Distributed-Memory Parallel Algorithms for Sparse Matrix and Sparse Tall-and-Skinny Matrix Multiplication

Isuru Ranawaka*, Md Taufique Hussain†, Charles Block‡, Gerasimos Gerogiannis§, Josep Torrellas¶, Ariful Azad‖
* Indiana University, Bloomington, IN, USA (isjarana@iu.edu)
† Indiana University, Bloomington, IN, USA (mth@iu.edu)
‡ University of Illinois at Urbana-Champaign, IL, USA (coblock2@illinois.edu)
§ University of Illinois at Urbana-Champaign, IL, USA (gg24@illinois.edu)
¶ University of Illinois at Urbana-Champaign, IL, USA (torrella@illinois.edu)
‖ Indiana University, Bloomington, IN, USA (azad@iu.edu)

***Abstract*—We consider a sparse matrix-matrix multiplication (SpGEMM) setting where one matrix is square and the other is tall and skinny. This special variant, *TS-SpGEMM*, has important applications in multi-source breadth-first search, influence maximization, sparse graph embedding, and algebraic multigrid solvers. Unfortunately, popular distributed algorithms like sparse SUMMA deliver suboptimal performance for TS-SpGEMM. To address this limitation, we develop a novel distributed-memory algorithm tailored for TS-SpGEMM. Our approach employs customized 1D partitioning for all matrices involved and leverages sparsity-aware tiling for efficient data transfers. In addition, it minimizes communication overhead by incorporating both local and remote computations. On average, our TS-SpGEMM algorithm attains 5× performance gains over 2D and 3D SUMMA. Furthermore, we use our algorithm to implement multi-source breadth-first search and sparse graph embedding algorithms and demonstrate their scalability up to 512 Nodes (or 65,536 cores) on NERSC Perlmutter.**

I. Introduction

Multiplication of two sparse matrices (SpGEMM) is a prevalent operation in scientific computing [1, 2], graph analytics [3]–[5], and machine learning [6]. Within these diverse applications, SpGEMM appears in three main variations: (a) $\mathbf{AA}$, which involves squaring a sparse matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and is used in Markov clustering [4] and triangle counting [7]; (b) $\mathbf{AA}^\mathsf{T}$, which entails multiplying a matrix by its transpose and is useful in calculating Jaccard similarity [8, 9], performing sequence alignments [10], and conducting hypergraph partitioning [2]; and (c) $\mathbf{AB}$, which involves multiplying two distinct sparse matrices and is used in multi-source breadth-first search (BFS) [11], the initial phase of Algebraic Multigrid solvers [1], influence maximization [12], and generating sparse graph embeddings. In this paper, we focus on a special instance of the third variant referred to as Tall-and-Skinny-SpGEMM (TS-SpGEMM), where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a square matrix and $\mathbf{B} \in \mathbb{R}^{n \times d}$ is a tall and skinny matrix with $d \ll n$.

TS-SpGEMM plays a crucial role in both graph analytics and scientific computing applications. For instance, iterations of multi-source BFS traversals on a graph are equivalent to TS-SpGEMM operations, where $\mathbf{A}$ is the adjacency matrix of the graph and $\mathbf{B}$ represents the BFS frontiers for all searches. Such multi-source BFS operations are central to calculations of influence maximization [12] and closeness centrality [11], where the sparsity of $\mathbf{B}$ may vary significantly across iterations. Additionally, TS-SpGEMM is applicable to sparse graph embedding algorithms, where each row of $\mathbf{B}$ corresponds to a d-dimensional sparse embedding of a vertex, and is used in graph neural networks (GNNs) that support sparse embeddings [13]. In the context of Algebraic Multigrid (AMG) methods, TS-SpGEMM is utilized during the setup phase, where $\mathbf{B}$ is the restriction matrix created from a distance-2 maximal independent set computation [1].

Despite its wide range of applications, TS-SpGEMM has not received focused attention in the literature (see Table II). Current distributed-memory SpGEMM algorithms, such as Sparse SUMMA [14, 15] in CombBLAS [5] and 1-D partitioning-based algorithms in Trilinos [16] and PETSc [17], perform well for standard scenarios but fall short in the TS-SpGEMM context, as demonstrated by our experiments.

Our work aims to address this gap by introducing a scalable distributed-memory algorithm specifically designed for TS-SpGEMM. Our algorithm follows the principles of Gustavson's algorithm [18], which constructs the output row-by-row. Since both the $\mathbf{B}$ matrix and the output matrix (which we call $\mathbf{C}$) are tall and skinny, 1-D partitioning is better suited for them. For instance, in many cases, the number of columns in matrix $\mathbf{B}$ is lower than the number of processes. However, a basic implementation of Gustavson's algorithm in distributed memory might necessitate fetching a substantial portion of $\mathbf{B}$ into a process, potentially exceeding local memory capacity. We mitigate this issue by utilizing a 2-D virtual layout for matrix $\mathbf{A}$ and conducting multiplications tile by tile, where each tile represents a submatrix of $\mathbf{A}$ stored within a process. During the execution of a tile, only the memory footprint of $\mathbf{B}$ required by the tile needs to be stored in the local memory, reducing the concurrent memory footprint. By adjusting tile widths and heights, we can manage the granularity of computation and communication.

We develop TS-SpGEMM with two computation modes. In the *local compute* mode, $\mathbf{A}$ and $\mathbf{C}$ remain stationary, while data from $\mathbf{B}$ is communicated to perform local multiplications. Conversely, in the *remote compute* mode, $\mathbf{B}$ and $\mathbf{C}$ remain sta-

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

tionary, while data from $\mathbf{A}$ is transferred to remote processes where multiplication is performed, and partial results are then returned back to their respective processes. The choice of local or remote computations is determined for each tile based on its sparsity pattern. By balancing local and remote computations, we can reduce communication costs for certain applications. As for local computations, we adaptively select between a sparse accumulator (SPA) [19] or a hash-based accumulator [20] for local SpGEMM and merging of partial results. These optimizations substantially enhance the performance of our TS-SpGEMM algorithm compared to existing distributed SpGEMM methods in CombBLAS and PETSc.

We have implemented two graph algorithms using TS-SpGEMM: multi-source BFS and sparse graph embedding. In multi-source BFS, the $\mathbf{C}$ matrix from one iteration serves as the $\mathbf{B}$ matrix in the subsequent iteration. Consequently, for scale-free graphs, the sparsity of $\mathbf{B}$ fluctuates dramatically across BFS iterations. For sparse embedding, we implemented a force-directed graph embedding algorithm [21], maintaining the embedding sparse without compromising its quality. In both scenarios, TS-SpGEMM offered substantial performance benefits over other SpGEMM alternatives.

We summarize key contributions of this paper as follows:

- **Algorithm**: We develop a distributed-memory algorithm for TS-SpGEMM. Our algorithm uses tiling to reduce memory requirements and selectively employs local or remote computations to reduce communication.
- **Comparison with SpMM and SUMMA**: We demonstrate the conditions under which TS-SpGEMM outperforms sparse and tall-and-skinny dense matrix multiplication (SpMM) and Sparse SUMMA.
- **Performance and Scalability**: For TS-SpGEMM (with $d = 128$), our algorithm runs on average $5\times$ faster than alternative SpGEMM implementations in CombBLAS and PETSc. TS-SpGEMM scales well to 512 nodes (65,536 cores) on the Perlmutter supercomputer
- **Applications**: Multi-source BFS utilizing our TS-SpGEMM runs upto $10\times$ faster than SUMMA-enabled algorithms in CombBLAS.

Availability: TS-SpGEMM is publicly available at https://github.com/HipGraph/DistGraph. The repository includes scripts to reproduce results presented in this paper.

II. Background and Related Work

A. The TS-SpGEMM problem

The generalized sparse matrix multiplication (SpGEMM) multiplies two sparse matrices $\mathbf{A}$ and $\mathbf{B}$ and computes another potentially sparse matrix $\mathbf{C}$. In this paper, we consider TS-SpGEMM that multiplies a square matrix $\mathbf{A}\in\mathbb{R}^{n\times n}$ with a tall and skinny matrix $\mathbf{B}\in\mathbb{R}^{n\times d}$ and computes another tall and skinny matrix $\mathbf{C}\in\mathbb{R}^{n\times d}$, where $d\ll n$. Although our TS-SpGEMM algorithms can cover the broader SpGEMM scenarios, our emphasis lies specifically on the TS-SpGEMM variant, which is used in numerous applications mentioned earlier. TS-SpGEMM can be also performed on an arbitrary semiring $\mathbb{S}$ instead of the usual (x,+) semiring. For example, we used a $(\wedge, \vee)$ semiring in our implementation of multi-source BFS. Given a matrix $\mathbf{A}$, $nnz(\mathbf{A})$ denotes the number of nonzeros in $\mathbf{A}$, and flops denotes the number of multiplications needed to compute $\mathbf{AB}$.

B. Related work

SpGEMM is a well-studied problem with many sequential, shared-memory and distributed-memory parallel algorithms discussed in the literature. Gao et al. [22] provided an excellent survey of the field.

