UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EVALUATING SPARSE COLLECTIVE COMMUNICATION ALGORITHMS FOR STRUCTURED GRAPHS AND MATRICES

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By DYLAN ZEMLIN

Norman, Oklahoma

2026

EVALUATING SPARSE COLLECTIVE COMMUNICATION ALGORITHMS FOR STRUCTURED GRAPHS AND MATRICES

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Richard Veras, Chair

Dr. Sridhar Radhakrishnan

Dr. Le Gruenwald

Acknowledgments

First, I would like to thank Dr. Richard Veras for his continued support throughout not only my work on this thesis, but also generally through my time at OU, even through my late (very late sometimes) emails containing ramblings of someone who has had one too many energy drinks.

I am extremely grateful to all of my friends I have met in Sooner Competitive Robotics (SCR). Although the work we do there is not directly applicable to this thesis, it did help me gain the confidence over the years in my work to be able to work on a project of this level.

And of course, I want to thank my family. My mom, my dad, and both of my brothers, your constant support and encouragement helped me more than I could ever explain.

The computing for this project was performed at the OU Supercomputing Center for Education & Research (OSCER) at the University of Oklahoma (OU).

This material is based upon work supported by the National Science Foundation under Grant No. 2440646.

Table of Contents

List of Figures

List of Tables

Abstract

Efficient collective communication is a key challenge for scaling applications on distributed memory architectures. Prior analyses have largely focused on dense matrix computations, where data often moves in predictable ways. However, many real-world problems are expressed in terms of graphs whose adjacency matrices are sparse. The structure and storage format of these matrices can introduce irregular communication patterns in distributed networks. This thesis looks into the study of how matrix size, sparsity pattern, and storage format affect the performance of parallel communication collectives and sparse matrix multiplication. Using the SUMMA algorithm as a baseline, we evaluate communication and computation costs across a range of matrix aspect ratios, sparsity patterns, and sparse storage formats, including CSR and CSC. Our experiments use both synthetic matrices with controlled density and structure, as well as Kronecker-generated graphs, and are implemented using collective communication primitives in MPI. Furthermore, we look to determine the limitations imposed by matrix shape and sparsity on statistics such as memory bandwidth, communication overhead, and overall multiplication performance.

Chapter 1

Introduction

Many real world computational problems are naturally represented as a series of sparse matrices. In these matrices, the majority of entires are zero, and thus only a small subset of elements actually contain meaningful information. These matrices arise in various domains, including graph analysis, computer graphics, science simulations, network modeling, and machine learning [1]. In many of these applications, the sparse matrices represent the structure of some underlying system, such as the adjacency matrix of a graph.

Unlike dense matrices, where every element contributes equally to computation and communication, sparse matrices encode structural information about the data. For example, the adjacency matrix of a social network contains nonzero entries only where connections exist between users in the network. A basic example of a dense matrix and a sparse matrix side by side is visualized in Figure 1.1.

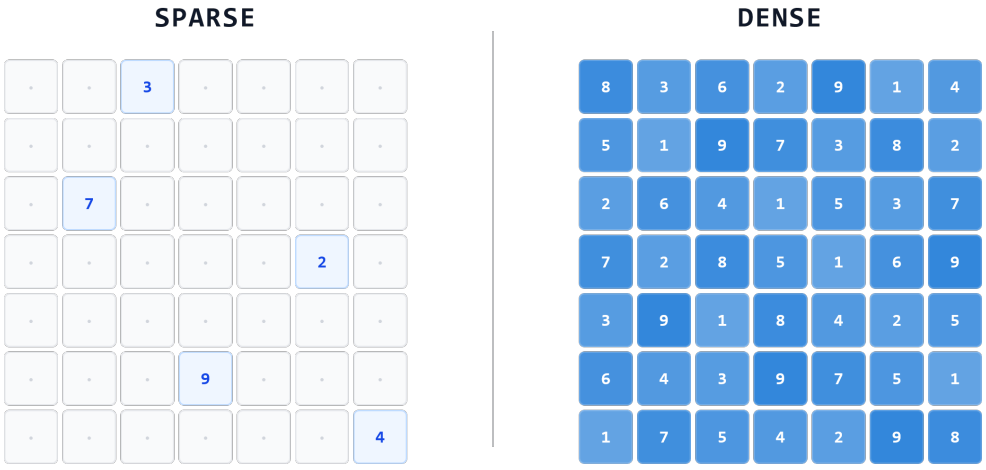


Figure 1.1: Example of dense and sparse matrix structures. Sparse matrices contain far fewer nonzero elements.

Because sparse matrices only store nonzero entries, they offer significant advantages in both memory usage and computational cost. However, the irregular nature of sparsity introduces challenges that do not appear in dense matrix computations, such as

1. Irregular memory access
2. Uneven distribution of elements
3. Additional metadata for representing these matrices
4. Unpredictable communication patterns in parallel systems

So algorithms designed for dense matrices cannot always be applied directly to sparse matrices. Understanding how the sparsity of a matrix, or the amount of nonzero values, interacts with algorithm design is a critical problem in modern day high performance computing.

1.1 Structure and Patterns

A key characteristic of sparse matrices is that their performance is not only determined by the number of nonzero entries, but also by how the entries are arranged. Two matrices may contain the same number of nonzero elements, however, due to their sparsity pattern may have drastically different performance characteristics. There are various common sparsity patterns that will be discussed in detail, but a few examples include:

1. Uniform Random Sparsity: Nonzero elements are distributed evenly across the matrix.

2. Banded Structures: Nonzero entries cluster near the diagonal of the matrix.

3. Block Structures: Nonzero entries form dense sub regions of the matrix.

4. Kronecker Product Matrices: Mimic the structure of many real world graphs through a repeated product of itself.

Figure 1.2 showcases two different sparsity structures, a social network and a fem stiffness matrix, side by side. The fem stiffness matrix also demonstrates a specific class of sparsity pattern, a banded matrix, which will be discussed in further detail later.

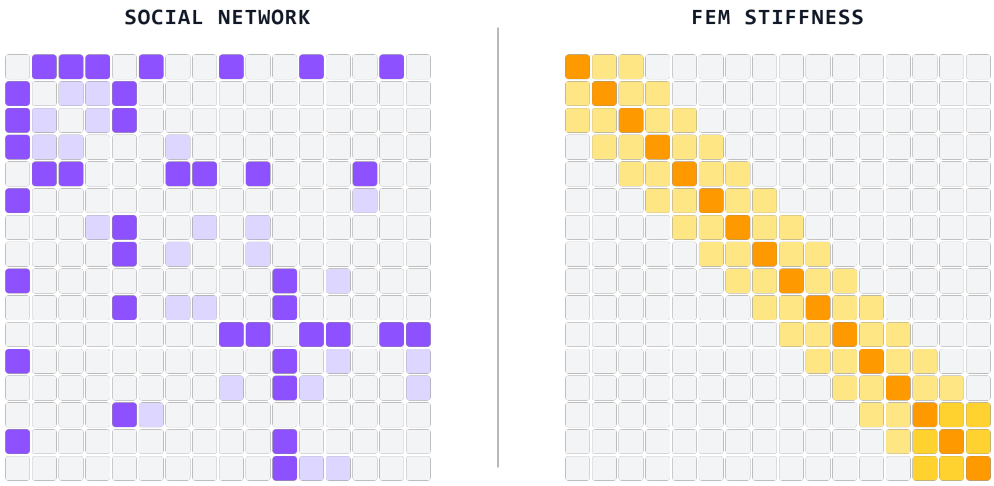


Figure 1.2: Two different sparsity patterns commonly observed in real world matrices. The left matrix represents a social network, the right matrix is a stiffness matrix which represents a system of linear equations.

