM)

$$C = A \times B,\ A \in \mathbb{R}^{m\times k},\ B \in \mathbb{R}^{k\times n},\ C \in \mathbb{R}^{m\times n}. \quad (1)$$

The iteration space for (1) can be viewed as an $m \times k \times n$ cuboid. Each unit volume in the cuboid corresponds to one scalar multiplication and addition. The computation of each unit volume is independent. Parallelizing the arithmetic operations of the iteration space is equivalent to partitioning the cuboid and assigning the partitions to processes. A parallel MM algorithm can be categorized into 1D, 2D, or 3D algorithms if it parallelizes over 1, 2, or 3 dimensions of the iteration space. In this work, we are interested in finding a low communication parallelization scheme for SpMM, where A is a general sparse matrix, and B and C are general dense matrices.

A. 1D Parallelization Schemes for SpMM

One-dimensional (1D) parallelization schemes are fundamental and commonly used approaches for parallelizing SpMV and SpMM. Categorized based on the partitioned dimension of the MM iteration space, 1D parallelization schemes include m-dimension, k-dimension, and n-dimension parallelization schemes.

An m-parallelization corresponds to a 1D row partitioning of both A and C. The rows of A and C are partitioned and assigned to different processes in a consistent manner. The rows of A and C owned by each process might be discontiguous. Each process needs to gather a portion of B matrix rows for its local SpMM calculation, with no additional communication required.

A k-parallelization corresponds to a 1D partitioning of both the columns of A and the rows of B. The columns of A and the rows of B are assigned to different processes in a consistent manner, and each process might have discontiguous columns of A. No communication is needed before the local SpMM computation. Following the local SpMM, the partial results of C need to be reduced and added to obtain the final C matrix.

To balance the computational workload, the rows of A owned by each process in the m-parallelization should contain approximately the same number of non-zeros. As a process may own discontiguous rows of A, various algorithms can be used to compute row partitionings of A with approximately balanced non-zero distributions. It should be noted that permuting the rows of A and assigning a row block with contiguous rows to a process is equivalent to assigning a set of possibly discontiguous rows of the original matrix to the same process. Similarly, a dual discussion holds for the k-parallelization.

An n-parallelization corresponds to a 1D column partitioning of both B and C. This method divides the original SpMM calculation into sub-tasks, each of which computes a portion of the input vectors through SpMM or SpMV operations. This approach is simpler than partitioning along the m or k dimension because different input vectors are identical in terms of both computation and communication. The only required communication involves replicating A between processes.

B. 2D and 3D Parallelization Schemes for SpMM

The 1D parallelization schemes are independent of each other and can be combined to get 2D and 3D parallelization schemes. Without loss of generality, we arrange p processes as a 3D grid of size $p_m \times p_k \times p_n = p$, where $p_m, p_n, p_k \geq 1$. This 3D process

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

grid results from combining a p_m-way m-parallelization, a p_n-way n-parallelization, and a p_k-way k-parallelization.

We first consider the 2D parallelization that partitions both the m and n dimensions. This mn-parallelization involves 1D row partitioning of A and 1D column partitioning of B, both of which are independent of each other. Thus, a p_m-way m-parallelization can be directly combined with a p_n-way n-parallelization. Similarly, a p_k-way k-parallelization can be directly combined with a p_n-way n-parallelization to obtain a kn-parallelization.

The direct combination of k-parallelization and m-parallelization will result in a 2D checkerboard partitioning of A, generally leading to an imbalance distribution of non-zeros. Below, we will introduce various algorithms that can be employed to calculate a 2D sparse matrix decomposition, ensuring a nearly balanced distribution of non-zeros and achieving an optimal or near-optimal SpMV communication cost. Nevertheless, the computation of a high-quality 2D decomposition could be computationally expensive.

The 3D mnk-parallelization can be considered as having p_n groups of processes, where each group's $p_m \times p_k$ processes perform the computation of multiplying A with n/p_n columns of B using a 2D partitioning of A. This naturally inherits all properties of mk-parallelization.

All 2D and 3D parallelization schemes inhabit an extensive design space which is spanned by two perpendicular subspaces. The first subspace involves the dimensions of a generalized 3D process grid $p_m \times p_n \times p_k = p$, where $1 \leq p_m, p_n, p_k \leq p$. The second subspace constitutes the solution space for partitioning the sparse matrix A. If p has numerous prime factors, multiple triplets satisfying $p_m \times p_n \times p_k = p$ can exist. Following the selection of the number of row and column partitions (the values of p_m and p_k), various algorithms can be used to compute a decomposition of A.

C. Existing Parallel SpMM Algorithms

In this section, we locate existing parallel SpMM algorithms in the huge design space of parallelizing SpMM, and discuss their strengths and weaknesses.

We first consider SpMV-based parallel SpMM algorithms. SpMV is one of the well-studied topics in parallel computing. Many optimizations for SpMV has been proposed, including

- New sparse matrix storage schemes (e.g., sliced ELLPACK [19], ESB [20], SELL-C-σ [21], CSR5 [22]);
- Workload partitioning and load balancing strategies (e.g., merge-based partitioning [23], adaptive partitioning for using the CSR format on GPUs [24], [25]);
- (Hyper)graph-based-partitioning schemes (e.g., [14], [15], [16]) and other partitioning algorithms (e.g., [17], [18]) for minimizing communication volume;
- Inspector-Executor (I-E) frameworks that inspect a matrix's sparsity structure and choose a matrix-specific storage format, parallelization scheme, and/or even generated source code (e.g., [26], [27]).

Most of these optimizations fall into the categories of m-parallelization and mk-parallelization. For the latter, the sparse matrix can be partitioned using 2D checkerboard partitioning [16], 2D jagged-like partitioning (the sparse matrix is first partitioned into multiple row or column blocks, then each block is partitioned independently) [16], 2D fine-grain partitioning (non-zeros are individually assigned to processes) [15], [16], and other methods like the Mondrian algorithm [17], [18] and the merge-based SpMV [23]. Some hierarchical parallelization and optimization techniques proposed for sparse tensor-dense vector multiplication [28] can also be applied to SpMV. Bienz et al. [29] proposed a node aware parallel SpMV algorithm to improve inter-node communication performance with system topology knowledge.

Driven by the need to accelerate GNN and sparse DNN, multiple new shared-memory parallel SpMM algorithms targeting multi-core CPUs [30], [31] and GPUs [9], [12], [32], [33], [34] have surfaced in recent years. These algorithms mainly focus on cache blocking, efficient vectorization, and other low-level optimizations. Both new CPU SpMM algorithms use m-parallelization. Due to the difference in programming models, some GPU SpMM algorithms can be categorized into either mk-parallelization or mnk-parallelization.

In terms of distributed-memory parallel SpMM, multiple studies reuse or modify parallel dense general matrix multiplication (GEMM) algorithms for SpMM. In the work of Koanantakool et al. [35], the authors assumed a uniform distribution of non-zeros in A and analyzed the communication costs of applying 2D SUMMA (mn-parallelization) and 3D SUMMA (mnk-parallelization) GEMM algorithms to SpMM directly. They further proposed three new 1.5D SpMM algorithms, which can be categorized into n-parallelization and nk-parallelization. Some application papers describe the parallel SpMM algorithms used by higher-level driver algorithms. MPI_FAUN [6] uses a 1.5D algorithm (mk-parallelization) and CAGNET [7] uses a 2D algorithm (mn-parallelization). These algorithms are designed for handling tall-skinny B matrices. Selvitopi et al. [36] implemented these algorithms and another 2D algorithm (mn-parallelization) in the CombBLAS library. All aforementioned GEMM-based parallel SpMM algorithms partition the sparse matrix A into equal-size blocks. As a result, some A blocks can be empty, leading to severe load imbalance and many unnecessary communications of B matrix elements. Block et al. partition SpMM inter-node communication into sparsity-aware/unaware parts and optimize the communication using collective and one-sided operations [37].

