Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication

Oguz Selvitopi
Lawrence Berkeley Nat. Laboratory
Berkeley, CA, USA
roselvitopi@lbl.gov

Benjamin Brock
University of California, Berkeley
Berkeley, CA, USA
brock@berkeley.edu

Israt Nisa
Lawrence Berkeley Nat. Laboratory
Berkeley, CA, USA
isratnisa@lbl.gov

Alok Tripathy
University of California, Berkeley
Berkeley, CA, USA
alokt@berkeley.edu

Katherine Yelick
University of California, Berkeley
Berkeley, CA, USA
yelick@berkeley.edu

Aydın Buluç
Lawrence Berkeley Nat. Laboratory
Berkeley, CA, USA
abuluc@lbl.gov

ABSTRACT

Sparse times dense matrix multiplication (SpMM) finds its applications in well-established fields such as computational linear algebra as well as emerging fields such as graph neural networks. In this study, we evaluate the performance of various techniques for performing SpMM as a distributed computation across many nodes by focusing on GPU accelerators. We examine how the actual local computational performance of state-of-the-art SpMM implementations affect computational efficiency as dimensions change when we scale to large numbers of nodes, which proves to be an unexpectedly important bottleneck. We consider various distribution strategies, including **A**-Stationary, **B**-Stationary, and **C**-Stationary algorithms, 1.5D and 2D algorithms, and RDMA-based and bulk synchronous methods of data transfer. Our results show that the best choice of algorithm and implementation technique depends not only on the cost of communication for particular matrix sizes and dimensions, but also on the performance of local SpMM operations. Our evaluations reveal that with the involvement of GPU accelerators, the best design choices for SpMM differ from the conventional algorithms that are known to perform well for dense matrix-matrix or sparse matrix-sparse matrix multiplies.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

KEYWORDS

Sparse linear algebra, Sparse matrices, Graphics accelerators, Parallel algorithms, RDMA

ACM Reference Format:
Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2021. Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication . In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447818.3461472

1 INTRODUCTION

Multiplying a sparse matrix with a tall-skinny dense matrix (SpMM) has traditionally found uses in scientific computing. The SpMM operation, also known as sparse matrix times multiple dense vector multiplication, has long been the workhorse of block iterative solvers such as LOBPCG [21]. These block iterative methods have increasingly gained popularity following the trends in computer architectures.

Within the last decade, low-rank matrix factorization methods such as alternating least-squares (ALS) made SpMM even more popular. However, the primary revolution that made research into SpMM explode in the last few years is the success of Graph Neural Networks (GNN) in a variety of supervised and semi-supervised learning scenarios. As shown in recent publications, SpMM is the workhorse of training GNNs [18, 19, 30]. In addition, several groups are also uncovering the potential of exploiting sparsity in convolutional and recurrent deep neural networks (DNNs) [14].

Thanks to the need to accelerate GNNs and sparse DNNs, an impressive number of new GPU [15, 32, 33] and CPU [4, 23] implementations of SpMM have recently surfaced. However, GNNs and sparse DNNs are often trained on distributed-memory clusters and the primary bottleneck in SpMM performance at large scale is the communication costs. There has been little or no work that is directly targeting the SpMM operation on distributed memory.

Many known algorithms from dense matrix multiplication [27, 28] as well as sparse-sparse matrix multiplication [5, 9] and sparse-dense matrix multiplication [22] are applicable to SpMM. However, the tradeoffs and constraints are vastly different for SpMM compared to these other problems. We discuss these differences in detail in Section 3. Algorithmic papers targeting higher level problems such as non-negative matrix factorization [20] and graph neural network training [30] implemented and studied their own subset of parallel SpMM algorithms.

In this paper, we consider a much larger family of distributed-memory parallel algorithms for SpMM on GPU and CPU equipped clusters. At a high-level, we consider both **A**-Stationary and **C**-Stationary algorithms [27]. We also dive deeper on the performance scaling of the local SpMM computation that often do not strong scale. Doing so, we uncover previously unknown tradeoffs between 1.5D and 2D algorithms. In particular, 1.5D achieves much better scaling in local SpMM computations because (1) our 1.5D algorithm formulation not cut the column dimension of the tall-skinny dense matrices, and (2) each process only performs one local SpMM as opposed to $\sqrt{p}$, creating a batching effect. Finally, we also study the impact of synchronization costs by studying two different communication backends with different synchronization semantics: bulk-synchronous with MPI and asynchronous NVSHMEM with BCL (Berkeley Container Library) [7].

Overall, distributed-memory SpMM algorithms provide a large three-dimensional design space even at the coarse level: based on data decomposition, based on which matrix or matrices to keep stationary, and based on the communication semantics. Here we present algorithms, analyses, and implementations for 5 of them in depth, which collectively cover all dimensions of the design space.

Our results show that, unlike for dense matrix-dense matrix (GEMM) and sparse matrix-sparse matrix multiplication (SpGEMM), 2D algorithms are not strongly scalable to large process counts for the SpMM problem. While the one-sided asynchronous 2D algorithm is faster than the bulk-synchronous 2D algorithm, neither scale particularly well. By contrast, the bulk-synchronous 1.5D algorithm strong scales well on all but one input. However, the overheads of bulk-synchronous implementation cause both 1.5D and 2D algorithms to run slower than the asynchronous 2D implementation that is much closer to bare metal.

The rest of the paper is organized as follows. Section 2 presents the necessary background and Section 3 surveys the related work. Section 4 presents the parallel SpMM algorithms. Section 5 describes two different methods to implement the presented algorithms and analyzes these implementations' various costs. Section 6 evaluates five different SpMM algorithms. Section 7 summarizes the findings of our work under the light of the discussions in Section 6. Section 8 concludes.

2 BACKGROUND

The SpMM of form $\mathbf{C} = \mathbf{AB}$ involves one sparse matrix $\mathbf{A}$ of size $m \times n$ and two dense matrices $\mathbf{B}$ and $\mathbf{C}$ of sizes $n \times k$ and $m \times k$, respectively. We assume the dense matrices are tall and skinny, i.e., $m, n \gg k$. For our analyses, we assume a uniform distribution of nonzeros in $\mathbf{A}$ where there are d_r and d_c nonzeros per row and column, respectively. We use the function $\text{nnz}(\cdot)$ to denote the number of nonzeros in a sparse (sub)matrix.

We assume p processes participate in computing SpMM in parallel. These processes may be organized into a 1D or 2D structure depending on the algorithm. For the former, we use $P(i)$ to denote the process at location i and for the latter, we assume that processes are organized into an $r \times c = p$ structure and we use $P(i, j)$ to indicate a process at the corresponding location. The set of processes at row i and column j are respectively indicated with $P(i, :)$ and $P(:, j)$. This notation extends to the matrices in the SpMM, where the submatrix $\mathbf{M}(i)$ or $\mathbf{M}(i, j)$ is associated with process $P(i)$ or $P(i, j)$.

The collective communications play an important role in our analysis. For communication analysis, we assume the cost of sending a message of size w from a process to another is given by $\alpha + \beta w$, where α is the message startup time and β is the per-word transfer time. There are four particular collective communication operations that are of interest to our work: `MPI_Bcast`, `MPI_Reduce`, `MPI_Allgather`, and `MPI_Reduce_scatter`. For the costs mentioned below, we do not consider computational costs that are required for collectives involving a reduction. The `MPI_Bcast` collective broadcasts w words to all processes in a communicator. For this operation, we assume a tree algorithm, which has $O(\alpha \log p + \beta w \log p)$ cost. The `MPI_Reduce` collective reduces w words from all processes at a single process. We also assume a tree algorithm for this operation, which has $O(\alpha \log p + \beta w \log p)$ cost. The `MPI_Allgather` collective gathers at each process w words from all processes. For this collective, we assume a recursive doubling algorithm, which has $O(\alpha \log p + \beta w p)$ cost. `MPI_Reduce_scatter` collective w words from each process and scatters w/p words to each process. We assume a recursive halving algorithm for this collective, which has $O(\alpha \log p + \beta w)$ cost. For details of these algorithms, refer to the survey by Chan et al. [12].

