Two-Face: Combining Collective and One-Sided Communication for Efficient Distributed SpMM

Charles Block*
University of Illinois at Urbana-Champaign, IL, USA
coblock2@illinois.edu

Gerasimos Gerogiannis*
University of Illinois at Urbana-Champaign, IL, USA
gg24@illinois.edu

Charith Mendis
University of Illinois at Urbana-Champaign, IL, USA
charithm@illinois.edu

Ariful Azad
Indiana University Bloomington, IN, USA
azad@iu.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, IL, USA
torrella@illinois.edu

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.

CCS Concepts: • **Computing methodologies** → **Distributed algorithms**.

*Both authors contributed equally to this research. Order is alphabetical.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640427

Keywords: High-performance computing, distributed algorithms, sparse matrices, SpMM

ACM Reference Format:
Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. 2024. 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 Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620665.3640427

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 [11]. It is the bottleneck primitive in various Graph Neural Networks [17, 29, 30] and an integral part of popular graph learning frameworks such as PyTorch Geometric (PyG) [18] and Deep Graph Library (DGL) [55].

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, 48]. 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 [8].

Communication can be scheduled in a sparsity-unaware manner, such as with 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 [48] 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 [3], 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].

In this work, we observe that individual sparse matrices often contain regions that are relatively denser and regions that are relatively sparser. Based on this observation, we develop 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*. 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.

Overall, this paper's contributions are:

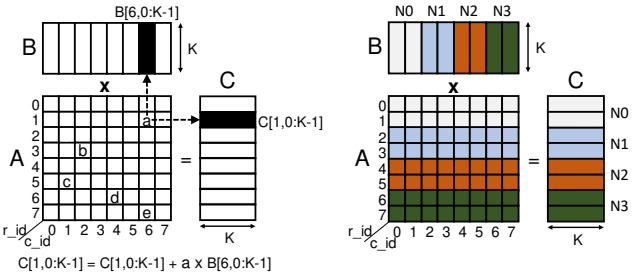
- The *Two-Face* algorithm for distributed SpMM, which is based on a mix of collective and one-sided communication.
- A low-overhead model and method to partition sparse matrices into regions corresponding to the two access types.
- An evaluation of *Two-Face* on a supercomputer and a comparison with the state-of-the-art.

2 Background

In this section, we provide a background on the memory access patterns in SpMM, the 1D partitioning method for distributing the SpMM data structures in a multi-node system, and the differences in the communication patterns of *SUT* and *SAT*.

2.1 SpMM

In SpMM, a sparse matrix A and a dense matrix B are multiplied, and the result is a dense matrix C, as expressed by $C = A \times B$. We refer to the number of columns in the dense matrices as K. Figure 1a illustrates the memory accesses and computation of this kernel. Note that B is shown transposed *in the whole paper* to help visualization. For each nonzero of the input sparse matrix A, one row of B and one row of C are accessed. Those rows are indexed by the nonzero's coordinates: the row index of the nonzero (r_id) indexes C, while the column index of the nonzero (c_id) indexes B. For example, in the figure, nonzero a triggers read accesses from the dense input row of B in black, and read and write accesses to the dense output row of C in black. The B row is scaled by a and added to the C row. This process is repeated for all the nonzeros of the sparse matrix. When the dense matrices are distributed in a multi-node machine, the nonzero structure of A affects the inter-node data transfer patterns.



(a) Computation in SpMM **(b)** 1D partitioning for 4-nodes

Figure 1. SpMM and 1D partitioning.

2.2 1D Partitioning

When executing SpMM on a distributed system, the sparse and dense matrices should be partitioned across nodes. In this work, we use 1D partitioning, which in prior work [8] was shown to display good performance for many sparse input matrices. Figure 1b illustrates 1D partitioning for a system consisting of 4 nodes ($N0...N3$). The matrices are partitioned according to the node colors. Each node is responsible for the nonzeros in a set of consecutive rows of A. It additionally hosts a set of consecutive B and C rows as shown in Figure 1b. The read and write accesses to the dense output matrix C are always local. The accesses to the dense input matrix B are often remote, except for nonzeros with c_ids that index to the local portion of B (e.g., nonzeros with c_id equal to 0 or 1 in Node 1). Overall, with this partitioning scheme, remote data accesses occur only for B. From now on, we will use the term *remote transfers* to mean transferring remote elements of the B matrix.

2.3 Sparsity-unaware and Sparsity-aware Transfers

We now discuss the communication patterns associated with sparsity-unaware (*SUT*) and sparsity-aware transfers (*SAT*).

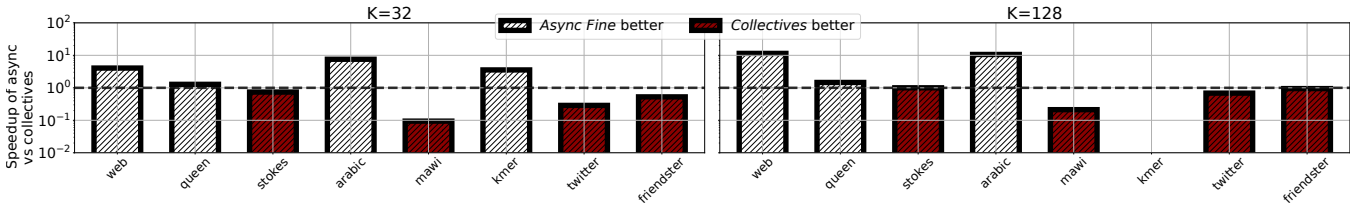
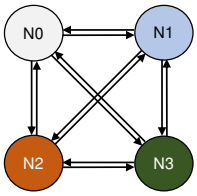
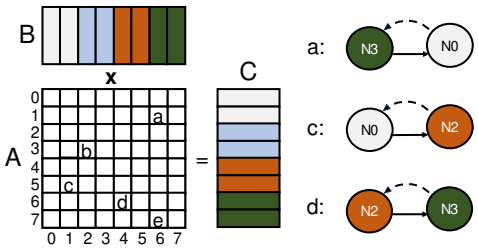


Figure 2. Speedup of *Async Fine* over a full-replication AllGather collective implementation for $K = 32$ (left) and $K = 128$ (right). There is no data for kmer with $K = 128$ due to the large memory consumption of the full-replication collective algorithm.

Figure 3a illustrates one possible *SUT* pattern. Each node sends its local rows of B to all of the other nodes. This is a conservative approach: all local rows are transferred to all the other nodes regardless of whether the rows are actually useful to all the destination nodes. This pattern can be implemented either with collectives [48] such as AllGather [12, 51], which fully replicates the dense input, or with shifting algorithms [8] that perform the transfers iteratively in a cyclic manner. A shifting algorithm consists of a series of computation and communication steps.



(a) SUT pattern



(b) SAT pattern

Figure 3. Examples of *SUT* and *SAT* patterns.