Shared-memory parallel algorithms. Most shared-memory parallel SpGEMM algorithms can be categorized into two main classes. The first and widely adopted approach is based on Gustavson's algorithm [18], which constructs the output column-by-column (or row-by-row) [15, 20, 23]–[25]. For performance, these algorithms rely on various accumulators based on heap [15], hash table [20], and a dense vector called SPA [19, 23]. The second approach uses the expand-sort-compress strategy [26]–[30], which generates intermediate results through outer products of input matrices and then merges duplicated entries to obtain the final results. The performance of these algorithms depends on factors such as the sparsity of input matrices, compression ratio (the ratio of floating-point operations to nonzeros in the output matrix), number of threads, and efficient utilization of memory and cache. Even in shared memory, most prior work evaluated algorithms with $\mathbf{AA}$ and $\mathbf{AA}^\mathsf{T}$ settings.

Distributed-memory parallel algorithms and libraries. Distributed-memory algorithms for SpGEMM can be classified based on the data distribution method they employ. Algorithms utilizing 1D partitioning distribute matrices across either the row or column dimension. Buluç and Gilbert [31] showed for the $\mathbf{AA}$ case that 1D algorithm fails to scale due to communication cost. The variant of 1D algorithm they considered forms output row-by-row while cyclicly shifting $\mathbf{B}$ to avoid high memory cost. To mitigate communication costs in 1D algorithms, preprocessing with graph/hypergraph partitioning models has been proposed [32]. However, this preprocessing step can pose new scalability challenges, as it often does not scale efficiently. In 2D distribution, the matrices are partitioned into rectangular blocks within a 2D process grid. For example, CombBLAS [33] used the Sparse SUMMA algorithm [14, 34], while Borvstnik et al. [35] used Cannon's algorithm [36] on the 2D distribution of matrices. In the 3D (or 2.5D) variant of the Sparse SUMMA algorithm, each sub-matrix is further divided into layers. This approach exhibits better scalability at larger node counts [15, 37, 38], where the multiplied instances become more likely to be latency-bound.

Given the broad range of applications, most libraries covering sparse linear algebra include an implementation of SpGEMM. As illustrated in Table I, well-known libraries like CombBLAS [5], DBCSR [35], PETSc [17], Trilinos [16], and CTF [39] all feature various SpGEMM algorithms.

Experimental settings. Over the years, researchers have evaluated distributed SpGEMM algorithms under various con-

TABLE I: Libraries with implementations of distributed SpGEMM.

Library	Data Distribution	Algorithm
CombBLAS [5]	2D, 3D	Sparse SUMMA
DBCSR [35]	2D, 3D	Sparse Cannon, One sided MPI
Saena [40]	1D	Recursive
PETSc [17]	1D	Distributed Gustavson
Trillinos [16, 41]	1D	Distributed Gustavson
CTF [39]	1D, 2D, 3D	Sparse SUMMA

TABLE II: Distributed SpGEMM algorithms with published experimental settings.

Algorithm	Data Distribution	Experiments
Sparse SUMMA [5, 14, 15]	2D and 3D	$\mathbf{AA}$, $\mathbf{AA}^\mathsf{T}$, $\mathbf{AB}$
Sparse Cannon [35]	2D	$\mathbf{AA}$
Recursive [40]	1D	$\mathbf{AA}$
Hypergraph Partitioning [32]	1D	$\mathbf{AA}$, $\mathbf{AA}^\mathsf{T}$, $\mathbf{AB}$
Survey [22]	1D	$\mathbf{AA}$, $\mathbf{AA}^\mathsf{T}$
TS-SpGEMM (this paper)	1D (virtual 2D)	$\mathbf{AB}$

TABLE III: List of notations used in the paper

Symbol	Description
$\mathbf{A}$	The $n \times n$ square matrix, row-wise 1D partitioned
$\mathbf{A}^c$	The $n \times n$ square matrix, column-wise 1D partitioned
$\mathbf{B}$	The $n \times d$ tall and skinny matrix, row-wise 1D partitioned
$\mathbf{C}$	The $n \times d$ output matrix, , row-wise 1D partitioned
P_i	The ith process
$\mathbf{A}_i$	The $\frac{n}{p} \times n$ submatrix of $\mathbf{A}$ stored at P_i
$\mathbf{A}_i^c$	The $n \times \frac{n}{p}$ submatrix of $\mathbf{A}^c$ stored at P_i
p	The number of processes
h	The height of a computing tile
w	The width of the tile
t	The number of threads

figurations of input matrices. Table II demonstrates that the majority of algorithms were evaluated for $\mathbf{AA}$ and $\mathbf{AA}^\mathsf{T}$ scenarios. Among these evaluations, 3D Sparse SUMMA [15] and Akbudak et al. [32] conducted experiments for $\mathbf{AB}$ cases, where $\mathbf{B}$ comprised rectangular matrices. However, their experimental settings focused on the setup phase of the AMG solvers, where the number of columns in $\mathbf{B}$ was typically substantial and frequently comparable to that of $\mathbf{A}$. Consequently, most existing algorithms have not been evaluated for TS-SpGEMM scenarios as discussed in this paper.

III. Distributed-Memory Algorithms

A. Distributed TS-SpGEMM based on Gustavson's algorithm

In the row variant of Gustavson's algorithm, to compute the rth row of $\mathbf{C}$, the rth row of $\mathbf{A}$ is scanned. For each nonzero element stored at location (r, c) of $\mathbf{A}$, the cth row of $\mathbf{B}$ is scaled with that particular non-zero value of $\mathbf{A}$ and merged together to obtain $\mathbf{C}(r, :)$ as follows:

$$\mathbf{C}(r,:) = \sum_{c:\mathbf{A}(r,c)\neq 0} \mathbf{A}(r,c)\mathbf{B}(c,:) \tag{1}$$

To implement Gustavson's algorithm in a distributed setting, we use 1-D row partitioning of all matrices, where $\mathbf{A}_i \in \mathbb{R}^{\frac{n}{p} \times n}$, $\mathbf{B}_i \in \mathbb{R}^{\frac{n}{p} \times d}$, and $\mathbf{C}_i \in \mathbb{R}^{\frac{n}{p} \times d}$ are submatrices of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ stored by the ith process P_i. Table III shows the list of notations used in the paper.

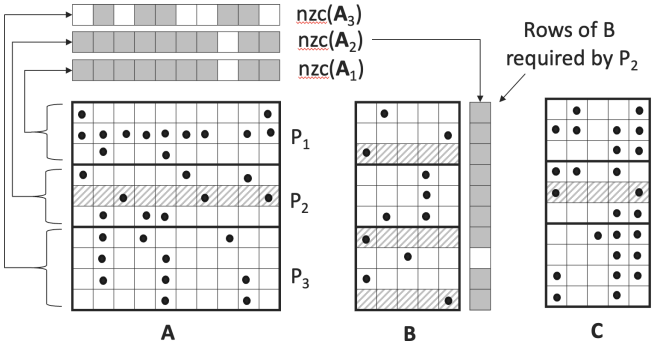

Fig. 1: Distributed memory Gustavson's algorithm using a toy example of 10×10 square sparse matrix and 10×5 tall-and-skinny sparse matrix distributed over 3 processes in a row partitioned manner (dark lines represent process boundary). Shaded rows of $\mathbf{A}$ and $\mathbf{B}$ represent parts of the input matrices involved in the computation of the shaded row of $\mathbf{C}$. Shaded elements in the nzc vectors represent columns with at least one non-zero (non-zero columns) of the local matrix owned by each process. Thus, the nzc vector of each process determines which rows of $\mathbf{B}$ are accessed by this process. Note that while the sparsity patterns of $\mathbf{A}_1$ and $\mathbf{A}_2$ differ significantly, both require all but one row of $\mathbf{B}$.

Algorithm 1 Overview of Naive TS-SpGEMM

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times d}$ distributed in p processes.
Output: $\mathbf{C} \in \mathbb{R}^{n \times d}$ distributed in p processes.

1: **procedure** TS-SpGEMM-Naive($\mathbf{A}_i$, $\mathbf{B}_i$) at P_i
2: nzc($\mathbf{A}_i$) $\leftarrow$ Non-zero column ids in $\mathbf{A}_i$
3: $r \leftarrow$ AllToAll(nzc($\mathbf{A}_i$)) ▷ Requested rows of $\mathbf{B}_i$
4: $\mathbf{B}_{recv} \leftarrow$ AllToAll($\mathbf{B}_i(r, :)$)
5: $\mathbf{C}_i \leftarrow$ LocalSpGEMM($\mathbf{A}_i, \mathbf{B}_{recv}$)
return $\mathbf{C}$

Alg. 1 shows a naive implementation of TS-SpGEMM where $\mathbf{A}$ and $\mathbf{C}$ stays stationary while $\mathbf{B}$ moves. We call this algorithm TS-SpGEMM-Naive. In this algorithm, each process sends requests to potentially all other processes to collect the necessary rows of $\mathbf{B}$ using two AllToAll communication (Line 3 and 4, Alg. 1). After the necessary portion of $\mathbf{B}$ is received, the output is generated by a local SpGEMM. We use a toy example in Fig. 1 to explain the process. Variants of this algorithm are implemented in popular libraries such as PETSc [17] and Trillinos [16]. We identify several avenues to optimize this algorithm.

