A HYBRID COMMUNICATION PATTERN AND ALGORITHM FOR DISTRIBUTED SPARSE-TIMES-DENSE MATRIX MULTIPLICATION

BY

CHARLES BLOCK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Sparse matrix dense matrix multiplication (SpMM) is commonly used in applications ranging from scientific computing to graph neural networks. Typically, when SpMM is executed in a distributed platform, communication costs dominate. Such costs depend on how communication is scheduled. If it is scheduled in a sparsity-unaware manner, such as with collectives, execution is often inefficient due to unnecessary data transfers. On the other hand, if communication is scheduled in a fine-grained sparsity-aware manner, communicating only the necessary data, execution can also be inefficient due to high software overhead. We observe that individual sparse matrices often contain regions that are denser and regions that are sparser. Based on this observation, we develop a model that partitions communication into sparsity-unaware and sparsity-aware components. Leveraging the partition, we develop a new algorithm that performs collective communication for the denser regions, and fine-grained, one-sided communication for the sparser regions. We call the algorithm Two-Face. We show that Two-Face attains an average speedup of 2.11x over prior work when evaluated on a 4096-core supercomputer. Additionally, Two-Face scales well with the machine size.

To Julia & Mabel.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Josep Torrellas, and the other co-authors of the *Two-Face* paper, Gerasimos Gerogiannis, Professor Charith Mendis, and Professor Ariful Azad. Without their efforts, this work would have not been possible. I would also like to thank my wife, Julia, for her continued support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Sparse matrix dense matrix multiplication (SpMM) is a key kernel in sparse linear algebra. It has applications across a wide range of domains. For example, SpMM is a key operation in Latent Dirichlet Allocation, Non-negative Matrix Factorization, and Alternating Least Squares [1]. It is the bottleneck primitive in various Graph Neural Networks [2, 3, 4] and an integral part of popular graph learning frameworks such as PyTorch Geometric (PyG) [5] and Deep Graph Library (DGL) [6].

The ever-increasing computing and memory demands of sparse matrix computations introduce the need for efficient *distributed* SpMM. However, designing efficient distributed SpMM algorithms is challenging [7, 8, 9]. Due to the low arithmetic intensity of this kernel, the communication cost, rather than the computation cost, typically dominates the execution time. Such cost depends on how communication is scheduled between nodes [8].

Communication can be scheduled in a sparsity-unaware manner, such as with bulk collective communications. For example, assume that each node originally hosts a block of the dense input matrix. This block may be sent to all the other nodes by using collectives or RDMA accesses to fully replicate it [9] or by using shifting algorithms [8] similar to those that would be used for dense computation. With this strategy, execution is often inefficient due to redundant data transfers, since parts of the dense input matrix may not be needed by some nodes.

On the other hand, communication can be scheduled in a fine-grained sparsity-aware manner [10], communicating only the truly necessary data and computing asynchronously. Specifically, when a node processes a sparse element but does not own the necessary dense row for that computation, it gets that row with a fine-grained one-sided request. With this strategy, execution can also be inefficient due to high software overheads and the need for more network round-trips [8].

We observe that individual sparse matrices often contain a mix of relatively dense and relatively sparse regions. Based on this observation, this thesis devlopes a model that partitions computation and communication into sparsity-unaware and sparsity-aware portions. Relatively denser regions of the sparse matrix are broken down into *Synchronous Stripes* and transfer the corresponding parts of the dense input matrix with *Sparsity-Unaware Transfers (SUT)*. Relatively sparser regions are broken down into *Asynchronous Stripes* and transfer the corresponding parts of the dense input matrix with *Sparsity-Aware Transfers (SAT)*.

Leveraging the partition, we develop a new algorithm that performs collective communication for the synchronous stripes, and fine-grained, one-sided communication and asynchronous computation for the asynchronous stripes. We call the algorithm *Two-Face* [11]. The synchronous and asynchronous parts of the sparse matrix are processed in parallel, and the model aims at equalizing the runtimes of the two parts.

We evaluate *Two-Face* on a CPU-based supercomputer using large matrices and compare it to state-of-the-art baselines. For a system with 32 nodes, 128 cores per node, and dense matrices with 128 columns, *Two-Face* attains an average speedup of 2.11x against dense shifting [8], a high-performing baseline. In addition, *Two-Face* is a scalable algorithm: its average speedup over dense-shifting increases to 2.21x for 64 nodes. Finally, the overhead introduced by the necessary matrix preprocessing step is small enough to make *Two-Face* suitable for applications that use the same sparse matrix only a few dozen times..

CHAPTER 2: BACKGROUND

This chapter provides background material helpful for understanding later chapters. First, it describes the concept of sparsity in matrices and how this sparsity can be exploited for data compression and performance. It then discusses the sparse-times-dense matrix multiplication operation, both in the abstract and in the case of a distributed system. Finally, this chapter concludes with a discussion of some related works in the domain of processing sparse workloads.

2.1 SPARSITY IN MATRICES

Consider the matrices B and S shown below:

$$B = \begin{bmatrix} 0 & 2 & 8 & 8 \\ 4 & 1 & 9 & 7 \\ 1 & 6 & 9 & 3 \\ 9 & 9 & 3 & 7 \end{bmatrix}, \quad S = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.1}$$

Both B and S are 4×4 matrices containing integer values (i.e., $B, S \in \mathbb{Z}^{4\times 4}$). However, S is *sparse*: it contains very few elements which are non-zero. Instead of individually defining every element of S, we can instead define the matrix only in terms of its non-zero elements, and assume that any unspecified elements are zero, as shown below.

$$S_{1,2} = 2, \quad S_{3,3} = 9, \quad S_{3,4} = 3 \tag{2.2}$$

Representing sparse matrices in terms of only their non-zero elements can reduce their memory footprint in computer systems. In the case of the matrix S, only $\frac{3}{16}$ of the elements are non-zero. Representing S as a set of (*row*, *column*, *value*) integer tuples, requires a total of 9 integers, which reduces the memory footprint by 43.75% compared to naïvely storing all the elements of the matrix. Additionally, when using S in computations such as matrix multiplications, taking advantage of the sparsity reduces amount of computation that needs to be performed, since zeros can be skipped over implicitly. These memory and performance gains become even more pronounced with greater sparsity, such as the sub-1% densities observed in adjacency matrices of large graphs.

2.2 SPARSE MATRIX DATA FORMATS

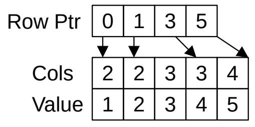
There exist a wide variety of data structures for representing sparse matrices. Some optimize for compression, some optimize for performance during computation, and the optimal choice is often dependent on the matrix's sparsity pattern.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) An example sparse matrix.

Rows	1	2	2	3	3
Cols	2	2	3	3	4
Value	1	2	3	4	5

(b) The matrix in COO form.



(c) The matrix in CSR form.

Figure 2.1: A sparse matrix shown in three different representations: (a) A traditional matrix format, which explicitly represents zeros, (b) a coordinate-list format, and (c) a compressed sparse row format.

The sparse matrix representation used in Section 2.1, which represents matrices as a list of coordinates and values, is commonly known as the Coordinate List (COO) format. COO represents the sparse matrices in a simple, easily understandable way, but it increases the memory requirement of each non-zero element substantially, since it stores both of its coordinates, along with its value. Figure 2.1a shows an example sparse matrix, and Figure 2.1b shows how this matrix might be encoded using COO. In this example, the non-zero located at $(1, 2)$ with the value 1 is stored in the sparse structure as the 3-tuple $(row = 1,\ col = 2,\ value = 1)$. The other nonzeros follow. In this case, the nonzeros are stored in row-major order, but this is not required when using COO.

An alternative data format, which attempts to reduce the memory requirement of COO, is the Compressed Sparse Row (CSR) format. Rather than storing a row index for each nonzero, CSR stores each row contiguously and maintains pointers to the beginning of each. Every non-zero entry is still stored with a column index. Figure 2.1c shows an example of the CSR format. In this example, the row pointers store the indices of the beginning of each of the four rows in the matrix. The first row (containing only the nonzero with the value 1) begins at index 0 of the structure, which is indicated by the first entry in the row pointer list. The nonzero is additionally specified by a column (2) and value (1). The next row contains two nonzeros, and begins at index 1 of the sparse structure. Along with the shared row pointer, each nonzero in this row is specified by a column/value pair. A related format to CSR is Compressed Sparse Column (CSC), which orders the non-zeros by column instead of row and maintains column pointers instead of row pointers.

The optimal data format depends on the sparse matrix and the computation [12]. For example, CSR may be better than COO when the matrix has many nonzeros in each row. COO may be best when the matrix has many empty rows and columns, since CSR and CSC would both maintain "useless" row/column pointers.

2.3 SPMM: SPARSE-TIMES-DENSE MATRIX MULTIPLICATION

The specific operation most relevant to this thesis is the multiplication of a sparse matrix, A, with a dense matrix, B, to produce another dense matrix, C. I.e.,

$$C = AB, \quad A \in \mathbb{R}^{m\times n},\ B \in \mathbb{R}^{n\times k},\ C \in \mathbb{R}^{m\times k}, \tag{2.3}$$

where A is sparse. SpMM commonly appears during the aggregation phase in graph neural networks [2, 3, 4], in which the sparse matrix A is the adjacency matrix and the dense matrix B represents the node embeddings. It also appears in latent dirichlet allocation, non-negative matrix factorization, alternating least squares [1], and graph learning frameworks [5, 6]. In these iterative applications, A is square and reused during each iteration. The output matrix, C, becomes the next iteration's input matrix, B.

An example of an SpMM operation is shown in Figure 2.2. The access pattern to the dense matrices are determined by the location of the non-zero elements of A. In the figure, B is shown transposed to highlight this relationship. The non-zero $A_{i,j}$ will read from the j^{th} row of B and accumulate onto the i^{th} row of C. For example, in Figure 2.2, $A_{1,6}$ is non-zero, so the row $B[6, 0 : K - 1]$ is accessed to load the corresponding dense input and the result of the scalar-vector multiplication is accumulated onto row $C[1, 0 : K - 1]$. Note that in this example, the rows of A with indices 0, 2, and 4 do not contain any non-zeros, so there is no need to access the corresponding row of C. Likewise, columns 0, 3, 5, and 7 of A are empty, so there are no accesses to their corresponding rows in B.

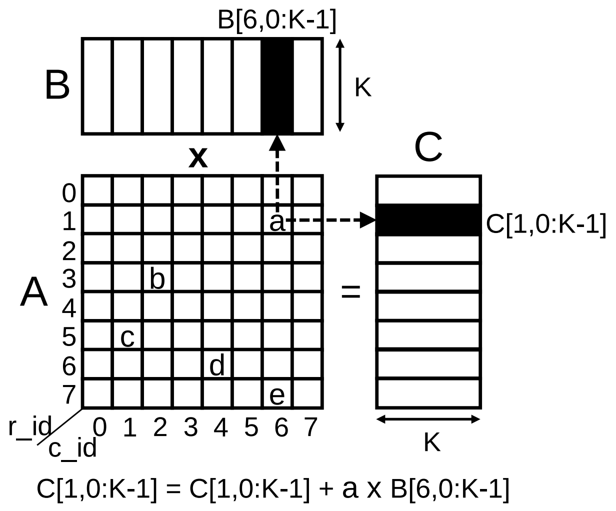


Figure 2.2: An illustration of an SpMM operation. The dense matrix B (shown transposed here) is multiplied by the sparse matrix A to produce the dense matrix C.