3 RELATED WORK

Algorithms for matrix-matrix multiplication is probably one of the most well-studied problems in parallel computing. The majority of this work is for the dense-dense case (GEMM). Commonly used algorithms, which are both 2D, are Cannon's algorithm [11] and the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [31]. Communication-avoiding algorithms trade memory for reduced communication, and are often called 3D [2] or 2.5D [28] algorithms depending on the relative length of the third processor dimension to the first two processor dimensions. Recent work focused on making the distributed GEMM algorithms perform optimally when process grids or input matrices are rectangular [13, 24].

Work on the sparse-sparse (SpGEMM) case include 2D [9] and 3D [5] algorithms, both of which are $\mathbf{C}$-stationary. The classification of algorithms based on which matrix or matrices communication is due to Schatz et al. [27].

Parallel GEMM benefits from the so-called surface-to-volume ratio, which means that the flops scale with n^3 for multiplying two $n \times n$ matrices whereas the data size, which primarily determines the communication costs, scales with n^2. Parallel SpGEMM and SpMM do not benefit from this favorable computation-to-data scaling. Somewhat counter-intuitively, it is easier to scale SpGEMM than it is to scale SpMM to large concurrencies. To understand why, consider an $n \times n$ sparse matrix $\mathbf{A}$ that has k nonzeros on average per row, and an $n \times k$ dense matrix $\mathbf{B}$. Computing both $\mathbf{AA}$ and $\mathbf{AB}$ take the same sized input data (in terms of bytes), hence incur similar communication costs assuming the use of the same output stationary algorithm (more formally defined in Section 4.2). However, SpMM attains a significantly higher fraction of peak on local compute nodes, hence completes its local computation steps faster. For example, the reported performance rate for SpMM on the NVIDIA P100 GPU range in 100−500 GFlops [17], yet SpGEMM can only achieve 1−10 GFlops on the same GPU [25]. Consequently, SpGEMM more effectively hides its communication costs with local computation, compared to SpMM.

In terms sparse-dense multiplication, Koanantakool [22] analyzed the communication costs of various distributed-memory algorithms including three 1.5D variants that have a higher memory footprint than 2D variants but reduce the latency by bulking up computation. The focus of Koanantakool work was multiplying two matrices where both the theory and common sense suggest keeping the significantly larger (in terms of number of elements) dense matrices stationary and only moving the sparse matrix around. Our 1.5D algorithm presented in this paper is of the same spirit as Koanantakool's in the sense that it has an asymptotically higher maximum memory footprint and reduces the numbers of stages and latency costs proportionally. However, our algorithm uses different data distribution. Most importantly, in our 1.5D algorithm the sparse matrix **A** is stationary and hence it moves two dense matrices: the input dense matrix as well as the output dense matrix. By contrast, the 1.5D algorithms presented by Koanantakool only move the input sparse matrix. In that sense, our 1.5D algorithm is closer in spirit to 2D algorithms where as Koanantakool's 1.5D algorithm is an extension of 1D algorithms.

Albeit indirectly, sparse-dense matrix multiplication has also been the subject of a work for general-purpose distributed platforms (i.e., Apache Spark). In this work, Park and Lee [26] consider SpGEMM, but they convert one of the sparse matrices into dense for performance reasons. Therefore, the resulting operation becomes sparse-dense matrix multiplication. They consider different sparsity levels for the tall-skinny matrices and these matrices can be treated as sparse or dense according to the algorithm being used. The work of Park and Lee focuses on predicting the performance of SpGEMM (or sparse-dense matrix multiplies in certain cases) on general purpose distributed computing platforms. They do not investigate parallelism and scalability aspects, which are central to our work. Our work also largely differs from that of Park and Lee with its HPC setting, where we use one of the faster supercomputers on earth, with GPU accelerators on each node.

Various distributed-memory parallel sparse matrix times tall-skinny dense matrix (SpMM) algorithms are buried in application papers. In particular, MPI_FAUN [20] implements the 1.5D algorithm we use in our paper and CAGNET [30] implements the bulk-synchronous 2D algorithm presented here. Neither work study the impacts of partitioning strategy in local computation costs as we do here. Furthermore, neither MPI_FAUN nor CAGNET implement communication using one-sided operations, something we show to outperform bulk-synchronous approaches in this paper.

In another work [1], the authors investigated tailored 1D sparse matrix partitioning models to improve the parallel performance of SpMM by focusing on its communication aspects. By arguing that SpMM's parallel communication performance is bandwidth-bound rather than being latency-bound, they propose a generic framework in which several important metrics related to communication volume can be encapsulated and optimized. Their approach relies on recursive bipartitioning to incrementally capture the communication volume information during the partitioning process and uses multiple vertex weights to formulate and optimize different volume objectives. Our work does not consider intelligent partitioning and uses a simple 2D partitioning of the sparse matrix.

4 PARALLEL ALGORITHMS

In this section we describe our parallel algorithms for SpMM. We assume a 2D partition of the sparse matrix (**A**) and we consider 1D and 2D partitions of the dense matrices (**B**, **C**). We only consider a 2D partition of the sparse matrix as it usually leads to better load balance than a 1D partition, which is not true for dense matrices. When dense matrices are tall and skinny, it is worth investigating the choice between 1D and 2D partitions of these matrices as this choice has important parallel performance implications related to both communication and computation. For 1D partition of dense matrices, we utilize replication with a fixed factor, which leads us to 1.5D algorithms. In the rest of this section, we assume a square process grid (i.e., $r = c = \sqrt{p}$) for 2D partitions.

Table 1: The combinations of the parallel algorithms for SpMM that follow from 2D partition of the sparse matrix and 1D or 2D partition of the dense matrices. The shaded cells indicate the algorithms that are described, analyzed, and evaluated in this work. The acronyms in each shaded cell show the communication methods used in the implementations of those variants (BS: bulk-synchronous, AS: asynchronous)

	1.5D	2D
A-Stationary	BS	BS, AS
B-Stationary		
C-Stationary		BS, AS

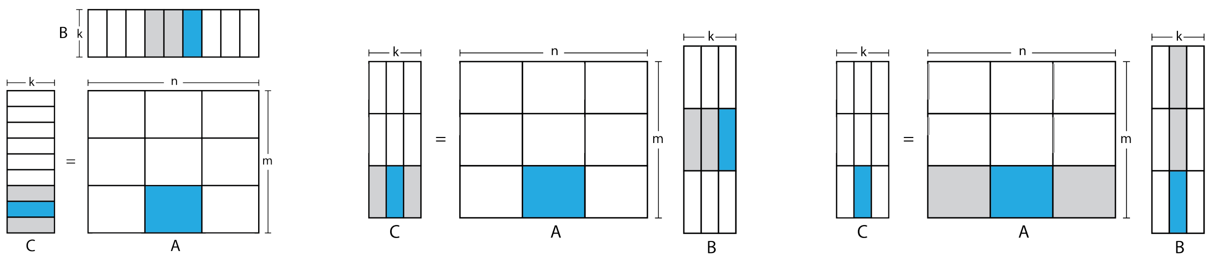
Table 1 shows the combinations of the SpMM algorithms under the above-described setting. Among these six combinations, we focus on three of them: 1.5D **A**-Stationary, 2D **A**-Stationary, and 2D **C**-Stationary. The **B**-Stationary algorithms have the same computational and communication characteristics with the **C**-Stationary algorithms when the sparse matrix is square, which is the case for the matrices evaluated in this work. Moreover, when the sparse matrix is rectangular, the choice of selecting **B**-Stationary or **C**-Stationary is trivial, which should be the algorithm that results in the communication of the smaller of the two dense matrices. We also do not analyze or evaluate 1.5D **C**-Stationary SpMM. This is because this algorithm necessitates each process to have the entire **B** or the entire **A** when **C** is rowwise or columnwise partitioned, respectively. This makes 1.5D **C**-Stationary SpMM prohibitively expensive, especially when one considers the communication overheads to more likely be a bottleneck in the presence of accelerators. This is not the case for the three evaluated algorithms in this work.

All algorithms have the same overall local computation complexity although they differ in the aspects of whether these computations are split across $\sqrt{p}$ stages or the matrices involved in the multiplication are the same in different stages, given that the SpMM algorithm runs in stages. The local multiplication is $O(\text{nnz}(\mathbf{A})/p \cdot k)$, or $O((nd_c)/p \cdot k)$.

All 1D partitions of dense matrices are rowwise, reasons of which will be clear later. We also briefly describe the memory requirements of the parallel algorithms as they can prove vital when relatively limited memory of the GPU accelerators is taken into account.