Only a few parallel SpMM algorithms that consider the sparsity of A have been proposed. Acer et al. [38] proposed a generic framework that uses both graph and hypergraph partitioning for minimizing communication costs of SpMM using 1D m-parallelization. Recently, Gianinazzi et al. proposed an "arrow matrix decomposition" approach [39] for parallel SpMM. This approach decomposes the sparse matrix A of the form $A = \sum_{i=1}^{l} P_i B_i P_i^{\mathsf{T}}$, where P_i is a permutation matrix, B_i is an arrowhead sparse matrix with a user-specified tile size b. An L-shape decomposition is applied to B_i for parallelizing the SpMM calculation, without utilizing n-dimension parallelism. Each arrow decomposition requires a specific number of processes, which cannot be calculated easily, for running SpMM.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

Given that calculating an arrow decomposition is even more expensive than finding a 2D decomposition using hypergraph partitioning or the Mondrian algorithm, this method is hard to use in practice.

Some high-level multi-purpose libraries also support parallel SpMM computation. The Cyclops Tensor Framework (CTF) [40] uses the 2.5D algorithm [41] (mnk-parallelization) for parallel sparse and dense matrix multiplications. The Portable, Extensible Toolkit for Scientific Computing (PETSc) library [42], [43] also supports parallel SpMM computation. PETSc uses 1D row partitioning for sparse matrices [44], so its parallel SpMV and SpMM use 1D m-parallelization. Neither CTF nor PETSc is fine tuned for parallel SpMM.

III. Data Transfers in Parallel SpMM

In this section, we discuss the data transfer size, referring to the number of matrix elements requiring communication in a parallel SpMM algorithm. We will not consider any possible special properties (for example, symmetry) of A. Additionally, we assume that all p processes share one copy of A and B at the beginning, and one copy of C at the end.

A. Notation

We consider a parallel algorithm using p processes for computation. All processes have the same number of cores and the same size of memory. A network connects all processes and inter-process data exchange must go through the network. Let P be the grid of all p processes. P can be organized as a 1D, 2D, or 3D grid, and is indexed with tuples of the same dimension. We use one-based indexing for the process grid and MATLAB colon notation to indicate a slice of processes or a slice of a matrix, for example, $P(2,:)$ refers to all processes in the second row of a 2D process grid, and $A(1:4,:)$ refers to the first four rows of matrix A. To represent row and column index sets, we use capital letters I and J, respectively. For example, if $I_1 = \{1,2,3,4\}$, then $A(I_1,:)$ refers to the first four rows of matrix A. The $nnz(\cdot)$ function denotes the number of non-zeros in a matrix or matrix block. We introduce two new functions to facilitate the discussion of communication costs:

- The $nec(\cdot)$ function denotes the set of non-empty column (columns that have non-zeros) indices in a matrix block;
- The $ner(\cdot)$ function denotes the set of non-empty row (rows that have non-zeros) indices in a matrix block.

We use $|\cdot|$ to denote the size of a set. Finally, m, k, and n are the matrix dimensions defined in (1).

B. Data Transfers in 1D Parallelization

The m-parallelization. Denote the A matrix rows owned by process P_i as $A(I_i,:)$. The output matrix C is divided according to the row partitioning of A to avoid communication: process P_i owns $C(I_i,:)$ and computes $C(I_i,:) = A(I_i,:) \times B$. To avoid communication, we require that each row of B is owned by one process that uses this row in its local SpMM (a common practice for 1D row parallel SpMV). The only communication required in this parallelization scheme is gathering rows of matrix B. The total number of matrix elements to be transferred is

$$S_m(p) = \left(\sum_{i=1}^{p} |nec(A(I_i,:))| - k \right) n. \tag{2}$$

The k-parallelization. This scheme is analogous to an m-parallelization. Denote the column block of matrix A owned by process P_i as $A(:,J_i)$. Correspondingly, P_i owns $B(J_i,:)$ and possesses $|ner(A(:,J_i))|$ rows of partial C matrix. We also require that each row of C is owned by one process with a partial result of this row to avoid communication. The total number of matrix elements to be transferred is

$$S_k(p) = \left(\sum_{i=1}^{p} |ner(A(:,J_i))| - m \right) n. \tag{3}$$

The n-parallelization. The B and C matrices are distributed in the same way. This scheme only requires replicating A $p-1$ times so that each process has a full copy of A. The total number of matrix elements to be transferred is

$$S_n(p) = (p-1) \cdot nnz(A). \tag{4}$$

C. Data Transfers in 2D and 3D Parallelization

We first consider the mn-parallelization and the kn-parallelization. As discussed in Section II-B, these 2D parallelization schemes can be achieved by directly combining an existing m-parallelization or k-parallelization with an existing n-parallelization. Using the formulas in Section III-B, the total number of matrix elements to be transferred in an mn-parallelization is

$$S_{mn}(p_m,p_n) = \left(\sum_{i=1}^{p_m} |nec(A(I_i,:))| - k \right) n + (p_n - 1) \cdot nnz(A), \tag{5}$$

and the total number of matrix elements to be transferred in a kn-parallelization is

$$S_{kn}(p_k,p_n) = \left(\sum_{i=1}^{p_k} |nec(A(:,J_i))| - m \right) n + (p_n - 1) \cdot nnz(A). \tag{6}$$

For an mk-parallelization, the total number of matrix elements to be transferred depends on the 2D sparse matrix decomposition algorithm. If a 2D checkerboard or a 2D jagged-like partitioning is used, the total data transfer size can be expressed in the form of

$$S_{mk}(p_m,p_k) = \sum_{i=1}^{p} |nec(A(I_i,J_i))| + \sum_{i=1}^{p} |ner(A(I_i,J_i))| - K, \tag{7}$$

where K is a constant that depends on the distribution of B and C for reducing communication. The total number of matrix elements to be transferred for the mnk-parallelization also depends on the 2D partitioning of A.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SUMMARY OF NOTATION

Notation	Meaning
$nec(\cdot)$	The set of non-empty column indices in a block
$ner(\cdot)$	The set of non-empty row indices in a block
$S_m(p)$	m-parallelization total communication size
$S_n(p)$	n-parallelization total communication size
$S_k(p)$	k-parallelization total communication size
$S_{mn}(p_m, p_n)$	mn-parallelization total communication size
$S_{kn}(p_k, p_n)$	kn-parallelization total communication size
$S_{mk}(p_m, p_k)$	mk-parallelization total communication size

D. Examples of Parallel SpMM Data Transfer

Table I summarizes the notation used in Sections III-B and III-C. We use an example to illustrate some notation.