2.4 DISTRIBUTED-MEMORY COMPUTING

A distributed memory system is a parallel system of multiple independent *processors* or *nodes*, each with its own memory space. Each processor may consist of multiple parallel *cores*. The processors typically communicate via *message passing*, using an API such as the Message Passing Interface (MPI) [13, 14]. Processors may us traditional interconnects such as Ethernet or specialized interconnects such as Slingshot [15].

One example of a distributed-memory system is the National Center for Supercomputing Application's Delta Supercomputer [16]. This system contains 132 CPU nodes, each with a dual-socket configuration of two AMD EPYC 7763 processors, as well as an additional 207 GPU nodes. Each of these nodes contains between 256GB-2TB of main memory and runs its own operating system (in this case, Red Hat Enterprise Linux [17]). These nodes can be programmed as logically separate machines, or they can take advantage of their high-speed interconnects to perform a distributed computation for a single application.

2.4.1 Programming Distributed-Memory Systems

Each node in a distributed-memory system runs programs independently, synchronizing with the others only when explicitly directly to. To run distributed HPC applications, many systems are programmed using MPI [14]. This framework places the same code on each machine (although the machines are aware of their unique IDs) and provides message passing primitives to allow the machines to communicate.

This message passing may consist of point-to-point communication, where one node directly sends or receives data from one other node. Alternatively, the messages may take the form of *collectives*, such as *Broadcast*, *AllGather*, *Reduce*, or others. These collectives can use optimized communication patterns to reduce latency and and better utilize system bandwidth. For example, a Broadcast collective may use a tree pattern to communicate data to all p processors in a system in $O(\log p)$ steps.

2.4.2 Modeling Bandwidth and Latency

In distributed-memory computing, the performance of an application often depends on the network properties. Relvant properties include the bisection bandwidth, network latency, and topology [18]. A first-order approximation of the cost of communicating data of size n can often be modeled as $C(n) = \beta n + \alpha$, where β is $\frac{1}{\text{bandwidth}}$ and α is the latency of the network. In supercomputers, typically $\beta < \alpha$, so when the data transfer is small, the latency dominates the transfer cost, while when the data size is large, the bandwidth is the limiting factor.

2.5 DISTRIBUTED SPARSE MATRIX PROCESSING

Prior works have investigated using distributed-memory systems for a variety of sparse kernels, including SpGEMM [19, 20], SDDMM [8, 19], and, most relevant to this work, SpMM [7, 8, 9]. In this section, we discuss a simple scheme for partitioning matrices across compute nodes in a distributed system and how a variant of this scheme has previously been applied in practice.

In distributed SpMM, the input matrices, A and B, and the output matrix, C, are partitioned and/or replicated across the nodes. Each node in the system then computes its portion of the operation. One possible way to partition an SpMM computation, called 1D partitioning, is shown in Figure 2.3. In 1D partitioning, each node owns a set of contiguous rows from each of the matrices.

A straightforward algorithm for computing SpMM with using 1D partitioning has each node compute the output rows of C belonging to its local partition. This requires iterating over the sparse non-zeros belonging to its local partition of A and multiplying each non-zero by the vector in the corresponding row of B. In this algorithm, all accesses to elements in A and C are *local accesses*, since they only require data found on the local node. However, some accesses to B will be *remote accesses*, since they will require fetching data from another node.

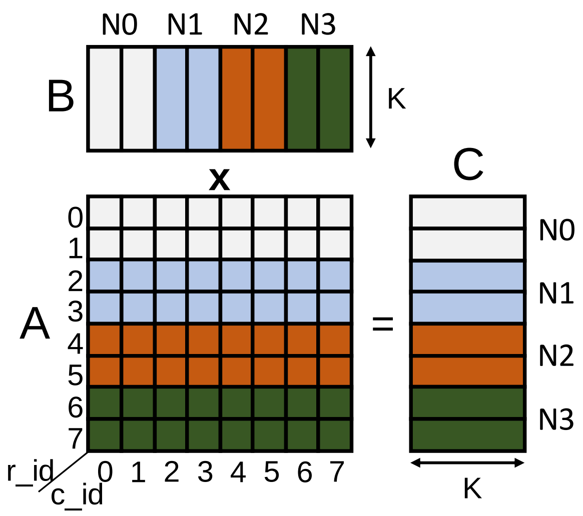


Figure 2.3: 1D partitioning for SpMM using a four-node configuration.

For highly-sparse workloads, the communication costs of remote accesses are known to dominate the cost of the SpMM operation.

2.6 PRIOR WORKS

There have been many recent works on sparse kernels, including distributed sparse kernels, GNN training, and hardware for both local processing and network support of these operations. Here, we discuss some of these prior works and their relation to the *Two-Face* algorithm [11] described later in this thesis.

2.6.1 Sparse Matrix Processing

Sparse matrices have become more prominent in machine learning and graph-related workloads. This thesis is concerned primarily with highly-sparse matrices, like those found in GNN aggregation steps. Thus, we will primarily discuss works related to these kinds of sparsity, as opposed to sparsity found in pruned DNNs.

Distributed SpMM Existing work on distributed SpMM is rather limited, but there are recent works exploring the topic. Bharadwaj et al. [8] investigate distributed SpMM, SDDMM, and methods of fusing the two for machine learning applications. We use their implementation of an algorithm which cyclically shifts the dense input matrix as a baseline in the evaluation of Two-Face (Section 5). In addition to this algorithm, Bharadwaj et al. [8]

also present a sparse-matrix shifting implementation. In our work, we did not evaluate this approach, since it partitions the dense input and output matrices in a way that requires additional all-to-all communication for GNNs or other applications that interleave SpMM with a row-wise operator. In their work, Bharadwaj et al. [8] compare their SpMM implementations to the SpMM provided by PETSc [7]. Separately, Selvitopi et al. [9] investigate multiple algorithms for SpMM, including algorithms that use bulk-synchronous collective communication and algorithms that use one-sided asynchronous RDMA communication. This is directly relevant to the work presented later in this thesis. However, unlike the approach presented in Section 4, they do not investigate combining these communication primitives in a single algorithm.

Other Distributed Sparse Kernels Other distributed sparse kernels have recently received attention. CombBLAS [19] is a library for distributed sparse kernels such as SpGEMM and SpMV. CombBLAS provides a number of GPU SpMM implementations using different partitioning and communication patterns. All of these SpMM implementations them use sparsity-unaware collectives to communicate. In contrast, this thesis will discuss a hybrid communication pattern that combines these collectives with sparsity-aware accesses. Hussain et al. [20] investigate communication-avoiding algorithms for SpGEMM. DGCL [21] is a library for distributed GNN training that partitions graphs and processes GNN computations at the level of nodes in the graphs, without explicitly expressing the computation with SpMM operations.

Non-Distributed Sparse Kernels SpMM optimization has been the topic of several investigations. Works such as WACO [22], WISE [23], and DDB [24] attempt to optimize sparse computations by using machine learning techniques to predict the performance of various configurations. Many CPU and GPU tiling techniques and implementations have been published [2, 25, 26, 27]. Other sparse kernels have also been subject to several investigations aiming to tame irregular access patterns [28, 29, 30, 31]. These optimizations for non-distributed kernels may be applicable to the distributed case, but they tend to assume a shared memory system, and they are largely orthogonal to our work.

Recently, SpMM, SpMV, and SpGEMM kernels for heterogeneous hardware have been proposed [32, 33, 34]. Cheng et al. [32] tackle SpGEMM on asymmetric multicore processors. HotTiles [34] partitions the SpMM sparse input matrix into two types of regions and assigns each region type to a different accelerator by solving an optimization problem.

2.6.2 GNN Training

Prior work has addressed the issue of large graphs in GNN training via sampling techniques [35]. However, the benefits of sampling can come at a cost to accuracy, leading prior work to investigate full-batch distributed GNN training [36, 37]. However, in Tripathy et al. [37], dense matrices used in SpMM operations are only transferred in a coarse-grained sparsity-unaware fashion. Conversely, Jia et al. [36], using Lux [38], assume a GNN runtime that operates via pushing/pulling node embeddings in a fine-grained manner. This is distinct from *Two-Face*, which uses a combination of coarse and fine-grained transfers to leverage the benefits of both approaches.

2.6.3 Domain-specific Architectures and Network Support:

Several architectural designs that offer hardware support for SpMM computation have been recently proposed [39, 40, 41, 42, 43]. SPADE [39] is an accelerator for SpMM and SDDMM designed to be tightly coupled with CPU cores. Tensaurus [40] and ExTensor [41] are accelerators for a variety of sparse kernels. We believe that algorithms such as *Two-Face* can be useful in orchestrating the communication in scaled-up multi-node versions of these accelerators or for other large-scale graph analytics architectures [44, 45, 46]. In addition, we believe that such algorithms can also be beneficial for inter-cube or inter-chip communication in PIM-based architectures for graph analytics [47, 48, 49, 50]

Finally, scheduling algorithms for collectives such as Themis [51] have been proposed to maximize the bandwidth utilization of multidimensional, heterogeneous networks. These works could inspire network hardware support for *Two-Face*, but one would also require innovations to support the asynchronous communication operations.

CHAPTER 3: COMMUNICATION IN DISTRIBUTED SPMM

The choice of how data is communicated in a distributed computation tends to have a large impact on performance. In the case of sparse workloads such as SpMM, algorithms can either disregard the sparsity patterns, essentially treating all the matrices as dense for the purpose of communication scheduling, or the algorithms can introspect the sparse workload to determine the necessary data to transfer. Both these sparsity-unaware and sparsity-aware communication techniques have trade-offs, as this chapter discusses. This chapter also discusses a possible compromise between these approaches, which attempts to get the best of both.

3.1 SPARSITY-UNAWARE COMMUNICATION

Algorithms which use sparsity-unaware transfers (SUTs) for SpMM do not schedule communication based on the content of the sparse matrix A. Rather, they use regular communications patterns, similar to those that would be used in dense matrix multiplications. An algorithm using SUTs might communicate the whole dense input matrix B (shown in Figure 2.2) to all nodes in the system using a broadcast operation, as illustrated in Figure 3.1.

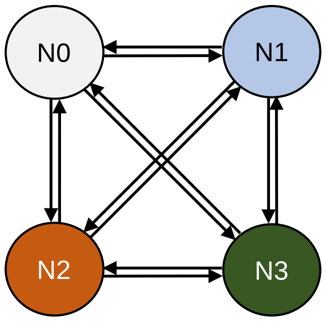


Figure 3.1: Sparsity-unaware transfers may use coarse-grained all-to-all patterns.

This will often send unecessary data between nodes, since one particular node may not need all rows of B to perform its local computation. However, the software overhead of transferring large, contiguous blocks of data is relatively low when compared to parsing sparse data structures and initiating many smaller transfers. Thus, SUTs are often good choices for matrices exhibiting only limited sparsity or sparsity patterns that still require large transfers.