Figure 3b illustrates an example of the *SAT* approach. Each node traverses its local partition of the sparse matrix A and issues fine-grained read requests for rows of B. For example, Node N0 is responsible for the nonzero a, but the corresponding B row that this nonzero requires is hosted in N3. Hence, as shown in the figure, N0 issues a request to N3 for the row (dashed arrow) and N3 sends the row (solid arrow). Similar requests are issued for nonzeros c and d, but not for b and e, since the required B rows of the latter are already local.

The *SAT*s are fine-grained – only a single row is transferred instead of all the rows of a node as in the *SUT* approach. In addition, requests are typically initiated by the receiver and are asynchronous (i.e., the set of all nodes does not need to synchronize for the request to be completed). For these reasons, we refer to this particular communication pattern for distributed SpMM as *Async Fine*. This is in contrast to the execution strategies that use *SUT*s, where transfers are coarse-grained and require more synchronization.

3 Motivation

We now show that, for some matrices, the *SAT* pattern is more suitable, while for others the *SUT* is better. Then, we provide an example to motivate that a combination of the two approaches for the same matrix can yield the best results.

3.1 Choice of SAT & SUT SpMM is Input Dependent

Both the *SUT* and *SAT* approaches have pitfalls: *SUT*s can lead to unnecessary data transfers, while *SAT*s can have high software overhead and more round-trips between nodes. To compare their performance, we profile the execution of distributed SpMM using *Async Fine* (a *SAT* approach) and AllGather collectives (a *SUT* approach). We use a distributed machine with 32 nodes and 128 CPU cores per node, running the two algorithms for 8 large sparse matrices from SuiteSparse [15], and for two different values of K (32 and 128). Section 6 gives more details about our methodology. Figure 2 displays our findings. The figure shows the speedup of *Async Fine* over the AllGather implementation (*Collectives*) for $K = 32$ (left) and $K = 128$ (right). We do not include results for the kmer matrix with $K = 128$ because the single-node memory demand of *Collectives* exceeds the single-node capacity in our system.

We see that, for half of the matrices, *Async Fine* outperforms *Collectives*, while the opposite is the case for the other half. The speedup of *Async Fine* over *Collectives* reaches 11.5x, while the speedup of *Collectives* over *Async Fine* reaches 10.5x. Clearly, whether *SUT* or *SAT* works best depends on the sparsity pattern of the input matrix.

3.2 Combining SUT & SAT for a Single SpMM

Typically, the nonzeros in a sparse matrix are not evenly distributed. Consequently, combining the *SAT* and *SUT* communication flavors for a single SpMM could be beneficial. An example of this idea is shown in Figure 4. We split the nonzeros into three categories: (1) *local-input nonzeros (NNZs)* are those for which the dense input rows needed are already

local to the node with the nonzeros and, therefore, no remote transfers are needed; (2) *sync nonzeros* are those for which the dense input rows needed are better off transferred through *SUTs*; and (3) *async nonzeros* are those for which it is more beneficial to transfer the rows through *SAT*.

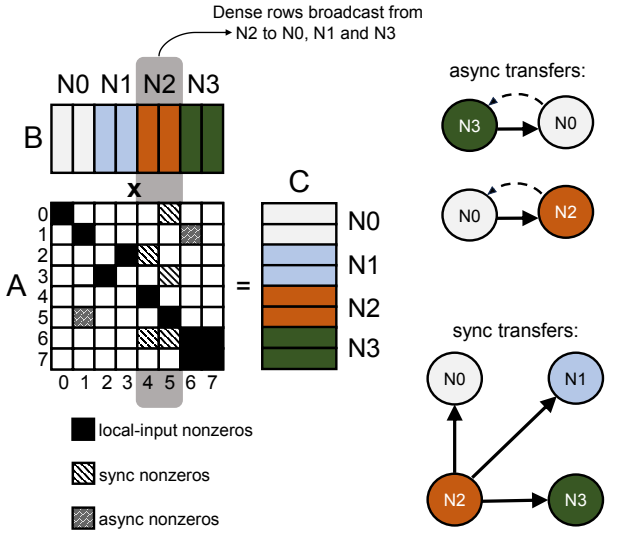


Figure 4. Example of how combining the two communication flavors can be beneficial.

To understand which nonzeros are *sync* and which are *async*, consider the example. Columns 4 and 5 of the sparse matrix are quite dense. This means that the corresponding dense input rows of B (shaded in the figure) are useful to many nodes. Specifically, B[4,0:K-1] is needed by N1, N2, and N3, while B[5,0:K-1] is needed by all the nodes. Hence, it is likely beneficial to transfer the whole group of rows hosted by N2 to all the other nodes through a collective broadcast operation. Hence, we classify the nonzeros at (0,5), (2,4), (3,5), (6,4), and (6,5) as *sync*. On the other hand, B[0,0:K-1] is not needed by any of the remote nodes and B[1,0:K-1] is only needed by the N2 remote node—N0 also accesses both rows, but they are already local and no transfers are needed. Transferring the whole group of dense rows that N0 hosts through a collective broadcast would lead to many unnecessary data transfers. Thus, the best strategy is likely for N2 to issue a one-sided request to N0 to get B[1,0:K-1], without synchronizing with the rest of the nodes. Therefore, the nonzero at (5,1) is classified as *async*. A similar situation occurs with the nonzero at (1,6), which is an *async* nonzero. Note that a collective broadcast operation does not necessarily need to include all nodes as destinations. For example, if the nonzero at (0, 5) was absent, then N2 could still issue a multicast transfer directed only to N1 and N3.

4 Overview of Two-Face

In this section, we translate the intuition about combining the *SUT* and *SAT* approaches into an algorithm called *Two-Face*. *Two-Face* combines sparsity-unaware and sparsity-aware transfers for efficient distributed SpMM. In this section, we describe how sparse matrices are analyzed and partitioned into two types of regions.

4.1 Sparse Matrix Partitioning

Before the SpMM execution begins, the sparse matrix is preprocessed in order to determine which of the data transfers will use coarse-grained multicast operations, and which will use fine-grained one-sided communication. Then, during runtime, both types of transfers and their corresponding computations will proceed in parallel.

Two-Face adopts 1D partitioning (Subsection 2.2). Thus, each node, which contains one MPI rank, is responsible for the nonzeros in a group of consecutive rows of sparse matrix A. In addition, the node hosts the corresponding group of consecutive rows of the dense output matrix C, and a group of consecutive rows of the dense input matrix B. As discussed earlier, the accesses to C and A are always local, while the accesses to B can either be local or require a remote data transfer, depending on the *c_id*s of the processed nonzeros. The matrices are partitioned in the following way:

Megatile. As shown in Figure 5, we logically divide the matrix A into *megatiles* (MT). Given the matrix A with N rows and M columns, and given p nodes in the distributed system, a megatile is formed with N/p consecutive rows and M/p consecutive columns. Node i stores the i^{th} row of megatiles. We logically divide the matrices B and C based on the width and height of a megatile, respectively. Figure 5 shows the breakdown of the matrices for $p = 4$. The chunks of the B and C matrices are distributed across the nodes as shown in the figure, where node i is labeled Ni.