We use Fig. 1 as the example. Let $I_1 = \{1, 2, 3, 4\}$, $I_2 = \{5, 6, 7, 8\}$, $I_3 = \{9, 10, 11, 12\}$, and $I_4 = \{13, 14, 15, 16\}$. We have

$$nec(A(I_1, :)) = ner(A(:, I_1)) = \{1, 2, 3, 4, 5, 6, 9, 11\},$$

$$nec(A(I_2, :)) = ner(A(:, I_2)) = \{3, 4, 5, 6, 7, 8, 13, 15\},$$

and so on. We first consider a m-parallelization (Fig. 1(a)). Process P_1 owns $B(I_1, 1 : n)$ and it needs to receive $B(\{5, 6, 9, 10\}, 1 : n)$ for computing $C(I_1, :) = A(I_1, 1 : n) \times B$. Process P_2 owns $B(I_2, 1 : n)$ and it needs to receive $B(\{3, 4, 13, 15\}, 1 : n)$ for computing $C(I_2, :) = A(I_2, 1 : n) \times B$. Similar analysis can be applied to P_2, P_3 and P_4. The total communication size is

$$S_m(4) = \left(\sum_{i=1}^{4} |nec(A(I_i, :))| - 16 \right) n = 16n.$$

Now we consider an mn-parallelization for this matrix with $p_m = p_n = 2$ (Fig. 1(b)). Let $\tilde{I}_1 = I_1 \cup I_2$ and $\tilde{I}_2 = I_3 \cup I_4$. Let P_1 and P_3 own $A(\tilde{I}_1, :)$, and let P_2 and P_4 own $A(\tilde{I}_2, :)$. Process P_1 now needs to receive $B(\{9, 11, 13, 15\}, 1 : n/2)$ for computing $C(\tilde{I}_1, 1 : n/2) = A(\tilde{I}_1, 1 : n) \times B(\tilde{I}_1, 1 : n/2)$. Similar analysis applies to P_2, P_3, and P_4. The total communication size is

$$S_{mn}(2, 2) = \left(\sum_{i=1}^{2} |nec(A(\tilde{I}_i, :))| - 16 \right) n + (2 - 1)nnz(A) = 8n + 60.$$

If $n \geq 8$, the mn-parallelization has a smaller communication cost than the m-parallelization.

E. Near-Optimal Parallel SpMM

As we discussed in Section II-B, the parallelization of SpMM encompasses an extremely large solution space, defined by two perpendicular subspaces. Finding an optimal SpMM parallelization is thus very challenging. This is true not even mentioning that finding an optimal SpMM parallelization would involve hypergraph partitioning as a sub-problem, which is known to be NP-hard [45].

Instead of finding an optimal SpMM parallelization, a near-optimal parallelization can be computed via a brute-force approach in a restricted design space. This involves enumerating all combinations of $p_m \times p_k \times p_n = p$, followed by computing a 1D or 2D partitioning of A that minimizes the data transfer size of parallel SpMV using $p_m \times p_k$ processes (the restricted design space), and then choosing the parallelization that has the lowest SpMM data transfer size. This approach is computationally expensive when there are many ways to factor p, or when A is very large.

In the next section, we propose an approach that is more practical than this brute-force approach, and that we also show can work well in practice.

IV. THE CRP-SPMM ALGORITHM

In this section, we present the Communication-Reduced Parallel SpMM (CRP-SpMM) algorithm. CRP-SpMM optimizes the process grid geometry for SpMM using the sparsity pattern of matrix A and the cost models developed in Section III. It can be implemented easily by reusing existing 1D m-parallel SpMV / SpMM algorithms. In this section, we will continue to use the notation defined in Section III.

A. Process Grid Geometry and Matrix Partitioning

The CRP-SpMM algorithm is grounded in the SpMM data transfer size models discussed in Section III. The exhaustive approach outlined in Section III-E for identifying an optimal SpMM parallelization scheme can be very expensive. Instead of seeking the optimal solution, our objective is to employ a cost-effective method to identify a parallelization scheme with lower communication costs than the traditional 1D m-parallelization.

Existing distributed parallel SpMM algorithms have primarily been constrained by the types of partitioning utilized: either partitioning solely the sparse matrix like in parallel SpMV, or partitioning all matrices into equal-size blocks like in parallel GEMM. We do not consider mk-parallelization since it is the traditional parallelization by using a 2D partitioning of A without utilizing n-dimension parallelism. In this work, we aim to harness n-dimensional parallelization to reduce SpMM communication costs compared to this and other approaches already commonly used.

CRP-SpMM uses a generalized 2D mn-parallelization instead of a generalized 3D parallelization for three reasons. First, the solution space of $p_m \times p_n = p$ is smaller than that of $p_m \times p_n \times p_k = p$ for the same p. Additionally, as discussed in Section III-C, finding a balanced 2D partitioning of a sparse matrix is more computationally expensive than finding a balanced 1D row partitioning. Lastly, handling irregular replications of B matrix rows is easier than dealing with irregular reductions of partial results in the C matrix. Hence, we opt for mn-parallelization over kn-parallelization.

Given that we will use the generalized 2D mn-parallelization, CRP-SpMM must determine 1) a $p_m \times p_n = p$ process grid and 2) a 1D p_m-way row partitioning for the sparse matrix A. It seems that p_m of the process grid must be chosen first, followed by the 1D p_m-way partitioning. However, CRP-SpMM does these

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

two steps together. (Hyper)graph-partitioning-based 1D row partitioning methods can yield favorable solutions, particularly in terms of data transfer size. However, running these algorithms multiple times can be computationally expensive. One potential approach involves initially computing a p-way row partitioning of A, followed by generating different p_m-way row partitionings through the merging of row blocks from the initial p-way row partitioning. We are unaware of the existence of an algorithm for this specific task. A possible greedy approach involves iteratively merging two row blocks with the largest communication size until only p_m row blocks with approximately the same number of non-zeros remain. We leave this as a topic for future study.

Assigning discontiguous rows to the same process is equivalent to reordering the matrix rows and dividing A into multiple row blocks, each containing contiguous rows of A. The reverse Cuthill–McKee algorithm [46] and other algorithms can be used for reordering the sparse matrix and improve parallel SpMV performance [47], [48], [49]. CRP-SpMM allows the caller to select a proper reordering algorithm and other methods for calculating a 1D p-way row partitioning of A, and uses this partitioning as an *input* and a baseline in searching for the 2D process grid.

CRP-SpMM uses a greedy algorithm to determine the 2D process grid geometry, utilizing the following formula for calculating communication size:

$$S(p_m, p_n) = r(A) \cdot \left(\sum_{i=1}^{p_m} |nec(A(I_i, :))| - k \right) n + f(A) \cdot (p_n - 1) \cdot nnz(A). \quad (8)$$

Formula (8) differs from Formula (5) in two places. First, Formula (8) takes the reuse of A into consideration. In iterative solvers and some other algorithms, A remains unchanged while B is updated between iterations. Hence, A only needs to be replicated once if $p_n > 1$. The constant $r(A)$ is used to specify the number of times A will be reused. A driver algorithm can estimate a lower bound of $r(A)$, or simply set it as 1 if the lower bound of $r(A)$ is hard to estimate. Second, Formula (8) accounts for the storage of a row or column index whenever a nonzero value is stored. Formula (8) contains the constant $f(A)$, defined as the total memory size required to store A divided by the total memory size needed to store only the non-zeros values (excluding their row and column indices). To illustrate, if A is stored in the compressed sparse row (CSR) format using a 4-byte integer data type for column indices and an 8-byte floating point data type for non-zeros values, $f(A) = 1.5$.

The search of a 2D process grid for minimizing Formula (8) is based on the following assumption.

Assumption 1: The number of B matrix elements to be transferred in a 1D row-wise parallel SpMM increases monotonically with the number of processes.

In other words, using a smaller p_m is likely to reduce the first term on the RHS of Formula (8). If this reduction outweighs the increase in the second term on the RHS, then a more favorable 2D process grid is identified. This observation also leads us to Assumption 2.