3.2 SPARSITY-AWARE COMMUNICATION

Algorithms which use sparsity-aware transfers (SATs) for SpMM schedule their communication based on the sparsity pattern of A. A simple SAT algorithm would iterate over each non-zero element of A and compute its product with the corresponding input row of B. When that corresponding row is not in the local compute node, a remote access is used to get it. Figure 3.2 illustrates an exaple of a SAT pattern. In it, we can see that nodes N0, N2, and N3 communicate, since each must either request or provide data to another, but N1 does not communicate with any other node, since it only requires rows of B that are in its own partition.

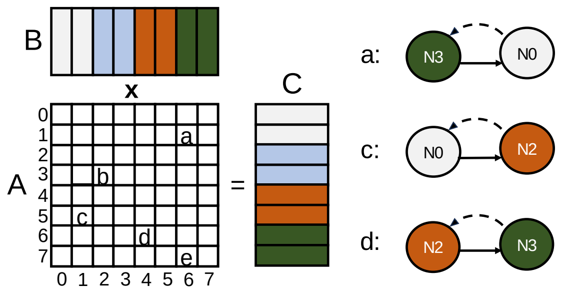


Figure 3.2: SATs use a collection of fine-grained one-to-one transfers.

Compared to SUTs, this may reduce the total number of data elements communicated during an SpMM operation. However, SATs often require more round trips and higher software overheads, since their requests take place at a finer granularity. Typically, SAT patterns are a good fit for highly-sparse matrices.

3.3 A MATRIX-DEPENDENT CHOICE

Due to the trade-offs presented by sparsity-aware and sparsity-unaware algorithms, the optimal choice between the two is generally matrix-dependent [8]. To illustrate this, we profiled the performance of distributed SpMM using a sparsity-unaware algorithm using *Collectives* and a sparsity-aware algorithm using *Async Fine*-grained accesses. The *Collectives* algorithm performs a single communication step, where the entirety of B is sent to all compute nodes using an AllGather operation, before the computation begins. The *Async Fine* algorithm is similar to the algorithm described in Section 3.2, where each node computes asynchronously and performs a remote access for necessary data. Figure 3.3 shows the measurements, which were evaluated on 32 nodes, each with 128 CPU cores. The figure shows the speedup of *Async Fine* over *Collectives* for $K = 32$ and $K = 128$. One workload is omitted for $K = 128$ since it exceeded the memory capcity of the nodes in our system.

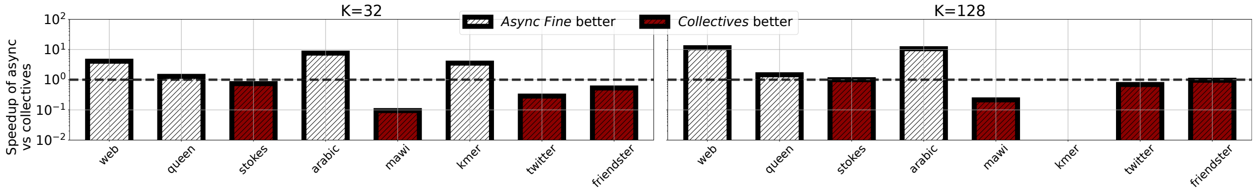


Figure 3.3: Comparison of the performance of *Async Fine* and *Collectives* on sparse matrices, for two different dense-matrix widths. The *Collectives* implementation ran out of memory when evaluating kmer for $K = 128$. For details on experimental methodology, see Section 5.1.

From Figure 3.3, we can see that the best algorithm depends on the matrix. *Async Fine* is able to reduce the communication cost substantially on some matrices, but the additional overheads that come from many small transfers make it inefficient on others. In particular, three of the workloads that *Async Fine* performs comparatively well have very few off-diagonal elements, and the other workload has a very large sparse matrix width, which correlates to a very high communication cost when transferring the entire dense input between nodes. Figure 3.3 also shows that as the size of the dense matrix increases, so does the benefit of using *Async Fine*. Intuitively, this is because as the size of the dense matrix rows increases, the amount of time saved by not transmitting unnecessary rows increases.

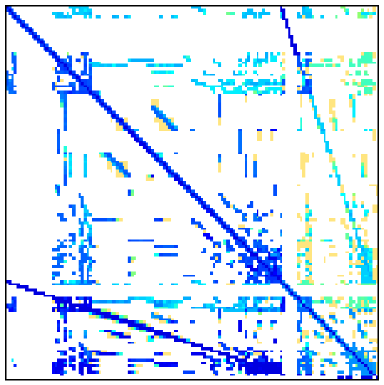


Figure 3.4: Density map of the VLSI/stokes matrix, from SuiteSparse [52]. Darker regions are denser. Note how different regions of the matrix exhibit different sparsities, suggesting different communication patterns.

3.4 A HYBRID COMMUNICATION PATTERN

In the previous section, we illustrated how the optimal choice of communication pattern for distributed SpMM can be matrix-dependent. However, sparse matrices often have complex internal structures. Some parts of the matrix may be highly-sparse, while others may be denser. To illustrate this, consider the heatmap of the stokes [52] matrix depicted in Figure 3.4. Although it is highly sparse, this matrix also contains regions that are relatively dense. During a distributed SpMM operation, we should expect the communication patterns from *Async Fine* to be beneficial in the regions with low density, while the coarser-grained *Collectives* should be more beneficial in denser regions.

Therefores, we propose tailoring the communication patterns of the SpMM operation to particular regions of the sparse matrix A. The high-level idea is shown in Figure 3.5. In this example, the matrix A has non-zero elements on the diagonal, which do not require data transfer, and off-diagonal non-zeros, which require remote accesses to rows of B.

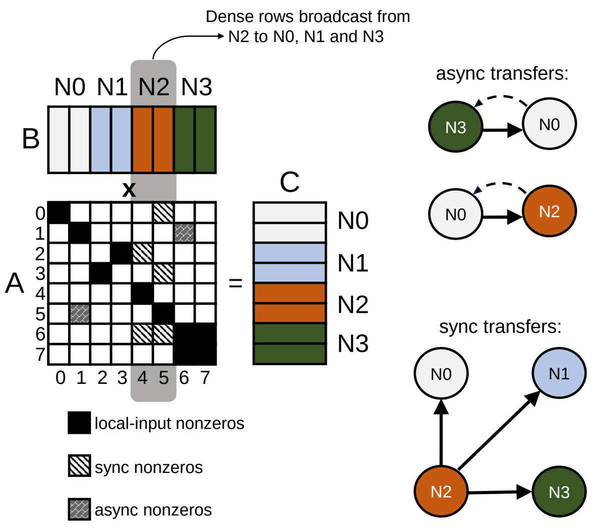


Figure 3.5: An illustration of how SUTs and SATs may be combined.

Two of these off-diagonal elements, $A_{1,6}$ and $A_{5,1}$, are in low-density regions: Node 0 only requires one row of B from Node 3, and Node 2 only requires one row of B from Node 0. These rows of B are also not required by any other node. Thus, we choose to transfer these elements in a sparsity-aware manner with fine-grained transfers.

In contrast, the shaded region of Figure 3.5 shows a relatively dense region of the sparse matrix. Every node requires data from Node 2, and most nodes require both of its rows. Since it would not make sense to have each node individually request each of these rows, we prefer to use coarse-gained SUTs to avoid software overheads and reduce the number of network round-trips. Note that these colletive operations do not necessarily have to include all nodes; they can be multicast operations that include only a subset. For certain systems, it may be a good decision to have nodes N1, N2, and N3 participate in this SUT collective, while N0 performs a fine-grained SAT-style access for the single row it requires.

By combining fine-grained point-to-point transfers and coarse-grained collectives, we can schedule our communication to get the best of both. In the next chapter, we will discuss how this idea can be implemented in practice, in the form of the Two-Face algorithm [11].

CHAPTER 4: DESIGN OF THE TWO-FACE SPMM ALGORITHM

Building on the idea of a hybrid communication pattern for SpMM, this chapter details the Two-Face algorithm [11]. Two-Face consists of a preprocessing step that performs a one-time analysis of a matrix and a runtime algorithm that actually evaluates the SpMM operation. It partitions a matrix into small blocks, which we call *stripes*. The preprocessing step determines the most appropriate communication method for each stripe, and the runtime algorithm follows that communication plan.

In this chapter, we will discuss how Two-Face partitions a matrix, the details of the preprocessing algorithm used to plan communication, and the design of the main Two-Face algorithm itself. Finally, we will discuss our evaluation of Two-Face and the preprocessor.

4.1 PARTITIONING THE MATRIX

Two-Face deals with a modified 1D matrix partitioning for distributed SpMM, shown in Figure 4.1. The sparse matrix is partitioned between the nodes in a 1D fashion (as previously shown in Figure 2.3). Additionally, the sparse matrix is further logically partitioned into *megatiles* and *sparse stripes*. We define a *megatile* as the largest collection of non-zeros that share a common input tile (in B) and output tile (in C). For example, all nonzeros in the shaded megatile in Figure 4.1 use input data from N0 and write their output to N0. The megatile to the right of the shaded one reads data from N1 and writes data to N0.

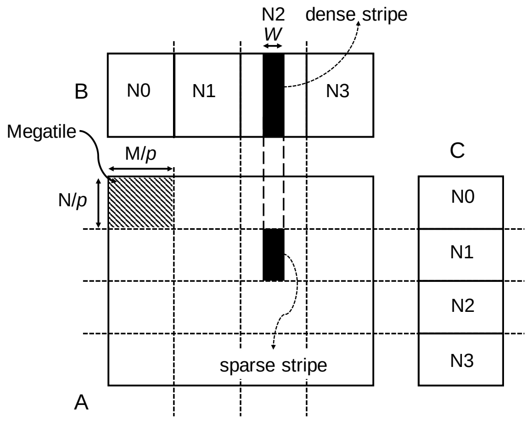


Figure 4.1: Partitioning a sparse matrix into megatiles and stripes.

Each megatile consists of a number of 1D partitions called *sparse stripes*, which are sets of contiguous columns (painted black in Figure 4.1). Each sparse stripe in A also has a corresponding *dense stripe* in B that it potentially requires rows from for its portion of the computation. In Two-Face, we classify these stripes as either *local-access stripes*, whose corresponding dense stripes reside on the local node, *sync stripes*, which use coarse-grained collectives to retrieve entire dense stripes, or *async stripes*, which use fine-grained one-sided remote accesses to retrieve a subset of rows from their corresponding dense stripe.

Finally, computation on the local-input and sync stripes operates on *row panels* (not shown): groups of contiguous rows of nonzeros. This is primarily a work sharing mechanism: when parallelizing computation, each row panel is assigned to a thread.

4.2 THE PREPROCESSING ALGORITHM

To determine which stripes should be treated as *sync* or *async*, Two-Face uses a preprocessing step. This preprocessing algorithm takes a sparse matrix and description of the system as input and outputs a classification of stripes as either *async* or *sync*.

During the main Two-Face algorithm these sync and async stripes will be processed in parallel. Therefore, the preprocessor attempts to balance the estimated execution time of the sync and async stripes. To do this, we create a cost model for each type of sparse stripe.