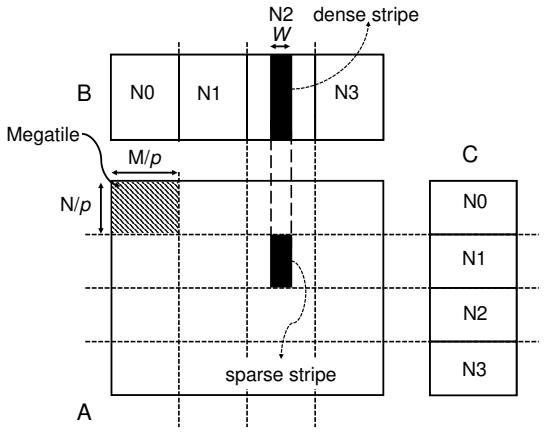


Figure 5. *Two-Face* megatiles and sparse and dense stripes.

Sparse stripe. Each megatile is further divided into *sparse stripes*, which determine the communication patterns. A sparse stripe has the same number of rows as a megatile and a fixed number of columns W. We choose to divide megatiles into sparse stripes to allow for partitioning the

sparse matrix into two types of regions at a fine granularity. Sparse stripes are classified as *local-input* if their corresponding B rows are owned by the local node; otherwise, they are *remote-input*. A remote-input sparse stripe can either trigger a coarse-grained collective transfer or a fine-grained asynchronous one. At a high level, remote-input stripes requiring many rows of the dense input matrix B will be marked as *synchronous* as, during execution, they will benefit from the coarse-grained collectives. On the contrary, remote-input stripes requiring few rows of B will be marked as *asynchronous* as, during execution, they will benefit from fine-grained asynchronous transfers. During execution, multiple nodes containing synchronous stripes that require the same data from B will participate in the same collective multicast operation to receive that data. It is possible that a "multicast" transfers data to only a single destination node.

Dense stripe. All the sparse stripes that have the same range of *c_id*s in A access the same group of rows in matrix B. We call this group of dense rows a *dense stripe*. A synchronous sparse stripe will trigger the coarse-grained transfer of a dense stripe using a collective operation with, potentially, additional destination nodes; an asynchronous sparse stripe will trigger the fine-grained one-sided transfer of individual rows (or groups of adjacent rows) within the dense stripe. These asynchronous transfers will only transfer the rows of the dense stripe that are needed for the computation. If a node does not need any of a dense stripe's rows, that dense stripe will not be communicated to it at all.

In the next sections, we use *stripe* to refer to sparse stripes. Any mention of dense stripes is explicit.

Stripes are classified as asynchronous or synchronous during a preprocessing step using a model that tries to minimize the expected execution time. We present the model in Section 4.2. During the actual execution after the preprocessing step, the local threads in an MPI rank operate in parallel and are split into two groups: (1) synchronous threads, which handle the data transfers for the synchronous stripes as well as the computation for synchronous and local-input stripes, and (2) asynchronous threads, which handle the data transfers and computation for the asynchronous stripes. All synchronous communication is completed before any synchronous computation begins. On the other hand, asynchronous communication and computation overlap: a thread may compute on one asynchronous stripe while another thread transfers data for a second asynchronous stripe.

To optimize computation and communication efficiency, the nonzeros in sparse stripes are ordered in row-major order in synchronous stripes and column-major order in asynchronous stripes. In synchronous stripes, the nonzeros are stored in row-major order because this benefits computation. Specifically, a thread can process the nonzeros of a whole row of the stripe and buffer the results in a thread-local buffer before updating C. Then, the thread uses a single synchronization operation to accumulate the contents of the thread-local buffer into the corresponding C row. In asynchronous stripes, the nonzeros are stored in column-major order to benefit communication. Specifically, a column-major format allows a thread to quickly traverse the nonzeros and determine the unique *c_id*s of the nonzeros in a stripe, which in turn identify the rows of matrix B that need to be transferred. This comes at the cost of computational inefficiency since this format makes buffering the output of a thread's computations hard. As a result, the thread must typically use one synchronization operation for each nonzero to accumulate results onto C.

4.2 Preprocessing Model

During execution, *Two-Face* will process the asynchronous stripes in parallel with the synchronous and local-input stripes. Consequently, the optimal choice to partition the sparse matrix into synchronous and asynchronous stripes is one that equalizes the execution times of asynchronous stripes and synchronous/local-input stripes. To this end, we create a model of execution based on the following ideas.

For the synchronous stripes, the model assumes that the computation time will be negligible compared to the synchronous communication time. The reason is that the row-major format of the nonzeros lends itself to efficient execution: the output of the nonzeros in a row of the stripe is reused through a thread-local buffer and accumulated into the corresponding C row with a single synchronization operation. In addition, we take advantage of this parallelizability by assigning more parallel threads to the computation of synchronous/local-input stripes than for the asynchronous stripes. For the local-input stripes, since they do not need communication, the model neglects both communication and computation time.

In contrast, the computation time for asynchronous stripes may be significant because the column-major format of the nonzeros lends itself to inefficient execution: thread-local buffers are not used and we need to perform a synchronization operation for every nonzero. In addition, because synchronization may be a bottleneck, we assign fewer threads to the computation of asynchronous stripes than to the others. Therefore, for the asynchronous stripes, the model considers both computation and communication.

We model the cost of synchronous communication ($Comm_S$), asynchronous communication ($Comm_A$), and asynchronous computation ($Comp_A$) for a particular node as:

$$Comm_S = S_S\,(\beta_S KW + \alpha_S)$$
$$Comm_A = \beta_A KL_A + \alpha_A S_A$$
$$Comp_A = \gamma_A KN_A + \kappa_A S_A$$

Consider $Comm_S$ first. S_S is the number of synchronous stripes processed by the node, W is the stripe width, and K is the number of columns in the dense matrices. β_S is the cost of synchronous transfer per element of B (i.e., it is inversely proportional to the bandwidth), and α_S is other per-stripe overheads of synchronous transfers.

Next, for $Comm_A$, S_A is the number of asynchronous stripes processed by the node, and L_A is the total number of rows of the dense matrix B transferred for these stripes via fine-grained accesses. β_A and α_A represent the same costs as β_S and α_S but for asynchronous accesses.

Finally, for $Comp_A$, N_A is the total number of nonzeros in the asynchronous stripes processed by the node. γ_A is the computational cost per operation, and κ_A is the additional per-stripe software overhead of asynchronous computation.

The coefficients β_S, α_S, β_A, α_A, γ_A, and κ_A are determined via a linear regression [40] calibration step (details in Section 6.2). These parameters are dependent on the system configuration. For example, a system with a large bisection bandwidth should have small β terms. The α terms may be reduced by reducing the round-trip communication latency, including the latency incurred in software libraries/drivers and in the network.