Algorithm 1: Computing a 2D Process Grid for CRP-SpMM.

```
Input: SpMM problem dimensions $m, k, n$, sparsity matrix $A$, number of processes $p$, a 1D row partitioning $\{I_i\}_{i=1}^p$
Output: A 2D process grid $p_m \times p_n = p$
1: Set: $p_m = p, p_n = 1$, and $p_f = -1$.
2: Compute $S_{opt} = S(p_m, p_n)$ using Formula (8).
3: Prime factorization $p = \prod_{i=1}^{s} p_i,\ p_i \geq p_{i+1}$.
4: for $i = 1$ to $s$ do
5:     if $(p_i == p_f)$ or $(p_n p_i > n)$ then continue to next $i$.
6:     Set: $p'_n = p_n p_i, p'_m = p/p'_n$.
7:     Merge $p'_n$ contiguous row blocks of $\{I_i\}_{i=1}^p$ to get a new $p'_m$-way 1D partition.
8:     Compute $S(p'_m, p'_n)$ using Formula (8).
9:     if $S(p'_m, p'_n) < S_{opt}$ then
10:        Update: $p_m = p'_m, p_n = p'_n, S_{opt} = S(p'_m, p'_n)$.
11:        Set $p_f = -1$.
12:    else
13:        Set $p_f = p_i$.
14:    end if
15: end for
```

Assumption 2: For the same sparse matrix, the optimal p_n increases when p and/or n increases.

CRP-SpMM finds a 2D process grid that minimizes Formula (8) with a greedy algorithm detailed in Algorithm 1. Instead of exhaustively evaluating all possible combinations of $p_m \times p_n = p$, the greedy algorithm starts with a 1D row partitioning and iteratively increases p_n while calculating the communication size using Formula (8). Importantly, it maintains p_n as a factor of p through this process. Algorithm 1 computes a new p'_m-way 1D partition by merging multiple contiguous row blocks of the initial p-way 1D partitioning. This way of merging matrix blocks can, but not always, reduce the cost of replicating the rows of matrix B (the first term in the RHS of Formula (8)). If the matrix is derived from a structural grid, merging row blocks corresponding to adjacent subdomains can better reduce the communication costs.

In some cases, CRP-SpMM may reduce to either a 1D m-parallelization or a 1D n-parallelization. Let p_s denote the smallest prime factor of p. By using Formula (8), if $S(p/p_s, p_s) > S(p, 1)$, CRP-SpMM will reduce to a 1D m-parallelization. Conversely if $S(1, p) < S(p_s, p/p_s)$, CRP-SpMM will reduce to a 1D n-parallelization. This observation also gives rise to Corollary 1.

Corollary 1: If the SpMV can be computed in a matrix-free way (for example, stencil calculation), $f(A) = 0$ and $p_n = \min(p, n)$ minimizes the SpMV / SpMM communication cost.

After selecting a $p_m \times p_n$ process grid, each of the p_m processes in process grid column j computes a column block $C(:, J_j) = A \times B(:, J_j)$ using 1D row-wise parallel SpMM. The matrix partitionings and communication operations for a 2D mn-parallel SpMM are detailed in Algorithm 2. Steps 1-3in Algorithm 2 are conceptual and do not involve any actual operation. The distributions of A, B, and C adhere to the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: SpMM With a 2D mn-Parallelization.

```
Input: A 1D or 2D process grid $p_m \times p_n$, sparse matrix $A$
  and dense matrix $B$ distributed on $p$ processes.
Output: 1D or 2D partitioned $C = A \times B$ distributed on $p$
  processes
1: Partition $A$ into $p_m$ row blocks s.t.
   $nnz(A(I_i, :)) \approx nnz(A)/p_m, 1 \le i \le p_m$.
2: Partition $B$ into a $p_m \times p_n$ grid of equal size
   sub-matrices s.t. $B(I_i, J_j)$ has $k/p_m$ rows and $n/p_n$
   columns.
3: Partition $C$ into a $p_m \times p_n$ grid of different size
   sub-matrices s.t. $C(I_i, J_j)$ has the same rows as $A(I_i, :)$
   and the same columns as $B(:, J_j)$.
4: if $p_n > 1$ then process $P(i, j)$ replicates $A(I_i, :)$ in
   process grid row $P(i, :)$ using allgather operation.
5: Process $P(i, j)$ packs local $B$ matrix rows needed by
   other processes in process grid column $P(:, j)$ and posts
   non-blocking send and receive operations.
6: Process $P(i, j)$ multiples the diagonal block of $A(I_i, :)$
   with the local $B$ matrix rows.
7: Process $P(i, j)$ waits for all non-blocking receive
   operations to complete.
8: Process $P(i, j)$ multiples the off-diagonal block of
   $A(I_i, :)$ with the received $B$ matrix rows.
```

requirements discussed in Section III-B. The driver algorithm of CRP-SpMM may adjust the matrix distribution after computing a 2D process grid or redistribute A and B before calling Algorithm 2. Steps 5–8 in Algorithm 2 constitute a standard 1D m-parallelization SpMM. Additionally, Algorithm 2 assumes that all processes possess sufficient memory to store all required matrix blocks. The topic of incorporating memory constraints in our communication-reduced algorithm is left as a topic for future study.

B. Complexity Analysis of CRP-SpMM

In this section, we will analyze the number of arithmetic operations, memory usage, communication size, and communication latency of CRP-SpMM. We assume that $\min(p_m, p_n) > 1$, and we further assume butterfly network collectives for communication size and latency analysis [50], which are optimal or near-optimal in the α-β model. The cost of collective operations (assuming "large" messages) used in the analysis are listed here, where n is the message size, P is the number of processes, α is network latency, and β is the inverse of network bandwidth

$$T_{\text{allgather}}(n, P) = \alpha \log_2(P) + \beta n \frac{P-1}{P},$$

$$T_{\text{alltoall}}(n, P) = \alpha(p-1) + \beta n.$$

We define the number of arithmetic operations F and the memory usage M as the maximum number of floating point operations and the maximum number of matrix elements (where each A matrix non-zero is counted as $f(A)$ elements), respectively, on any process in Algorithm 2. After step 5, process $P(i, j)$ stores $A(I_i, :)$ and $B(nA_i, J_j)$. Since $|nA_i| \le k$ always holds, we have

$$F = \mathcal{O}\left(\frac{nnz(A)n}{p}\right), \tag{9}$$

$$M = \mathcal{O}\left(f(A) \cdot \frac{nnz(A)}{p_m} + \frac{kn}{p_n}\right). \tag{10}$$

We define the communication size Q and the communication latency L as the maximum number of matrix elements transferred and the maximum number of messages sent, respectively, by any process in Algorithm 2. Process $P(i, j)$ needs to receive $nnz(A)/p_m$ A matrix elements in the allgather operation and receive $|nA_i|n/p_n \le kn/p_n B$ matrix elements in the alltoall operation, which gives

$$Q = \mathcal{O}\left(f(A) \cdot \frac{nnz(A)}{p_m} \cdot \frac{p_m - 1}{p_m} + \frac{kn}{p_n}\right), \tag{11}$$

$$L = \mathcal{O}\left(\log_2(p_n) + p_m\right). \tag{12}$$

C. CRP-SpMM and Shared-Memory Optimizations

CRP-SpMM focuses on minimizing *inter-process* communication costs in *distributed-memory* parallel SpMM. Its process grid geometry optimization is hardware-independent, presenting both advantages and disadvantages. On the positive side, CRP-SpMM can collaborate with hardware-oriented optimizations, such as techniques for enhancing vectorization, register and cache reuse, and ensuring performance portability. Some shared-memory optimization techniques, including novel sparse matrix storage schemes and inspector-executor (I-E) frameworks, also leverage sparsity structures to enhance SpMV and SpMM performance. These techniques run in parallel with the process grid geometry optimization in CRP-SpMM, offering the potential for a more substantial performance improvement when combined. However, on the negative side, there may be instances where CRP-SpMM determines a parallelization scheme in which each process only computes with a small number of input vectors in local SpMM. This scenario might be unfavorable for vectorization and could adversely impact the performance of local SpMM calculations.