The cost model intentionally neglects the time taken to perform local computation on sync/local-input stripes. This is reasonable, since the sync and local-input stripes are stored in a format that provides good locality and are processed in a highly-parallel manner. In contrast, the cost model includes the cost of computing on the async stripes, since they are stored in a format with less locality and processed with fewer threads. More details on the rationale behind these decisions can be found in Section 4.3.1.

The costs of sync communication, async communication, and async computation for a node are modeled as shown below.

$$Comm_S = S_S\left(\beta_S K W + \alpha_S\right) \tag{4.1}$$

$$Comm_A = \beta_A K L_A + \alpha_A S_A \tag{4.2}$$

$$Comp_A = \gamma_A K N_A + \kappa_A S_A \tag{4.3}$$

Equation 4.1 describes $Comm_S$, the cost of synchronous communication. This is a function of S_S, the number of synchronous stripes processed by the node; K, the width of the dense input matrix B; and W, the width of a stripe. β_S and α_S are empirical coefficients that describe the per-element communication cost and per-stripe overheads, respectively.

Next, Equation 4.2 describes $Comm_A$, the cost of asynchronous communication. This is also a function of K, as well as the number of rows of B transferred by the asynchronous accesses, L_A, and the number of asynchronous stripes, S_A. The empricial coefficients β_A and

α_A represent the same costs as their counterparts in Equation 4.1, but for the asynchronous stripes.

The last part of the cost model, Equation 4.3, describes $Comp_A$, the cost of computing on the asynchronous stripes. In addition to K and S_A, this is also a function of N_A, the total number of nonzeros among all asynchronous stripes. This equations emprical coefficients, γ_A and κ_A, correspond to the per-nonzero computational cost and per-stripe computational overheads, respectively.

The empirical coefficients β_S, α_S, β_A, α_A, γ_A, and κ_A, are determined by a process of linear regression [53] during a calibration step. These parameters are dependent on the system's configuration. A system with a large bisection bandwidth should have relatively small β terms. A system with high round-trip latency (including latency introduced in the software stack) should have relatively high α terms.

As discussed previously, the preprocessing step should aim to balance the execution time of the sync and async stripes. Thus, we aim to satisfy Equation 4.4.

$$Comm_S = Comm_A + Comp_A \tag{4.4}$$

To rework this equation, we define $S_T = S_S + S_A$ to be the total number of non-local-input stripes processed by a node. We can then reformulate the requirement expressed in Equation 4.4 as:

$$\begin{aligned} & S_S(\beta_S KW + \alpha_S) = \beta_A K L_A + \alpha_A S_A + \gamma_A K N_A + \kappa_A S_A && (4.5)\\ \implies\quad & (S_T - S_A)(\beta_S KW + \alpha_S) = K(\beta_A L_A + \gamma_A N_A) + S_A(\alpha_A + \kappa_A) && (4.6)\\ \implies\quad & S_T(\beta_S KW + \alpha_S) = K(\beta_A L_A + \gamma_A N_A) + S_A(\alpha_A + \kappa_A + \beta_S KW + \alpha_S) && (4.7) \end{aligned}$$

Although Equation 4.7 may initially appear to be more complex than what we started with, it is really more useful. The left-hand side depends only on S_T, K, and W, which are all independent of how the stripes are classified. The right-hand side of the equation is expressed in terms of S_A, the number of async stripes. Using this equation, we can construct a greedy algorithm for stripe classification as described below.

We begin with all non-local stripes marked as sync stripes. This makes the right-hand side of Equation 4.7 equal to 0. Then, for each stripe i, we compute its hypothetical contribution to the right-hand side of that equation if it were instead to be classified as an async stripe.

This contribution is given by z_i:

$$z_i = v_i + u, \tag{4.8}$$

$$\text{where } v_i = K(\beta_A l_i + \gamma_A n_i), \tag{4.9}$$

$$u = \alpha_A + \kappa_A + \beta_S KW + \alpha_S, \tag{4.10}$$

where stripe i requires l_i dense rows from B and contains n_i nonzeros. The variable u depends only on the empirically-derived coefficients, K, and W, so this is constant for all stripes in the sparse matrix A.

Because the synchronous communication cost is directly proportional to number of sync stripes, we attempt to minimize S_S while meeting the constraint imposed by Equation 4.7. Equivalently, this means maximizing the number of async stripes within this constraint. Thus, we sort all the node's sparse stripes by their z_i in ascending order. We then greedily classify stripes as async stripes until we have r async stripes, where r is the greatest number that satisfies

$$S_T(\beta_S KW + \alpha_S) \geq \sum_{i=0}^{r-1} z_i. \tag{4.11}$$

The remaining stripes are classified as sync stripes.

The preprocessing algorithm is summarized as Algorithm 4.1. Each node in the system is processed individually within the loop starting at Line 3. We iterate over each of a node's non-local sparse stripes (Lines 4-5). We initially classify each stripe as *sync* (Line 6) and compute the z_i values according to Equation 4.8 (Lines 7-10). We then sort the stripes by their z_i (Line 13). Finally, we compute the limit imposed by Equation 4.7 (Line 15) and greedily classify stripes as async until the limit until that limit is reached (Lines 18-22).

It is worth noting that there may be other preprocessing schemes that are worthy of investigation, but which are not treated here. For example, the algorithm presented here does not consider whether two or more nodes both require the same data when making its decisions. Such information could potentially allow for better stripe classification. However, inter-node analysis would also increase the runtime of the preprocessor, which may limit the applicability of the algorithm as a whole. We provide an analysis of our preprocessor's overheads in Section 5.4.

Algorithm 4.1 The Two-Face preprocessing algorithm

```
1: procedure PREPROC($p$, $A$, $K$, $W$, $\beta_S$, $\alpha_S$, $\beta_A$, $\alpha_A$, $\gamma_A$, $\kappa_A$)
2:     $u \leftarrow \alpha_A + \kappa_A + \beta_S KW + \alpha_S$
3:     for $j \in 0 \ldots p$ do                    ▷ Each node in the system is processed separately.
4:         $stripes \leftarrow A$.getStripes($j$)
5:         for $i \in 0 \ldots |stripes|$ do    ▷ Compute the stripe's $z_i$ according to Equation 4.8.
6:             $stripes[i].class \leftarrow$ sync
7:             $l_i \leftarrow stripes[i]$.numNonzeroColumns()
8:             $n_i \leftarrow stripes[i]$.numNonzeros()
9:             $v_i \leftarrow K(\beta_A l_i + \gamma_A n_i)$
10:            $z_i \leftarrow v_i + u$
11:        end for
12:
13:        $stripes \leftarrow$ sort($stripes$, $z$)
14:
15:        $limit \leftarrow |stripes|(\beta_S KW + \alpha_S)$
16:        $sum \leftarrow 0$
17:        $i = 0$
18:        while $sum + z_i < limit$ do ▷ Greedily classify async stripes while Equation 4.11
    is satisfied.
19:            $stripes[i].class \leftarrow$ async
20:            $sum \leftarrow sum + z_i$
21:            $i \leftarrow i + 1$
22:        end while
23:    end for
24: end procedure
```

4.3 THE TWO-FACE ALGORITHM

Once the matrix A has been preprocessed and we know the sets of sync and async stripes, the Two-Face algorithm can compute an SpMM operation. In this section, we will discuss the details of the data structures used in Two-Face, the Two-Face algorithm itself, and the portability and applicability of Two-Face to different systems and applications.

4.3.1 Sparse Matrix Representation

Two-Face represents the sparse input matrix A using three separate data structures. The sync stripes and local-input stripes are stored together using a COO format and row-major ordering. Additionally, Two-Face maintains pointers to the beginning of each row panel to assist with locally parallelizing computation. The async stripes are stored in a separate COO structure, which additionally maintains pointers to the beginning of each stripe. This async sparse matrix stores stripes in row-major order, but stores the nonzeros within each stripe in column-major order. We store the async nonzeros in column-major order so that we can quickly iterate over them to identify the unique indices for remote accesses of B. An example of the sync/local-input and async structures can be seen in Figure 4.2.

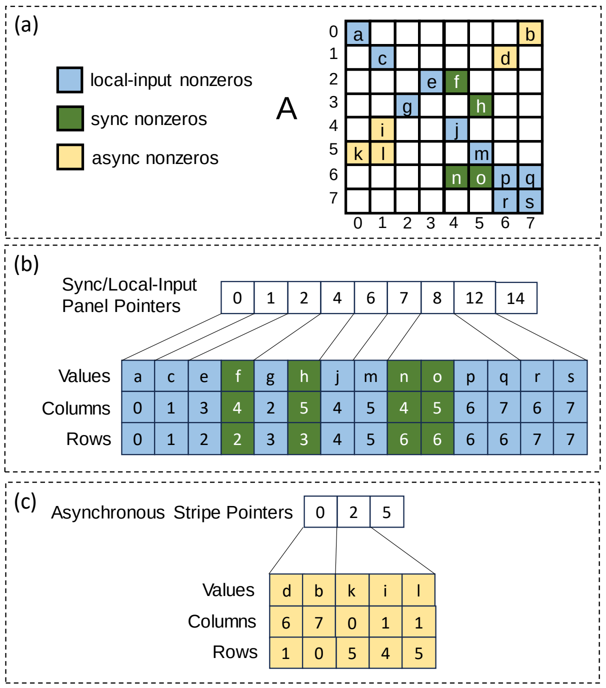

(a)
local-input nonzeros
sync nonzeros
async nonzeros
A
(b)
Sync/Local-Input Panel Pointers
0 1 2 4 6 7 8 12 14
Values: a c e f g h j m n o p q r s
Columns: 0 1 3 4 2 5 4 5 4 5 6 7 6 7
Rows: 0 1 2 2 3 3 4 5 6 6 6 6 7 7
(c)
Asynchronous Stripe Pointers
0 2 5
Values: d b k i l
Columns: 6 7 0 1 1
Rows: 1 0 5 4 5

Figure 4.2: An example of the sparse matrix representation used by Two-Face, assuming a 4-node system, $W = 2$, and a row panel height of 1. (a) An example input sparse matrix A with nonzeros classified as either local-input, sync, or async. (b) The corresponding sync/local-input sparse matrix structure. (c) The corresponding async sparse matrix structure.

4.3.2 Two-Face Algorithm Description

We describe the Two-Face algorithm in three parts: the top-level algorithm, which describes how the sync/non-local and async stripes are processed in parallel; the algorithm for computing on sync row panels; and the algorithm for communicating and computing on async stripes. Figure 4.3 graphically depicts the control flow of the algorithm. The rest of this sub-section describes the algorithm's parts in more detail.