4.1 A-Stationary algorithms

We describe two **A**-Stationary algorithms for parallel SpMM. In **A**-Stationary algorithms, the sparse matrix remains in-place while the dense matrices are replicated or communicated. In the first variant of **A**-Stationary algorithms, we consider a 1D partition of dense matrices and replicate **B** among $\sqrt{p}$ processes. This results



(a) sA-1.5D **(b) sA-2D** **(c) sC-2D**

Figure 1: The submatrices $P(2,1)$ interacts in various parallel SpMM algorithms. The blue submatrices are originally stored by $P(2,1)$ and the gray submatrices are the submatrices which $P(2,1)$ participates in communicating/replicating.

in a 1.5D **A**-Stationary algorithm, which we refer to as **sA**-1.5D. In the second variant, we consider a 2D partition of dense matrices do not make use of replication. We refer to this variant as **sA**-2D.

4.1.1 1.5D **A**-*Stationary SpMM (***sA***-1.5D).* In **sA**-1.5D SpMM, **B** is originally partitioned in 1D and initially $P(i,j)$ has the submatrices $\mathbf{A}(i,j)$, $\mathbf{B}(j\sqrt{p}+i)$, and $\mathbf{C}(i\sqrt{p}+j)$. **B** is then replicated among $\sqrt{p}$ processes in each column of the process grid and after this operation $P(i,j)$ has $\sqrt{p}$ blocks of **B**, which are given by $\mathbf{B}(j\sqrt{p}+\ell)$ for $0 \le \ell < \sqrt{p}$. The processes then perform their local SpMM of form $\mathbf{C}^j(i\sqrt{p}+\ell) = \mathbf{A}(i,j)\mathbf{B}(j\sqrt{p}+\ell)$, for $0 \le \ell < \sqrt{p}$, and compute their partial results for the output matrix. Here, $\mathbf{C}^j$ denotes the partial dense matrix computed by the process at the corresponding jth column of the process grid. These partial dense matrices are then summed to get the final result matrix at each process with $\mathbf{C}(i\sqrt{p}+j) = \sum_\ell \mathbf{C}^\ell(i\sqrt{p}+j)$. The left of Fig. 1 illustrates the submatrices $P(2,1)$ stores and replicates/communicates in **sA**-1.5D algorithm for $p=9$ processes.

In terms of memory requirements, $P(i,j)$ needs to store its portion of the sparse matrix, replicated dense matrix **B** and the partial dense matrix. These dense matrices have $\sqrt{p}$ times the number of elements in the dense matrices $P(i,j)$ had at the beginning of the algorithm. The memory requirement of **sA**-1.5D SpMM at each process is therefore $\text{nnz}(\mathbf{A})/p + k(m+n)/\sqrt{p}$.

The local computations, which overall take $O((\text{nnz}(\mathbf{A})/p)\cdot k)$ time, are performed in one big stage.

4.1.2 2D **A**-*Stationary SpMM (***sA***-2D).* In **sA**-2D, all matrices are partitioned in 2D and $P(i,j)$ has the submatrices $\mathbf{A}(i,j)$, $\mathbf{B}(j,i)$, and $\mathbf{C}(i,j)$. The **sA**-2D proceeds in stages and at each stage the computations regarding $\mathbf{A}\,\mathbf{B}(:,j)$ are performed by all processes in parallel.

The **sA**-2D algorithm consists of $\sqrt{p}$ stages and each stage involves a local multiply sandwiched between two communication operations that involve input and output dense matrices. At stage s, $P(i,s)$ broadcasts its dense submatrix $\mathbf{B}(j,s)$ to $\sqrt{p}$ processes that are in the same column of the process grid. Then the processes perform their local SpMM of form $\mathbf{C}^j(i,s) = \mathbf{A}(i,j)\,\mathbf{B}(j,s)$, where $\mathbf{C}^j$ denotes partial dense submatrix computed by the process at the corresponding jth column of the process grid. These partial dense submatrices are then summed to get the final dense submatrices by $\mathbf{C}(i,s) = \sum_\ell \mathbf{C}^\ell(i,s)$ at processes that are in the sth column of the process grid. The middle of Fig. 1 illustrates the submatrices $P(2,1)$ stores and communicates in **sA**-2D algorithm for $p=9$ processes.

In terms of memory requirements, in addition to the submatrices initially stored by $P(i,j)$, it also needs store store the input dense submatrix it receives and the partial output dense submatrix it computes. The memory requirement of **sA**-2D SpMM at each process is therefore $(\text{nnz}(\mathbf{A}) + 2k(m+n))/p$.

The local computations, which overall take $O((\text{nnz}(\mathbf{A})/p)\cdot k)$ time, are split across $\sqrt{p}$ stages and each stage takes $O((\text{nnz}(\mathbf{A})/p)\cdot(k/\sqrt{p}))$ time. Among the two multiplied matrices at each stage, the sparse one does not change while the dense submatrix changes at each stage.

4.2 C-Stationary algorithms

We next describe a **C**-Stationary algorithm for parallel SpMM. In **C**-Stationary algorithms, the output dense matrix remains in place while the sparse matrix and the input dense matrix are replicated or communicated. We only consider a 2D algorithm, which we refer to as **sC**-2D. We do not describe the 1.5D variant as this requires the communication of the entire sparse matrix or the entire input dense matrix. For **sC**-2D SpMM, we do make use of replication.

4.2.1 2D **C**-*Stationary SpMM (***sC***-2D).* In **sC**-2D, all matrices are partitioned in 2D and $P(i,j)$ has the submatrices $\mathbf{A}(i,j)$, $\mathbf{B}(i,j)$, and $\mathbf{C}(i,j)$. There are several different algorithms [3, 31] to perform parallel SpMM with a 2D partition of all matrices. Here, we focus on the 2D variant of SUMMA algorithm [31]. The SUMMA algorithm for **sC**-2D distributes the computations regarding the outer product of $\mathbf{A}(:,j)$ and $\mathbf{B}(j,:)$ among all processes.

The **sC**-2D algorithm consists of $\sqrt{p}$ stages and each stage involves of successive broadcast operations and local multiplies. At stage s, $P(i,s)$ broadcasts its sparse submatrix $\mathbf{A}(i,s)$ to $\sqrt{p}$ processses that are in the same row of the process grid and $P(s,j)$ broadcasts its dense submatrix $\mathbf{B}(s,j)$ to $\sqrt{p}$ processses that are in the same column of the process grid. Each process then computes the partial local multiplies $\mathbf{C}^s(i,j) = \mathbf{A}(i,s)\mathbf{B}(s,j)$ and updates its own output dense matrix with this partial matrix by $\mathbf{C}(i,j) = \mathbf{C}(i,j) + \mathbf{C}^s(i,j)$. The right of Fig. 1 illustrates the submatrices $P(2,1)$ stores and communicates in **sC**-2D algorithm for $p=9$ processes.

In terms of memory requirements, in addition to the matrices initially stored by $P(i,j)$, it also needs to store the sparse and dense submatrices it receives via communication. This effectively doubles the storage required by **A** and **B** at $P(i,j)$. The memory requirement of **sC**-2D SpMM at each process is therefore $(2\text{nnz}(\mathbf{A})+k(m+2n))/p$.

The local computations, which overall take $O((\mathrm{nnz}(\mathbf{A})/p) \cdot k)$ time, are split across $\sqrt{p}$ stages and each stage takes $O((\mathrm{nnz}(\mathbf{A})/p) \cdot k/\sqrt{p})$ time. Among the two multiplied matrices at each stage, both the sparse and the dense submatrix change at each stage.

5 IMPLEMENTATION

5.1 Collective-Based Implementations

In this section we describe the implementation details of the algorithms in Section 4 realized via a bulk-synchronous approach through MPI collectives. We also analyze their communication costs and discuss various trade-offs.

We rely on CombBLAS library [10] for parallel SpMM. CombBLAS is a distributed memory parallel graph library that is based on sparse matrix and vector operations on arbitrary semirings. It is originally designed for distributed memory execution on CPUs. We enhance this library in two main directions to realize the parallel SpMM algorithms described in this work.

In the first direction, we introduce support for storing distributed dense matrices, and 1D and 2D partition options for them. Furthermore, we realize the **sA**-1.5D, **sA**-2D, and **sC**-2D SpMM algorithms. The CombBLAS library utilizes a 2D decomposition for sparse matrices, which are in agreement with the decomposition used for the sparse matrix **A**. It provides compressed sparse column and doubly-compressed sparse column storage formats for sparse matrices. We utilize the former as it is more common and allows a seamless integration with the GPU libraries.