D. Implementation of CRP-SpMM

We implement CRP-SpMM in C + MPI + MKL. We use the CSR format for matrix A with a 4-byte integer data type (`int`) for indices and an 8-byte floating point data type (`double`) for non-zeros. B and C are stored and used in row-major style. We note that CRP-SpMM can use other formats for A, B, and C. Algorithm 2 uses `MPI_Iallgatherv` and overlaps multiple non-blocking operations to better utilize the network bandwidth (a technique proposed in [51]). Algorithm 2 step 5 uses `MPI_Isend` and `MPI_Irecv`. For local SpMM computation, we use the MKL I-E sparse BLAS routine `mkl_sparse_d_mm`. We re-index the local A matrix block's columns to reduce memory usage and improve memory access locality. For example, before column re-indexing, a

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

TABLE II
PROPERTIES OF SPARSE MATRICES USED IN NUMERICAL EXPERIMENTS

Matrix Name	# rows $\times 10^6$	# cols $\times 10^6$	$nnz(A)$ $\times 10^6$	Avg. nnz per row	Problem Kind
PFlow_742	0.7	0.7	37	50	3D structural
Serena	1.4	1.4	65	46	
Geo_1438	1.4	1.4	63	44	
StocF-1465	1.5	1.5	21	14	
Long_Coup_dt6	1.5	1.5	87	59	
Hook_1498	1.5	1.5	61	41	
Flan_1565	1.6	1.6	117	75	
Bump_2911	2.9	2.9	128	44	
com-Orkut	3.1	3.1	234	76	NNMF
Amazon	14.3	14.3	230	16	GNN
reddit	0.2	0.2	115	495	GNN
cage15	5.1	5.1	99	19	DNA
kmer_V2a	55.0	55.0	117	2	Protein
delaunay_n23	8.4	8.4	50	6	Graph
wb-edu	9.8	9.8	57	6	Graph
nlpkkt160	8.3	8.3	229	28	Optimization
Hardesty3	8.2	7.6	40	5	Visualization
ss	1.6	1.6	35	22	Semiconductor
circuit5M	5.6	5.6	59	11	Circuit

process needs to compute a local SpMM

$$\begin{bmatrix} 0 & a & b & 0 & c & 0 & d & 0 \\ 0 & 0 & e & 0 & f & 0 & 0 & g \end{bmatrix} \times [1,2,3,4,5,6,7,8]^{\mathsf{T}}.$$

After column re-indexing, this process computes

$$\begin{bmatrix} a & b & c & d & 0 \\ 0 & e & f & 0 & g \end{bmatrix} \times [2,3,5,7,8]^{\mathsf{T}}.$$

Our 1D m -parallelization SpMM implementation (including re-indexing) is very similar to the parallel SpMV implementation in PETSc (Section IV-A in [44]), while the only difference is that PETSc's implementation uses the PETSc scalable communication layer instead of MPI routines.

V. NUMERICAL EXPERIMENTS

All experiments in this section are performed on the Georgia Tech PACE-Phoenix cluster. Each CPU node has two CPU sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Gold 6226 12-core processor. The compute nodes are connected with 100 Gbps InfiniBand networking.

We use 19 sparse matrices from different fields in numerical experiments. Table II shows the properties of these matrices. Matrix `Amazon` is from [7], while all other matrices are downloaded from the SuiteSparse Matrix Collection [52].

A. Reducing Communication Sizes With 2D Parallelization

In this section, we evaluate the theoretical communication sizes of 1D m-parallelization and 2D mn-parallelization SpMM to illustrate the impact of reducing SpMM communication sizes with 2D parallelization. We utilize Formula (8) with $r(A) = 1$ and $f(A) = 1.5$ for computing communication sizes.

We first consider the first eight matrices in Table II, which are symmetric matrices obtained from 3D structural problems discretized by the finite element method (FEM). Typically, these matrices are partitioned using a domain decomposition method or a graph partitioning algorithm. For our study, we employ the widely used METIS library [13] to compute 1D row partitionings

TABLE III
THE 1D PARTITIONING ("PT"), 2D GRID SEARCH ("GS") TIMINGS (IN SECONDS), AND COMMUNICATION SIZES ("CS", NUMBER OF DOUBLE-PRECISION WORDS) OF 1D m-PARALLELIZATION ("m-PARA.") AND 2D mn-PARALLELIZATION ("mn-PARA.") FOR $p = 256$ PROCESSES AND $n = 1024$ INPUT VECTORS

Matrix Name	1D PT Time (s)	2D GS Time (s)	m-para. CS ($\times 10^6$)	mn-para. $p_m \times p_n$	mn-para. CS ($\times 10^6$)
PFlow_742	8.08	0.01	359	128×2	315
Serena	5.78	0.01	666	128×2	625
Geo_1438	5.08	0.02	674	128×2	630
StocF-1465	4.14	0.01	442	64×4	361
Long_Coup_dt6	7.10	0.02	836	128×2	802
Hook_1498	5.13	0.01	620	128×2	557
Flan_1565	6.90	0.02	604	128×2	603
Bump_2911	13.22	0.04	1160	128×2	1136
com-Orkut	0.01	0.53	107752	8×32	25760
Amazon	0.01	2.17	120815	4×64	39391
reddit	0.01	0.14	34333	16×16	5893
cage15	0.01	0.15	29125	8×32	11803
kmer_V2a	0.01	1.83	115058	1×256	44835
delaunay_n23	0.01	0.13	10070	256×1	10070
wb-edu	0.01	0.13	737	256×1	737
nlpkkt160	0.01	0.24	22121	32×8	12566
Hardesty3	0.01	0.17	8805	8×32	3844
ss	0.01	0.06	6588	32×8	2655
circuit5M	0.01	0.11	22395	16×16	19821

of these eight matrices. We implemented Algorithm 1 using C + OpenMP. Both METIS and our code are compiled using the Intel C/C++ compiler v19.1, and the tests are performed on a single computing node. The last 11 matrices in Table II come from different areas and lack a mesh or grid with a physical structure. As these matrices do not originate from a structured mesh or grid, we refrain from utilizing METIS for calculating 1D row partitionings (although some symmetric matrices may still employ METIS for this purpose). Instead, we use the simple 1D partitioning approach described in Section IV-A.

Table III lists the communication sizes of 1D m-parallelization and 2D mn-parallelization SpMM with $p = 256$ processes and $n = 1024$ input vectors. Despite METIS providing high-quality 1D row partitionings for these sparse matrices derived from 3D FEM, Algorithm 1 demonstrates the capability to find better 2D process grid dimensions, resulting in further reductions in SpMM communication sizes. If a more effective algorithm for computing 1D row partitionings for different p_m values (Algorithm 1 step 7) is employed, the communication sizes of mn-parallelization can be further decreased. The cost of determining a 2D process grid is negligible in comparison to the cost of computing a 1D row partitioning using METIS.

On non-FEM matrices, using a 2D parallelization scheme significantly reduces the total communication sizes for matrices `com-Orkut`, `Amazon`, `reddit`, and `ss`. Additionally, Algorithm 1 outputs a 1D process grid for matrices`delaunay_n23` and `wb-edu`. If a hypergraph partitioning algorithm is used for computing 1D row partitionings, determining the baseline p-way 1D row partitioning might be computational expensive. However, the cost of computing optimized 2D process grid dimensions using Algorithm 1 is likely to be negligible when compared to the cost of computing hypergraph partitioning.