Eliminating communication needed to send requests. In Alg. 1 Line 3, we send column indices of $\mathbf{A}$ to fetch necessary rows of $\mathbf{B}$. For example, consider $\mathbf{A}$ to be an $n \times n$ Erdős-Rényi matrix of an average degree of k_A. Then, the expected number of nonzero columns per process would be $\frac{n\,k_A}{p}$. Hence, each process sends $\frac{n\,k_A}{p}$ indices in the AllToAll communication at Line 3 of Alg. 1. We eliminate this communication by keeping another copy of $\mathbf{A}$ and partitioning it column-wise among processes, where P_i stores $\mathbf{A}_i^c \in \mathbb{R}^{n \times \frac{n}{p}}$. By

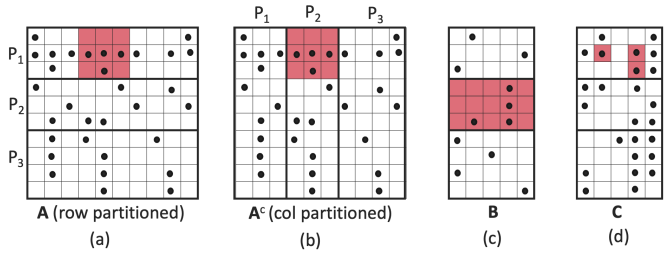

Fig. 2: Distribution of the same matrices as in Fig.1 involved in our algorithm. The highlighted regions in each subfigure represent (a) a 3×3 tile in $\mathbf{A}$, (b) the same tile in $\mathbf{A}^c$, (c) the rows of $\mathbf{B}$ used by this tile, and (d) the nonzeros in $\mathbf{C}$ produced by these tiles.

utilizing $\mathbf{A}^c$, each process can precisely identify which rows of its local copy of $\mathbf{B}$ are needed by other processes, thereby eliminating the necessity to communicate indices beforehand. Fig. 2 explains the benefit of keeping $\mathbf{A}^c$. In this example, P_2 is trying to determine which rows of $\mathbf{B}_2$ will be needed by P_1. By using $\mathbf{A}_i^c$, P_2 can calculate the necessary rows (in this example all rows of $\mathbf{B}_2$) that are sent to P_1 requiring P_1 to communicate the corresponding indices. Thus, our distribution strategy decreases communication at the cost of doubling the memory requirement for $\mathbf{A}$.

Reducing memory requirement. At the end of communication, each process may receive a considerable number of rows of $\mathbf{B}$ from other processes. For example, if $\mathbf{A}_i$ has a dense row, it will need the entire $\mathbf{B}$ to compute $\mathbf{C}_i$. Fig. 1 illustrates this scenario where the second row of $\mathbf{A}_1$ is nearly dense, necessitating all rows except one from $\mathbf{B}$ to be stored in P_1. This memory bottleneck can also arise when $\mathbf{A}$ is an Erdős-Rényi matrix with an average degree of k_A. Let each row of $\mathbf{B}$ have k_B nonzeros. As the expected number of nonzero columns of $\mathbf{A}$ in a process is $\frac{n\,k_A}{p}$, the total memory requirement in a process could be $\frac{n\,k_A\,k_B}{p}$. Thus, depending on the average number of nonzeros in each row of $\mathbf{A}$ and $\mathbf{B}$, the memory requirement to receive remote rows of $\mathbf{B}$ can be prohibitively large.

We address the high-memory requirement, by partitioning $\mathbf{A}_i$ into tiles. A $w \times h$ tile is a submatrix of $\mathbf{A}_i$, where $\mathbf{A}_i(w,h) \in \mathbb{R}^{w \times h}$ with $h \leq n/p$ and $w \leq n$. We conduct computations tile by tile, where each process receives rows of $\mathbf{B}$ corresponding to the nonzero columns in the current tile of $\mathbf{A}$. Fig. 2 shows an example where P_1 computes with the highlighted tile of $\mathbf{A}$ by storing $\mathbf{B}_2$ (also highlighted in Fig. 2) in P_1. We discuss the tile selection policy in the next section.

Improving memory locality in computation In Line 5 of Alg. 1, local matrix multiplication is performed to generate the final output. Since the local sparse matrices are stored in CSR format, this computation can potentially involve accessing rows of $\mathbf{B}_{recv}$ randomly depending on the sparsity pattern of $\mathbf{A}_i$. By computing tile by tile, we also maintain a reasonable memory locality for this random access pattern.

Improving communication by alternating between moving $\mathbf{B}$ and $\mathbf{C}$. Up to this point, we have exclusively focused on communicating submatrices of $\mathbf{B}$ needed for a tile for localized computation of $\mathbf{C}$ within their assigned processes. We refer to this method of computation as the *local mode of computation.* However, if there is a dense row of $\mathbf{A}_i$ in a tile, we can further optimize the communication required to compute the relevant portion of $\mathbf{C}_i$. For example, consider the tile highlighted in Fig. 2. This tile affects the highlighted entries of $\mathbf{C}$ in P_1 (Fig. 2d). If we perform this computation at P_1, 4 nonzero elements of $\mathbf{B}$ need to be communicated from P_2 to P_1 (Fig. 2c), but this computation affects only 3 nonzeros of $\mathbf{C}$ (Fig. 2d). Instead, because we maintain a column partitioned copy $\mathbf{A}^c$, P_2 can compute the relevant entries in $\mathbf{C}$ and send the result back to P_1, reducing the required amount of communication. We refer to this method of computation as the *remote mode of computation.* This communication optimization via remote computation only works when the number of output nonzeros produced for a tile is less than the number of nonzeros required from $\mathbf{B}$. Due to maintaining two copies of $\mathbf{A}$, it is possible to mark each tile as a local or remote compute tile through a symbolic step without requiring any communication. In this way, it is possible to switch between the move-$\mathbf{B}$ and move-$\mathbf{C}$ variants of TS-SpGEMM.

Improving load balance by alternating between moving $\mathbf{B}$ and $\mathbf{C}$. Line 5 of Alg. 1 may cause computation and memory load imbalance when there is a dense row in matrix A. Unlike the 2D distribution of matrices, the memory imbalance is inherent in 1D distribution, a characteristic that persists in our algorithm. However, by delegating computations to remote processes, we can partially balance the computational workload, which enhances the scalability of our algorithm.

B. Distributed TS-SpGEMM with tiling

In the previous section, we described several modifications of Alg. 1 to reduce memory and communication requirements and improve load balance. This section discusses the improved algorithm with all those improvements.

Overall data distribution and storage. As mentioned before, we use 1-D row partitioning of all matrices, where $\mathbf{A}_i \in \mathbb{R}^{\frac{n}{p} \times n}$, $\mathbf{B}_i \in \mathbb{R}^{\frac{n}{p} \times d}$, and $\mathbf{C}_i \in \mathbb{R}^{\frac{n}{p} \times d}$ are submatrices of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ stored the ith process P_i. Additionally, we use 1-D column partitioning of $\mathbf{A}$ where P_i stores $\mathbf{A}_i^c \in \mathbb{R}^{n \times \frac{n}{p}}$. At the ith process, $\mathbf{A}_i$ is divided into $w \times h$ tiles, where each tile is a submatrix of $\mathbf{A}_i$ with $h \leq n/p$ and $w \leq n$. Similarly, $\mathbf{A}_i^c$ is divided into $h \times w$ tiles.

Overview of our distributed TS-SpGEMM algorithm. Algorithm 2 provides a high-level description of our algorithm from P_i's point of view. At fist, we generate $w \times h$ tiles for $\mathbf{A}_i$ and $h \times w$ tiles for $\mathbf{A}_i^c$ (Algorithm 2, Line 9)). The optimal tile width and height depend on the available memory and sparsity of input matrices. We tune them empirically as discussed in the result section. After the tiles are generated, we categorize them in local, remote and diagonal tiles (Algorithm 2, Line 10)). The algorithm used to group tiles is discussed in the next section.