In the second direction, we introduce GPU support limited to the operations related to parallel SpMM. The local multiplies are performed with NVIDIA's cuSPARSE library[1] provided within the CUDA Toolkit. We utilize row-major storage for dense matrices, which is the recommended option by NVIDIA and which is also found to be the more efficient storage in our own evaluations. For the coordination of MPI collectives, we investigate using both the CPUs and the CUDA-aware MPI.

5.1.1 Communication analyses. **sA-1.5D.** **sA**-1.5D SpMM necessitates communication on two dense matrices, one for replicating **B** before local multiplication and one for reducing partial dense matrices after the local multiplication. The replication is realized with an `MPI_Allgather` among $\sqrt{p}$ processes in each column of the process grid. The replication begins at each process with nk/p words and thus the respective `MPI_Allgather` has a cost of $O(\alpha \log \sqrt{p} + \beta nk/\sqrt{p})$. The reduction of partial dense matrices is a reduction with summation operator and is realized with an `MPI_Reduce_scatter` among $\sqrt{p}$ processes in each row of the process grid. The reduction begins at each process with $mk/\sqrt{p}$ words and thus the respective `MPI_Reduce_scatter` has a cost of $O(\alpha \log \sqrt{p} + \beta mk/\sqrt{p})$. The total communication cost of **sA**-1.5D SpMM is therefore

$$O(2\alpha \log \sqrt{p} + \beta \frac{k(m+n)}{\sqrt{p}}). \tag{1}$$

sA-2D. **sA**-2D SpMM necessitates communication on input dense matrix before the local SpMM and on output dense matrix after local SpMM. These two communication operations are respectively realized with `MPI_Bcast` and `MPI_Reduce` collectives. The broadcast of the input dense matrix involves sending out nk/p words to $\sqrt{p}$ processes at each stage and thus the respective `MPI_Bcasts` have a total cost of $O(\alpha\sqrt{p} \log \sqrt{p} + \beta nk \log \sqrt{p}/\sqrt{p})$. The reduction of the output dense matrix involves summing mk/p words form $\sqrt{p}$ processes at each stage and thus the respective `MPI_Reduces` have a total cost of $O(\alpha\sqrt{p} \log \sqrt{p} + \beta mk \log \sqrt{p}/\sqrt{p})$. The total communication cost of **sA**-2D SpMM is therefore

$$O(2\alpha\sqrt{p} \log \sqrt{p} + \beta \frac{k(m+n) \log \sqrt{p}}{\sqrt{p}}). \tag{2}$$

sC-2D. **sC**-2D SpMM necessitates communication on one sparse and one dense matrix, each prior to local multiplies. These communication operations are both realized with `MPI_Bcast` collectives, each of which contains $\sqrt{p}$ processes. The broadcast of the sparse submatrix involves sending out $\mathrm{nnz}(\mathbf{A})/p$ words to $\sqrt{p}$ processes at each stage and thus the respective `MPI_Bcast` has a cost of $O(\alpha\sqrt{p} \log \sqrt{p} + \beta \mathrm{nnz}(\mathbf{A}) \log \sqrt{p}/\sqrt{p})$. The broadcast of dense submatrix of **B** follows a similar pattern except that it involves communicating nk/p words and hence has a cost of $O(\alpha\sqrt{p} \log \sqrt{p} + \beta nk \log \sqrt{p}/\sqrt{p})$. The total communication cost of **sC**-2D SpMM is therefore

$$O(2\alpha\sqrt{p} \log \sqrt{p} + \beta \frac{(\mathrm{nnz}(\mathbf{A}) + nk) \log \sqrt{p}}{\sqrt{p}}). \tag{3}$$

5.1.2 Remarks. Here we discuss various aspects of the collective-based implementations by comparing them against each other. We also mention computational issues related to utilizing accelerators.

Computation and memory. In terms of local SpMM computations, the 1.5D algorithms differ from the 2D algorithms in the sense that they do not "split" the computations across several stages. They perform the local multiplies at once where the 2D algorithms spread it across $\sqrt{p}$ stages in which the local multiplies are separated by various communication operations. On the one hand performing local multiplies at once may result in running out of memory when we take the limited memory of GPU accelerators into account. The 1.5D algorithms usually have a higher memory footprint those of 2D. For example, **sA**-1.5D necessitates more memory compared to **sA**-2D as long as $p > 4$. On the other hand, spreading computations across several stages may result in reduced memory bandwidth utilization and lower occupancy on the GPUs. Depending on where the submatrices are initially stored (device or host memory), splitting local multiplies may also lead to worse utilization of the link between the host and the device.

Communication costs. The 2D algorithms trade off increased communication costs for reduced memory footprint compared to the 1.5D algorithms. They have an extra $\sqrt{p}$ factor in the latency costs and an extra $\log \sqrt{p}$ factor in the bandwidth costs. Another important aspect is the sizes of sparse and dense submatrices being communicated. This is an aspect of the **A**-stationary vs. **C**-stationary algorithms, where the former passes around two dense matrices while the latter passes around one sparse and one dense matrix. When we compare communication costs of **sA**-2D and **sC**-2D, the only differing quantities are mk and $\mathrm{nnz}(A)$, which respectively relate to the sizes of the output dense matrix and the sparse matrix. When we consider that our work targets tall and skinny dense matrices, these two quantities become comparable

[1] https://docs.nvidia.com/cuda/cusparse/index.html

```
for i in 0..M-1:
  for j in 0..N-1:
    if C.owner(i, j) == rank():
      for k in 0..K-1:
        local_a = A.get_tile(i, k)
        local_b = B.get_tile(k, j)
        local_c = C.tile_ref(i, j)

        local_c += local_a*local_b
barrier()
```

Figure 2: RDMA-based 2D C-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for k in 0..K-1:
    if A.owner(i, k) == rank():
      for j in 0..N-1:
        local_a = A.tile_ref(i, k)
        local_b = B.get_tile(k, j)

        local_c = local_a*local_b

        queue[C.owner(i, j)].push(local_c)

while local_c = queue[rank()].pop():
  C.my_tile() += local_c

barrier()
```

Figure 3: RDMA-based 2D A-Stationary SpMM. M, N, and K represent tile dimensions.

and this makes them one of the key indicators of communication performance. Similar arguments apply to memory requirements as well.

Communication orchestration. We consider two options for orchestrating communication: conventional CPU-based communication and CUDA-aware MPI. In the former, the submatrices originally stay on host memory and they are copied back and forth between host and device memory. All communicated data pass through the host memory before leaving for the network. In the latter, the submatrices stay on device memory and they are directly communicated between devices on different nodes by bypassing the host memory. The latter option is arguably faster but we include both options in our analyses as not all MPI vendors or architectures may have support for GPUDirect RDMA.

B-Stationary algorithms. One of the design issues regarding the parallel SpMM algorithms is the **B**-Stationary algorithms. When $m = n$, **B**-Stationary algorithms have similar properties to those of **C**-Stationary algorithms. If this is not the case and the sparse matrix is rectangular, **B**-Stationary algorithms should be preferable to their **C**-Stationary counterparts for when $m < n$. We do not investigate **B**-Stationary algorithms in our study and leave it as a future research direction. All matrices used in our evaluations are square.

Finally, there are certain shortcomings to the communication analyses given in Section 5.1.1. First, our analyses did not take the hierarchical nature of the supercomputer systems we conduct our experiments on. On these systems, there often exist multiple GPUs per node and the analyses consider the maximum amount of data received per GPU whereas what matters is the maximum amount of data received per node. However, the analyses should hold when there is one GPU on a node. Second, they are not valid anymore when we consider asynchronous execution. The time spent in communication can be spread over time and overlapped by computations - which enables better load balancing and better utilization of the network over the algorithm runtime. We next turn our focus to such algorithms.

5.2 RDMA-Based Implementations

An alternate method to using MPI collectives to communicate matrix tiles in bulk synchronous steps is to use *RDMA operations* to transfer individual matrix tiles between nodes on demand. RDMA (Remote Direct Memory Access) allows for processes to issue `get` and `put` requests that can arbitrarily read or write data from any location within the shared segment of another process. These operations are entirely one-sided, since they can be executed directly by the NIC (Network Interface Card) on the remote node. On GPUs, GPUDirect RDMA allows for direct transfer of data between two remote GPUs over an Infiniband network. Today there are two user-level libraries for taking advantage of RDMA when using GPUs, (1) NVIDIA's NVSHMEM library, which provides an extension of the OpenSHMEM communication library, and (2) CUDA-aware MPI, for which there are some implementations, such as OpenMPI, that take advantage of GPUDirect RDMA.

For sparse matrix multiplication, RDMA-based implementations are particularly interesting because they allow for *decoupling* of the inner loop, which in a bulk synchronous implementation is normally executed in lockstep by all processes. This lockstep execution can lead to load imbalance problems when processes have differing amounts of local computation to perform, which is common in sparse matrix multiplication due to nonuniform distributions of nonzeros in the sparse matrix. In an RDMA-based implementation, processes can retrieve tiles of the matrix on demand, without any required involvement from remote processes. Thus, no synchronization with other processes is required until the very end of the computation, when processes synchronize to ensure the entire computation has finished.

Pseudocode for our RDMA-based **C**-Stationary algorithm is shown in Figure 2. Each process will iterate through the **C** matrix, to find the tiles of **C** that it owns. Then, it iterates through the corresponding row block of **A** and column block of **B**, issues `get` operations to retrieve the tiles, and then performs a local SpMM operation, accumulating into its local tile of **C**. Note that no synchronization is required until the very end, where a barrier is issued to ensure that all processes have finished the computation before continuing.

Our **A**-Stationary algorithm, shown in Figure 3, is somewhat similar, but re-orders the iterations of the loops. The two outer loops iterate through tiles of **A**, with each process identifying the tiles of **A** which it owns. For each of its tiles of **A**, a process will iterate through the corresponding row of **B**, retrieving each tile with a remote `get` operation. For each tile, it will perform a local SpMM operation between the local tile of **A** and the newly fetched tile of **B**, producing a partial product **C** that needs to be accumulated into