In the optimal case, $Comm_S = Comm_A + Comp_A$, so that there is a perfect overlap of the asynchronous and synchronous components. Defining $S_T = S_S + S_A$ to be the total number of non-local-input stripes processed by a particular node and rearranging this equation gives the following:

$$\begin{aligned} S_S(\beta_S KW + \alpha_S) &= \beta_A KL_A + \alpha_A S_A + \gamma_A KN_A + \kappa_A S_A \\ \implies (S_T - S_A)(\beta_S KW + \alpha_S) &= K(\beta_A L_A + \gamma_A N_A) \\ &\quad + S_A(\alpha_A + \kappa_A) \\ \implies S_T(\beta_S KW + \alpha_S) &= K(\beta_A L_A + \gamma_A N_A) \\ &\quad + S_A(\alpha_A + \kappa_A + \beta_S KW + \alpha_S) \end{aligned} \tag{1}$$

We now classify the stripes requiring communication as either synchronous or asynchronous. Initially, we assume that all stripes are synchronous, which makes the right-hand side of Equation 1 equal to 0. Then, we take each stripe i and consider classifying it as asynchronous, instead. In this case, the stripe's contribution (call it z_i) to the right-hand side of Equation 1 would be given by:

$$\begin{aligned} z_i &= v_i + u, \\ \text{where } v_i &= K(\beta_A l_i + \gamma_A n_i), \\ u &= \alpha_A + \kappa_A + \beta_S KW + \alpha_S, \end{aligned}$$

where stripe i requires l_i dense rows from matrix B and contains n_i nonzeros. Note that u depends only on the stripe width (W) and other constants, and is therefore constant for all the stripes in the matrix.

To identify the most beneficial stripes to classify as asynchronous, we look for stripes with low values of z_i. This is because a low z_i implies that, if the stripe is classified as asynchronous, it requires few dense rows to be transferred from a remote node and contains few nonzeros. Therefore, its communication and computation costs are relatively low. On the other hand, if the stripe is classified as synchronous, it has a constant communication cost.

Consequently, we sort all of this node's stripes by their z_i in ascending order. Then, we take one stripe at a time, in order, and classify it as asynchronous, until we have taken the first r stripes, where r is the greatest number that satisfies

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

The rest of the stripes are classified as synchronous. With this method, the two sides of Equation 1 are approximately equal, which is a necessary condition to attain optimal execution time. Indeed, following this method, the total runtimes of the synchronous and asynchronous stripes in *Two-Face* should be nearly equal, assuming the simplified cost model that we use does hold. Additionally, sorting the stripes by their z_i maximizes the number of stripes classified as asynchronous and minimizes the number of synchronous stripes. Because the cost of communication for a synchronous stripe is constant for a given K and W, this strategy minimizes $Comm_S$ and, therefore, the total cost of the operation.

There are other possible methods of classifying stripes. One such method is to analyze columns of stripes in the sparse matrix and classify a stripe as synchronous when its corresponding dense stripe is needed by many nodes and, therefore, is likely to benefit from optimized multicast operations. We leave the investigation of such methods for future work.

5 The Two-Face Algorithm

In this section, we provide greater details about *Two-Face*. We discuss the sparse matrix representation, the *Two-Face* algorithm, its tuning and portability, and its applicability to GNN training.

5.1 Sparse Matrix Representation

Two-Face represents the sparse matrix A in a modified COO format. The nonzeros in asynchronous stripes are extracted from A and are stored in an *Asynchronous* sparse matrix; the nonzeros in synchronous/local-input stripes are extracted into a *Synchronous/Local-Input* sparse matrix. Because we use a compressed sparse matrix representation, this format does not significantly increase overall memory use.

Figure 6 shows: (a) an example of an input sparse matrix A, (b) its corresponding synchronous/local-input sparse matrix, and (c) its corresponding asynchronous sparse matrix. The figure assumes that there are four nodes and one sparse stripe for each 2x2 megatile. Assume that, after running the preprocessing step, the stripes have been classified as local-input, synchronous, and asynchronous such that the nonzeros in A end up in the categories shown in Figure 6a.

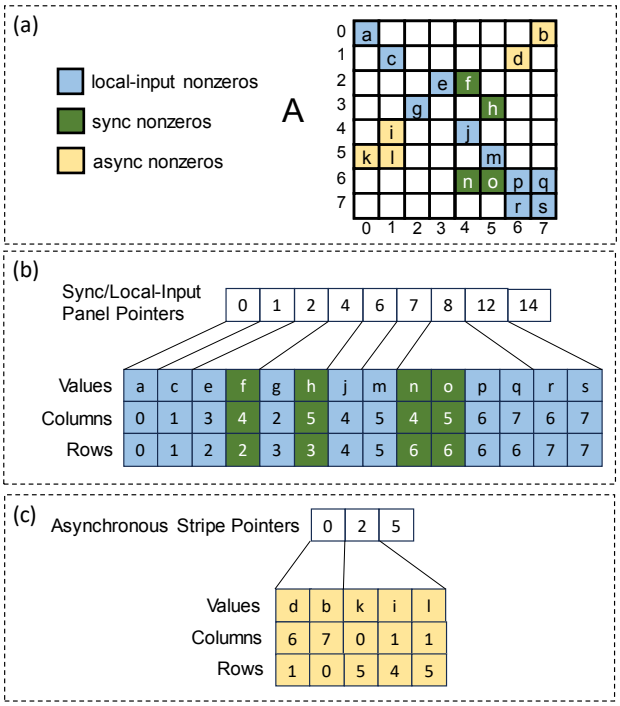


Figure 6. Sparse matrix representation in *Two-Face*: (a) an input sparse matrix A, showing an example of nonzeros classified into the local-input, sync, and async categories; (b) the corresponding synchronous/local-input sparse matrix; and (c) the corresponding asynchronous sparse matrix. This figure assumes a 4-node system, a stripe width of 2, and a row panel height of 1.

The corresponding synchronous/local-input sparse matrix (Figure 6b) organizes the synchronous/local-input nonzeros in a row-major order structure. The elements in this structure are divided into row panels—e.g., nonzeros e and f in Figure 6 are in one row panel. Row panels are the units of work assigned to threads *computing* on synchronous/local-input nonzeros. In Figure 6b, these panels are one row tall, and an array of *Synchronous/Local-input Panel Pointers* points to the beginning of each panel.

The corresponding asynchronous sparse matrix (Figure 6c) organizes the asynchronous nonzeros within stripes in a column-major order structure. The order of the stripes themselves is row-major, to simplify the distribution of the asynchronous sparse matrix across nodes at runtime. An array of *Asynchronous Stripe Pointers* points to the beginning of each asynchronous stripe.

At runtime, each node will only store those portions of the synchronous/local-input and asynchronous sparse matrices that are relevant to its computation. In addition, for each dense stripe of B, the preprocessing step generates metadata containing a list of nodes that are destinations of the collective transfer of that stripe. At runtime, this metadata is replicated across all nodes.