Top-Level Algorithm Algorithm 4.2 describes the top-level control flow of the Two-Face algorithm. The algorithm begins by initializing a flag and two atomic queues for worksharing between local threads (Lines 2-3). Then, the algorithm enters a parallel region. One thread begins all the sync stripe transfers (Lines 5-8) while a separate group of threads processes

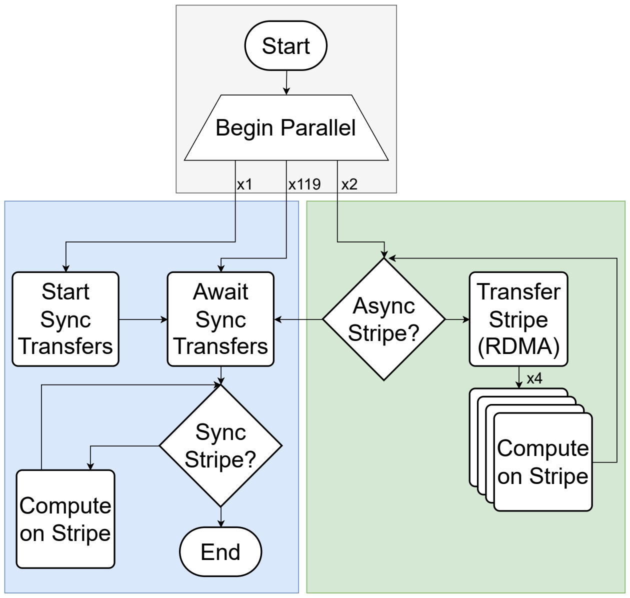


Figure 4.3: Two-Face's control flow. The *sync* processing (blue, left) and *async* processing (green, right) threads execute in parallel, motivating the cost-balancing discussed in Section 4.2.

the async stripes (Lines 9-14). Once the dense stripes from the sync transfer have been received (Line 15), computation on them begins (Lines 16-20).

Processing Synchronous Row Panels Algorithm 4.3 describes the process of computing on sync and local-input row panels. The operation starts by initializing a thread-local *Accumulation Buffer* to zero (*acc* in Line 2) and reading the row panel (*panel* in Line 3). Then, the algorithm iterates through all of the nonzeros in the row panel, accumulating each result onto *acc* (Line 10). When we either complete a row of nonzeros (Line 7) or complete the whole row panel (Line 13), we add *acc* to the corresponding row of C. Atomics are required in this operation because some threads operating on asynchronous stripes may also be writing to the same rows of C.

Processing Asynchronous Stripes Algorithm 4.4 describes the process of communicating and computing on async stripes. A thread reads the asynchronous stripe (*stripe* in Line 2) and iterates over the nonzeros in the stripe to identify the unique c_ids of the nonzeros (Line 3). These determine the indices of the dense rows from B that are required. The asynchronous thread then initiates the remote access of the dense rows by calling `GetRemoteRows` (Line 4). This procedure uses `MPI_Rget` and a custom MPI datatype defined with `MPI_Type_indexed` to select only the rows of interest for the transfer.

Algorithm 4.2 Top-Level Two-Face Pseudo-code.

```
1: procedure DistSPMM($A$, $B$, $C$)
2:     $sync\_transfer\_done \leftarrow$ False
3:     $async\_q$, $sync\_q \leftarrow InitQueues(A)$
4:     DoParallel
5:         if $tid = 0$ then                                  ▷ Sync Transfers
6:             TransferDenseStripes($A$, $B$)
7:             $sync\_transfer\_done \leftarrow$ True
8:         end if
9:         if $tid \in AsyncThreads$ then                    ▷ Async Processing
10:            while $async\_q$.nonempty() do
11:                $n \leftarrow async\_q$.pop()
12:                ProcessAsyncStripe($A$, $B$, $C$, $n$)
13:            end while
14:        end if
15:        WaitForFlag($sync\_transfer\_done$)
16:        while $sync\_q$.nonempty() do                      ▷ Sync Compute
17:            $n \leftarrow sync\_q$.pop()
18:            ProcessSyncRowPanel($A$, $B$, $C$, $n$)
19:        end while
20:    EndParallel
21: end procedure
```

Once the dense rows arrive, they are stored in *drows* (Line 4), and multiple threads begin computing on them. Each thread processes a subset of the nonzeros in the sparse stripe. Each nonzero is multiplied with the corresponding row of *drows* and accumulated into C (Line 6). Atomics are required for correct accumulation into C, just as in the synchronous stripe case. However, since asynchronous nonzeros are stored in column-major order, we cannot easily use thread-local buffers to reduce the number of atomics.

To reduce transfer overheads, inside the `GetRemoteRows` routine, we coalesce the transfer of nearby rows of B. For example, if a sparse stripe requires B rows $\{2, 3, 6, 8\}$, we transfer three groups of rows, with $(offset, size)$ pairs equal to $\{(2, 2), (6, 1), (8, 1)\}$. This optimization reduces software overheads. For small K, we also coalesce rows separated by unused rows, potentially reducing the software overhead further, but transferring some useless data. Using the example from before, we might transfer groups of rows $\{(2, 2), (6, 3)\}$, retrieving one unnecessary row (row 7).

Algorithm 4.3 Two-Face Sync Compute Pseudo-code

```
 1: procedure ProcessSyncRowPanel($A$, $B$, $C$, $n$)
 2:     $acc \leftarrow \{0, ..., 0\}$                                  ▷ Output row buffer
 3:     $panel \leftarrow A.panel\_ptrs[n]$
 4:     $prev\_row \leftarrow panel[0].row$                             ▷ Initialize to first row
 5:     for $nz \in panel$ do
 6:         if $nz.row \neq prev\_row$ then
 7:             AtomicAdd($C[prev\_row]$, $acc$)
 8:             $acc \leftarrow \{0, ..., 0\}$
 9:         end if
10:         $acc \leftarrow acc + nz.val * B[nz.col]$
11:         $prev\_row \leftarrow nz.row$
12:     end for
13:     AtomicAdd($C[prev\_row]$, $acc$)
14: end procedure
```

Algorithm 4.4 Two-Face Async Pseudo-code

```
1: procedure ProcessAsyncStripe($A$, $B$, $C$, $n$)
2:     $stripe \leftarrow A.async\_stripe\_ptrs[n]$
3:     $drow\_ids \leftarrow stripe$.UniqueColIDs()
4:     $drows \leftarrow$ GetRemoteRows($drow\_ids$)
5:     for $nz \in stripe$ do in parallel
6:         AtomicAdd($C[nz.row]$, $nz.val * drows[nz.col]$)
7:     end for
8: end procedure
```

4.3.3 Portability

Two-Face has several parameters that may need to be calibrated for each individual system to achieve maximal performance. Among these are the coefficients used in the preprocessing cost model (β_S, α_S, β_A, α_A, κ_A, γ_A). As mentioned before, in our evaluation, we determine the values of these coefficients via linear regression on a small number of workloads. These parameters only need to be calibrated once for a system, possibly at installation time.

In addition to the preprocessing cost model coefficients, the runtime algorithm is parameterized by the number of threads assigned to sync/async stripe processing, the aggressiveness of row coalescing in async stripe transfers, the height of the row panels used for computation in the sync stripes, and the width of the stripes. The optimal choice for these parameters may vary between systems and workloads, but we show in Chapter 5 that choosing reasonable, static values can provide good performance. In practice, these parameters could be determined at installation time similarly to the preprocessing coefficients.

Thus, although *Two-Face* relies on knowledge of system characteristics to make decisions about how to schedule the work, porting to a new system just requires a one-time profiling step during installation.

4.3.4 Applicability to GNN Training

While SpMM is used in a variety of domains, one of the most important ones is GNN training. GNN training is often done in relatively small-scale systems, where the amount of memory is a limitation. To alleviate this problem, GNN training algorithms have recently resorted to the use of sampling [35] and mini-batching. While these techniques reduce the memory footprint requirements, they may introduce some inaccuracy [35, 36], and may introduce runtime overhead which can sometimes increase the end-to-end execution time [54]. In *Two-Face*, like in most of the prior work in distributed GNN training [36, 37], we are less concerned about the lack of memory because we can use the full aggregate memory of a very large cluster [37]. As a result, we do not consider sampling or mini-batching and, instead, support full-graph GNN training.

It is interesting, however, to consider the application of *Two-Face* to an environment with sampling or mini-batching. In principle, in its current form, *Two-Face* is incompatible with sampling or mini-batching. This is because, in sampling or mini-batching algorithms, different iterations of the SpMM computation use a different reduced (or sampled) matrix. As a result, *Two-Face* would have to re-run the preprocessing step every time the reduced matrix changes.

Future work may involve adapting *Two-Face* to apply to GNNs with sampling. One possible approach may involve making preprocessing decisions offline once, based on the expected stripes' densities, given knowledge of the sampling to be done at runtime. Then, stripes that are expected to be dense enough even after sampling would still be classified as synchronous, and the other stripes would be classified as asynchronous. At runtime, the graph would still be stored as shown in Figure 4.2, but with the addition of masks to filter nonzeros eliminated by the sampling at each iteration.

In current full-graph GNN training [55, 56], the preprocessing cost can be easily amortized. We quantify the exact cost of the preprocessing step in Section 5.4. In GNN training, the same sparse matrix is used for hundreds or even thousands of SpMM iterations. Additionally, in many GNN applications, the same graph is used for both training and inference. This is the case in most GNNs for semi-supervised node classification applications [55, 56]. Since the sparse matrix does not change, the preprocessing done in training can be reused for inference. Thus, the overhead of *Two-Face* preprocessing in full-graph GNN training is negligible.

CHAPTER 5: EVALUATION OF TWO-FACE

In this chapter, we discuss our evaluation of the Two-Face algorithm's performance in a distributed system. First, we discuss our methodology, including our system configuration, parameterization of the algorithm, and the baselines for our comparison. Then, we present the results of our evaluations, including runtime comparisons against our baselines, a breakdown of the algorithm's runtime, and a scaling analysis. We finally conclude with a discussion of the preprocessing cost and the preprocessor's sensitivity to the system's characterization.

5.1 METHODOLOGY

Here, we describe our evaluation methodology. We begin with details about the hardware configuration and software libraries used to evaluate *Two-Face*, as well as the sparse matrices used as benchmarks. Then, we discuss how we determined the values of various parameters. Finally, we describe the other algorithms which we use as baselines to compare to *Two-Face*.

5.1.1 Overview & Workloads

We evaluate *Two-Face* and multiple baseline algorithms using large matrices on Delta [16], a supercomputer at the National Center for Supercomputing Applications (NCSA). We use up to 64 CPU nodes, with a default of 32 CPU nodes. Each Delta node is a dual-socket system with two 64-core AMD EPYC 7763 processor chips running at 2.45 GHz and a total of 256 GiB of DRAM. The nodes are connected through a Cray Slingshot interconnect [15].

We build on the code published by Bharadwaj et al. [8], adapting it as necessary to support our algorithms and larger matrices. We use hybrid OpenMP / MPI programming, with one MPI rank and 128 OpenMP threads per node. We use OpenMP 4.5 [57] and Open MPI 4.1.2 [13, 14] with UCX 1.11.2 [58]. All of the baseline algorithms use the Intel Math Kernel Library [59] (version 2022.0.2) for local SpMM computations. These baselines also rely on CombBLAS [19] for I/O. Our implementation of *Two-Face* handles I/O by way of custom data loaders for our preprocessed sparse matrix format. All algorithms used in these experiments make use of Eigen [60] for handling dense matrices locally.

We use eight large sparse matrices from SuiteSparse [52], described in Table 5.1. These matrices are derived from a variety of domains, including internet traffic, social networks, web crawls, and scientific applications.

Our evaluation in Section 5.2 supports the claim that distributed SpMM is typically a

Table 5.1: Matrices used in the evaluation. All matrices are among the largest in SuiteSparse [52] and are square. Stripe widths are chosen to scale with the number of columns.