1.2 Distributed Sparse Matrices

Many applications that use sparse matrices often operate at scales that exceed the capabilities of a single machine. With the ever increasing dependency on Artificial Intelligence, this is becoming the truth more than ever due to the extensive workloads that span across distributed networks of machines.

In these distributed systems, data is exchanged between processes to complete computations. These exchanges are typically implemented using collective communications, which are critical to coordinating communication across groups of processes and machines. Examples of this to be discussed in detail later include broadcast, all-gather, reduce-scatter, and all-reduce. Figure 1.3 showcases an example MPI Broadcast operation, where one process sends its data to all other processes in the connected network. For dense operations, these collective communications are relatively predictable because the data is often stored in contiguous arrays. Sparse matrices, however, introduce additional complexity. Since sparse matrices are stored with only nonzero entries alongside

the indexing metadata, communication often contains:

1. values and index information
2. irregular message sizes
3. data dependent communication patterns

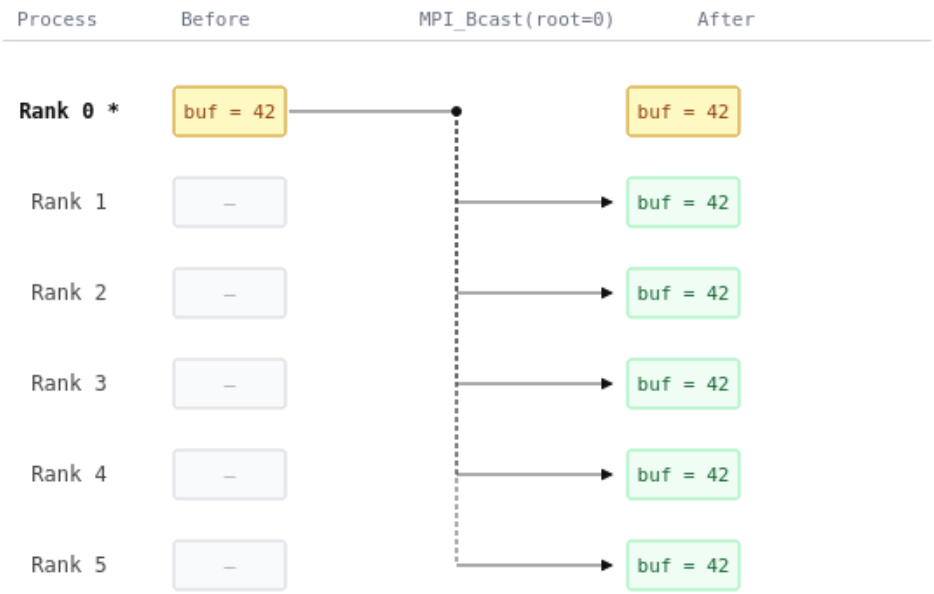


Figure 1.3: The broadcast collective communication, showing that one process sends all of the data to other connected processes.

1.3 Thesis Outline

This thesis will primarily consist of work on

1. Distributed dense and sparse matrices.
2. Collective communication operations as a measurement for communication cost for sparse algorithms.

3. Parallel matrix multiplication using the SUMMA algorithm to evaluate both computation and communication.

4. Matrices with controlled sparsity patterns.

And will be structured by building on sparse concepts from sparsity patterns to distributed matrix multiplication

1. Chapter 2 provides theoretical and practical context for sparse matrix formats, collective communications, and distributed algorithms.

2. Chapter 3 analyzes how collective communication operations behaves when applied to sparse data rather than dense arrays.

3. Chapter 4 extends the analysis to sparse SUMMA, examining how sparsity interacts with the algorithms two phase broadcast and multiply structure.

4. Chapter 5 describes the experimental methodology. It covers the matrix shapes and patterns used in testing, the storage formats and variants implemented, as well as the hardware and software environment and tools.

5. Chapter 6 presents and analyzes the experimental results. It evaluates the impact of sparsity pattern, sparsity shape, and storage format on both collective communication and SUMMA performance.

6. Chapter 7 presents and analyzed the experimental results specifically for Kronecker matrices. It evaluates many of the same parameters as in Chapter 6, with a focus on results that can be used for algorithm references.

7. Chapter 8 outlines the role of visualization in the communication of complex sparse and distributed behavior and problems.

8. Chapter 9 summarizes the contribution of this thesis and summarizes previous work.

Chapter 2

Background

2.1 Sparse Matrix Representations

2.1.1 Coordinate (COO) Format

The Coordinate format is one of the simplest representations for sparse matrices [1]. In the COO format, each nonzero element is simply stored as a tuple containing its row index, column index, and value. Thus, the underlying structure for a COO Matrix may look like:

```
struct COOMatrix {
    int rows, cols, nnz;
    std::vector<int> row_indices;
    std::vector<int> col_indices;
    std::vector<float> values;
};
```

Listing 1: An example COO representation in C++.

This structure is easy to construct and is commonly used as an intermediate format when building sparse matrices [1, 2]. However, the COO format does not support any

efficient random access or row/column traversal due to the nature of how the data is stored, and thus is not often used for computations and is converted to other formats such as CSR or CSC. Figure 2.1 shows an example of a COO matrix and the individual mapping to the lower level arrays.

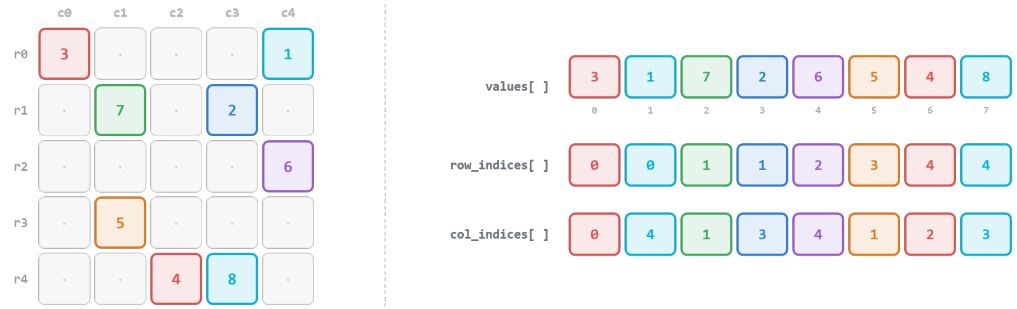


Figure 2.1: An example of the COO format, showing the underlying data and visual index associations.