5.2 *Two-Face* Algorithm Description

The algorithm consists of three parts: top-level algorithm, processing synchronous row panels, and processing asynchronous stripes. We describe each part in turn.

5.2.1 Top-Level Algorithm. Algorithm 1 shows the top-level operation of the *Two-Face* algorithm. All nodes of the distributed system execute Algorithm 1 in parallel. First, the node initializes a flag and two atomic queues for work-sharing (Lines 2-3). These queues provide indices of asynchronous stripes (*async_q*) and indices of row panels (*sync_q*). In the example of Figure 6, the *sync_q* of N2 is {4, 5}, which are the indices of the two pointers in the Sync/Local-Input Panel Pointers array used by N2 (pointing to rows containing nonzeros j and m). The *async_q* of N2 is {1}, which is the index of the pointer in the Asynchronous Stripe Pointers array that points to the asynchronous stripe assigned to N2.

Algorithm 1 Top-Level Two-Face Pseudo-code.

```
1:  procedure DistSPMM(A, B, C)
2:      sync_transfer_done ← False
3:      async_q, sync_q ← InitQueues(A)
4:      DoParallel
5:          if tid = 0 then                          ▷ Sync Transfers
6:              TransferDenseStripes(A, B)
7:              sync_transfer_done ← True
8:          end if
9:          if tid ∈ AsyncThreads then ▷ Async Processing
10:             while async_q.nonempty() do
11:                 n ← 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 ← sync_q.pop()
18:             ProcessSyncRowPanel(A, B, C, n)
19:         end while
20:     EndParallel
21: end procedure
```

Then, the code starting at Line 4 is executed by all the threads of the node in parallel. Specifically, Thread 0 initiates the transmission/reception of the dense stripes needed for synchronous operations (Lines 5-8). Data transmission is implemented as non-blocking, but data reception is blocking. These transfers are done via a series of calls to `MPI_Bcast`. The destination nodes are determined via metadata produced by the preprocessing step, and these transfers occur at the stripe granularity.

In parallel, all the threads assigned to asynchronous stripes begin processing those stripes (Lines 9-14). Once all needed dense input data from collectives has been received (Line 15), all threads (including the asynchronous ones after they have processed the asynchronous stripes) process the synchronous row panels (Lines 16-19).

5.2.2 Processing Synchronous Row Panels.

Algorithm 2 describes the processing of a row panel. 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.

Algorithm 2 Two-Face Sync Compute Pseudo-code

```
1:  procedure ProcessSyncRowPanel(A, B, C, n)
2:      acc ← {0, ..., 0}                          ▷ Output row buffer
3:      panel ← A.panel_ptrs[n]
4:      prev_row ← panel[0].row                    ▷ Initialize to first row
5:      for nz ∈ panel do
6:          if nz.row ≠ prev_row then
7:              AtomicAdd(C[prev_row], acc)
8:              acc ← {0, ..., 0}
9:          end if
10:         acc ← acc + nz.val * B[nz.col]
11:         prev_row ← nz.row
12:     end for
13:     AtomicAdd(C[prev_row], acc)
14: end procedure
```

5.2.3 Processing Asynchronous Stripes.

Algorithm 3 shows the algorithm to process an asynchronous stripe. A thread reads the asynchronous stripe (*stripe* in Line 2) and iterates over the nonzeros in the stripe to identify the unique *c_id*s 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.

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.

Algorithm 3 Two-Face Async Pseudo-code

```
1: procedure ProcessAsyncStripe(A, B, C, n)
2:     stripe ← A.async_stripe_ptrs[n]
3:     drow_ids ← stripe.UniqueColIDs()
4:     drows ← GetRemoteRows(drow_ids)
5:     for nz ∈ stripe do in parallel
6:         AtomicAdd(C[nz.row], nz.val * drows[nz.col])
7:     end for
8: end procedure
```

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).

5.3 Tuning Knobs and 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 Sections 6.2 and 7 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.

5.4 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 [25] and mini-batching. While these techniques reduce the memory footprint requirements, they may introduce some inaccuracy [25, 33], and may introduce runtime overhead which can sometimes increase the end-to-end execution time [23]. In *Two-Face*, like in most of the prior work in distributed GNN training [33, 53], we are less concerned about the lack of memory because we can use the full aggregate memory of a very large cluster [53]. 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 6, but with the addition of masks to filter nonzeros eliminated by the sampling at each iteration.

In current full-graph GNN training [35, 54], the preprocessing cost can be easily amortized. We will quantify the exact cost of the preprocessing step in Section 7.3. 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 assumption in most GNNs for semi-supervised node classification applications [35, 54]. Since the sparse matrix does not change, the preprocessing done in training can be reused for inference. For these reasons, the overhead of *Two-Face* preprocessing in full-graph GNN training is negligible.

6 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*.

6.1 Overview

We evaluate *Two-Face* and multiple baseline algorithms using large matrices on Delta [2], 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 [16].

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 [43] and Open MPI 4.1.2 [19, 39] with UCX 1.11.2 [49]. All of the baseline algorithms use the Intel Math Kernel Library [14] (version 2022.0.2) for local SpMM computations. These baselines also rely on CombBLAS [6] 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 [46] for handling dense matrices locally.

We use eight large sparse matrices from SuiteSparse [15], described in Table 1. These matrices are derived from a variety of domains, including internet traffic, social networks, web crawls, and scientific applications.

Table 1. Matrices used in the evaluation. All matrices are among the largest in SuiteSparse [15] and are square. Stripe widths are chosen to scale with the number of columns.

Matrix Name		# Rows (Mill)	# Nonzeros (Mill)	Stripe Width
Long	Short			
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

Our evaluation in Section 7 supports the claim that distributed SpMM is typically a 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.

6.2 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 1 shows the stripe widths we chose.

All run-time parameters other than the stripe width are held constant across matrices. Table 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 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 [40]. We collect data by processing the twitter matrix [36] 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 3.

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 7.4 of our evaluation, we evaluate the impact of different values of these coefficients.

6.3 Algorithms Evaluated

In our evaluation, we compare *Two-Face* to other algorithms shown in Table 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 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}

Table 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

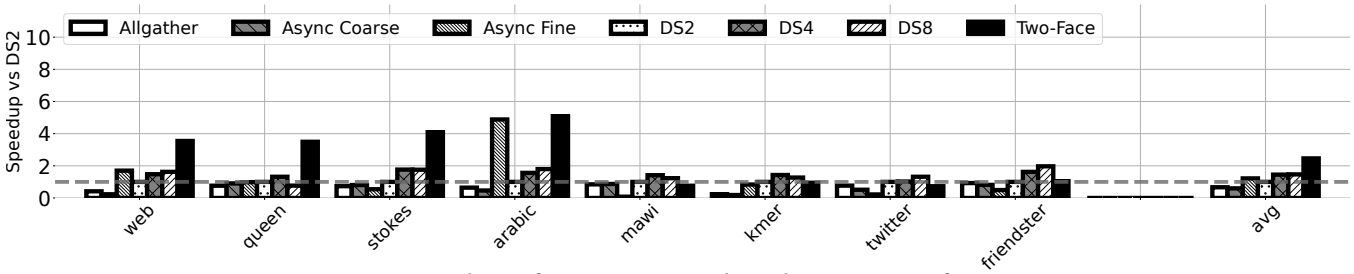


Figure 7. Speedups of various SpMM algorithms over DS2 for $K = 32$.