Processing remote tiles (lines 11-18 in Algorithm 2). Let A_i^{remote} be a $w \times h$ tile stored at P_i. Let P_j denote the process where the required portion of $\mathbf{B}$ necessary for this

Algorithm 2 Distributed TS-SpGEMM algorithm at P_i

1: **Inputs:**
2: $\mathbf{A}_i$: Row partition of $\mathbf{A}$ stored at P_i
3: $\mathbf{A}_i^c$: Column partition of $\mathbf{A}$ stored at P_i
4: $\mathbf{B}_i$: Row partition of $\mathbf{B}$ stored at P_i
5: $\mathbb{X}$: Tile mode (local/remote) selection policy
6: **Output:**
7: $\mathbf{C}_i$: Row partition of the output stored at P_i
8: **procedure** DIST-TS-SPGEMM($\mathbf{A}_i, \mathbf{A}_i^c, \mathbf{B}_i, h, w$) at P_i
9: GENERATETILES($\mathbf{A}_i, \mathbf{A}_i^c, h, w$)
10: DECIDEMODE($\mathbf{A}_i, \mathbf{A}_i^c, \mathbf{B}_i, \mathbb{X}$)
11: ▷ Compute output for remote tiles
12: **for** each remote tile $\mathbf{A}_i^{\text{remote}}$ at P_i **do**
13: $P_j \leftarrow$ the remote computation process for this tile
14: At P_j, extract $\mathbf{A}_i^{\text{remote}}$ from $\mathbf{A}_j^c$
15: $\mathbf{B}_j^{\text{remote}} \leftarrow$ At P_j, extract the submatrix from $\mathbf{B}_j$ corresponding to nonzero columns of the remote tile
16: $\mathbf{C}_j^{\text{remote}} \leftarrow$ LOCALSPGEMM($\mathbf{A}_i^{\text{remote}}, \mathbf{B}_j^{\text{remote}}$)
17: Send $\mathbf{C}_j^{\text{remote}}$ back to P_i
18: $\mathbf{C}_i =$ MERGE($\mathbf{C}_i, \mathbf{C}_j^{\text{remote}}$) at P_i
19:
20: ▷ Compute output for diagonal tiles
21: $\mathbf{C}_i^{\text{diag}} \leftarrow$ LOCALSPGEMM($\mathbf{A}_i^{\text{diag}}, \mathbf{B}_i$)
22: $\mathbf{C}_i =$ MERGE($\mathbf{C}_i, \mathbf{C}_i^{\text{diag}}$)
23:
24: ▷ Compute output for local tiles
25: **for** each local tile $\mathbf{A}_i^{\text{local}}$ at P_i **do**
26: $P_j \leftarrow$ Process storing the necessary $\mathbf{B}$ submatrix
27: $\mathbf{B}_j^{\text{local}} \leftarrow$ Fetch $\mathbf{B}$ submatrix from P_j
28: $\mathbf{C}_i^{\text{local}} \leftarrow$ LOCALSPGEMM($\mathbf{A}_i^{\text{local}}, \mathbf{B}_j^{\text{local}}$)
29: $\mathbf{C}_i =$ MERGE($\mathbf{C}_i, \mathbf{C}_i^{\text{local}}$)

tile is stored[1]. Given the column partitioned matrix $\mathbf{A}^c$, P_j can access the remote tile A_i^{remote} from $\mathbf{A}^c$ without any communication with P_i. Then, P_j extracts rows of $\mathbf{B}_j$ corresponding to nonzero columns of this remote tile and store it in $\mathbf{B}_j^{\text{remote}}$ (line 15). After multiplying the remote tile with $\mathbf{B}_j^{\text{remote}}$, the result is sent back to P_i, where the partial result is merged with $\mathbf{C}_i$.

Processing diagonal tiles (lines 20-22 in Algorithm 2). Let $A_i^{\text{diag}} = \mathbf{A}_i \cap \mathbf{A}_i^c$ be a tile on the diagonal of $\mathbf{A}$. For this tile, the corresponding entries of $\mathbf{B}$ are also available in P_i. Hence, this multiplication is performed locally without any communication.

Processing local tiles (lines 24-29 in Algorithm 2). Let A_i^{local} be a $w \times h$ tile stored at P_i. Let P_j denote the process where the required portion of $\mathbf{B}$ necessary for this tile is stored. Given the column partitioned matrix $\mathbf{A}^c$, P_j can access the local tile A_i^{local} from $\mathbf{A}^c$ and extracts necessary portion of $\mathbf{B}_j$ at $\mathbf{B}_j^{\text{local}}$ (line 27). Then, P_j send $\mathbf{B}_j^{\text{local}}$ to P_i. Upon receiving $\mathbf{B}_j^{\text{local}}$, P_i performs local computations and merge the results with $\mathbf{C}_i$.

[1]When the tile width is greater than n/p, the necessary rows of $\mathbf{B}$ is distributed across multiple processes. For simplicity, we refer to a single process P_j in line 13 of Algorithm 2

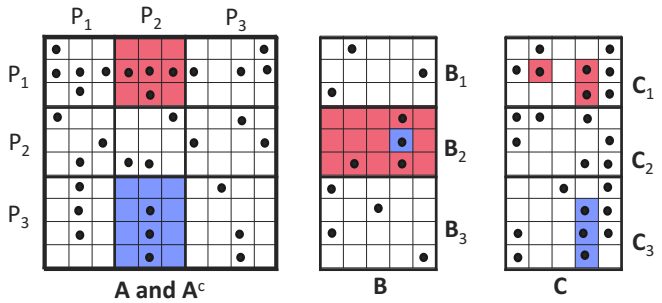

Fig. 3: P_2 decides the mode of two tiles shown in red and blue within $\mathbf{A}$. To compute $\mathbf{C}_1$, the red tile is multiplied with the entire $\mathbf{B}_2$. Since $nnz(\mathbf{B}_2)$ is greater than the affected nonzeros in $\mathbf{C}_1$, the red tile is marked as a remote tile within $\mathbf{A}_1$. To compute $\mathbf{C}_3$, the blue tile is multiplied with just one nonzero in $\mathbf{B}_2$ shown in blue. In this case, it is beneficial to communicate necessary data from $\mathbf{B}$ and hence, the blue tile is marked as a local tile within $\mathbf{A}_3$.

Consolidated communication. For simplicity, Alg. 2 discusses communication concerning P_i. As all processes engage in computations for both local and remote tiles, communication for the ith tile across all processes is consolidated into a single AllToAll communication at lines 17 and 27 of Alg. 2.

C. Local computations

Algorithm 2 performs two computations: (1) SpGEMM involving local or remote tiles and (2) merge results from a tile with the results from other tiles. For both of these operations, we use SPA or hash-based accumulators. Previous work [20, 42] demonstrated that hash-based merging performs the best for $\mathbf{AA}$ and $\mathbf{AA}^\mathsf{T}$ operations. However, for tall-and-skinny $\mathbf{B}$ matrices, we observed that the SPA outperforms hash-based SpGEMM and merging techniques. This is attributed to the fact that each row of the output matrix is of length d. Therefore, a row-by-row SpGEMM necessitates a dense vector for SPA with a length of d. When d is small, it can easily fit into the lowest level of cache, resulting in enhanced performance. We parallelize local computations by assigning different rows of the output to t threads. For $d > 1024$, we opt for a hash-based SpGEMM, as at large values of d, SPA tends to spill out of the cache.

D. Tile mode selection

We use a symbolic step to categorize tiles into local and remote modes with an aim of reducing communication. In this step, P_i computes the modes of tiles available in $\mathbf{A}_i^c$ based on the exact communication overheads of possible local and remote computations for each tile. Fig. 3 explains this step with three processes. In this example, the red tile is located in P_1 as part of $\mathbf{A}_1$, but it is also available in P_2 as part of $\mathbf{A}_2^c$. Hence, P_2 can multiply the red tile with its local $\mathbf{B}_2$ to estimate the contribution of this tile to $\mathbf{C}_1$. In Fig. 3, the entire $\mathbf{B}_2$ (four nonzeros) is needed for the red tile to generate three nonzeros in $\mathbf{C}_1$. Hence, a remote computation of this red tile at

P_2 and sending the results back to P_1 reduces communication. Hence, the red tile is marked as a remote tile by P_2.

On the other hand, the blue tile is located in P_3 as part of $\mathbf{A}_1$, but it is also available in P_2 as part of $\mathbf{A}_3^c$. Hence, P_2 can multiply the red tile with its local $\mathbf{B}_2$ to estimate the contribution of this tile to $\mathbf{C}_3$. In Fig. 3, only one nonzero from $\mathbf{B}_2$ is needed for the red tile to generate three nonzeros in $\mathbf{C}_3$. Hence, a local computation of this blue tile at P_3 by getting necessary data from P_2 reduces communication. Thus the blue tile is marked as a local tile by P_2. Following these steps, every process can categorize tiles from their respective column partition of $\mathbf{A}^c$. Note that the selection tiles do not require any communication. After the modes of all tiles are finalized, the modes of the tiles are shared with all processes via an AllToAll communication step. The cost of this communication is not significant since it only communicates a binary value (local or remote) for each tile.

E. Communication and space complexity

Communication complexity of Algorithm 2. The communication complexity depends on the cost of (a) receiving remotely computed outputs (2, lines 17), and (b) sending rows from $\mathbf{B}$ to other processes for local computations (2, line 27). In both cases, we use AlltoAll collective communication to transfer data. To analyze the communication complexity, we used the $\alpha - \beta$ model [43], where α is the latency constant corresponding to the fixed cost of communicating a message, 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.

Let k_A, k_B, and k_C denote the average number of nonzeros in each row of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively. We consider an $n/p \times n/p$ tile $\mathbf{A}^{\text{tile}}$ that is multiplied with $\mathbf{B}^{\text{tile}}$ to generate partial result $\mathbf{C}^{\text{tile}}$. Here, $nnz(\mathbf{B}^{\text{tile}}) = n\,k_B\,/p$ and $nnz(\mathbf{C}^{\text{tile}}) = n\,k_C\,/p$. Assuming the pairwise exchange algorithm typical for long messages in MPI implementations, the communication cost for a remote tile is $\mathcal{O}(\alpha p + \beta \frac{(p-1)n\,k_C}{p})$. Similarly, the communication cost for a local tile is $\mathcal{O}(\alpha p + \beta \frac{(p-1)n\,k_B}{p})$. Since a tile is either local or remote depending on the communication cost, the overall communication costs for a tile is $\mathcal{O}(\alpha p + \beta \frac{(p-1)n \min\{k_B, k_C\}}{p})$.