```
for i in 0..M-1:
  for j in 0..N-1:
    if C.owner(i, j) == rank():
      k_offset = i + j
      buf_a = A.async_get_tile(i, k_offset % K)
      buf_b = B.async_get_tile(k_offset % K, j)
      for k_ in 0..K-1:
        k = (k_ + k_offset) % K

        local_a = buf_a.get()
        local_b = buf_b.get()
        local_c = C.tile_ref(i, j)

        if (k_ + 1 < K):
          buf_a = A.async_get_tile(i, (k+1) % K)
          buf_b = B.async_get_tile((k+1) % K, j)

        local_c += local_a*local_b
barrier()
```

Figure 4: Optimized RDMA-based 2D C-Stationary SpMM. M, N, and K represent tile dimensions.

into a corresponding tile of **C**, likely remote. To do this, a process will append a pointer to the **C** partial product onto a queue in the remote memory of the destination process. Once it has completed its matrix multiplications, each process will pop the matrices it needs to accumulate off of its queue, accumulating them into its local block of **C**.

There are two important optimizations that we applied to our RDMA-based algorithms in order to achieve good performance, which is shown in Figure 4. The first optimization is applying an *iteration offset* to the inner loop of the matrix multiply, shown in the pseudocode as `k_offset`. In the case where `k_offset` is equal to 0, every process in a row will issue a `get` operation to the first tile in that row in the first iteration, and in the next iteration to the second block, and so on. This is suboptimal, since the NIC associated with each process will become overloaded with many requests. Instead, we apply an iteration offset equal to the sum of the current **C** tile's coordinates to the inner loop. This ensures that each process will access a unique block within its row and column in each iteration, as well as ensuring that for a regular block 2D distribution the first `get` operations will be to local tiles of **A** and **B**. The second optimization is asynchronously prefetching the tiles needed for the next iteration of the computation, which allows for overlap of communication and computation. These optimizations are applied to both the **C**-Stationary and **A**-Stationary algorithms, but for brevity we only show their application to the **C**-Stationary algorithm in Figure 4.

6 EVALUATION

6.1 Setup

We consider a wide variety of sparse matrix inputs that represent problems from graph neural networks (reddit and amazon [30]), eigensolvers in nuclear structure calculations (nm7 and nm8 [4]), low-rank or non-negative matrix factorization (com-Orkut [20]), and bioinformatics (isolates [6]). Various properties of these matrices are presented in Table 2. Load imbalance is defined as the ratio of the maximum number of nonzeros assigned to a processor to the average number of nonzeros in each processor, or formally:

$$\frac{p \cdot \max_{(i,j)\in P} \operatorname{nnz}(\mathbf{A}(i,j))}{\operatorname{nnz}(\mathbf{A})}.$$

Table 2: Properties of matrices used in our evaluations. The values under "load imb." column present the load imbalance in terms of the sparse matrix elements for 100 processes (i.e., 10×10 2D partition of A),

Sparse matrix (**A**)	kind	$m = n$	avg degree	nnz(**A**)	load imb.
amazon	GNN	14.3M	16.2	230M	1.01
com-Orkut	NMF	3.1M	76.3	234M	8.15
isolates	biology	17.5M	299.6	5.2B	1.00
nm7	eigen	5.0M	129.9	648M	6.38
nm8	eigen	7.6M	78.1	592M	6.48
reddit	GNN	233K	492	115M	1.08

We use three different column dimension sizes for the dense matrices: 128, 256, and 512.

In our evaluations, we test three variants of SpMM algorithms: 1.5D **A**-Stationary (s**A**-1.5D), 2D **A**-Stationary (s**A**-2D), and 2D **C**-Stationary (s**C**-2D). For s**A**-1.5D, we only consider the bulk-synchronous version (Section 5.1) in which we use collectives. We use the suffix "BS" to indicate the bulk-synchronous variant, i.e., s**A**-1.5D-BS. For the rest of the two algorithms, we consider both the bulk-synchronous (Section 5.1) and asynchronous versions (Section 5.2). We use the suffix "AS" to indicate the asynchronous variant, i.e., s**A**-2D-AS or s**C**-2D-AS. Hence, we have five SpMM algorithms in our evaluations under two category:

- three bulk-synchronous algorithms s**A**-1.5D-BS, s**A**-2D-BS, and s**C**-2D-BS, and
- two asynchronous algorithms s**A**-2D-AS and s**C**-2D-AS.

The bulk-synchronous versions make use of MPI collectives and they rely on CPUs to orchestrate communication, i.e., they do not make use of CUDA-Aware MPI and explicitly copy back and forth the data needed for local multiplies between host and device (the reasons of which will be clear in Section 6.3). The asynchronous version, on the other hand, relies on RDMA operations.

We conduct our experiments on IBM Summit system at Oak Ridge National Laboratory. Each node in this system has two sockets and is equipped with two PowerPC 9 CPUs and six NVIDIA Volta V100 GPUs. Each of the CPUs has 22 cores and 256 GB of memory and each of the GPUs has 80 SMs and 16 GB of memory. The CPUs are not utilized for computation and they are only responsible for communication in bulk-synchronous algorithms.

The bulk-synchronous algorithms are implemented within CombBLAS [10] and the asynchronous algorithms are implemented within BCL [7]. BCL utilizes NVSHMEM v2.0.2. All codes are compiled with the host compiler gcc v8.1.1 and the device compiler nvcc v11.0.3. For MPI, IBM's Spectrum MPI is utilized. For local multiplies, we rely on NVIDIA's cuSPARSE library and utilize row-major storage for dense matrices. Specifically, we use the generic API function `cusparseSpMM` with the choice of option `CUSPARSE_SPMM_CSR_ALG2`, which is the recommended option by the manual when dense matrices are stored in row-major order. We benchmarked alternative

SpMM implementations within cuSPARSE but found this recommended option to be the fastest overall. For all matrices, we utilize single precision for matrix elements. For the storage of indices in sparse matrices, we use 32-bit integers for all matrices except the isolates matrix.

In our evaluations, for the bulk-synchronous implementations, we use six different numbers of MPI processes $\{9, 25, 49, 100, 196, 400\}$, corresponding to $\{2, 5, 9, 17, 33, 67\}$ Summit nodes. For the asynchronous implementations, we also utilize six different numbers of NVSHMEM processes $\{12, 24, 48, 96, 192, 384\}$, corresponding to $\{2, 4, 8, 16, 32, 64\}$ Summit nodes. We will call both "processes" going forward without quantifying whether they are MPI or NVSHMEM processes, to avoid clutter. In parallel evaluations, we always use six processes per node and assign one process to control a GPU. We use the default task assignment for scheduling processes, which assigns the tasks to compute resources linearly. The discrepancy between these two different sets stems from the requirement that CombBLAS requires the number of processes to be a perfect square. For the asynchronous algorithm, we choose a number that is a multiple of six (the number of GPUs on a node) and close to the number of processes utilized by the bulk-synchronous algorithms as much as possible. This small difference should not affect the direction of the arguments.