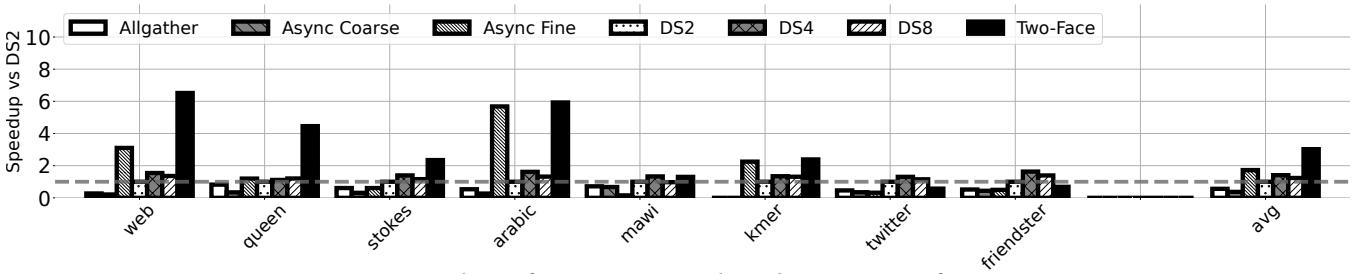


Figure 8. Speedups of various SpMM algorithms over DS2 for $K = 128$.

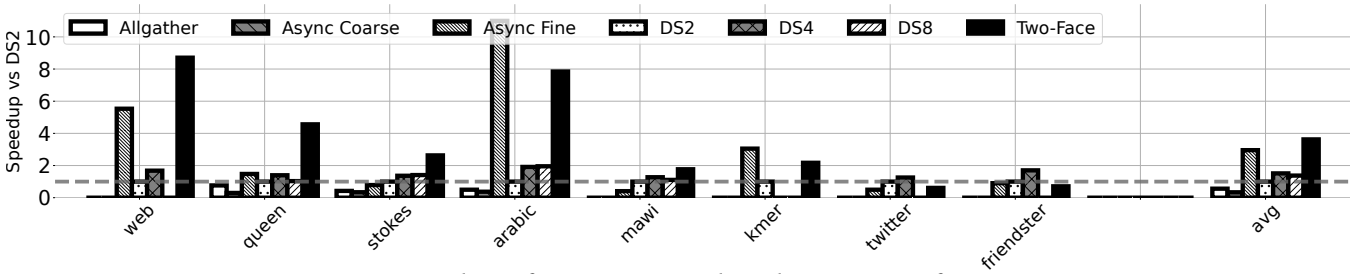


Figure 9. Speedups of various SpMM algorithms over DS2 for $K = 512$.

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.

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.

7 Evaluation

In this section, we evaluate *Two-Face*. First, we compare the performance of *Two-Face* to the various baselines and discuss any bottlenecks observed. Next, we discuss the scaling behavior of *Two-Face* as we vary the number of nodes in the system. Finally, we analyze the preprocessing cost of *Two-Face* and the sensitivity of *Two-Face* to the choice of the preprocessing parameters.

7.1 Comparing *Two-Face* to Various Baselines

Figures 7, 8, and 9 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.

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* 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

Table 5. Absolute execution times of DS2 and *Two-Face* for the experiments in Figures 7, 8, and 9. The numbers are the average of five SpMM operations.

		web	queen	stokes	arabic	mawi	kmer	twitter	friendster
K=32	DS2 (seconds)	1.97	0.28	0.96	1.32	6.46	6.33	4.17	8.79
	Two-Face (seconds)	0.56	0.08	0.23	0.26	8.50	6.70	5.71	8.41
K=128	DS2 (seconds)	7.198	0.86	2.22	3.85	19.78	35.77	11.57	20.61
	Two-Face (seconds)	1.10	0.19	0.94	0.65	15.18	14.98	20.24	30.08
K=512	DS2 (seconds)	38.86	2.89	9.34	21.46	97.95	136.21	52.77	83.02
	Two-Face (seconds)	4.46	0.634	3.552	2.74	55.40	62.77	86.62	117.31

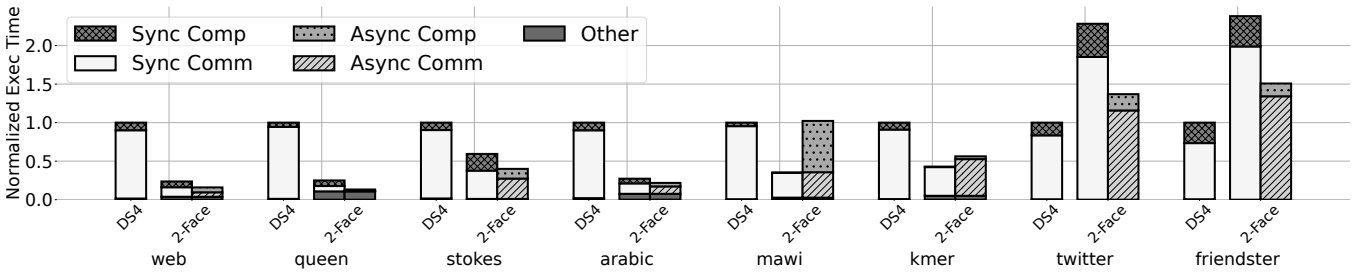


Figure 10. Breakdown of the total execution times of DS4 and *Two-Face* for $K = 128$. *Two-Face*'s time is divided into synchronous and asynchronous components (left and right bars, respectively), which operate in parallel. These are further broken down into computation (*Comp*) and communication (*Comm*). 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.

fails in one matrix. As a reference, Table 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 10 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.