Space complexity of Algorithm 2. In 1-D partitioning, there could be storage imbalance if different row partitions have different numbers of nonzeros. This inherent problem of 1-D partitions also exists in our algorithm. We analyze the additional memory requirements for an $n/p \times n/p$ tile, which can be translated to other tile sizes. We use the nonzero settings discussed in the communication analysis. For a local tile, the additional memory required to store received submatrices of $\mathbf{B}$ is $\mathcal{O}(n\,k_B\,/p)$. Similarly, for a remote tile, the additional memory required to store partial results of $\mathbf{C}$ is $\mathcal{O}(n\,k_C\,/p)$. These estimates are not precise, as the actual memory requirements can be substantially lower depending on the sparsity of the tile. For local SpGEMM, the memory requirement depends on the accumulator (SPA/Hash) used. For SPA, where each of the t threads maintain their private SPA, the memory requirement is $\mathcal{O}\Big(t \times d + \frac{nnz(\mathbf{C})}{p} + n\Big)$, and for hash-based SpGEMM the memory complexity is $\mathcal{O}\Big(\frac{nnz(\mathbf{C})}{p} + n\Big)$.

Algorithm 3 Multi-source BFS

Input: Adjacency matrix $\mathbf{A} \in \mathbb{B}^{n\times n}$ and vector $\mathbf{f} \in \mathbb{B}^{n\times 1}$ with d non-zero entries representing sources of BFS traversal.
Output: Tall and Skinny matrix $\mathbf{S} \in \mathbb{B}^{n\times d}$ representing vertices reachable from d sources.

1: **procedure** DIST-MSBFS($\mathbf{A}$, $\mathbf{f}$)
2: $\mathbf{F} \leftarrow$ INIT($\mathbf{f}$) ▷ Initialize frontier, $\mathbf{F} \in \mathbb{B}^{n\times d}$
3: $\mathbf{S} \leftarrow \mathbf{F}$ ▷ Mark sources as visited
4: SR $\leftarrow$ SEMIRING($\wedge, \vee$)
5: **while** $nnz(\mathbf{F}) > 0$ **do**
6: $\mathbf{N} \leftarrow$ TS-SPGEMM($\mathbf{A}, \mathbf{F}$, SR) ▷ Discover next frontier
7: $\mathbf{F} \leftarrow \mathbf{N} \setminus \mathbf{S}$ ▷ Remove already visited vertices
8: $\mathbf{S} \leftarrow \mathbf{S} \vee \mathbf{N}$ ▷ Update so far visited list
9: **return** $\mathbf{S}$

IV. ALGORITHMS IMPLEMENTED WITH TS-SPGEMM

To demonstrate the utility of TS-SpGEMM in practical settings, we implemented two graph algorithms that require the multiplication of a square matrix with a tall-and-skinny matrix. We briefly discuss these algorithms in this section.

A. Distributed multi-source BFS

BFS traversal from a single source can be translated as a sequence of sparse matrix-sparse vector (SpMSpV) operations [44] with $(\wedge, \vee)$ semiring (or a (sel2nd,min) semiring when the reconstruction of the BFS tree is desired) [25, 45]. In this formulation, the input vector represents the current BFS frontier, while the output vector, after eliminating already visited vertices, indicates the next frontier. In the beginning, the input vector contains exactly one non-zero, indicating the source of the traversal. For scale-free graphs, the BFS frontier initially becomes denser and then gradually becomes sparser as more vertices are discovered [46].

A multi-source BFS from d sources runs concurrent BFSs on the same input graph as described in Alg. 3. When implemented with TS-SpGEMM, the multi-source BFS maintains a tall-and-skinny sparse matrix $\mathbf{F}$ where the ith column represents the frontier corresponding to the ith source vertex. To keep track of visited vertices, we maintain a tall-and-skinny sparse matrix $\mathbf{S}$ where the ith column represents all vertices visited from the ith source vertex. At first, we build $\mathbf{F}$ from the set of source vertices such that each column of $\mathbf{F}$ has just one nonzero (Line 2, Alg. 3). We continue expanding frontiers until $\mathbf{F}$ is empty. In each iteration, we discover the vertices reachable from the current frontier through the multiplication of the adjacency matrix by the frontier matrix with $(\wedge, \vee)$ semiring (Line 6, Alg. 3). After each multiplication, we remove already visited vertices from $\mathbf{F}$ and then update the set of visited vertices across all BFSs (Line 7 and 8, Alg. 3) The updated frontier $\mathbf{F}$ becomes the input to the next iteration.

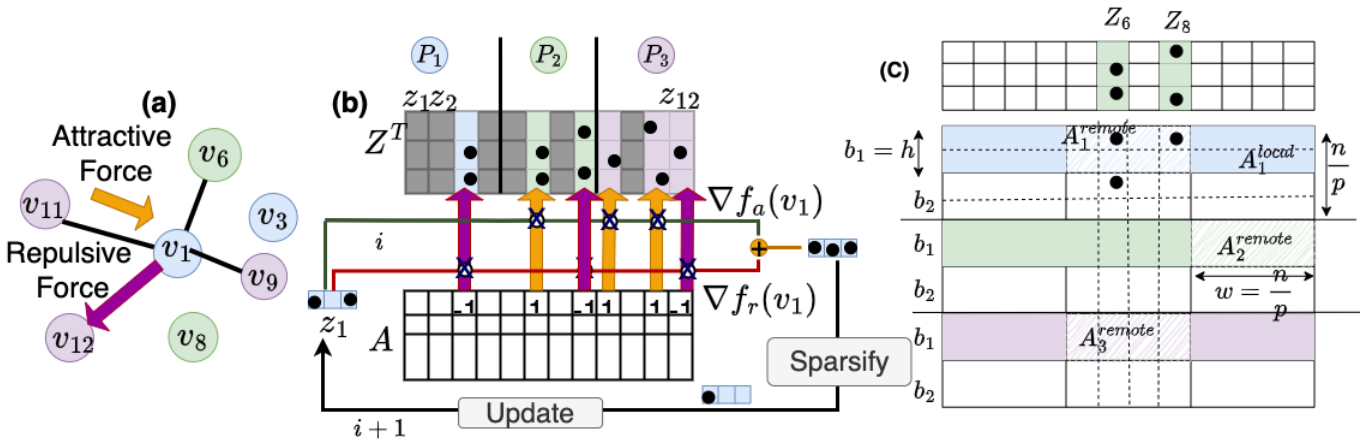

Fig. 4: (a) An illustration of the force-directed node embedding, where neighboring vertices generate attractive forces and non-neighboring vertices generate repulsive forces. (b) The matrix $\mathbf{A}$ represents the adjacency matrix of the graph and $\mathbf{Z}^\mathsf{T}$ represents the sparse embedding matrix. Then, force computations are mapped to a TS-SpGEMM operation. (c) The minibatch computation, where we set tile height to be equal to the batch size.

As the sparsity of matrix $\mathbf{F}$ changes significantly throughout iterations, this algorithm serves as an excellent testing ground for TS-SpGEMM.

B. Distributed sparse embedding

We consider a node embedding problem where each vertex in a graph is embedded in a d-dimensional vector space. Typically, the embedding matrix $\mathbf{Z} \in \mathbb{R}^{n \times d}$ is a dense matrix, but it can be sparsified without compromising the quality of embedding. Hence, in sparse embedding, the embedding matrix $\mathbf{Z} \in \mathbb{R}^{n \times d}$ takes the form of a tall-and-skinny sparse matrix, presenting an interesting application for our TS-SpGEMM algorithm.

We aim to implement a force-directed node embedding algorithm called Force2Vec [21] that uses attractive and repulsive forces among vertices to compute embeddings. We specifically choose Force2Vec to demonstrate TS-SpGEMM due to its existing implementation with SpMM [47]. By inducing sparsity in the embedding matrix, we develop a sparse variant of Force2Vec using distributed TS-SpGEMM. Figure 4 shows a sample graph (left figure) and matrix representation with embedding computation (right figure). We use synchronous SGD with negative samples to compute the embedding. The matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ represents the adjacency matrix of the graph $G(V, E)$, where 1 indicates the neighbor vertices and -1 indicates the negative sampled non-neighbor vertices. The embedding matrices $\mathbf{Z} \in \mathbb{R}^{n \times d}$, and $\mathbf{Z^T} \in \mathbb{R}^{d \times n}$ represents the tall-and-skinny embedding matrix and it's transpose, respectively. Both $\mathbf{A}$, $\mathbf{Z}$, and $\mathbf{Z^T}$ are 1-D partitioned and stored in each process in CSR format.

As depicted in Figure 4, each embedding vector in a minibatch is updated parallelly by calculating the attractive force gradient $\Delta f_a(v_i)$, and repulsive force gradient $\Delta f_r(v_i)$ through a sequence of SpGEMM operations of TS-SpGEMM and update each embedding vector using SGD. Afterward, the updated embedding matrix is sparsified by selecting the required number of nonzero entries to achieve the target sparsity by keeping the highest valued entries. This sparsified output is used as the input to the next iteration.