Matrix Name		**# Rows**	**# Nonzeros**	**Stripe**
Long	Short	**(Million)**	**(Million)**	**Width**
mawi_201512020030	mawi	68.86	143.41	128K
Queen_4147	queen	4.15	316.55	8K
stokes	stokes	11.45	349.32	32K
kmer_V1r	kmer	214.01	465.41	512K
arabic-2005	arabic	22.74	640.00	64K
twitter7	twitter	41.65	1,468.37	128K
GAP-web	web	50.64	1,930.29	128K
com-Friendster	friendster	65.61	3,612.13	128K

communication-bound workload. Thus, we expect that extending *Two-Face* to other computing hardware would provide similar results. For example, using GPUs in the nodes may accelerate the local computation, but communication will remain a bottleneck. Thus, we expect that *Two-Face* will still see speedups if used with GPUs. Here, we evaluate a CPU implementation.

5.1.2 Two-Face Parameterization

Two-Face is a parameterizable algorithm. To determine its parameters for our system, we analyzed several combinations of parameters on a small set of workloads.

To determine the appropriate width of stripes, we analyzed the performance of SpMM using the queen, arabic, and twitter matrices with various choices for W. There was increasing overhead in both the preprocessing and runtime steps as the number of stripes grew, suggesting that the stripe width should not be made too small, relative to the size of the matrix. We decided to scale the stripe width proportionally to the dimensions of the matrices, rounding to the nearest power of two. Table 5.1 shows the stripe widths we chose.

All run-time parameters other than the stripe width are held constant across matrices. Table 5.2 shows these parameters. Each node runs 128 OpenMP threads. Since a large number of one-sided transfers results in high resource contention, we limit the number of threads communicating asynchronous data to 2 per node. We allow each of these threads to fork up to four ways (for a total of 8 threads) when computing on the asynchronous stripes. We dedicate the remaining 120 threads in the node to computation on the synchronous and local-input stripes. We define the maximum row coalescing distance for asynchronous transfers to be proportional to $\frac{1}{K}$, since the cost of transferring unnecessary dense rows grows with K.

Table 5.2: Constant runtime parameters used in *Two-Face*.

Parameter Name	**Value**
Async Communication Threads per Node	2
Async Computation Threads per Node	8
Sync/Local-Input Computation Threads per Node	120
Max Async Coalescing Distance	$(127/K)+1$
Row Panel Height of Sync/Local-Input Sparse Matrix	32 rows

To determine the values of the preprocessing parameters used in stripe classification (Section 4.2), we employ linear regression [53]. We collect data by processing the twitter matrix [61] using $K = 32$, $p = 32$, and nine different combinations of stripe widths and asynchronous/synchronous stripe classifications. The number of samples is kept small to ensure that it is reasonable to calibrate these coefficients when installing *Two-Face* on a new system. The derived coefficient values, which we use when preprocessing all matrices in our evaluation (unless otherwise specified), are shown in Table 5.3.

Table 5.3: Coefficient values used in the preprocessing of matrices. The β parameters relate to the system bandwidth, the α parameters relate to other communication overheads, and the γ & κ terms relate to computational throughput and other overheads.

Coefficient	**Experimental Value**
β_S	1.95×10^{-10}
α_S	1.36×10^{-6}
β_A	3.61×10^{-9}
α_A	1.02×10^{-5}
γ_A	2.07×10^{-8}
κ_A	8.72×10^{-9}

These coefficients provide some insight into the performance difference between one-sided asynchronous and collective synchronous communication. For example, they suggest that asynchronous transfers are more expensive per transferred element of B than synchronous transfers by a factor of $\beta_A/\beta_S \approx 18.5$.

In Section 5.4.2 of our evaluation, we evaluate the impact of different values of these coefficients.

5.1.3 Algorithms Evaluated

In our evaluation, we compare *Two-Face* to other algorithms shown in Table 5.4. All the algorithms use 1D partitioning. We divide the dense input matrix B into as many equally-sized portions as the number of nodes p, and call each portion a "block". The B matrix is distributed across all nodes, where each node stores a single block.

Table 5.4: SpMM algorithms being compared.

Algorithm Name	**MPI Transfer Operations**
Dense Shifting [8]	`MPI_Allgather`, `MPI_Sendrecv`
Allgather	`MPI_Allgather`
Async Coarse-Grained	`MPI_Get`
Two-Face	`MPI_Rget`, `MPI_Ibcast`
Async Fine-Grained	`MPI_Rget`

Dense Shifting (DS) is a synchronous SpMM algorithm that has been investigated by Bharadwaj et al. [8] and found to be highly competitive compared to other state-of-the-art implementations. We use it as our main baseline. DS begins by using `MPI_Allgather` to replicate a certain number of blocks in each node, as determined by a replication factor c. It then continues by shifting the replicated blocks cyclically via `MPI_Sendrecv` after each computation step. For instance, with $c = 4$, this algorithm replicates each block such that each node holds four blocks at a time. It then performs p/c computation and shifting steps to complete the SpMM operation. In our experiments, we evaluate this algorithm for $c = 2$, $c = 4$, and $c = 8$, and refer to these settings as DS2, DS4, and DS8, respectively.

The next two algorithms replicate all or nearly all of the matrix B before beginning the computation. In *Allgather*, each node uses `MPI_Allgather` to broadcast its block of B to all others and receive theirs in turn. In *Asynchronous Coarse-Grained*, each node uses `MPI_Get` to obtain the blocks that it needs for its computation. In both cases, substantial memory has to be allocated, creating issues as the problem size scales.

Two-Face is the algorithm we propose. We use the parameters as described before. However, if the preprocessing algorithm determines that the chosen sync/async classification of stripes would result in too much memory consumption in one or more nodes during SpMM execution, it will classify additional stripes as async until the expected memory consumption in those nodes is feasible.

Asynchronous Fine-Grained is implemented in the same way as *Two-Face*, except that all stripes are asynchronous. This algorithm is used as an extreme example to illustrate the tradeoffs made by a balanced *Two-Face* implementation. This baseline was used in Section 3.3.

All algorithms are evaluated by averaging out the time of 5 consecutive SpMM operations. By default, our experiments use $p = 32$ and $K = 128$. Some experiments use $K = 32$ or $K = 256$, and others use $p = 1$, 2, 4, 8, 16, 32, or 64.

5.2 BASELINE COMPARISON

Figures 5.1, 5.2, and 5.3 show the speedups of *Two-Face* and the other SpMM algorithms over DS2 for $K = 32$, $K = 128$, and $K = 512$, respectively. We normalize to DS2 because, unlike DS4 or DS8, DS2 does not run out of memory for any matrices or value of K in our evaluation. From the figures, we see that, on average, across matrices and K values, *Two-Face* is the fastest algorithm, and delivers substantial speedups.

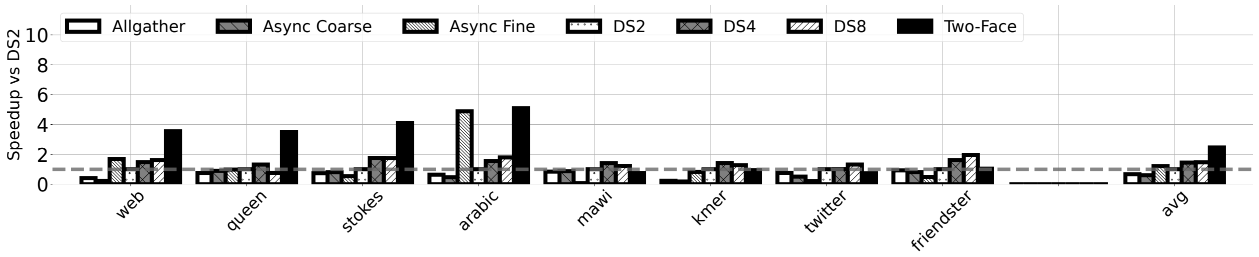


Figure 5.1: Speedups of various SpMM algorithms over DS2 for $K = 32$.

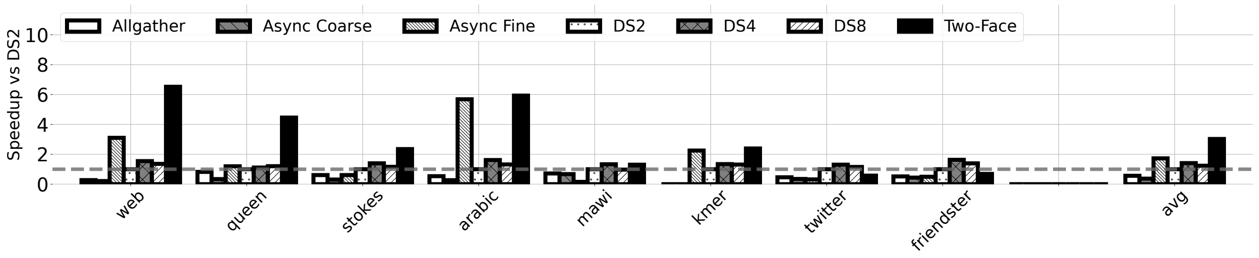


Figure 5.2: Speedups of various SpMM algorithms over DS2 for $K = 128$.

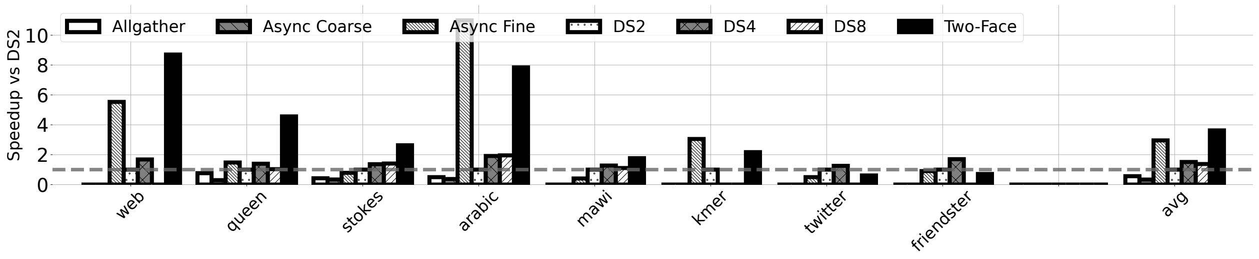


Figure 5.3: Speedups of various SpMM algorithms over DS2 for $K = 512$.

As K increases, the advantage of *Two-Face* over the dense shifting algorithms becomes more prominent. This is because the cost of transferring unnecessary rows in the dense shifting algorithms increases with K, providing a greater advantage to the fine-grained one-sided accesses of *Two-Face*. At $K = 32$, *Two-Face*'s average speedup over the dense shifting algorithm with the best choice of replication factor for each individual matrix is 1.53x. At $K = 128$, the same speedup is 2.11x, and at $K = 512$, is it 2.35x. The average speedup across all values of K shown here is 1.99x.

The *Async Fine* and dense shifting algorithms are on average faster than the *Async Coarse*

Table 5.5: Absolute execution times of DS2 and *Two-Face* for the experiments in Figures 5.1, 5.2, and 5.3. The numbers are the average of five SpMM operations. All times are given in seconds.