7.2 *Two-Face* Strong Scaling

Figure 11 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

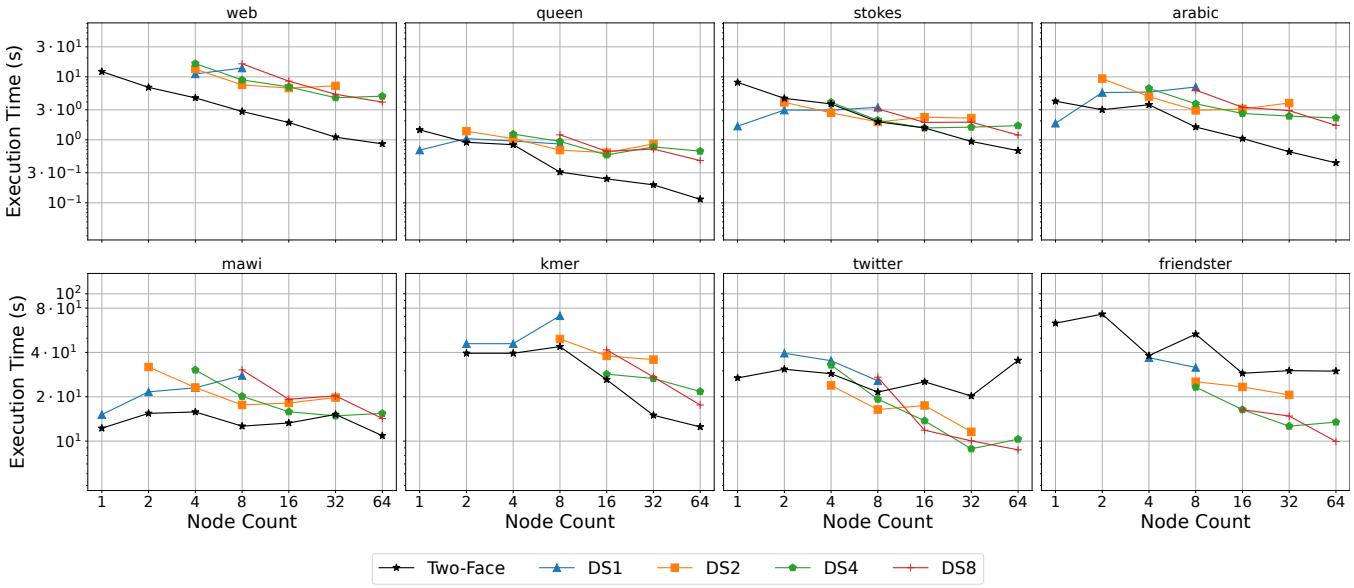


Figure 11. Execution time of *Two-Face* and the dense shifting algorithm with different replication factors (DS1, DS2, DS4, and DS8) as the number of nodes changes. The K value is 128. Some data points for the dense shifting algorithm are missing because they need too much memory or take too long to execute. Both axes in the plots use a logarithmic scale.

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 10 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 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.

7.3 *Two-Face* Preprocessing Cost

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. 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 6. The overhead of preprocessing in *Two-Face*, normalized to the cost of a single SpMM operation.

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

Table 6 shows the overhead of the preprocessing step for 32 nodes and K=128 for each matrix. Column 2 shows

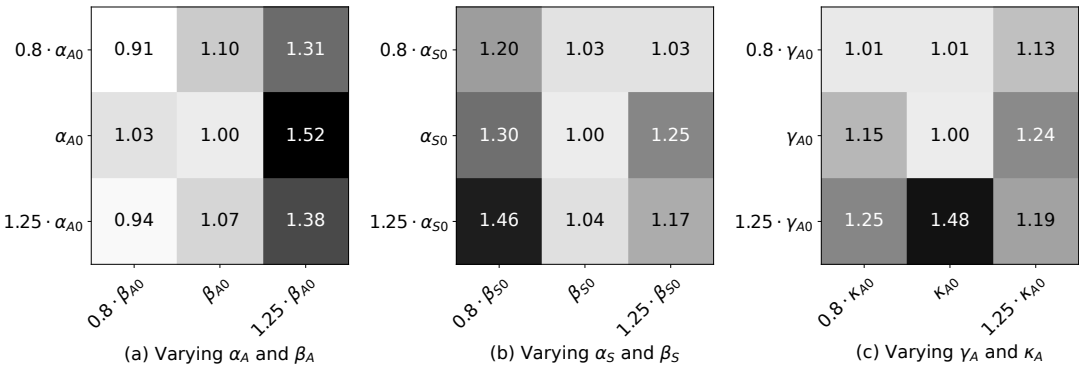


Figure 12. Sensitivity of *Two-Face*'s execution time to the values of the parameters of the execution model used during the preprocessing step. The default values of these parameters, as set in Section 6.2 and used in all the earlier experiments, are represented with the 0 subscript (e.g., α_{A0}).

$t_{norm\_I/O}$, 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 [9] 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.

7.4 Sensitivity to Parameter Values of the Preprocessing Model

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 6.2, we used linear regression to set their default values. We used such values in all the experiments so far.

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 12 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 12a 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.

8 Related Work

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. The implementations of dense shifting evaluated in our paper originate from their work. Additionally, Bharadwaj et al. [8] present a sparse shifting implementation. In our work, we did not evaluate their 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. Bharadwaj et al. [8] compare their implementations to the

SpMM provided by PETSc [7]. Selvitopi et al. [48] investigate multiple algorithms for SpMM, including algorithms that use bulk-synchronous collective communication and algorithms that use one-sided asynchronous RDMA communication. They do not, however, investigate combining these communication primitives in a single algorithm.

GNN Training: Prior work has addressed the issue of large graphs in GNN training via sampling techniques [25]. However, the benefits of sampling can come at a cost to accuracy, leading prior work to investigate full-batch distributed GNN training [33, 53]. However, in Tripathy et al. [53], dense matrices used in SpMM operations are only transferred in a coarse-grained sparsity-unaware fashion. Conversely, Jia et al. [33], using Lux [32], 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.

Non-Distributed Sparse Kernels: SpMM optimization has been the topic of several investigations. Works such as WACO [56], WISE [57], and DDB [58] 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 [27, 30, 37, 41]. Other sparse kernels have also been subject to several investigations aiming to tame irregular access patterns [24, 28, 42, 52]. 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 [13, 20, 38]. Cheng et al. [13] tackle SpGEMM on asymmetric multicore processors. HotTiles [20] 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.

Other Distributed Sparse Kernels: Other distributed sparse kernels have recently received attention. CombBLAS [6] 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 them use sparsity-unaware collectives. In contrast, *Two-Face* uses a hybrid approach. Hussain et al. [31] investigate communication-avoiding algorithms for SpGEMM. DGCL [10] 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.

Domain-specific Architectures and Network Support: Several architectural designs that offer hardware support for SpMM computation have been recently proposed [21, 26, 34, 44, 50]. SPADE [21] is an accelerator for SpMM and SDDMM designed to be tightly coupled with CPU cores. Tensaurus [50] and ExTensor [26] 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 [1, 4, 45]. 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 [5, 22, 59, 60]

Finally, scheduling algorithms for collectives such as Themis [47] 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.

9 Conclusion

Sparse matrices often contain regions that are denser and regions that are sparser. Based on the observation, this paper presented *Two-Face*, an algorithm for distributed SpMM that, leveraging a preprocessing model, performs 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.

Two-Face suggests that distributed sparse algorithms should be input-matrix aware, in that different sections of a sparse input matrix prefer using different communication methods. The 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. We are investigating these and other algorithms.