TABLE IV: Default parameters used in our experiments.

Parameter	Value
Number of OpenMP threads per process	16
Number of processes per node	8
Number of processes for application testing	64
Dimension of $\mathbf{B}$ matrix (d)	128
Height of a tile (h)	$\frac{n}{p}$
Width of a tile (w)	$16 \times \frac{n}{p}$
Default sparsity of $\mathbf{B}$	80%
Embedding mini-batch size (b)	256
Embedding learning rate	0.02

We set the batch size to the height of a tile, that enables minibatch SpGEMM for TS-SpGEMM . Reducing the height of a tile increases the communication volume. For instance, in the sub-figure (c) of Figure 4, the embedding vector Z_6 needs to be fetched twice in batches b_1, and b_2. But, if A_1^{remote} is computed remotely on P_2, the Z_6 only needs to be fetched once while computing b_2. Hence, the remote computations can reduce communication overhead that is incurred in the minibatch scenarios with tiling.

V. Results

A. Experimental setup

Table IV shows the default parameters used in the experiments. We identified these default parameters via extensive benchmarking. Users can use default parameters to obtain good performance. In particular, we observed that a tile width of $(16 \times n/p)$ and a tile height of n/p perform the best for most matrices. We use these default parameters in all experiments unless otherwise stated. Runtimes were reported as the average of five runs.

Experimental platforms. We evaluate the performance of our algorithm on the CPU partitions of the NERSC Perlmutter

TABLE V: Datasets used in our experiments.

Dataset	Alias	# Vertices	# Edges	Avg Degree
pubMed	pubmed	19,717	44,338	4.49
flicker	flicker	89,250	899,756	20.16
cora	cora	2708	5429	2
citeseer	citeseer	3312	4732	1.4
arabic-2005	arabic	22,744,080	639,999,458	28.1
it-2004	it	41,291,594	1,150,725,436	27.8
GAP-web	gap	50,636,151	1,930,292,948	38.1
uk-2002	uk	18,520,486	298,113,762	16.0
Erdős-Rényi	ER	40000000	320000000	8

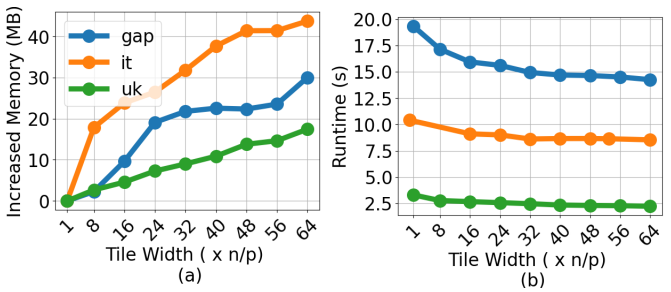

Fig. 5: The impact of an increasing tile width on memory and runtime on 8 nodes (64 processes). The x-axis shows the width of the tile ranging from n/p to n, expressed as multiples of n/p. The left subfigure shows the increase in memory consumption, while the right subfigure shows the impact on the runtime.

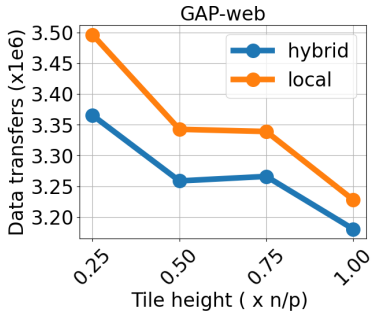

Fig. 6: The reduction in data transfers for hybrid mode and local mode. The hybrid mode enables both local and remote tiles, whereas the local mode only uses local tiles. We ran the experiments on 8 nodes for GAP-web.

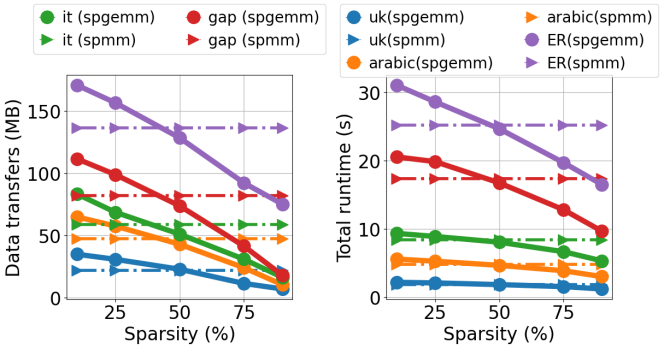

Fig. 7: The Figure shows the comparison results of the TS-SpGEMM and SpMM version. The left sub-figure shows the communication volume and the right sub-figure shows the runtime for different sparsity levels.

supercomputer. A single compute node of the Perlmutter CPU partition is equipped with two AMD EPYC 7763 CPUs with 64 cores and 512GB of memory. To compile the programs, SUSE Linux g++ compiler 12.3.0 is used with -O3 option. We used MPI+OpenMP hybrid parallelization for all experiments. For in-node multithreading, we experimented with various settings and found that 16 threads per process gave the best performance Unless otherwise stated, we used 8 MPI processes per node and 16 OpenMP threads per process. For MPI implementation, we used Cray-MPICH-8.1.28.

Datasets. Table.V describes the graphs used in our experiments. We collected these graphs from SNAP [48] and the Suitesparse Matrix Colelction [49]. Additionally, we generate uniformly random tall-and-skinny matrices for our experiments. In our experiments, the tall-and-skinny matrix $\mathbf{B}$ with $s\%$ sparsity means $s\%$ entries in each row of the $\mathbf{B}$ are zero.

Baselines. We compare our implementation with state-of-the-art algorithms such as 2-D Sparse SUMMA [14], 3-D Sparse SUMMA [50], and 1-D algorithm in PETSc [17]. We use 2-D and 3-D implementation available in CombBLAS-2.0 [5] and 1-D implementation available in PETSc-3.19.3.

B. Impacts of the tile width

To determine the optimal tile size, we ran experiments on 8 nodes (64 MPI processes) with three datasets. We use the maximum height $h = n/p$ and vary the width of a tile w from n/p to n. The results are shown in Figure 5, where tile widths are shown as multiples of n/p. Figure 5(a) shows that the memory consumption increases monotonically with the increase of tile width. This behavior is expected since a higher value of w requires fetching a larger fraction of $\mathbf{B}$ into the local process. Hence, to reduce memory consumption, we should use smaller tile widths. However, as shown in Figure 5(b), a small tile width results in longer runtimes due to increased communication rounds. Based on the observations from Fig. 5, we determine that $16\times n/p$ represents an optimal tile width for achieving the fastest performance with a manageable memory overhead. Hence, we use $16\times n/p$ as the default tile width for our experiments. Furthermore, we tested the data transfer cost for different tile heights, by fixing the tile width to $16\times n/p$. Figure 6 shows that in the hybrid mode where both remote and local computations are enabled, TS-SpGEMM reduces the data transfer cost compared to the pure local mode. A small tile height is useful to capture the computations in the sparse embedding application.

C. SpGEMM vs SpMM

When the tall-and-skinny input matrix ($\mathbf{B}$) is sufficiently dense, running an SpMM may run faster than TS-SpGEMM. To determine the sparsity threshold of $\mathbf{B}$ at which TS-SpGEMM begins to outperform SpMM, we implemented an SpMM with a dense $\mathbf{B}$ using the same communication patterns as TS-SpGEMM. We confirmed that our SpMM performs comparably or better than the 1.5D dense shifting algorithm [51, 52]. We ran this experiment on 32 nodes

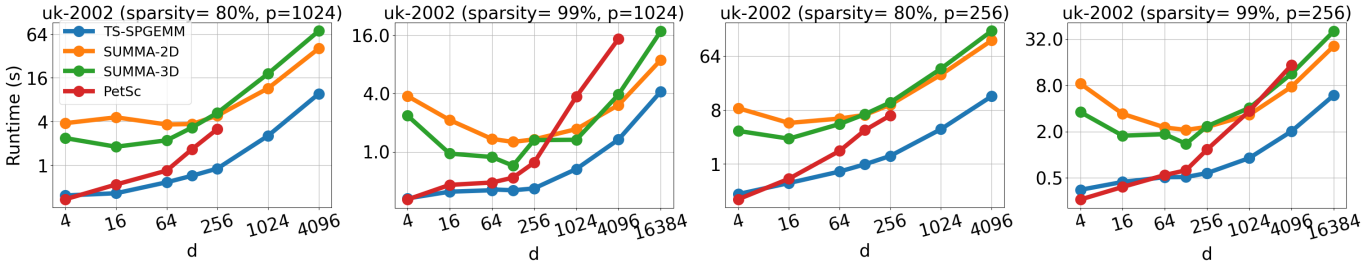

Fig. 8: This figure shows the runtime comparison of the TS-SpGEMM , 2-D SUMMA, 3-D SUMMA, and PETSc (1-D) for *uk-2002* dataset under different dimensions for sparsity 80%, and 99% on 128 nodes ($p = 1024$) and 32 nodes ($p = 256$). For 80% sparsity, we encountered out-of-memory issues with the PETSc binary converter, preventing us from converting inputs to binary format for dimensions $d \geq 256$.