K		**web**	**queen**	**stokes**	**arabic**	**mawi**	**kmer**	**twitter**	**friendster**
32	DS2	1.97	0.28	0.96	1.32	6.46	6.33	4.17	8.79
	Two-Face	0.56	0.08	0.23	0.26	8.50	6.70	5.71	8.41
128	DS2	7.20	0.86	2.22	3.85	19.78	35.77	11.57	20.61
	Two-Face	1.10	0.19	0.94	0.65	15.18	14.98	20.24	30.08
512	DS2	38.86	2.89	9.34	21.46	97.95	136.21	52.77	83.02
	Two-Face	4.46	0.634	3.552	2.74	55.40	62.77	86.62	117.31

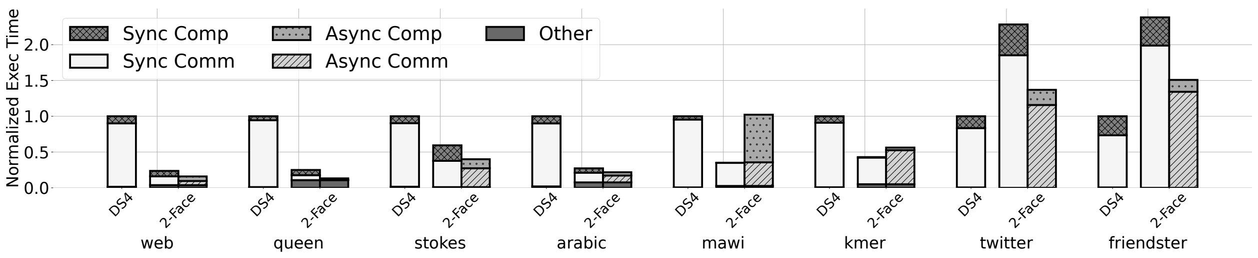


Figure 5.4: Breakdown of the total execution times of DS4 and *Two-Face* for $K = 128$. *Two-Face*'s time is divided into synchronous and asynchronous components (left and right bars, respectively), which operate in parallel. These are further broken down into computation (*Comp*) and communication (*Comm*). DS4 only has a Sync component. The *Other* category mainly consists of the initial setup of data structures for MPI. Execution times are normalized to DS4.

and *Allgather* algorithms. Dense shifting is sometimes unable to run with higher replication factors due to memory constraints. For example, for $K = 512$, DS8 fails to run for half of the matrices, and DS4 fails in one matrix. As a reference, Table 5.5 provides the absolute execution times of *Two-Face* and DS2 in these figures.

The figures also show that the speedups (or slowdowns) are highly dependent on the matrix. For example, *Two-Face* is not the fastest algorithm for twitter and friendster and, for $K = 32$, additionally for mawi and kmer. To understand this behavior, Figure 5.4 breaks down the total execution time of DS4 and *Two-Face* for each matrix for $K = 128$. For *Two-Face*, we break down the execution time into *Sync Comp*, *Sync Comm*, *Async Comp*, and *Async Comm*. We stack the Sync components in the left bar and the Async components in the right bar, and show both bars side-to-side, since the execution time is equal to the highest of the two bars. *Two-Face* also has some *Other* overheads, which mainly consist of initializing necessary MPI structures before the main communication/computation begins. For DS4, only *Sync Comp* and *Sync Comm* are relevant. For each matrix, the bars are normalized to DS4.

We see that the dominant contributor to DS4's execution time is its communication. *Two-Face* is able to attain significant speedups over DS4 by reducing the amount of communication through fine-grained accesses. In five of the matrices, we can see that the sum of the communication time spent by *Two-Face* in *Sync Comm* and *Async Comm* is significantly less than the amount of time spent by DS4 in its communication.

Two exceptions are twitter and friendster. In these matrices, *Two-Face*'s *Sync Comp* and *Sync Comm* have both increased over DS4, despite the fact that less data is being transferred. We note that *Two-Face*'s synchronous broadcast operations are significantly slower than the cyclic shifting operations in DS4 when a large portion of the input dense matrix is required by many nodes. When *Two-Face* operates on a matrix like friendster, each node participates in many more MPI calls than it does if dense shifting is used, due to the finer granularity of the transfers.

An interesting case is mawi, where *Two-Face* is unable to reduce the execution time over DS4 because of the cost of asynchronous computation. The mawi sparse matrix has regions that have a relatively high density of nonzeros. Computing on such asynchronous stripes is likely expensive due to the heavy use of atomics, as the nonzeros are organized in column-major order. During this work, we conducted initial tests into storing the nonzeros in row-major order instead. However, this change did not result in faster execution, as the cost of identifying which columns contained nonzeros (and therefore which dense rows were required) became drastically higher.

5.3 SCALING

Figure 5.5 shows the execution times of *Two-Face* and the dense shifting algorithm with different replication factors (DS1, DS2, DS4, and DS8) as we scale the number of nodes from 1 to 64. There is a plot for each matrix and both axes are in logarithmic scale. Some data points are missing, since some workloads either exceed the memory capacity of one or more nodes (at small node counts or high replication factors) or take too long to run.

The figure shows that, in most of the matrices, *Two-Face* scales well with the number of nodes and, in fact, as well or better than the dense shifting algorithm. The exceptions are mawi, twitter, and, to a lesser extent, friendster. With mawi, none of the algorithms scale particularly well due to the high load imbalance across nodes induced by the matrix. With twitter and friendster, we saw in Figure 5.4 that *Two-Face* is impacted by inefficient synchronous communication. This is the reason for the worse scaling performance.

To understand the behavior of twitter and friendster better, we profile the collectives in

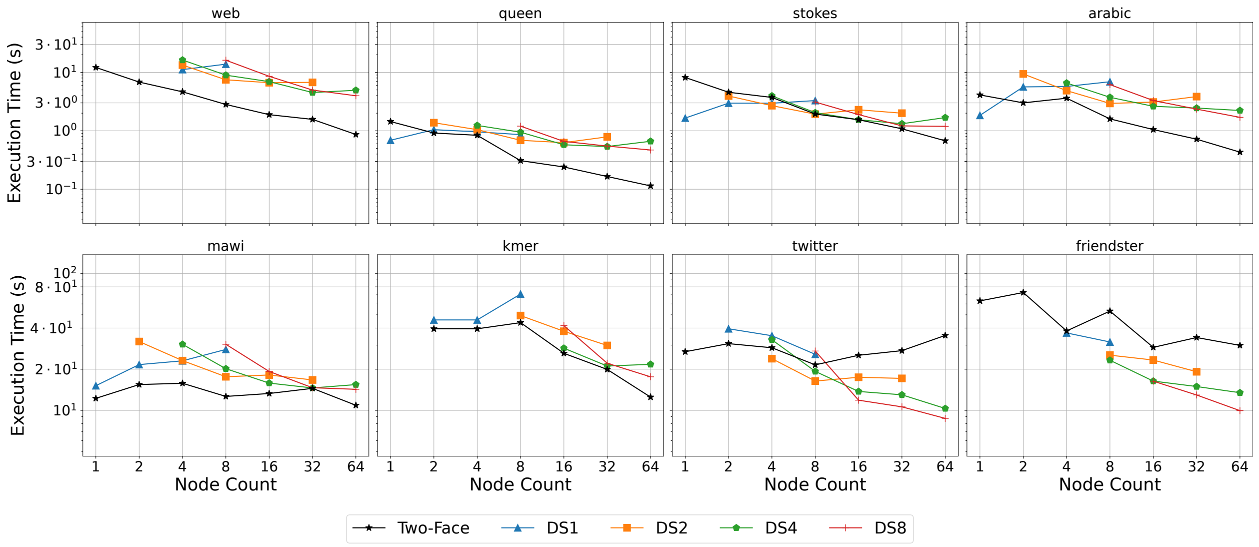


Figure 5.5: Execution time of *Two-Face* and the dense shifting algorithm with different replication factors (DS1, DS2, DS4, and DS8) as the number of nodes changes. The K value is 128. Some data points for the dense shifting algorithm are missing because they need too much memory or take too long to execute. Both axes in the plots use a logarithmic scale.

the 64-node runs. We measure the number of recipients of each multicast operation. On average, this number is 35.7 for twitter and 43.5 for friendster. In contrast, the matrix with the next largest average recipient count is kmer, with an average of only 5.7. It appears that the large collectives needed for *Two-Face* in twitter and friendster are responsible for the inefficient execution and limited scaling. This effect does not appear at low node counts, where the execution is primarily bottlenecked by local computation, but it dominates at high node counts. Future work should investigate methods to reduce the size of collectives in the algorithm or the design of more regular data movement patterns for the synchronous stripes.

Overall, the performance of *Two-Face* improves as we scale from 1 to 64 nodes by 7.47x on average, with a best-case speedup of 12.12x for queen and a worst-case of 0.76x for twitter. Moreover, compared to the dense shifting algorithm with the optimal replication factor, *Two-Face* sees an average speedup ranging from 1.25x at 4 nodes to 2.21x at 64 nodes.

5.4 PREPROCESSING

Two-Face requires a preprocessing step that involves, mainly: (1) running our model to classify the stripes into synchronous and asynchronous, and (2) creating the asynchronous and the synchronous/local-input sparse matrices. In this section, we give an idea of the execution time of the preprocessing step. Additionally, we provide some evaluation of the sensitivity of Two-Face to the preprocessing coefficients determined by system characterization (namely, β_S, α_S, β_A, α_A, γ_A, and κ_A).

Table 5.6: The overhead of preprocessing in *Two-Face*, normalized to the cost of a single SpMM operation for $K = 128$.

Matrix	$t_{norm\_I/O}$	t_{norm}
web	428.74	102.00
queen	302.55	23.60
stokes	116.70	11.18
arabic	180.35	36.57
mawi	2.58	1.50
kmer	6.16	3.25
twitter	17.89	7.29
friendster	19.81	8.79
Average	134.35	24.27

5.4.1 Preprocessing Cost

Two-Face's preprocessing step consitutes some additional overhead to the algorithm, which can be amortized by reusing the preprocessing analysis for multiple operations with a particular sparse matrix. Here, we provide details about the execution time of the preprocessor. Note that we have not fully optimized it; in particular, we have not parallelized it across multiple nodes. Therefore, the numbers reported are a pessimistic bound.

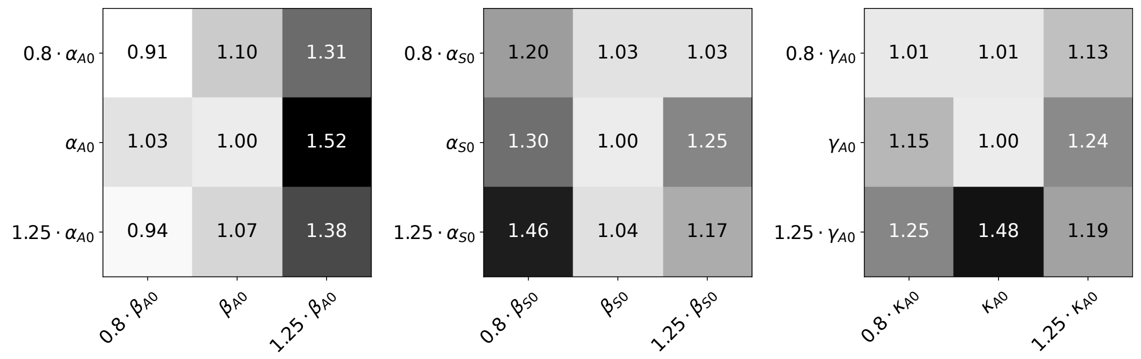
Table 5.6 shows the overhead of the preprocessing step for 32 nodes and K=128 for each matrix. Column 2 shows $t_{norm\_I/O}$, which is the time of the preprocessing step normalized to the time of one SpMM operation. On average, $t_{norm\_I/O}$ is 134.35. However, the preprocessing step is dominated by I/O time, as the original sparse matrix is read from the file system in a textual Matrix Market format [62] and the final asynchronous and synchronous/local-input sparse matrices are written to the file system in a bespoke binary format.