We first focus on the evaluations regarding local SpMM benchmarking (Section 6.2) and communication orchestration (Section 6.3). Then we present results for overall runtime of the evaluated parallel algorithms (Section 6.4) and detailed performance analysis (Section 6.5).

6.2 Local SpMM benchmarks

Many computational kernels that operate on sparse matrices already achieve a low fraction of peak performance. Evidence of this behavior is widely reported in the literature and several mitigating techniques have been developed. A lesser known but no less dramatic problem emerges when we try to strong scale sparse kernels on distributed memory machines. As the number of partitions (which are often but not always equal to the number of processes) grow, the local sparse computations start achieving even a smaller percentage of the peak. This causes unscalability even if the communication costs are made to be negligible, either via algorithmic innovations or communication-computation overlap.

Admittedly, the performance degradation when strong scaling is present in all computations but dense computations are dramatically more forgiving. For example, dense matrix-matrix multiplication can be made to achieve close to ($\geq$ 80%) peak achievable performance for dimensions as small as 16 for a single core Xeon [16] and as small as 384 for a Fermi GPU [29].

The reasons for performance degradation of sparse kernels are often fundamental. For example, when using 2D partitioning, the number of nonzeros per row or column in each local sparse matrix goes down as the number of partitions increases [8]. This increased sparsity puts a lower bound on the maximum achievable performance.

When a fixed sized computation is strong scaled to increasing number of devices, the aggregate memory available to the program inevitably increases. This prompted a fruitful line of research where

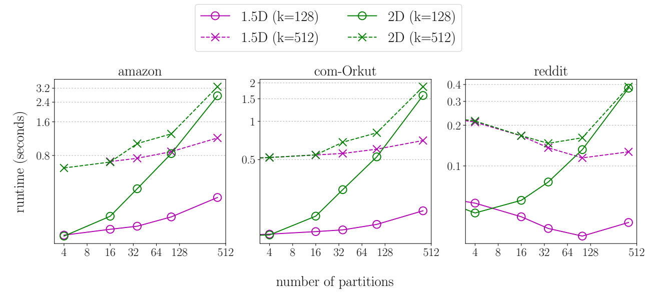


Figure 5: Total local SpMM time across all processors (partitions) when running stationary-A 1.5D and stationary-C 2D algorithms on k=$\{128, 512\}$. Ideal linear scaling is a flat line.

input or output matrices are replicated to reduce communication. A less appreciated benefit of communication-avoiding algorithms is their ability to batch computation and improve strong scaling of local computation.

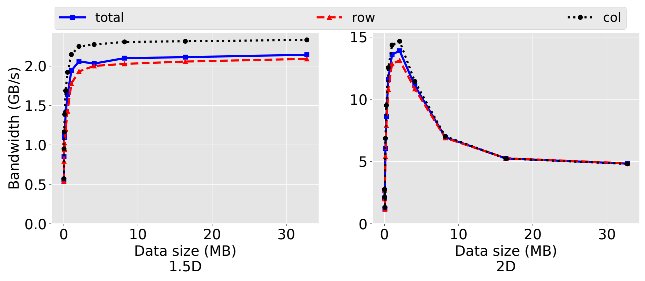
In Figure 5, we show the total time spent in local SpMM calls (`cusparseSpMM`) when the number of partitions (processes) increases. We see that 2D algorithm spends more and more time on local SpMM computations. This makes it not strongly scalable even if there were no communication costs and no other other parallelization overheads such as load imbalance or matrix creation. Our 1.5D algorithm, on the other hand, exhibits much better local SpMM scaling. While for com-Orkut, it shows a 2.8× increase in runtime for k=128 case and a 9.3× increase in runtime for k=512 case, these slowdowns are significantly better than the ones experienced by the 2D algorithm. Surprisingly for reddit, 1.5D algorithm even achieves a superlinear scaling in local SpMM computations.

We can explain the severe performance degradation observed for 2D algorithms (Figure 5) with a simple Arithmetic Intensity (AI) argument. Assuming perfect load balance, the arithmetic intensity of local SpMMs within a 2D algorithm running on a perfect square process grid is:

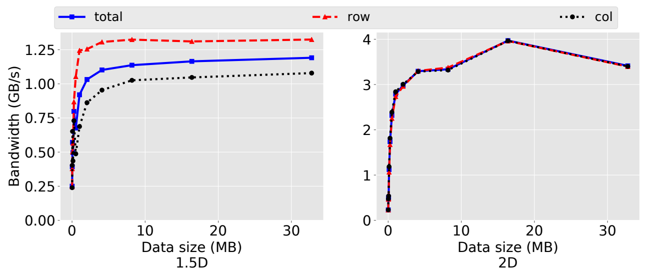
$$AI_{local2D} = \frac{(nnz(\mathbf{A})/p) \cdot (k/\sqrt{p})}{nnz(\mathbf{A})/p + nk/p} = \frac{nnz(\mathbf{A}) \cdot k}{\sqrt{p}\,(nnz(\mathbf{A}) + nk)}$$

Clearly, the arithmetic intensity of local SpMMs within a 2D algorithm goes down as the number of processors increases. By contrast, the arithmetic intensity is constant for 1.5D algorithms under the same conditions.

We also benchmarked another SpMM code called Sputnik [15]. We found that Sputnik performs as fast as or faster than the cuSPARSE implementation on a single GPU or when the SpMM is performed on a small ($\leq$ 16) number of GPUs. However, its performance degraded much faster than cuSPARSE's as we increased the number of GPUs. When the SpMM is performed on a larger (> 16) number of GPUs, cuSPARSE was consistently faster for all but one instance (reddit). This can be explained by the relative denser use cases Sputnik targets because the local sparse matrices get sparser as concurrency increases when 2D partitioning is employed. For consistency, we performed all subsequent experiments in this paper using cuSPARSE but it would be possible to swap a different implementation depending on the sparsity level of the input as well



(a) CPU MPI



(b) CUDA-Aware MPI

Figure 6: Per-process bandwidth microbenchmarks with a 6×6 process grid, 6 processes per node and 1 process per GPU, on Summit for both our 1.5D and 2D algorithms across varying message sizes. *row* refers to MPI calls done on process rows, such as `MPI_Reduce_scatter` in the 1.5D algorithm. *col* refers to MPI calls on process columns, such as `MPI_Allgather` in the 1.5D algorithm. *total* refers to the bandwidth of running the entire algorithm.

as the concurrency the computation is run on. Since identifying the optimal transition point between algorithms is non-trivial without first running the computation, we leave this tuning opportunity as an interesting future work.

6.3 Communication orchestration

We isolate the performance of communication primitives by microbenchmarking simultaneously-executing collective operations that are typical of 1.5D and 2D algorithms. We benchmark CPU and GPU-orchestrated communication and illustrate the obtained bandwidth results in Fig. 6. On the node level each node of Summit system has a 25 GB/s bi-directional bandwidth.

These figures show that the 2D algorithms take better advantage of available bandwidth. On a further note, the CUDA-Aware MPI seems to be not using NVIDIA GPUDirect as they are significantly worse than the bandwidth sustained by the CPU-orchestrated collectives. This is probably due to collective implementations of the underlying MPI implementation not taking advantage of the RDMAs yet. For this reason, in our bulk-synchronous SpMM algorithms we prefer CPU-orchestrated communication. The asynchronous algorithms already do not rely on MPI and utilize RDMA operations for data transfer.

Table 3: Runtime (in seconds) of SpMM on one GPU for two test instances.

k	com-Orkut	reddit
128	1.343	0.577
256	2.644	0.726
512	4.047	0.904

6.4 Runtime of parallel SpMM algorithms