2.1.2 Compressed Sparse Row

The Compressed Sparse Row (CSR) format is one of the most widely used sparse matrix representations in high performance computing [1]. In the CSR format, each nonzero element is stored as a tuple containing its value, column index, and compressed row index. Thus, the underlying structure for a CSR Matrix may look like:

```
struct CSRMatrix {
    int rows, cols, nnz;
    std::vector<int> row_ptrs; // length rows + 1
    std::vector<int> col_indices; // length nnz
    std::vector<double> values; // length nnz
};
```

Listing 2: An example CSR representation in C++.

The main compression outside of only storing non zero values comes from the *row_ptrs* array. Instead of storing a row index for every nonzero element like the COO format,

CSR only stores the starting offset for each row. This reduces the row index storage from $O(nnz)$ to $O(m)$ which becomes a significant factor as $nnz > m$. Retrieving all nonzero elements in some row i only requires a slice of values from $row\_ptrs[i]$ to $row\_ptrs[i+1]$ and thus row traversal is simply a $O(nnz_i)$ operation where nnz_i is the number of nonzeroes in that row. An example of this format can be seen in Figure 2.2 which illustrates the mapping between the visual structure of a matrix and the three underlying arrays.

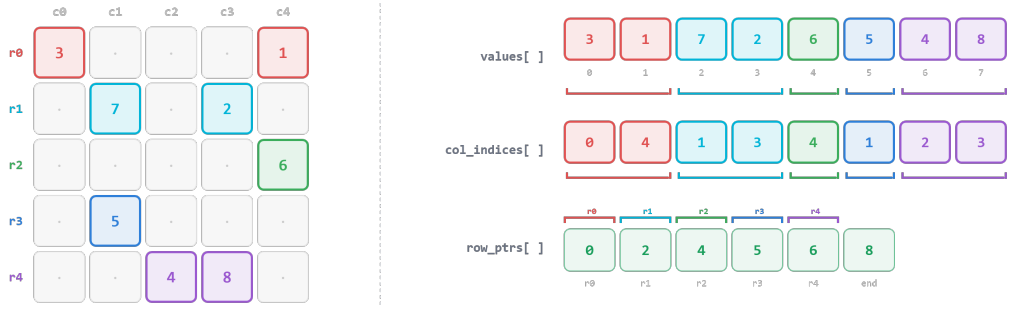


Figure 2.2: An example of the CSR format, showing the underlying data and visual index associations.

The row oriented access pattern makes CSR well suited for operations that proceed row by row, such as sparse matrix vector multiplication (SpMV) and the row broadcasts that will be discussed later in the SUMMA algorithm. Conversely, random access to individual elements requires a binary search within the *col_indices* array for a given row and thus is a $O(log(nnz_i))$ operation. As a result of this, CSR is typically only used for computational problems rather than constructing or modifying sparse matrices in place.

2.1.3 Compressed Sparse Column

The Compressed Sparse Column (CSC) format is the column major variant of the CSR format. The underlying arrays are identical in concept but transposed to compress the columns rather than the rows, as demonstrated below:

The CSC format follows all of the same computational behavior as CSR but with respect to columns instead of rows and thus makes CSC the natural format to be used

```cpp
struct CSCMatrix {
    int rows, cols, nnz;
    std::vector<int> col_ptrs; // length cols + 1
    std::vector<int> row_indices; // length nnz
    std::vector<double> values; // length nnz
};
```

Listing 3: An example CSC representation in C++.

when an algorithm needs to iterate over columns. This is similarly useful for the soon to be discussed SUMMA algorithm as there are points within the algorithm that require columns to be broadcast down the process grid and using the CSC format makes that a contiguous scatter operation.

CSC is also found in sparse matrices that arise from graph problems when adjacency is expressed column wise. For instance, in certain variants of graph neural networks or PageRank where incoming edge lists are column indexed [3, 4]. Figure 2.3 shows an example of the format visually.

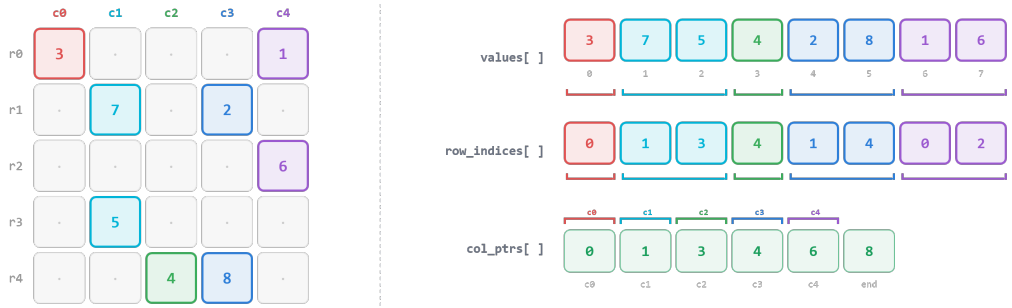


Figure 2.3: An example of the CSC format, showing the underlying data and visual index associations.

2.2 Sparsity Patterns and Matrix Structure

As established in Chapter 1, two sparse matrices with the same number of nonzero elements can exhibit very different performance characteristics depending on where those

elements are located. This distribution of nonzero entries is referred to as the sparsity pattern. This section describes many of the major sparsity patterns encountered in practice and discusses how each arises in the real world and what computational challenges they introduce.

2.2.1 Uniform Random Sparsity

Uniform Random Sparsity is the simplest sparsity pattern and serves as a good baseline against which all other patterns can be compared. In a uniformly random sparse matrix, every entry is independently nonzero with some probability p, where p is the target density of the matrix. This produces a sparsity pattern in which there is no regularity as nonzero entries are statistically scattered evenly across rows and columns. So, there should be no clustering, banding, or hierarchical structure. Figure 2.4 visualizes an example of a uniformly random sparse matrix.