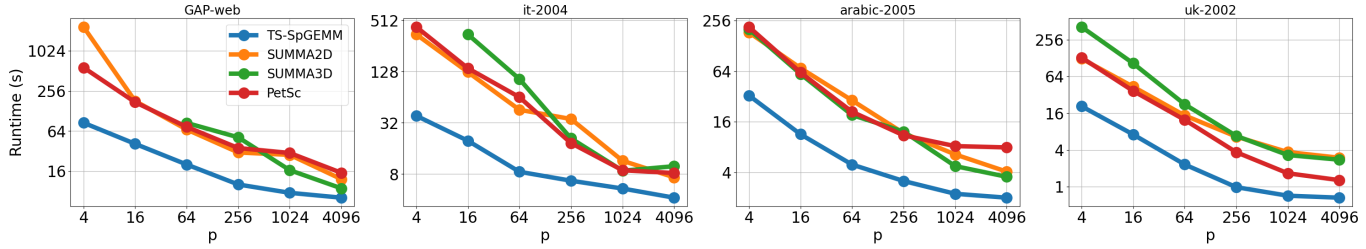

Fig. 9: The figures show the strong scaling runtime for TS-SpGEMM 2-D SUMMA, 3-D SUMMA, and PetSc for *GAP-web*, *it-2004*, *arabic-2005* and *uk-2002* datasets. We ran this experiment for $\mathbf{B}$ with 128 dimensions and 80% sparsity.

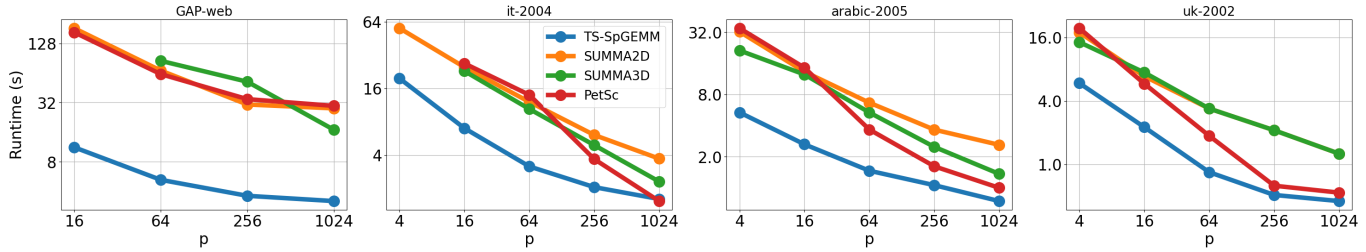

Fig. 10: The figures show the strong scaling runtime for TS-SpGEMM , 2-D SUMMA, 3-D SUMMA, and PetSc for *GAP-web*, *it-2004*, *arabic-2005*, and *uk-2002* datasets. We ran this experiment for $\mathbf{B}$ matrix with 128 dimensions and 99% sparsity.

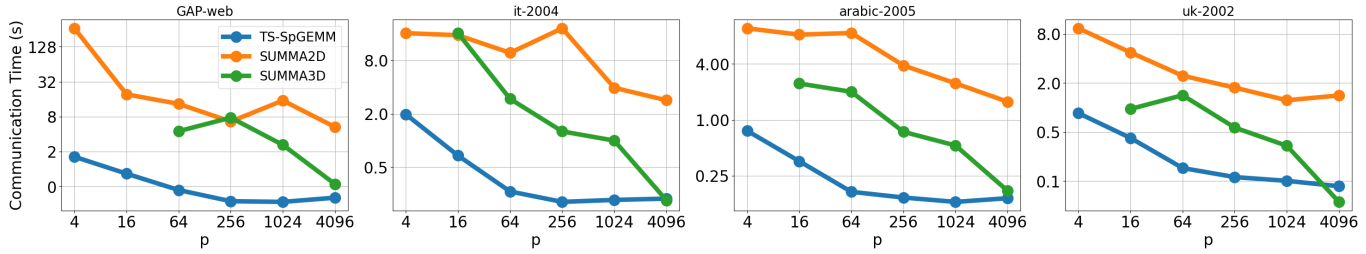

Fig. 11: The figures show the strong scaling communication time for TS-SpGEMM , 2-D SUMMA, 3-D SUMMA, and PetSc for *GAP-web*, *it-2004*, *arabic-2005*, and *uk-2002* datasets. We ran this experiment for $\mathbf{B}$ matrix with 128 dimensions and 80% sparsity.

with 256 MPI processes and show the results in Figure 7. For all datasets with sparsity exceeding 50%, TS-SpGEMM communicates less data and runs faster than SpMM. This sparsity threshold is justified by considering that TS-SpGEMM requires communication of both indices and values, whereas SpMM only communicates values. The reduction in communication may not translate into a proportional reduction of runtime because local SpGEMM computation is more costly than SpMM. Furthermore, the total runtime of SpGEMM is higher up to 50% sparsity compared to SpMM due to additional overheads involved with random memory access and computation. We recommend using TS-SpGEMM only for applications where the second matrix is at least 50% sparse.

D. Comparison with other distributed SpGEMM

We compared TS-SpGEMM with state-of-the-art implementations: PETSc (1-D algorithm), 2-D Sparse SUMMA, and 3-D Sparse SUMMA. We ran the experiments by varying

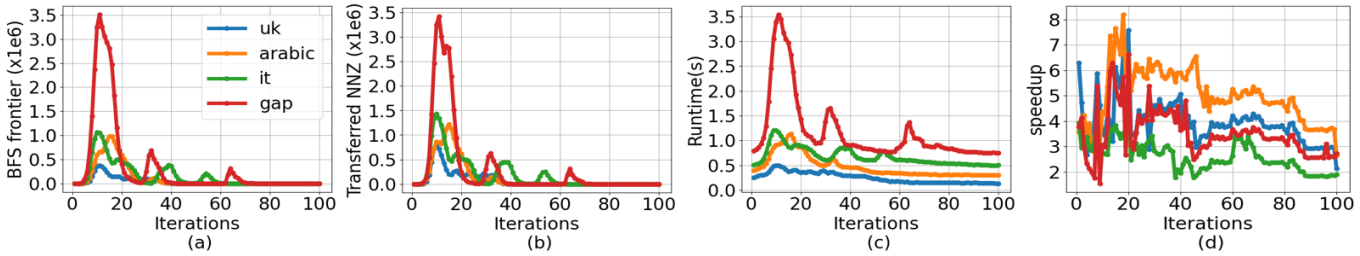

Fig. 12: The multi-source BFS implementation using TS-SpGEMM ran on 8 nodes (64 MPI processes). The sub-figure (a) shows the average BFS frontier. The sub-figure (b) shows the average communicated nnz. The sub-figure (c) shows the runtime for each iteration. The sub-figure (d) shows the speedup with respect to 2-D SUMMA on each iteration.

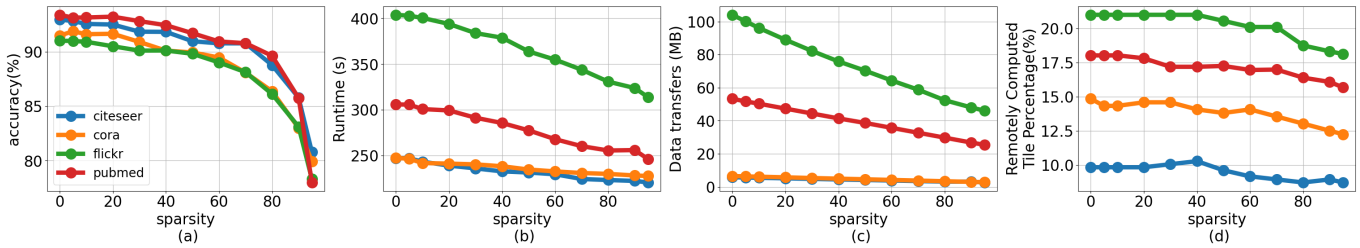

Fig. 13: The performance of the sparse embedding algorithm on 8 nodes (64 MPI processes). (a) The link-prediction accuracy for *citeseer*, *cora*, *flicker*, and *pubmed* with varying sparsity in the embedding matrix. (b) Total runtime with varying sparsity. (c) The communicated volume. (d) The percentage of remotely computed tiles.

d under 80% and 99% sparsity in 32 nodes and 128 nodes. The results are shown in the Figure 8. Our algorithm outperforms the others when varying the dimension of the second matrix $\mathbf{B}$ from 4 to $16,384$. According to the results, PETSc and TS-SpGEMM have similar runtimes at d=4, but PETSc SpGEMM performance drops noticeably at around d=64. This is because, at d=4 with 80% sparsity, it is feasible to store the entire $\mathbf{B}$ on a single process. Hence, there is no significant benefit of tiling when d is very small. By contrast, SUMMA-2D and SUMMA-3D do not perform well when $d < 256$, but their runtimes appear to be more competitive at higher d. The behavior of SUMMA-2D and SUMMA-3D is also predictable because these algorithms involve communication for both $\mathbf{A}$ and $\mathbf{B}$. In cases where d is small and $\mathbf{B}$ is sparse, it is advantageous to communicate only $\mathbf{B}$, as our algorithm and PETSc do. A more interesting result is TS-SpGEMM's superior performance at $d = 4,096$ and $d = 16,384$ where 1D SpGEMM in PETSc performs poorly. This improved performance of TS-SpGEMM is attributed to tiling and other optimizations discussed in the method section. Thus, Figure 8 demonstrates that TS-SpGEMM outperforms state-of-the-art algorithms when $\mathbf{B}$ is a tall-and-skinny matrix.