We compare five schemes **sA**-1.5D-BS, **sA**-2D-BS, **sC**-2D-BS, **sA**-2D-AS, and **sC**-2D-AS on six test matrices for three different values $k \in \{128, 256, 512\}$. For all schemes, we conduct our evaluation on the six matrices. We measure the overall SpMM time in seconds. The obtained results are illustrated in Fig. 7. The top, middle, and bottom rows of the figure presents the plots for $k = 128$, $k = 256$, and $k = 512$, respectively and both axes in the plots are in log-scale. The missing points in the plots are due to the device running out of memory while multiplying the respective instances, which may be due to both some instances being quite large and having high load imbalance. We also present SpMM runtime on one GPU for com-Orkut and reddit instances (the other instances run out of memory).

The plots in Fig. 7 show the asynchronous schemes achieve better parallel runtime than the bulk-synchronous schemes in almost test instances. Their better performance can be attributed to two contributing factors related to communication and computation. First, the asynchronous variants orchestrate the communication more efficiently by relying on RDMAs. Furthermore, they spread the possible communication imbalance across multiple stages in which one can hide communication overheads by overlapping them with local multiplies. In a similar manner, they are able to hide possible computational imbalance via asynchronous execution.

Among the compared schemes for the bulk-synchronous schemes, **sA**-1.5D-BS exhibits superior scalability. The replication here seems to be paying off as it is able to better make use of GPU resources. However, this increased memory footprint has the drawback of running out of memory at small node counts where the 2D SpMM algorithms successfully complete execution (i.e., amazon and isolates for $k = 512$). The lower runtimes of asynchronous variants on some small number of processes compared to **sA**-1.5D-BS can be attributed to its better device occupancy at small node counts compared to higher node counts.

Among the 2D bulk-synchronous algorithms, **sA**-2D-BS SpMM tends to scale better than **sC**-2D-BS SpMM, which is especially true for matrices such as nm7 and nm8 that suffer from high load imbalance. For such instances, communicating the sparse matrix should be more costly than communicating the dense matrices as dense matrices do not have balance issues. Thus, among the bulk-synchronous algorithms, it is safe to recommend the **A**-Stationary algorithms (1.5D or 2D) for load-imbalanced instances.

We next vary the column dimension of the dense matrices and plot it against runtime for amazon, com-Orkut, and reddit instances in Fig. 8. All schemes achieve lower runtime with decreasing k as expected. The rate of decrease is greater in instances whose dimensions are larger.

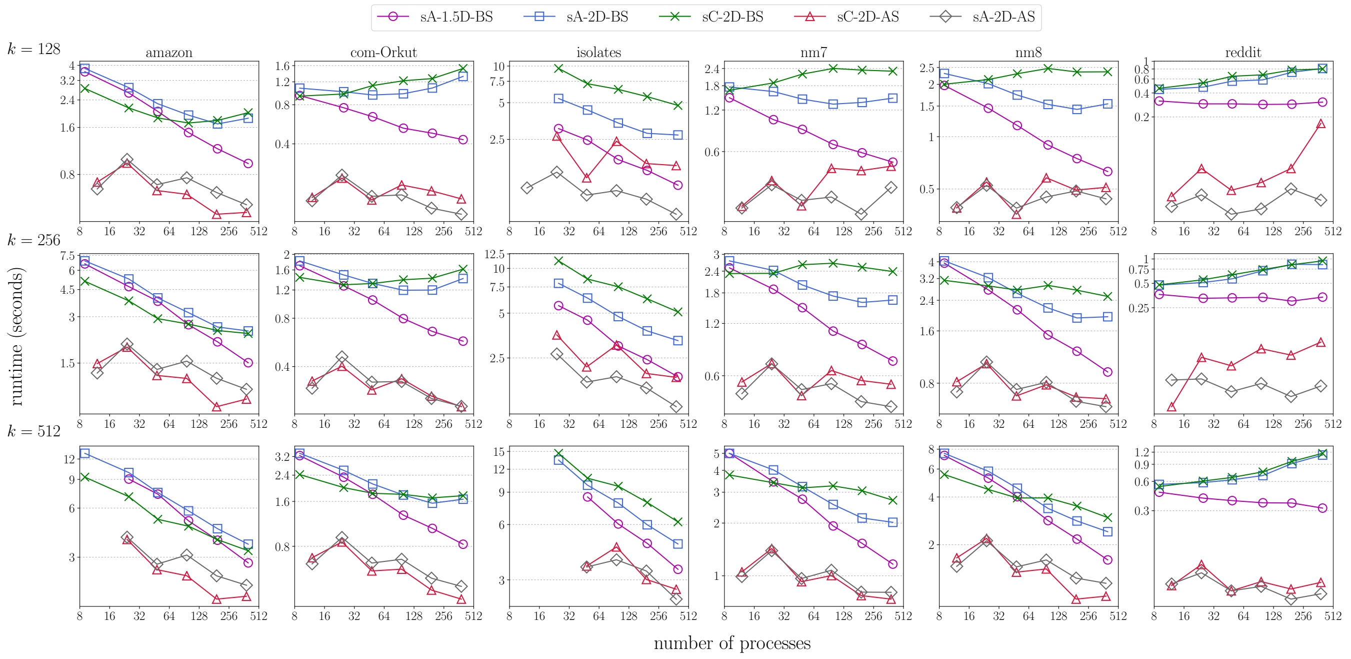


Figure 7: Parallel runtimes of five parallel SpMM algorithms on six evaluated instances. The row and column dimension of the figure respectively belongs to dense matrix column dimension k and matrix instance.

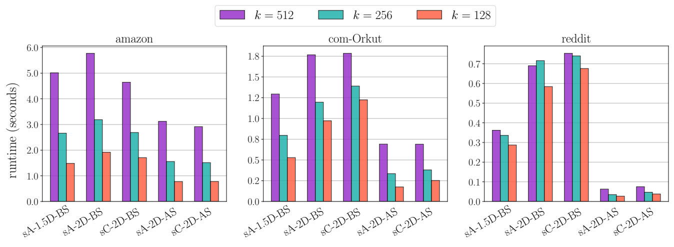


Figure 8: Parallel runtimes of five parallel SpMM algorithms for varying column dimension of the dense matrices. The runtimes belong to the instances at 100 processes.

6.5 Performance analysis of bulk-synchronous algorithms

We next focus on the details of the parallel performance of the bulk-synchronous algorithms **sA**-1.5D-BS, **sA**-2D-BS, and **sC**-2D-BS. We investigate these schemes' communication and computation components and identify the cases where each component is likely to be a bottleneck. In this regard, we focus the percentage of time spent in each component (Fig. 9) and these components' scalability (Fig. 10). For the experiments in Fig. 9 we fix the number of processes to 100 and for the experiments in Fig. 10 we fix $k = 256$.

Among the three bulk-synchronous algorithms, it can be said that the communication component of **sA**-2D-BS usually constitutes a smaller portion of the overall execution time compared to other two algorithms. **sC**-2D-BS seems to be negatively affected by the highly imbalanced instances. Although **sA**-1.5D-BS has a smaller communication cost than **sA**-2D-BS (Section 5.1.1), in practice the communication component constitutes quite a large portion in this algorithm. This can be attributed to the observations discussed in the previous section, where a monolithic and faster local multiply for **sA**-1.5D-BS renders communication operations to be more of a bottleneck. Finally, the communication component becomes more pronounced with increasing k. This is due to GPUs favoring regular access patterns with high flop per byte ratio.

We next investigate how each of these components scale in Fig. 10. Among 6 test instances, the ones that scale relatively better are the amazon and isolates instances. This is because these datasets have relatively better load balance. Among the instances that exhibit poor scalability, the matrices com-Orkut, nm7, and nm8 have the worst load balance and the reddit instance have the smallest sparse matrix dimensions. **sA**-1.5D-BS suffers less from load imbalance issues since it does not perform SpMM in stages and it does communicate the sparse matrix. **sC**-2D-BS's communication scalability is worst among the three algorithms. Hence, it can be said that **sA**-1.5D-BS is preferable over **sA**-2D-BS and **sC**-2D-BS when the sparse matrix distribution is especially imbalanced.

7 HIGHLIGHTS

We highlight the main findings of our work to provide a clear picture:

- The arithmetic intensity plays a more crucial role in the performance of parallel SpMM running on GPUs than on CPUs. The higher arithmetic intensity of 1.5D algorithms in local SpMM enables them to scale better than the 2D algorithms (Section 6.2).

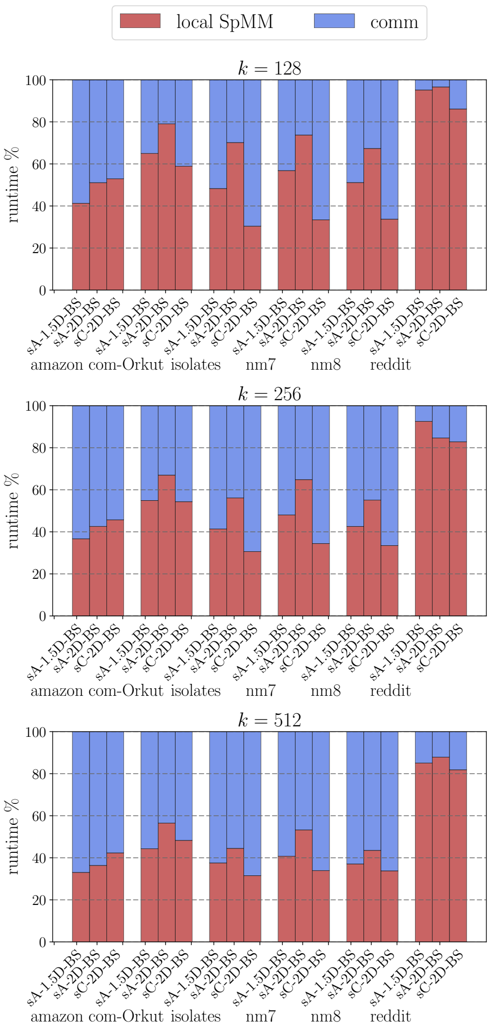


Figure 9: The percentage of time spent in local SpMM and communication for 100 processes.

- Regarding the communication performance of the investigated SpMM algorithms, the 2D ones are able to take better advantage of the available bandwidth (Section 6.3).
- The higher performance of the asynchronous implementations compared to the bulk-synchronous implementations can be attributed to their lower overhead communication via RDMAs, their ability to better utilize network bandwidth by spreading out the communication to the entire duration of the distributed SpMM, and their ability to avoid load imbalance that is caused by bulk-synchronous executions. (Section 6.4).
- The 1.5D algorithms often exhibit better scalability than the 2D algorithms as the replication pays off by making better utilization of the GPUs. The drawback of the replication in 1.5D algorithms, however, is that they are more likely to run out of memory for large instances at small node counts due to limited HBM available on GPUs (Section 6.4).
- Regarding the 2D bulk-synchronous algorithms, the **A**-Stationary algorithms are preferable to the **C**- or **B**-Stationary algorithms

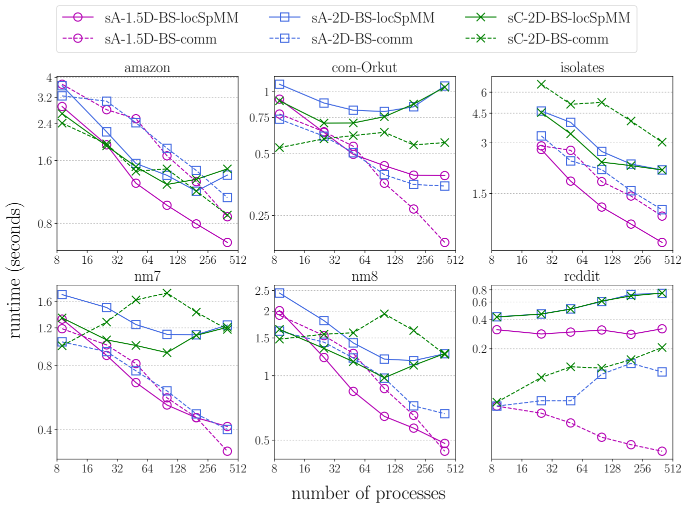