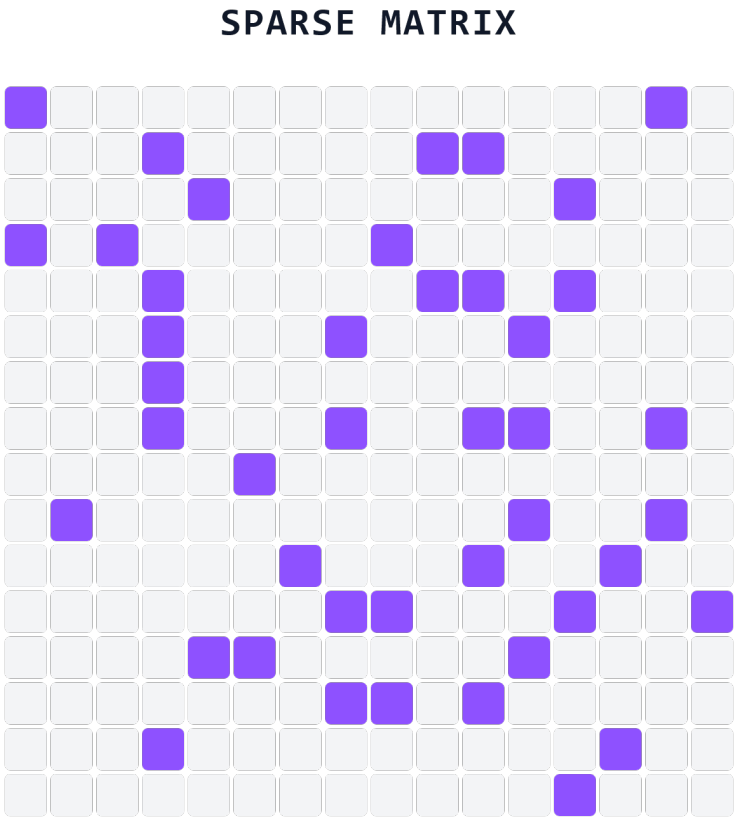


Figure 2.4: An example of a uniformly random sparse matrix.

In practice, truly uniform random sparsity is rare in real world data. Its primary utility, especially in this thesis, is used as a controlled experimental condition. Because the expected number of nonzeroes per row and column is constant and essentially equal to $p \times dimension$, a uniformly random sparse matrix distributes the load evenly across processes and avoids any imbalance effects. Similarly for collective communications, uniformly random sparsity presents the most predictable case. Since each process receives approximately the same amount of nonzero entries, variance in message sizes and thus costs are low relative to structured patterns. And so it can be most similarly applied to the dense case as it will have nearly uniform scaling.

2.2.2 Banded Sparsity

Banded sparsity can be characterized by nonzero entries that are concentrated near the main diagonal of the matrix. Banded matrices arise naturally from the discretizations of differential equations over regular grids. For example, a one dimensional finite difference discretization of the Poisson equation produces a tridiagonal matrix, while a two dimensional discretization on an $n \times n$ grid produces a matrix with a band width of approximately n [1, 5]. The fem stiffness matrix shown in Figure 1.2 is an example of this sparse matrix pattern.

Compared to Uniform Sparse Matrices, banded sparsity introduces a number of favorable properties when it comes to computation and communication. Both row wise and column wise access, depending on the format the matrix is stored in, can be efficient as the non zero entries would be stored in contiguous index space. For example, when a banded sparsity matrix is stored in the CSR format it would be efficient to access entire rows of nonzero entries as they are stored contiguously. Conversely, trying to access columns of data in a banded matrix stored in CSR format would be extremely inefficient. This also introduces natural load balancing as it would be efficient to send bands of the matrix at a time. Because both CSR and CSC provide efficient access to banded matrices, the choice between them will be driven by the access pattern of the algorithm consuming the matrix, and in particular whether or not it is column wise or row wise.

2.2.3 Block Sparsity

Block sparsity refers to matrices in which nonzero entries are organized into dense rectangular sub blocks. This structure is commonly observed in graph problems where nodes represent collections of entities and in some neural network weight matrices. Figure 2.5

visualizes an example of a sparse matrix with a block sparsity pattern.

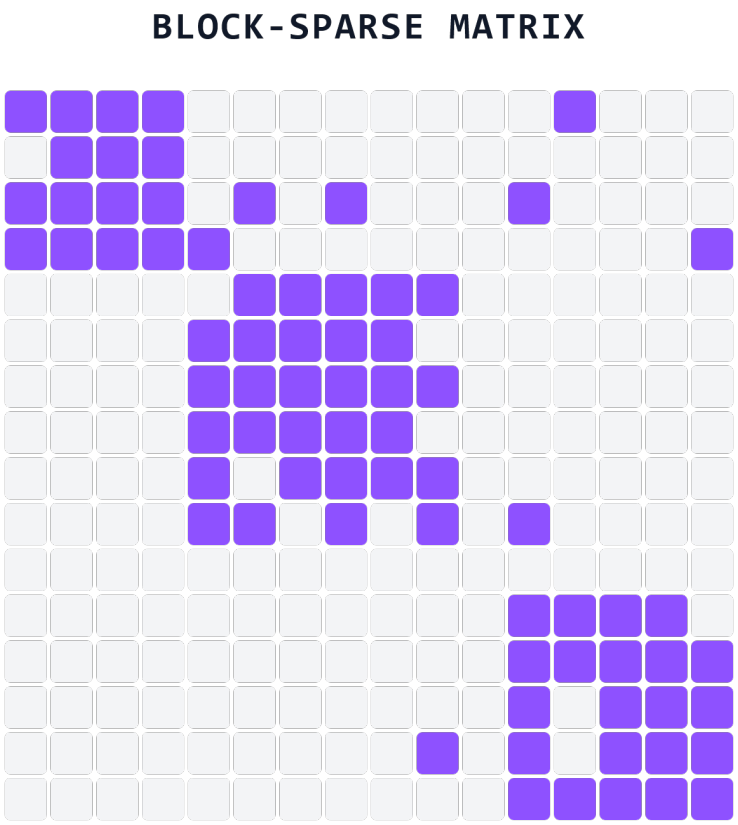


Figure 2.5: An example of a block sparsity matrix.

In practice, block sparsity offers several computational advantages over unstructured patterns. Because non zero entries are concentrated into dense rectangular regions, the local computation within each block can be handled by optimized BLAS kernels rather than general sparse routines [6]. This allows these algorithms to exploit both hardware level parallelism and the cache in ways that unstructured patterns cannot. Additionally, while not covered in this thesis, specialized formats like BCSR (Block Compressed Sparse Row) which extends CSR by grouping non zeroes into fixed blocks can also be used which reduce metadata overhead and allow for better memory access patterns [7].

For collective communications, block sparsity is advantageous as it partially helps with the message size predictability problem. When blocks align with the process bound-

aries, the number of non zeroes per process is more accurately determined by the block dimensions and thus can make buffer preallocation straightforward. The tradeoff however is that if blocks are not aligned carefully with the process grid than it can cause them to be split which destroys any advantages this solution created, and thus process grid alignment is an important consideration when using this format.

2.2.4 Kronecker Product Sparsity

A Kronecker product matrix is constructed by repeatedly applying the Kronecker product to a smaller initiator matrix [8, 9]. Given some initiator matrices A of size $m \times n$ and B of size $p \times q$, their Kronecker product $A \otimes B$ is a matrix of size $mp \times nq$ in which each entry a_{ij} of A is replaced by the scaled block $a_{ij} \times B$. Repeated application of this operation thus produces large matrices with a similar, hierarchical sparsity structure, as visualized in Figure 2.6.