Acknowledgments

We thank Fredrik Kjolstad and the reviewers for helping to improve this paper. This research was funded in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by a grant from the IBM-Illinois Discovery Accelerator Institute; by NSF grants PPoSS CCF 2316233, CNS 1956007 and CCF 2107470; by DOE grant DE-SC0022098; and by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 21-46756. This work used the Delta system at the National Center for Supercomputing Applications through allocation CIS230044 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296.

References

[1] Sriram Aananthakrishnan, Shamsul Abedin, Vincent Cavé, Fabio Checconi, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Wim Heirman, Jason Howard, Ibrahim Hur, Samkit Jain, Marek M. Landowski, Kevin Ma, Jarrod Nelson, Robert Pawlowski, Fabrizio Petrini, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, and Yves Vandriessche. 2023. The Intel® Programmable and Integrated Unified Memory Architecture (PIUMA) Graph Analytics Processor. *IEEE Micro* (2023), 1–11. https://doi.org/10.1109/MM.2023.3295848

[2] Bruno Abreu, Galen Arnold, Gregory Bauer, Brett Bode, Craig Steffan, et al. 2024. *Delta User Documentation.* National Center for supercomputing Applications. Retrieved Jan 2024 from https://docs.ncsa.illinois.edu/systems/delta/en/latest/

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

[4] Matthew Joseph Adiletta, Jesmin Jahan Tithi, Emmanouil-Ioannis Farsarakis, Gerasimos Gerogiannis, Robert Adolf, Robert Benke, Sidharth Kashyap, Samuel Hsia, Kartik Lakhotia, Fabrizio Petrini, Gu-Yeon Wei, and David Brooks. 2023. Characterizing the Scalability of Graph Convolutional Networks on Intel® PIUMA. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* 168–177. https://doi.org/10.1109/ISPASS57527.2023.00025

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture.* 105–117.

[6] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R. Gilbert, and Aydın Buluç. 2022. Combinatorial BLAS 2.0: Scaling Combinatorial Algorithms on Distributed-Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 989–1001. https://doi.org/10.1109/TPDS.2021.3094091

[7] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2023. PETSc Web page. https://petsc.org/. https://petsc.org/

[8] Vivek Bharadwaj, Aydın Buluc, and James Demmel. 2022. Distributed-Memory Sparse Kernels for Machine Learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE Computer Society, 47–58. https://doi.org/10.1109/IPDPS53621.2022.00014

[9] Ronald Boisvert, Roldan Pozo, and K Remington. 1996. The Matrix Market Exchange Formats: Initial Design.

[10] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21).* Association for Computing Machinery, 130–144. https://doi.org/10.1145/3447786.3456233

[11] John Canny and Huasha Zhao. 2013. Bidmach: Large-scale learning with zero memory allocation. In *BigLearning, NIPS Workshop.*

[12] Ernie Chan, Robert Van De Geijn, William Gropp, and Rajeev Thakur. 2006. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2–11.

[13] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) *(ICPP '23).* Association for Computing Machinery, 807–817. https://doi.org/10.1145/3605573.3605611

[14] Intel Corporation. 2023. *Intel® oneAPI Math Kernel Library.* Intel Corporation. Retrieved 2023 from https://intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[15] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[16] Hewlett Packard Enterprise. 2024. *HPE Slingshot interconnect.* Hewlett Packard Enterprise. Retrieved Jan 2024 from www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html

[17] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2023. Fast Sparse GPU Kernels for Accelerated Training of Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 501–511. https://doi.org/10.1109/IPDPS54959.2023.00057

[18] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds.*

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting.* 97–104.

[20] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. 2024. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE.

[21] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23).* Association for Computing Machinery, Article 19, 15 pages. https://doi.org/10.1145/3579371.3589054

[22] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–49.

[23] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W Fletcher, Christopher J Hughes, and Josep Torrellas. 2022. Graphite: Optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA).* 916–931.

[24] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. 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* (Virtual Event, USA) *(SPAA '20).* Association for Computing Machinery, 293–303. https://doi.org/10.1145/3350755.3400216

[25] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17).* Curran Associates Inc., 1025–1035.

[26] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, 319–333. https://doi.org/10.1145/3352460.3358275

[27] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, 300–314. https://doi.org/10.1145/3293883.3295712

[28] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, 710–726. https://doi.org/10.1145/3582016.3582051

[29] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41405.2020.00075

[30] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC41405.2020.00076

[31] Md Taufique Hussain, Oguz Selvitopi, Aydin Buluç, and Ariful Azad. 2021. Communication-Avoiding and Memory-Constrained Sparse Matrix-Matrix Multiplication at Extreme Scale. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 90–100. https://doi.org/10.1109/IPDPS49936.2021.00018

[32] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment* 11, 3 (Nov 2017), 297–310. https://doi.org/10.14778/3157794.3157799

[33] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with Roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[34] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. 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*. 600–614.

[35] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[36] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proc. the 19th Intl. Conf. on World Wide Web* (Raleigh, North Carolina, USA). ACM, 591–600. https://doi.org/10.1145/1772690.1772751

[37] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41404.2022.00042

[38] Wenxuan Li, Helin Cheng, Zhengyang Lu, Yuechen Lu, and Weifeng Liu. 2023. HASpMV: Heterogeneity-Aware Sparse Matrix-Vector Multiplication on Modern Asymmetric Multicore Processors. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 209–220.

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

[40] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2021. *Introduction to linear regression analysis*. John Wiley & Sons.

[41] NVIDIA. 2024. *cuSPARSE*. Retrieved Jan 2024 from https://developer.nvidia.com/cusparse

[42] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, 18–32. https://doi.org/10.1145/3582016.3582064

[43] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface Version 4.5. https://openmp.org/wp-content/uploads/openmp-4.5.pdf.

[44] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SOCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 817–830.

[45] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Dalorex: A data-local program execution and architecture for memory-bound applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 718–730.

[46] Eigen Project. 2023. *Eigen v3.4*. Retrieved Jan 2024 from https://eigen.tuxfamily.org

[47] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, 581–596. https://doi.org/10.1145/3470496.3527382

[48] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2021. Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, 431–442. https://doi.org/10.1145/3447818.3461472

[49] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[50] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

[51] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[52] Han D. Tran, Milinda Fernando, Kumar Saurabh, Baskar Ganapathysubramanian, Robert M. Kirby, and Hari Sundar. 2022. A scalable adaptive-matrix SPMV for heterogeneous architectures. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 13–24. https://doi.org/10.1109/IPDPS53621.2022.00011

[53] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00074

[54] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[55] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[56] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, 920–934. https://doi.org/10.1145/3575693.3575742

[57] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. 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* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, 329–341. https://doi.org/10.1145/3572848.3577506

[58] Serif Yesil, José E. Moreira, and Josep Torrellas. 2022. 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* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, Article 27, 14 pages. https://doi.org/10.1145/3524059.3532369

[59] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[60] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.