Figure 10: Scaling of local SpMM and communication in sA-1.5D-BS, sA-2D-BS, and sC-2D-BS.

when the sparse matrix is highly load-imbalanced because communicating the dense matrices is less likely to create performance bottleneck than communicating the sparse matrix, which is the source of the load imbalance (Section 6.4).
- Faster and monolithic local SpMM in 1.5D algorithms makes them less communication-bound than 2D algorithms, which run in stages. Specifically, regarding the bulk-synchronous algorithms, the 1.5D **A**-Stationary algorithm spends a smaller fraction of its time communicating, because it does not run in stages and does not communicate the sparse matrix. Hence, it is able to achieve the best overall parallel performance among the bulk-synchronous algorithms (Section 6.5).

8 CONCLUSIONS

In this work, we systematically analyzed, implemented, and evaluated various design choices regarding parallel SpMM on distributed memory nodes with GPUs. Our implementations and evaluation cover both the use of popular bulk-synchronous approach that utilizes MPI's collective communication routines, as well as a truly asynchronous RDMA-based approach using Berkeley Container Library (BCL) on top of NVIDIAâĂŹs NVSHMEM. Our asynchronous implementation does not synchronize across any subset of processors, except once at the very end of computation. Whenever there are multiple libraries to choose from, we identified the best performing variant via microbenchmarks before incorporating it into our implementation.

We found that the algorithmic trade-offs regarding distributed-memory parallel SpMM are quite distinct from the trade-offs for distributed-memory DGEMM and SpGEMM. In particular, the 2D **C**-Stationary algorithm, which is shown to scale well for both DGEMM and SpGEMM, exhibits poor scalability for SpMM. Despite its increased memory footprint, the 1.5D **A**-Stationary algorithm has shown better performance due to (1) its higher arithmetic intensity, which leads to better utilization of GPUs, and (2) reduced synchronization points. We also found our asynchronous implementation to be surprisingly fast in practice, yet its scalability was limited. Our work revealed shortcomings of certain preferred algorithms and showed that with the involvement of GPU accelerators, some kernels may need to take into account the design directions that are not conventionally considered in the absence of accelerators.

As future work, reordering of the sparse matrix can be used to minimize communication. Graph and hypergraph models [1] would be the choice of preference for this purpose. This is especially promising for the asynchronous variants as most partitioners only minimize the total volume and do not minimize the maximum volume per processor directly, consequently limiting performance gains of bulk-synchronous implementations that rely on collective communication. Compared to the sparse matrix-dense vector multiplication, which is arguably one of the most popular applications of partitioners, the potential savings in terms of communication volume reduction are larger by a factor of k for SpMM.

ACKNOWLEDGMENTS

This work is supported by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231, and by the National Science Foundation under Award No. 1823034. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. 2016. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Comput.* 59 (2016), 71 – 96. Theory and Practice of Irregular Applications.

[2] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.

[3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. 1990. Communication complexity of PRAMs. *Theoretical Computer Science* 71, 1 (1990), 3 – 28.

[4] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *28th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1213–1222.

[5] Ariful Azad, Grey Ballard, Aydın Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.

[6] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (01 2018), e33–e33.

[7] Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A cross-platform distributed data structures library. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[8] Aydın Buluç and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[9] Aydın Buluç and John R Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[10] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509.

[11] Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm.* Ph.D. Dissertation. Montana State University-Bozeman, College of Engineering.

[12] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective communication: theory, practice, and experience: Research Articles. *Concurr. Comput. : Pract. Exper.* 19, 13 (2007), 1749–1783.

[13] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *27th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 261–272.

[14] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2020. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14629–14638.

[15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[16] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[17] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[18] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[19] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[20] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2017. MPI-FAUN: an MPI-based framework for alternating-updating nonnegative matrix factorization. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2017), 544–558.

[21] Andrew V Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing* 23, 2 (2001), 517–541.

[22] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. 2016. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *30th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 842–853.

[23] Sureyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and Ponnuswamy Sadayappan. 2020. Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[24] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[25] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2019. Register-aware optimizations for parallel sparse matrix–matrix multiplication. *International Journal of Parallel Programming* 47, 3 (2019), 403–417.

[26] J. Park and K. Lee. 2020. Performance Prediction of Sparse Matrix Multiplication on a Distributed BigData Processing Environment. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 30–35.

[27] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. 2016. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing* 38, 6 (2016), C748–C781.

[28] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[29] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[30] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[31] R. A. Van De Geijn and J. Watts. 1997. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[32] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. In *29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[33] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*. Springer, 672–687.