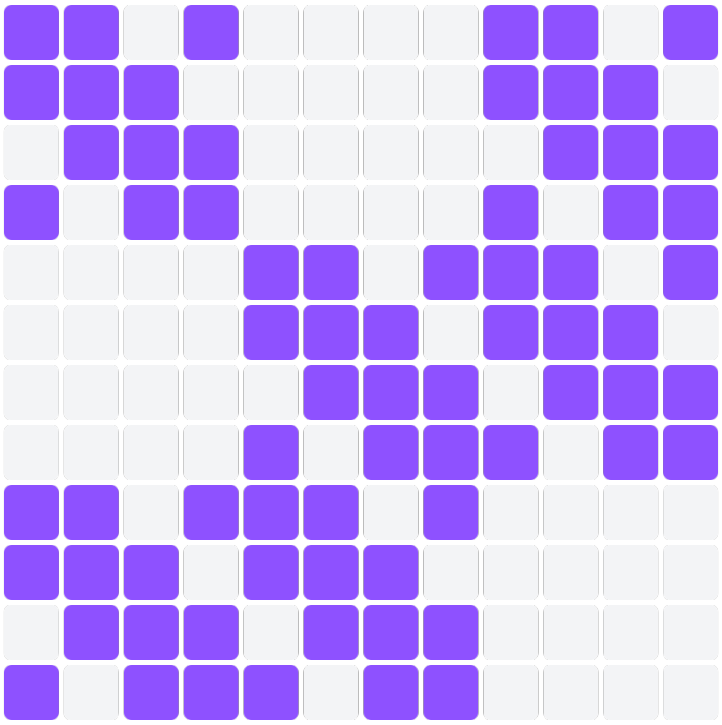


Figure 2.6: An example of a simple kronecker product matrix.

Kronecker product matrices are widely used in graph benchmarking, most notably in the Graph500 benchmark [10], because the resulting adjacency matrices present properties similar to real world graphs such as: skewed degree distributions and clustering. These properties make Kronecker graphs a more realistic and insightful test case as compared to uniformly random matrices. However, the same structure that makes them realistic also causes a significant load imbalance in our distributed setting since portions of the rows and columns with have far fewer non zero entries than the rest. This is discussed further in Chapter 3.

For all experiments performed on Kronecker matrices, they are generated using the Kronecker initiator matrices estimated by Leskovec et al. [9] from real world networks such as social graphs, topology graphs, and citation networks. Each initiator matrix

essentially encodes a structure of the source network. For each network, Kronecker powers from $k = 1$ to $k = 23$ are generated which produce matrices whose node count grows as N_1^k. So, even as the k grows the structural properties of the original source network will still be maintained, making it useful to determine what a network might behave like under similar conditions at varying node counts. Additionally, multiple samples of each k were generated and averaged to reduce the effect of sampling variance. The full set of source networks, their parameters, and their properties are described in Table 2.1.

Table 2.1: Kronecker source networks used to generate experimental matrices in this thesis. Network properties and estimated initiator parameters $\hat{\Theta}$ are taken from Leskovec et al. [9]. $S_1 = \sum_{ij} \theta_{ij}$ determines the densification rate across Kronecker powers.

Network	**Type**	**N**	**E**	$\bar{D}$	$\hat{\Theta}$ $(a,\ b,\ c,\ d)$	S_1
AS-ROUTEVIEWS	Internet	6,474	26,467	3.72	(0.987, 0.571, 0.571, 0.049)	2.178
AS-NEWMAN	Internet	22,963	96,872	3.83	(0.954, 0.594, 0.594, 0.019)	2.161
GNUTELLA-25	P2P	22,687	54,705	5.57	(0.746, 0.496, 0.654, 0.183)	2.079
GNUTELLA-30	P2P	36,682	88,328	5.75	(0.753, 0.489, 0.632, 0.178)	2.052
EPINIONS	Social	75,879	508,837	4.27	(0.999, 0.532, 0.480, 0.129)	2.140
ANSWERS	Social	598,314	1,834,200	5.72	(0.994, 0.384, 0.414, 0.249)	2.041
DELICIOUS	Social	205,282	436,735	6.28	(0.999, 0.327, 0.348, 0.391)	2.065
FLICKR	Social	584,207	3,555,115	5.42	(0.999, 0.474, 0.485, 0.144)	2.102
ATP-GR-QC	Bipartite	19,177	26,169	11.08	(0.902, 0.253, 0.221, 0.582)	1.958
CA-GR-QC	Collab	5,242	28,980	6.10	(0.999, 0.245, 0.245, 0.691)	2.180
CA-HEP-PH	Collab	12,008	237,010	4.71	(0.999, 0.437, 0.437, 0.484)	2.357
CA-HEP-TH	Collab	9,877	51,971	5.96	(0.999, 0.271, 0.271, 0.587)	2.128
CA-DBLP	Collab	425,957	2,696,489	6.75	(0.999, 0.307, 0.307, 0.574)	2.187
BIO-PROTEINS	Biological	4,626	29,602	4.24	(0.847, 0.641, 0.641, 0.072)	2.201
BLOG-NAT05-6M	Citation	31,600	271,377	3.40	(0.999, 0.569, 0.502, 0.221)	2.291
BLOG-NAT06ALL	Citation	32,443	318,815	3.94	(0.999, 0.578, 0.517, 0.221)	2.315
CIT-HEP-PH	Citation	30,567	348,721	4.33	(0.994, 0.439, 0.355, 0.526)	2.314
CIT-HEP-TH	Citation	27,770	352,807	4.20	(0.990, 0.440, 0.347, 0.538)	2.315
WEB-NOTREDAME	Web	325,729	1,497,134	7.22	(0.999, 0.414, 0.453, 0.229)	2.095

2.3 Distributed Memory Parallelism (MPI)

As noted in Chapter 1, in the modern age of computing single core algorithms are a relic of the past with modern CPUs being capable of performing hundreds of operations in parallel and GPUs being able to perform millions of operations in parallel. Additionally, many modern computational problems are often distributed across dozens of individual machines (nodes or ranks) which in and of themselves may contain multiple instances of a program working on a problem. This type of distributed memory system is critical for problems where a single machines worth of memory is physically impractical as problem sizes reach requirements in the hundreds of gigabytes.

The Message Passing Interface (MPI) is the standard API for programming these distributed memory systems [11]. MPI provides a portable, vendor neutral specification for sending and receiving messages between processes as well as an extensive set of collective operations that coordinate communication across groups of processes. OpenMPI is a widely used open source implementation of the MPI standard that provides high performance communication[12]. All distributed experiments in this thesis are implemented using OpenMPI.