Fig. 2 shows the changes of 2D process grid dimensions $p_m \times p_n = p$ concerning the number of processes (p) and input vectors (n) for the `StocF-1465` matrix. The results in this figure align with Assumption 2. The optimized 2D process grid

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

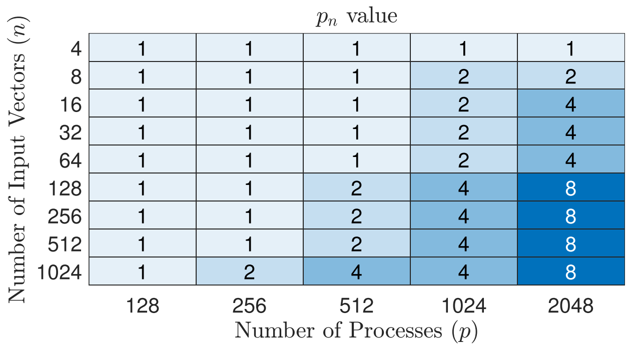


Fig. 2. Process grid dimensions for the `StocF-1465` matrix with different numbers of processes (p) and input vectors (n).

dimension experiences more rapid changes as n increases, primarily because the communication size for replicating B matrix rows increases proportionally to n. Similar observations can be applied to other sparse matrices with comparable application backgrounds.

TABLE IV
SINGLE SPMM EXECUTION TIME ("RT") OF THE BEST COMBBLAS IMPLEMENTATION ("CB-BEST"), 1D m-PARALLELIZATION ("1D m-PARA."), AND CRP-SPMM ON 128 NODES WITH 2 MPI PROCESSES PER NODE ($p = 256$) AND $n = 256$ INPUT VECTORS FOR ALL MATRICES IN TABLE II

Matrix	CB-best	1D m-para.		CRP-SpMM		
Name	RT (s)	PT (s)	RT (s)	$p_m \times p_n$	GS (s)	RT (s)
PFlow_742	0.79	8.08	0.01	256 × 1	0.01	0.01
Serena	1.07	5.78	0.01		0.01	0.01
Geo_1438	1.22	5.08	0.01		0.02	0.01
StocF-1465	0.55	4.14	0.01		0.01	0.01
Long_Coup_dt6	1.59	7.10	0.02		0.02	0.02
Hook_1498	1.19	5.13	0.01		0.01	0.01
Flan_1565	2.20	6.90	0.01		0.02	0.01
Bump_2911	2.46	13.22	0.01		0.04	0.01
com-Orkut	2.31	0.01	0.62	16 × 16	0.30	0.30
Amazon	2.85	0.01	0.69	16 × 16	0.95	0.42
reddit	0.59	0.01	0.21	32 × 8	0.05	0.07
cage15	2.26	0.01	0.08	32 × 8	0.10	0.07
kmer_V2a	5.99	0.01	0.69	32 × 8	1.05	0.70
delaunay_n23	1.97	0.01	0.17	256 × 1	0.10	0.17
wb-edu	2.25	0.01	0.02	256 × 1	0.12	0.02
nlpkkt160	4.65	0.01	0.26	64 × 4	0.17	0.17
Hardesty3	1.40	0.01	0.03	16 × 16	0.13	0.03
ss	0.60	0.01	0.04	64 × 4	0.02	0.03
circuit5M	4.43	0.01	(N/A)	256 × 1	0.07	(N/A)

The running time of calculating 1D baseline partitioning ("PT") and searching for 2D grid ("GS") are also listed. All timings are in seconds. We note that using CRP-SpMM needs to calculate a 1D baseline partitioning before searching for a 2D grid.

B. *Performance of Parallel SpMM Algorithms*

We test and compare seven distributed-memory SpMM algorithms. The 1.5D A-stationary, 2D A-stationary, and 2D C-stationary algorithms are implemented in the Combinatorial BLAS (CombBLAS) 2.0 library [36], [53]. We include our implementation of 1D m-parallelization SpMM and CRP-SpMM in the evaluation. CRP-SpMM directly reuses the 1D m-parallelization SpMM code. The CTF and PETSc libraries support distributed-memory parallel SpMM but are not specifically optimized for this task. We have also tested them and they are much slower than CombBLAS, our 1D m-parallelization SpMM code, and CRP-SpMM, so we do not present and discuss their results. We note that CombBLAS only supports a square process grid. We compile CombBLAS and our code using the Intel C/C++ compiler v19.1 with optimization flags "-xHost -O3". Both codes use Intel MKL v19.1 for shared-memory parallel SpMM and MVAPICH2 2.3.6 for the MPI backend.

Table IV lists the single SpMM execution time of all tested codes on 128 nodes for all matrices in Table II. The running time of calculating 1D baseline partitioning and searching for a 2D grid are also listed. All programs use 2 MPI processes per node. Each MPI process uses 12 OpenMP threads and is bonded to one CPU socket. CombBLAS always requires a square process grid (16 × 16 for 256 processes). Both 1D m-parallelization and CRP-SpMM use METIS for the first 8 matrices, and both algorithms use a simple 1D row partitioning for the last 11 matrices. One-time initialization costs, including initializing MPI communicators and allocating work buffers, are not included in the reported values.

Overall, both 1D m-parallelization and CRP-SpMM greatly outperform CombBLAS across all tested matrices. Although 1D m-parallelization needs to calculate a 1D row partitioning and CRP-SpMM needs to further search for 2D grid sizes, these costs can be be amortized by multiple SpMM executions. As discussed in Section II-C, the parallel SpMM algorithms implemented in CombBLAS are modified from parallel GEMM algorithms. For matrices `wb-edu`, `nlpkkt160`, `cage15`, and `Hardesty`, CombBLAS reported very high load imbalance ratios (defined as $p \max(nnz_i)/nnz$, where nnz_i is the number of non-zeros on process i) ranging from 15 to 18, which are much higher than other non-FEM matrices. Corresponding, the speedups of CRP-SpMM over CombBLAS on these four matrices, ranging from 28× to 112×, are also much larger than the speedups on other non-FEM matrices. These results show that CombBLAS suffers from load imbalance and unnecessary communications due to non-uniform non-zero distributions of A. Selvitopi et al. implemented not only the bulk-synchronous version of these algorithms in CombBLAS, but also the asynchronous version using remote direct memory access (RDMA) operations for SpMM on GPUs [36]. Their work reveals multiple performance tradeoffs for SpMM on GPUs. Since our work does not involve GPUs, the subsequent discussion will focus on comparing 1D m-parallelization and CRP-SpMM.

For the first 8 matrices, given that METIS already provides high-quality 1D row partitionings and the number of input vectors is not sufficiently large, CRP-SpMM reduces to 1D m-parallelization. We also tested $n = 1024$ for these 8 matrices (results omitted) on 256 nodes to evaluate the performance difference between 1D m-parallelization and the 2D mn-parallelization in CRP-SpMM. However, the running time of all 8 matrices under both parallelization schemes is remarkably small, falling within the range of 0.01 to 0.02 seconds. Such minuscule timings can be easily influenced by network performance fluctuations. Meanwhile, in Table III, the differences in communication sizes between 1D and 2D parallelization schemes are marginal, making it challenging to discern a impact on execution time.