E. Scalability

We ran scalability tests ranging from 1 node to 512 nodes for sparsity levels of 80% and 99%. The results are shown in Figures 9 and 10 respectively. TS-SpGEMM scales up to 512 nodes (4096 MPI processes; 65,536 cores) and outperforms other SpGEMM implementations. Runtime scales almost linearly until 1024 processes for both sparsity levels. Past this point, performance scaling has been reduced due to workload reduction. Figure 11 shows the results for communication scalability for 80% sparsity. TS-SpGEMM's communication scales up to 1024 MPI processes, after which latency begins to dominate. We did not plot the PETSc communication overhead as it does not report the communication time separately. Since SUMMA3D is a communication-avoiding algorithm, its communication scales much better than other algorithms. SUMMA3D communication can even beat TS-SpGEMM at 512 nodes by utilizing more layers. Still, applications relevant to TS-SpGEMM do not typically need more than 128 nodes to run, as the second matrix stays as tall and skinny.

F. Multi-source BFS

The multi-source BFS is the first application we implemented using TS-SpGEMM . We test the application using 8 nodes (64 MPI processes). We use four datasets to evaluate the BFS: *uk-2002*, *arabic-2005*, *it-2004*, and *GAP-web*. We consider 128 sources randomly selected as the starting nodes. Hence, $\mathbf{B}$ is of size $n \times 128$ and initially contains one randomly chosen non-zero per column. The evaluation results are given in the Figure 12. In the BFS implementation, we sparsify the output of each iteration such that it only contains the newly visited vertices and feed that output as input to the next iteration. We can see that the BFS frontier becomes denser only in a few iterations but remains sparse for the rest of the iterations. If there are multiple connected components, it is possible to have several peaks in the BFS frontier for the same dataset. If sparsity is greater than 50%, we use SpGEMM for the computation, otherwise we can utilize the SpMM version of the TS-SpGEMM. The communication and runtime closely follow the BFS frontier. Figure 12(d) shows the speedup with

respect to multi-source BFS implemented with 2-D SUMMA in CombBLAS. We can achieve up to a 10x speedup in some iterations and around a 5x speedup on average.

G. Sparse Embedding

As our final application, we implemented a sparse embedding algorithm discussed in Section IV-B. For improved accuracy, we use mini-batch SpGEMM where we set a batch size of $b = 0.5 \times \frac{n}{p}$. The tile height matches the batch size in the minibatch setting. We ran all the experiments on 8 nodes (64 MPI processes) and calculated the link prediction accuracy as given by the Force2Vec embedding algorithm [21]. Figure 13 shows that we can make the embedding 80% sparse by sacrificing less than 5% accuracy in link prediction. The runtime and data transfer plots reveal that we can achieve faster convergence and lesser communication overhead with increasing sparsity. Furthermore, sub-figure 13(d) shows that remote tiles play important roles in the minibatch setting.

VI. Conclusion

Popular distributed algorithms deliver suboptimal performance for SpGEMM settings where one matrix is square and the other is tall and skinny—a variant we call TS-SpGEMM. To address this limitation, we developed a novel distributed-memory algorithm for TS-SpGEMM that employs customized 1D partitioning for all matrices and leverages sparsity-aware tiling for efficient data transfers. At lower to moderate node counts (up to 128 nodes), TS-SpGEMM shows superior performance compared to 1D, 2D, and 3D SpGEMM algorithms. This trend persists even at 512 nodes, highlighting the effectiveness of our optimizations. Further, we use our algorithm to implement multi-source BFS and sparse graph embedding algorithms and demonstrate their scalability up to 512 Nodes on NERSC Perlmutter.

One limitation of our TS-SPGEMM algorithm is that it requires storing two copies of the first input matrix, $\mathbf{A}$, which increases the overall memory usage. However, most communication-avoiding algorithms, including SUMMA3D, use additional memory to reduce communication overhead. We believe that most graph and sparse matrix algorithms can accommodate an extra copy of $\mathbf{A}$ in a distributed-memory system. Another limitation of our algorithm is the use of 1D matrix partitioning, which can lead to load imbalances in scale-free graphs that typically have denser rows. While the memory imbalance is inherent to 1D partitioning, we addressed the computational imbalance using virtual 2D partitioning, which performs multiplication tile by tile. Nevertheless, the memory imbalance associated with input matrices can still pose challenges for scale-free graphs.

The optimizations used in TS-SpGEMM can be adapted to distributed SpMM [51, 52] and fused matrix multiplication [53] algorithms. We observed that while TS-SpGEMM is not faster than SpMM when matrix $\mathbf{B}$ is fully dense, it outperforms SpMM when $\mathbf{B}$ is 50% or more sparse. Additionally, TS-SpGEMM is not the optimal choice when $\mathbf{B}$ closely resembles $\mathbf{A}$ in shape and sparsity; however, it still outperforms SUMMA when multiplying a sparse matrix by another sparse matrix that is not tall and skinny.

VII. Acknowledgments

This research was funded in part by DOE grants DE-SC0022098 and DE-SC0023349; by NSF grants PPoSS CCF 2316233 and OAC-2339607; by SRC JUMP 2.0 ACE Center; and by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 21-46756.

References

[1] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[2] 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.

[3] 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.

[4] 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.

[5] A. Azad, O. Selvitopi, M. T. Hussain, J. R. Gilbert, and A. Buluç, "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 989–1001, 2021.

[6] 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.

[7] 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.

[8] 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 *IPDPS*. IEEE, 2020.

[9] E. Hassani, M. T. Hussain, and A. Azad, "Parallel algorithms for computing jaccard weights on graphs using linear algebra," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–7.

[10] O. Selvitopi, S. Ekanayake, G. Guidi, G. Pavlopoulos, A. Azad, and A. Buluç, "Distributed many-to-many protein sequence alignment using sparse matrices," in *SC*, 2020.

[11] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, "The more the merrier: Efficient multi-source graph traversal," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 449–460, 2014.

[12] M. Minutoli, M. Halappanavar, A. Kalyanaraman, A. Sathanur, R. Mcclure, and J. McDermott, "Fast and scalable implementations of influence maximization algorithms," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[13] M. K. Rahman and A. Azad, "Triple sparsification of graph convolutional networks without sacrificing the accuracy," *arXiv preprint arXiv:2208.03559*, 2022.

[14] 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.

[15] 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.

[16] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, 2005.

[17] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.5, 2014. [Online]. Available: http://www.mcs.anl.gov/petsc

[18] 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.

[19] J. R. 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.

[20] 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 Computing*, vol. 90, p. 102545, 2019.

[21] M. K. Rahman, M. H. Sujon, and A. Azad, "Force2vec: Parallel force-directed graph embedding," in *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 442–451.

[22] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[23] 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 *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

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

[25] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[26] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–11.

[27] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[28] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, p. 25, 2015.

[29] J. Liu, X. He, W. Liu, and G. Tan, "Register-aware optimizations for parallel sparse matrix–matrix multiplication," *International Journal of Parallel Programming*, vol. 47, no. 3, pp. 403–417, 2019.

[30] 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.

[31] A. Buluç and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *The 37th International Conference on Parallel Processing (ICPP'08)*, 2008, pp. 503–510.

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

[33] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, pp. 496 – 509, 2011.

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

[35] 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.

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

[37] 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 *PASC*, 2017, pp. 1–9.

[38] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, "Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale," in *IPDPS*. IEEE, 2021.

[39] E. Solomonik and T. Hoefler, "Sparse tensor algebra as a parallel programming model," *arXiv preprint arXiv:1512.00066*, 2015.

[40] M. Rasouli, R. M. Kirby, and H. Sundar, "A compressed, divide and conquer algorithm for scalable distributed matrix-matrix multiplication," in *The International Conference on High Performance Computing in Asia-Pacific Region*, 2021, pp. 110–119.

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

[42] M. T. Hussain, G. S. Abhishek, A. Buluç, and A. Azad, "Parallel algorithms for adding a collection of sparse matrices," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 285–294.

[43] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[44] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Proceedings of the IPDPS*. IEEE, 2017.

[45] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *IEEE HPEC*, 2016.

[46] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.

[47] I. Ranawaka and A. Azad, "Scalable node embedding algorithms using distributed sparse matrix operations," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 1199–1201.

[48] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[49] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw. (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[50] 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.

[51] 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 *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 431–442.

[52] C. Block, G. Gerogiannis, C. Mendis, A. Azad, and J. Torrellas, "Two-face: Combining collective and one-sided communication for efficient distributed spmm," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1200–1217.

[53] M. K. Rahman, M. H. Sujon, and A. Azad, "FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266.