In an MPI program, processes are identified by their rank within a communicator, which is a group of processes that can communicate with one another. A global communicator, MPI_COMM_WORLD, is defined as all processes that are launched by the MPI runtime. Custom communicators are available to be created for problems that see communication across common sets of processes, such as when rows and columns are assigned to individual processes and need to be distributed. Processes in the MPI runtime interact directly via a collection of MPI_Send and MPI_Recv operations, often wrapped as a set of collective communications. The simple point to point example in Listing 4 illustrates an example of Rank 0 sending an integer to Rank 1.

```cpp
// Fetch the local instances rank and the total world size
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int value = 0;
if (rank == 0) {
    value = 42;
    MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    std::cout << "Process 0 sent value " << value << std::endl;
    // "Process 0 sent value 42"
}
else if (rank == 1) {
    MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
    ↪ MPI_STATUS_IGNORE);
    std::cout << "Process 1 received value " << value << std::endl;
    // "Process 1 received value 42"
}
```

Listing 4: Simple point to point communication using MPI

2.4 Collective Communications

As noted previously, Collective Communication operations are designed to get a group of nodes to consolidate or distribute data. The operations are typically divided into two main categories: data redistribution operations (broadcast, scatter, gather, all gather) which move data around without any arithmetic, and data consolidation operations

(reduce, reduce-scatter, all reduce) which combine contributions from all nodes using operations like summation. It should be noted that this is not an exhaustive list of all collective communications, but a subset that is discussed in this thesis.

The cost of any of these collectives is analyzed under the $\alpha - \beta$ model, where sending some message of n items costs $\alpha + n\beta$ [13]. The term α is indicative of the per message start latency and β is the communication time per item. For reduction operations, there is also an extra term γ which is an additional per item compute cost.

Adapting the following collectives to operate on sparse formats introduces a completely different cost analysis which completely depends on the sparsity pattern of these matrices. Without a known sparsity pattern, doing static analysis of the cost is unpredictable. Implementation of these algorithms in the sparse world will be discussed further in Chapter 3.

2.4.1 MST Broadcast, Scatter, Gather, and Reduce

The minimum spanning tree (MST) algorithms are the standard for short vector algorithms[13] for these four operations. The latency lower bound $\lceil \log_2 p \rceil \alpha$ as noted by Chan is achieved by halving the number of nodes that need to be communicated at every step. The main algorithm is a recursive partition such that at each step, p nodes are split into a left half $\{0, \ldots, m\}$ and a right half $\{m+1, \ldots, p-1\}$ where $m = \lfloor p/2 \rfloor$. A root node from each half is chosen and a single message is exchanged across the two roots. This algorithm continues recursively choosing new roots until the operation is complete. Because the two separate halves proceed in parallel, the max depth of the communication is $\lceil \log_2 p \rceil$.

2.4.1.1 Broadcast

In the MST Broadcast operation, the root node holds some vector x that must be sent to every other node. At each level of recursion, the current root node sends the full vector to a node in the other half, after which both halves recursively perform that same operation. As shown in Figure 2.7, with a root of $n = 0$ and $p = 4$, the first step sends x from $n = 0$ to $n = 3$. The next step sends x simultaneously from $n = 0$ to $n = 1$ and from $n = 3$ to $n = 2$.

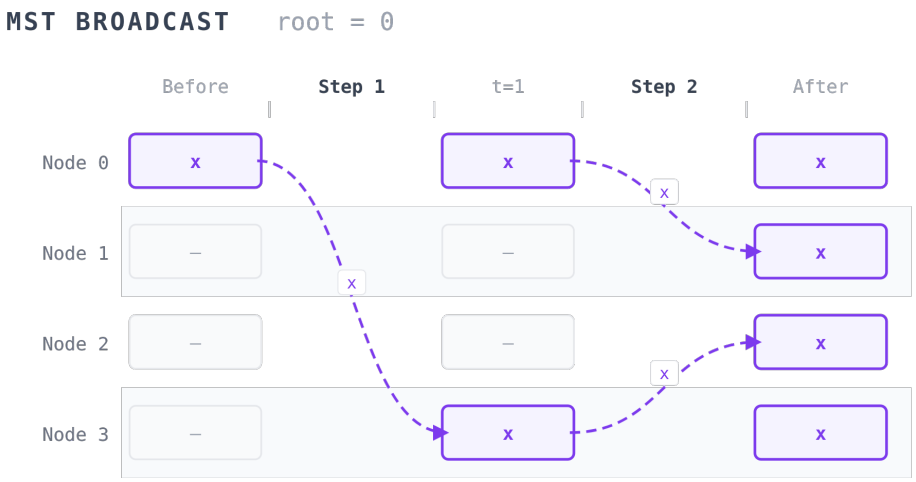


Figure 2.7: MST Broadcast on $p = 4$ nodes with root at Node $n = 0$. Node $n = 0$ sends x to Node $n = 3$ in Step $n = 1$, then both nodes fan out to the remaining two nodes in Step 2, achieving full coverage in $\lceil \log_2 p \rceil = 2$ rounds.

2.4.1.2 Scatter

The MST Scatter operation is a simple modification of a MST Broadcast, in which the root only sends a portion of the data to the receiving node rather than the entire vector. This is shown in Figure 2.8: Node $n = 0$ sends x_2, x_3 to $n = 3$ in Step 2, then sends x_1 to $n = 1$ while Node $n = 3$ sends x_2 to Node $n = 2$.

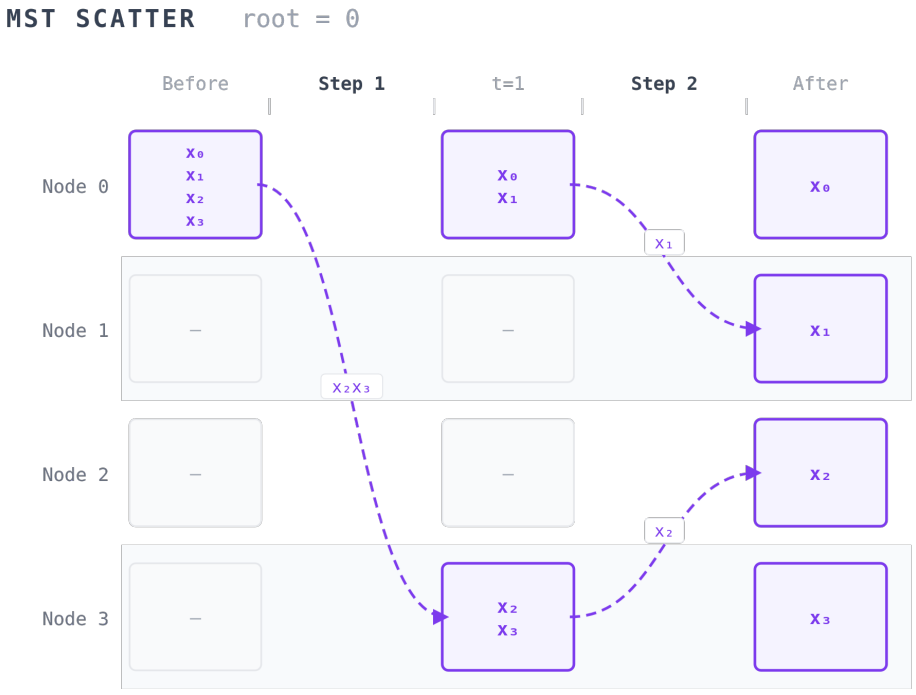


Figure 2.8: MST Scatter on $p = 4$ nodes with root at Node $n = 0$ holding $x = (x_0, x_1, x_2, x_3)$. Only the subvector belonging to each subnetwork is transmitted at each step, halving the message size at every level.