For the last 11 matrices, where both 1D m-parallelization and CRP-SpMM uses the same simple 1D row partitioning

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

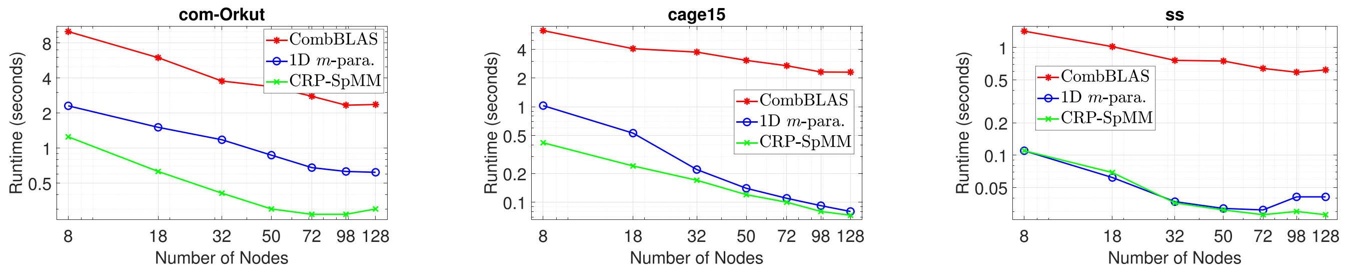


Fig. 3. The CombBLAS, 1D m-parallelization ("1D m-para."), and CRP-SpMM strong scaling results on matrices `com-Orkut`, `cage15`, and `ss`. Each node runs 2 MPI processes. CombBLAS always uses a square process grid.

method, the speedup of CRP-SpMM over 1D m-parallelization aligns with the communication size comparison in Table III. Notably, matrices`com-Orkut`, `Amazon`, `reddit`, and `ss` exhibit substantial speedups with CRP-SpMM over 1D m-parallelization. Additionally, two special cases are observed. For the `kmer_V2a` matrix, Table III shows that the communication size of 2D mn-parallelization is only slightly smaller than 1D m-parallelization. However, 2D mn-parallelization involves more communication operations and incurs a larger overhead cost. The `circuit5M` matrix has certain rows that contain over 5×10^6 non-zeros, resulting in one or more processes having a B matrix block larger than 4 GB for local SpMM, causing a crash in the SpMM routine within Intel MKL.

Fig. 3 shows the CombBLAS, 1D m-parallelization, and CRP-SpMM strong scaling results on matrices `com-Orkut`, `cage15`, and `ss`. CRP-SpMM runs much faster than CombBLAS on all three matrices. On the `com-Orkut` matrix, the CRP-SpMM communication size for using 128 nodes is slightly larger than that of using 98 nodes, which explains the increased runtime on 128 nodes. On the `cage15` matrix, CRP-SpMM uses $p_n = 4$ and $p_n = 9$ for 8 and 18 nodes respectively, leading to a significant reduction of communication sizes compared to 1D m-parallelization. For 32 and more nodes, CRP-SpMM uses $p_n = 8$, the communication size difference between CRP-SpMM and 1D m-parallelization becomes smaller. Therefore, the speedup of CRP-SpMM over 1D m-parallelization also becomes smaller. On the `ss` matrix, the difference in communication size and performance between CRP-SpMM and 1D m-parallelization becomes larger on 98 and 128 nodes.

Fig. 4 provides the runtime breakdown for 1D m-parallelization and CRP-SpMM in tests on 128 nodes for matrices`com-Orkut`, `cage15`, and `ss`. In both parallelization schemes, communication costs associated with replicating A and B dominate the overall runtime, with the communication costs of mn-parallelization being smaller than those of m-parallelization. For matrices `com-Orkut` and `ss`, we observe large communication size imbalances in the 1D m-parallelization reflected by the difference between the averaged and maximum running time of "Pack & Isend B". CRP-SpMM has less severe load imbalance since it uses smaller p_m values than 1D m-parallelization. This also shows the necessity of using a better 1D partitioning algorithm to balance the communication sizes on different processes. Additionally, we observe that the cost of packing B matrix data could surpass the cost of local SpMM calculations. This issue can be alleviated by

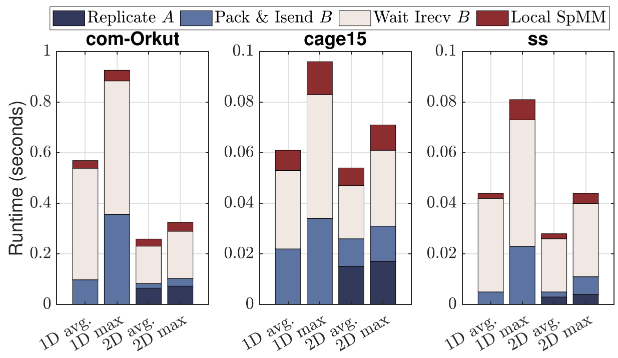


Fig. 4. The 1D m-parallelization ("1D") and CRP-SpMM ("2D") runtime (in seconds) breakdowns for 128 node tests on matrices`com-Orkut`, `cage15`, and `ss`. The "avg." and "max" labels refer to the average and maximum value over all processes. "Replicate A", "Pack & Isend B", and "Wait Irecv B " refers to steps 4, 5, and 7 in Algorithm 2, respectively. "Local SpMM" refers to the total runtime of steps 6 and 8 in Algorithm 2.

using MPI derived data types and other low-level performance optimizations. As our primary focus is not on low-level performance engineering in this work, we leave this issue for future study.

VI. Conclusion and Future Work

In this study, we tackle the optimization of process grid geometry to harness different levels of parallelism, aiming to reduce the communication costs of parallel SpMM. We first analyze the design space of parallel SpMM algorithms and formulate communication cost models for different parallelization schemes. Based on these cost models, we propose an algorithm to optimize the process grid geometry. Our algorithm begins with a 1D row partitioning of the sparse matrix, generated through (hyper)graph partitioning or other methods. Subsequently, it employs a greedy algorithm to explore and discover more effective 2D process grid geometries. Numerical experiments show that our algorithm can find better process grid geometries, even when high-quality 1D row partitionings are used as baselines. Notably, it significantly reduces the communication sizes of SpMM in certain cases. For sparse matrices derived from structural grids or meshes and a small or moderate number of input vectors, using a 1D row parallelization with a graph partitioning or a domain decomposition partitioning is usually good enough. For a large number of input vectors or sparse matrices that arise in

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

other areas, our algorithm is more likely to reduce the communication sizes. Furthermore, our implementation demonstrates a substantial performance advantage over existing distributed-memory parallel SpMM codes across a diverse range of process numbers and sparse matrices from different fields, exhibiting various sparsity patterns. Moreover, our theoretical analysis and experimental data offer insights into potential topics for future research.

The first topic involves designing a fast algorithm for computing 1D row partitionings for different p_m values (Algorithm 1 step 7) using an existing 1D p-way row partitioning. This may further reduce the communication sizes of mn-parallelization.

The second topic involves extending the parallel algorithm to support generalized 3D parallelization. This requires a careful redesign and implementation of the process grid geometry search algorithm to replace Algorithm 1, as well as the communication patterns in SpMM.

The third topic involves enhancing the performance of parallel SpMM through low-level optimizations. Leveraging MPI derived data types and/or MPI one-sided communication operations may help alleviate the overhead associated with manually packing and unpacking B matrix rows.

The code of this work is available in open-source form at https://github.com/scalable-matrix/CRP-SpMM. The methods proposed in this work and our codes can be integrated into higher-level algorithm libraries, for example, parallel iterative eigenvalue solver and non-negative matrix factorization libraries.

Acknowledgment

The authors thank the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology for providing HPC resources that have contributed to the research results reported within this article. The authors also thank Wenkai Shao for discussions on communication-reduced algorithms.

References

[1] A. V. Knyazev, “Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method,” *SIAM J. Sci. Comput.*, vol. 23, no. 2, pp. 517–541, 2001.

[2] Y. Zhou and Y. Saad, “Block Krylov–Schur method for large symmetric eigenvalue problems,” *Numer. Algorithms*, vol. 47, pp. 341–359, 2008.

[3] P. Maris et al., “Large-scale ab initio configuration interaction calculations for light nuclei,” *J. Phys.: Conf. Ser.*, vol. 403, no. 1, Dec. 2012, Art. no. 012019.

[4] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, “Improving the performance of dynamical simulations via multiple right-hand sides,” in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 36–47.