Since in many realistic environments, this I/O will not be present, Column 3 shows the more relevant t_{norm}, which is the preprocessing overhead without I/O normalized to the time of one SpMM operation. In this case, the numbers have reduced substantially. We see that t_{norm} ranges from 1.50 to 102.00, with an average of 24.27. For $K = 512$ (not shown in the table), the average value of t_{norm} is 6.15.

From these numbers, we see that the cost of the preprocessing step can be easily amortized. For the matrices where *Two-Face* demonstrates a speedup over dense shifting when $K = 128$, an average of only 15 SpMM operations need to be performed by *Two-Face* to already see a speedup when including preprocessing time. For $K = 512$, this decreases to only 3 SpMM operations, on average. In contexts such as GNN training, with hundreds of epochs, we can expect to perform many more SpMM operations with the same matrices than these numbers. In addition, the preprocessing step from training may be reusable during inference if the same graph is used in both steps.

5.4.2 Preprocessing Sensitivity

The model of execution that we use during preprocessing (Section 4.2) uses parameters α_A, β_A, α_S, β_S, γ_A, and κ_A. In Section 5.1.2, we used linear regression to set their default values. We used such values in all the experiments so far.



(a) Varying α_A and β_A (b) Varying α_S and β_S (c) Varying γ_A and κ_A

Figure 5.6: Sensitivity of *Two-Face*'s execution time to the values of the parameters of the execution model used during the preprocessing step. The default values of these parameters, as set in Section 5.1.2 and used in all the earlier experiments, are represented with the 0 subscript (e.g., α_{A0}).

In this section, we change the values of these parameters, repeat the experiments, and measure the changes in *Two-Face*'s execution time. We perform three sets of changes. In the first one, we vary α_A and β_A, keeping the other parameters unchanged. Specifically, if α_{A0} and β_{A0} are the default values of α_A and β_A, we consider all combinations of $\{0.8 \cdot \alpha_{A0},\ \alpha_{A0},\ 1.25 \cdot \alpha_{A0}\} \times \{0.8 \cdot \beta_{A0},\ \beta_{A0},\ 1.25 \cdot \beta_{A0}\}$. In the second set of changes, we vary α_S and β_S in the same way, keeping the other parameters unchanged. Finally, we vary γ_A and κ_A, again keeping the others unchanged.

Figure 5.6 shows the outcome of the three sets of changes for the average of three representative matrices: web (*Two-Face*'s best case), twitter (*Two-Face*'s worst case), and stokes (*Two-Face*'s median case). For example, Figure 5.6a corresponds to the experiments varying α_A and β_A. The number in each box is *Two-Face*'s execution time with the new parameters relative to *Two-Face*'s execution time with the default parameters. For example, if we use

$0.8 \cdot \alpha_{A0}$ and $1.25 \cdot \beta_{A0}$, the *Two-Face*'s execution time becomes 1.31x higher than when using the default values.

Overall, the figure shows that using the default parameters obtained using linear regression is a good choice. Changes to the parameter values typically end up increasing *Two-Face*'s execution time. The execution time decreases in only two cases, and the decrease is small.

CHAPTER 6: CONCLUSION

Sparse matrices often contain regions that are denser and regions that are sparser. Based on the observation, we introduced the *Two-Face* [11] algorithm for distributed SpMM, which, leveraging a preprocessing model, performs coarse-grained collective communications for the denser regions, and fine-grained one-sided communications for the sparser regions. *Two-Face* attains an average speedup of 2.11x over dense shifting when evaluated on a 4096-core supercomputer. Additionally, *Two-Face* scales well with the machine size.

The performance improvements that we observe with *Two-Face* suggests that distributed sparse algorithms should be input-matrix aware, since different sections of a sparse input matrix prefer using different communication methods. Such algorithms should also be communication-oriented, since minimizing communication is a first-class concern.

With simple modifications, the *Two-Face* algorithm should also be applicable to sparse kernels such as Sampled Dense-Dense Matrix Multiplication (SDDMM), which exhibits very similar patterns to SpMM. Likewise, with proper parameter tuning, *Two-Face* may also be applicable to accelerate SpMV, which is a special case of SpMM.

REFERENCES

[1] J. Canny and H. Zhao, "Bidmach: Large-scale learning with zero memory allocation," in *BigLearning, NIPS Workshop*, 2013.

[2] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–12.

[3] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "Featgraph: A flexible and efficient backend for graph neural network systems," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[4] R. Fan, W. Wang, and X. Chu, "Fast sparse gpu kernels for accelerated training of graph neural networks," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 501–511.

[5] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[6] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep Graph Library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[7] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc Web page," https://petsc.org/, 2023. [Online]. Available: https://petsc.org/

[8] V. Bharadwaj, A. Buluc, and J. Demmel, "Distributed-memory sparse kernels for machine learning," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 6 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00014 pp. 47–58.

[9] 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. [Online]. Available: https://doi.org/10.1145/3447818.3461472 p. 431–442.

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

[11] C. Block, G. Gerogiannis, C. Mendis, A. Azad, and J. Torrellas, "Two-face: Combining collective and one-sided communication for efficient distributed spmm," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating System*, ser. ASPLOS '24, vol. 2, New York, NY, USA, 2024.

[12] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276493

[13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, 6 2021, https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[14] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 9 2004, pp. 97–104.

[15] H. P. Enterprise, "Hpe slingshot interconnect," 2024, Hewlett Packard Enterprise. [Online]. Available: www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html

[16] B. Abreu, G. Arnold, G. Bauer, B. Bode, C. Steffan et al., "Delta user documentation," 1 2024, National Center for supercomputing Applications. [Online]. Available: https://docs.ncsa.illinois.edu/systems/delta/en/latest/

[17] I. Red Hat, "Red hat enterprise linux operating system," 2024, Red Hat, Inc. [Online]. Available: https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[19] 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, no. 4, pp. 989–1001, 2022.

[20] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, "Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 90–100.

[21] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447786.3456233 p. 130–144.

[22] J. Won, C. Mendis, J. S. Emer, and S. Amarasinghe, "Waco: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3575693.3575742 p. 920–934.

[23] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Wise: Predicting the performance of sparse matrix vector multiplication with machine learning," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3572848.3577506 p. 329–341.

[24] S. Yesil, J. E. Moreira, and J. Torrellas, "Dense dynamic blocks: Optimizing spmm for processors with vector and matrix units using machine learning techniques," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3524059.3532369

[25] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3293883.3295712 p. 300–314.

[26] NVIDIA, "cusparse," 2024. [Online]. Available: https://developer.nvidia.com/cusparse

[27] S. Li, K. Osawa, and T. Hoefler, "Efficient quantized sparse matrix operations on tensor cores," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–15.

[28] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3350755.3400216 p. 293–303.

[29] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3582016.3582051 p. 710–726.

[30] H. D. Tran, M. Fernando, K. Saurabh, B. Ganapathysubramanian, R. M. Kirby, and H. Sundar, "A scalable adaptive-matrix spmv for heterogeneous architectures," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 13–24.

[31] T. O. Odemuyiwa, H. Asghari-Moghaddam, M. Pellauer, K. Hegde, P.-A. Tsai, N. C. Crago, A. Jaleel, J. D. Owens, E. Solomonik, J. S. Emer, and C. W. Fletcher, "Accelerating sparse data orchestration via dynamic reflexive tiling," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3582016.3582064 p. 18–32.

[32] H. Cheng, W. Li, Y. Lu, and W. Liu, "Haspgemm: Heterogeneity-aware sparse general matrix-matrix multiplication on modern asymmetric multicore processors," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3605573.3605611 p. 807–817.

[33] W. Li, H. Cheng, Z. Lu, Y. Lu, and W. Liu, "Haspmv: Heterogeneity-aware sparse matrix-vector multiplication on modern asymmetric multicore processors," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2023, pp. 209–220.

[34] G. Gerogiannis, S. Aananthakrishnan, J. Torrellas, and I. Hur, "Hottiles: Accelerating spmm with heterogeneous accelerator architectures," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024.

[35] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[36] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.

[37] A. Tripathy, K. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[38] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, p. 297–310, 11 2017. [Online]. Available: https://doi.org/10.14778/3157794.3157799

[39] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, "Spade: A flexible and scalable accelerator for spmm and sddmm," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589054

[40] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 689–702.

[41] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358275 p. 319–333.

[42] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but mighty: designing and realizing scalable latency tolerance for manycore socs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 817–830.

[43] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, 2019, pp. 600–614.

[44] S. Aananthakrishnan, S. Abedin, V. Cavé, F. Checconi, K. Du Bois, S. Eyerman, J. B. Fryman, W. Heirman, J. Howard, I. Hur, S. Jain, M. M. Landowski, K. Ma, J. Nelson, R. Pawlowski, F. Petrini, S. Szkoda, S. Tayal, J. J. Tithi, and Y. Vandriessche, "The intel® programmable and integrated unified memory architecture (piuma) graph analytics processor," *IEEE Micro*, pp. 1–11, 2023.

[45] M. J. Adiletta, J. J. Tithi, E.-I. Farsarakis, G. Gerogiannis, R. Adolf, R. Benke, S. Kashyap, S. Hsia, K. Lakhotia, F. Petrini, G.-Y. Wei, and D. Brooks, "Characterizing the scalability of graph convolutional networks on intel® piuma," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 168–177.

[46] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A data-local program execution and architecture for memory-bound applications," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 718–730.

[47] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[48] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[49] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.

[50] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–49, 2022.

[51] S. Rashidi, W. Won, S. Srinivasan, S. Sridharan, and T. Krishna, "Themis: A network bandwidth-aware collective scheduling policy for distributed training of dl models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3470496.3527382 p. 581–596.

[52] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 12 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[53] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis.* John Wiley & Sons, 2021.

[54] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Graphite: Optimizing graph neural networks on cpus through cooperative software-hardware techniques," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 916–931.

[55] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[56] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[57] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," https://openmp.org/wp-content/uploads/openmp-4.5.pdf, 11 2015.

[58] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss et al., "Ucx: an open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[59] I. Corporation, "Intel® oneapi math kernel library," 2023, Intel Corporation. [Online]. Available: https://intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[60] E. Project, "Eigen v3.4," 2023. [Online]. Available: https://eigen.tuxfamily.org

[61] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proc. the 19th Intl. Conf. on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[62] R. Boisvert, R. Pozo, and K. Remington, "The matrix market exchange formats: Initial design," 12 1996.