2.4.1.3 Gather

The MST Gather operation is simply the inverse of the MST Scatter operation, such that after every step the amount of elements increases as the recursion reaches the target root node. Figure 2.9 shows that Node $n = 1$ sends x_1 to Node $n = 0$ and Node $n = 2$ sends x_2 to Node $n = 3$ in Step 1, then Node $n = 3$ sends the combined x_2, x_3 to Node $n = 0$ in Step 2.

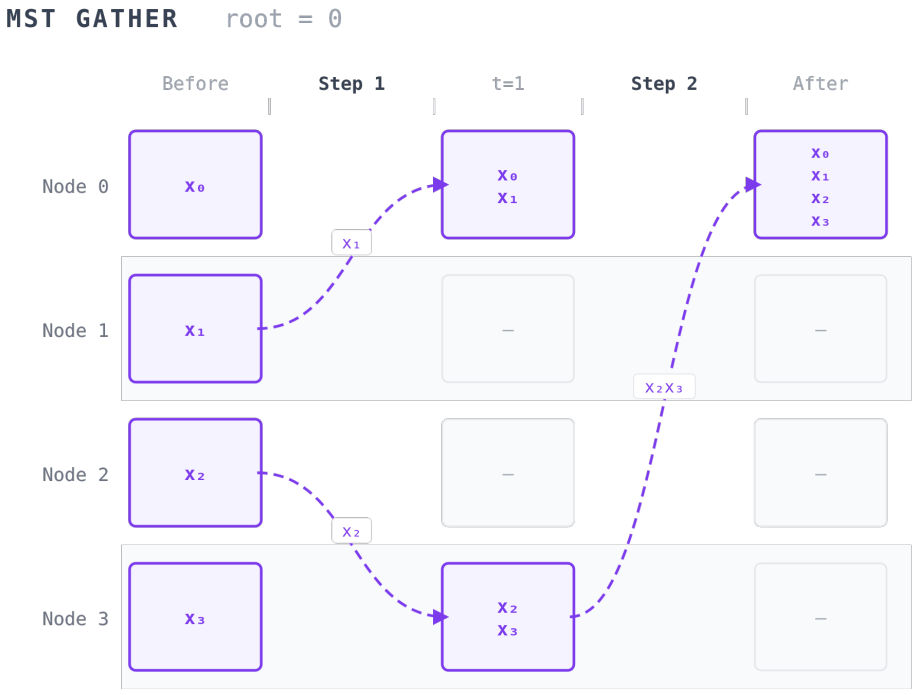


Figure 2.9: MST Gather on $p = 4$ nodes, collecting subvectors into Node $n = 0$. Messages travel toward the root, with each intermediate node forwarding an increasingly complete subvector up the tree.

2.4.1.4 Reduce

Finally the MST Reduce operation is a modification of an MST Broadcast. Whenever a node receives some value x, it will perform an operation such as summation on the incoming x with its own value, and then broadcast that value towards the other halves root node. Figure 2.10 in Step 1 has Node $n = 1$ send $x^{(1)}$ to Node $n = 0$, and Node $n = 3$ send $x^{(3)}$ to Node $n = 2$, forming two partial sums; Step 2 has Node $n = 2$ send the partial sum $x^{(2)} + x^{(3)}$ to Node $n = 0$, which adds it to its own partial sum to obtain the total. As previously mentioned, due to the addition of a computational cost an additional γ term is added to the total operation cost.

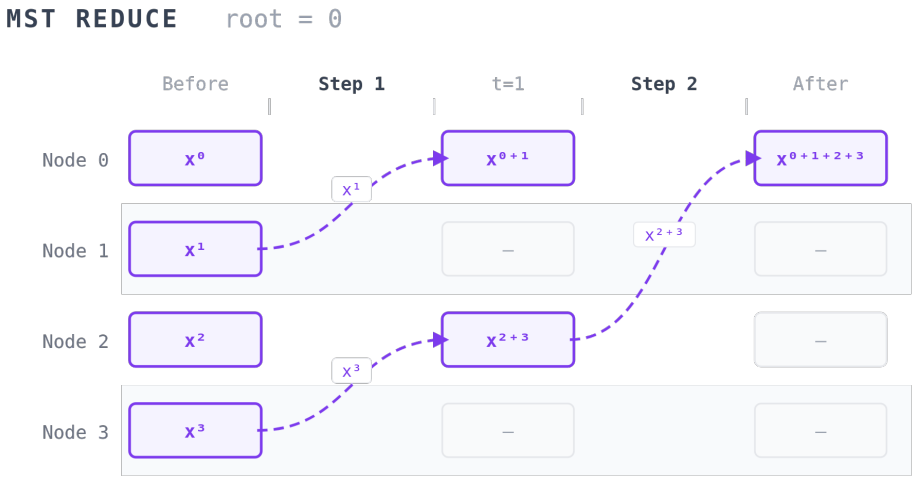


Figure 2.10: MST Reduce on $p = 4$ nodes with root at Node $n = 0$. Partial sums are accumulated bottom up: each node reduces within its half before forwarding the result toward the root, performing $\lceil \log_2 p \rceil$ additions in total.

2.4.2 Bucket All Gather and Reduce Scatter

The bucket algorithms are the standard approach for bandwidth optimal implementations of the All Gather and Reduce Scatter collectives. Unlike the MST algorithms which prioritize achieving optimal latency by minimizing the communication rounds, the bucket algorithms focus on minimizing the total volume of data transmitted per process. This matters a lot whenever the message size is large, as for example the MST broadcast operation sends the full message along every edge of the spanning tree, while the bucket approach routes each chunk of data once.

In both the bucket All Gather and Reduce Scatter collectives, all p processes are arranged logically in a ring topology, where the algorithm proceeds in $p - 1$ steps. In each step, each process simultaneously sends data to its right neighbor and receives data from its left neighbor. Each message thus contains at most $\frac{1}{p}$ of the total data, so the per step communication volume is $\frac{n}{p}$ items.

The Reduce Scatter operation is equivalent to performing a distributed reduction such that after the algorithm completes, each process holds the reduced result for its assigned $\frac{1}{p}$ chunk of the output data.

The All Gather is the inverse of the Reduce Scatter collective. Each process begins with some distinct $\frac{1}{p}$ chunk of the final result and must receive all other chunks. In each step, every process forwards the most recently received chunk clockwise around the ring, such that after $p-1$ steps every process has receives every chunk of the total data.