[5] J. Kim and H. Park, “Fast nonnegative matrix factorization: An active-set-like method and comparisons,” *SIAM J. Sci. Comput.*, vol. 33, no. 6, pp. 3261–3281, 2011.

[6] R. Kannan, G. Ballard, and H. Park, “MPI-FAUN: An MPI-based framework for alternating-updating nonnegative matrix factorization,” *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 3, pp. 544–558, Mar. 2018.

[7] A. Tripathy, K. Yelick, and A. Buluç, “Reducing communication in graph neural network training,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Atlanta, GA, USA, 2020, pp. 1–14.

[8] Y. Hu et al., “FeatGraph: A flexible and efficient backend for graph neural network systems,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Atlanta, GA, USA, 2020, pp. 1–13.

[9] G. Huang, G. Dai, Y. Wang, and H. Yang, “GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Atlanta, GA, USA, 2020, pp. 1–12.

[10] M. F. Balin, K. Sancak, and U. V. Çatalyürek, “MG-GCN: A scalable multi-GPU GCN training framework,” in *Proc. 51st Int. Conf. Parallel Process.*, 2023, pp. 1–11.

[11] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, “Fast sparse ConvNets,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 14617–14626.

[12] T. Gale, M. Zaharia, C. Young, and E. Elsen, “Sparse GPU kernels for deep learning,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Atlanta, GA, USA, 2020, pp. 1–14.

[13] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[14] U. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.

[15] U. V. Çatalyürek, “A fine-grain hypergraph model for 2D decomposition of sparse matrices,” in *Proc. 15th Int. Parallel Distrib. Process. Symp.*, San Francisco, CA, USA, 2001, pp. 1199–1204.

[16] U. V. Çatalyürek, C. Aykanat, and B. Uçar, “On two-dimensional sparse matrix partitioning: Models, methods, and a recipe,” *SIAM J. Sci. Comput.*, vol. 32, no. 2, pp. 656–683, 2010.

[17] B. Vastenhouw and R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication,” *SIAM Rev.*, vol. 47, no. 1, pp. 67–95, 2005.

[18] A.-J. N. Yzelman and R. H. Bisseling, “Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods,” *SIAM J. Sci. Comput.*, vol. 31, no. 4, pp. 3128–3154, 2009.

[19] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically tuning sparse matrix-vector multiplication for GPU architectures,” in *Proc. Int. Conf. High Perform. Embedded Architectures Compilers*, Berlin, Heidelberg: Springer, 2010, pp. 111–125.

[20] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, “Efficient sparse matrix-vector multiplication on x86-based many-core processors,” in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 273–282.

[21] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units,” *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.

[22] W. Liu and B. Vinter, “CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 339–350.

[23] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 678–689.

[24] J. L. Greathouse and M. Daga, “Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New Orleans, LA, USA, 2014, pp. 769–780.

[25] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, “Fast sparse matrix-vector multiplication on GPUs for graph applications,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New Orleans, LA, USA, 2014, pp. 781–792.

[26] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, “Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA, 2017, Art. no. 13.

[27] K. Cheshmi, Z. Cetinic, and M. M. Dehnavi, “Vectorizing sparse matrix computations with partially-strided codelets,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, Art. no. 32.

[28] Y. Chen, G. Xiao, M. T. Özsu, C. Liu, A. Y. Zomaya, and T. Li, “aeSpTV: An adaptive and efficient framework for sparse tensor-vector product kernel on a high-performance computing platform,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2329–2345, Oct. 2020.

[29] A. Bienz, W. D. Gropp, and L. N. Olson, “Node aware sparse matrix–vector multiplication,” *J. Parallel Distrib. Comput.*, vol. 130, pp. 166–178, 2019.

[30] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, “Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Phoenix, AZ, USA, 2014, pp. 1213–1222.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.

[31] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayyapan, "Efficient tiled sparse matrix multiplication through matrix signatures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Atlanta, GA, USA, 2020, pp. 1–14.

[32] C. Hong et al., "Efficient sparse-matrix multi-vector product on GPUs," in *Proc. 27th Int. Symp. High- Perform. Parallel Distrib. Comput.*, Tempe, AZ, USA, 2018, pp. 66–79.

[33] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," in *Proc. 24th Int. Conf. Parallel Distrib. Comput.*, Turin, Italy: Springer-Verlag, 2018, pp. 672–687.

[34] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, San Diego, CA, USA, 2020, pp. 376–388.

[35] P. Koanantakool et al., "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Chicago, IL, USA, 2016, pp. 842–853.

[36] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proc. ACM Int. Conf. Supercomputing*, New York, NY, USA, 2021, pp. 431–442.

[37] C. Block, G. Gerogiannis, C. Mendis, A. Azad, and J. Torrellas, "Two-face: Combining collective and one-sided communication for efficient distributed SpMM," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, 2024, pp. 1200–1217.

[38] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Comput.*, vol. 59, pp. 71–96, Nov. 2016.

[39] L. Gianinazzi et al., "Arrow matrix decomposition: A novel approach for communication-efficient sparse matrix multiplication," in *Proc. 29th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2024, pp. 404–416.

[40] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 813–824.

[41] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Proc. Eur. Conf. Parallel Process.*, Berlin, Heidelberg: Springer, 2011, pp. 90–109.

[42] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Basel, Switzerland: Birkhäuser Press, 1997, pp. 163–202.

[43] S. Balay et al., "PETSc/TAO users manual," Argonne Nat. Lab., Tech. Rep. ANL-21/39 - Revision 3.18, 2022.

[44] J. Zhang et al., "The PetscSF scalable communication layer," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 842–853, Apr. 2022.

[45] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Chichester, U.K.: Wiley, 1990.

[46] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proc. 24th Nat. Conf.*, 1969, pp. 157–172.

[47] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Rev.*, vol. 44, no. 3, pp. 373–393, 2002.

[48] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs," *Microprocessors Microsystems*, vol. 36, no. 2, pp. 65–77, 2012.

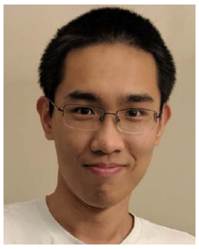
[49] J. D. Trotter et al., "Bringing order to sparsity: A sparse matrix reordering study on multicore CPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA, 2023, Art. no. 31.

[50] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, Feb. 2005.

[51] H. Huang and E. Chow, "Overlapping communications with other communications and its application to distributed dense matrix computations," in *Proc. IEEE 33rd Int. Parallel Distrib. Process. Symp.*, Rio de Janeiro, Brazil, 2019, pp. 501–510.

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

[53] A. Azad, O. Selvitopi, M. T. Hussain, J. Gilbert, and A. Buluç, "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 989–1001, Apr. 2022.



Hua Huang is currently working toward the PhD degree with the School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology. His research interests are in parallel computing, numerical linear algebra, quantum chemistry computations, and numerical methods for data science. He was awarded the 2023 ACM-IEEE CS George Michael Memorial HPC Fellowship Honorable Mention.

Edmond Chow is a professor with the School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology. He previously held positions with DE Shaw Research and Lawrence Livermore National Laboratory. His research interests are in numerical methods, particularly numerical linear algebra, for high-performance computers applied to scientific computing and data science problems, including for PDE models, quantum chemistry, molecular dynamics, Brownian/Stokesian dynamics, inverse problems, data assimilation, uncertainty quantification, Gaussian processes, and machine learning. He was awarded the 2009 ACM Gordon Bell prize and the 2002 Presidential Early Career Award for Scientists and Engineers (PECASE). He is a fellow of the Society for Industrial and Applied Mathematics (SIAM).

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on November 16,2024 at 23:36:12 UTC from IEEE Xplore. Restrictions apply.