For sparse data, the bucket algorithms present a challenge. Because each step transmits a fixed $\frac{1}{p}$ slice of the data, the slicing must be defined in terms of the sparse structure. For example, if the data is partitioned by rows then each step transmits approximately $\frac{nnz}{p}$ nonzero values, but is still fully dependent on how the nonzero entries are distributed across row slices. A highly skewed sparsity pattern can cause some steps to transmit for more data than others, destroying the balance that makes the bucket algorithm communication efficient.

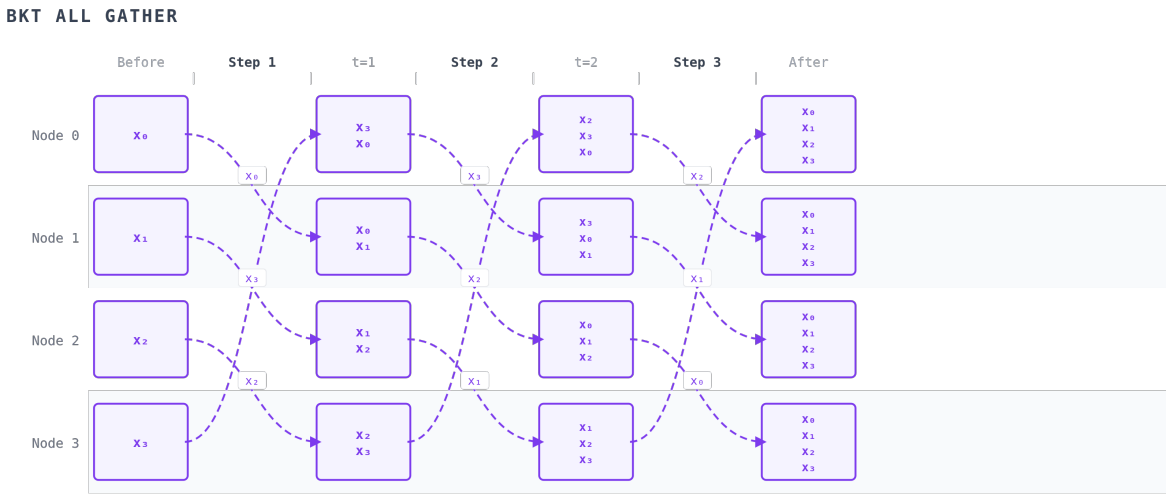


Figure 2.11: BKT Allgather on $p = 4$ nodes, collecting data from all nodes to all other nodes by passing to the next neighbor.

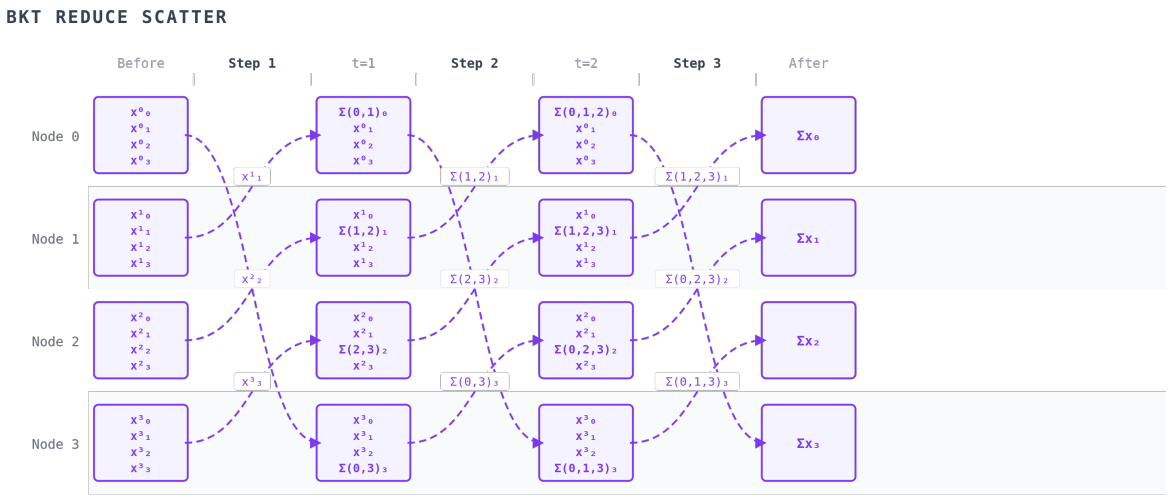


Figure 2.12: BKT Reduce-Scatter on $p = 4$ nodes, distributing data and reducing them based on the node index and data associated with that node.

2.5 Distributed Matrix Multiplication (SUMMA)

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) is a distributed matrix multiplication algorithm designed for two dimensional process grids [14]. Given a matrix multiplication $C = A \times B$ where A is $m \times k$, and B is $k \times n$, and C is $m \times n$, SUMMA distributed all three matrices across some $P_r \times P_c$ process grid. Each process (i, j) owns a local block of each matrix and proceeds by processing blocks of size b.

Choosing a two dimension process grid is an important part of SUMMA's design. A one dimension process grid would cause entire rows of A and C to be assigned to each process, which in turn requires every process to hold an $O(n^2)$ copy of B which eliminates any memory benefits. For a two dimension grid, this is solved because both input matrices are split across the process grid and thus requires processes to only hold no more than $O(n^2/P)$ memory.

The SUMMA algorithm operates in k/b steps, where b is the panel block size. At each step s, there are a few operations that are performed:

1. The process in column s of the grid broadcasts its local block of A along the entire process row, such that every process in that row receives the A panel of the column.

2. The process in row s of the the grid broadcasts its local block of B along its entire process column, such that every process in the column receives the B panel of the row.

3. Each prcoess computes the local outer product contribution $C_{ij} += A_{local} \times B_{local}$ accumulating into its local block of C.

Figure 2.13 demonstrates a single step of SUMMA where $k = 1$. The pseudocode in Listing 5 shows the corresponding OpenMPI implementation using MPI_BCast over individual row and column communicators for dense matrices.

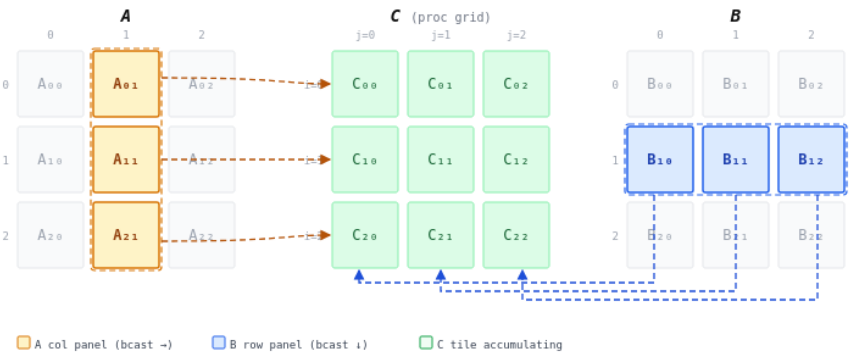


Figure 2.13: A single step of the SUMMA algorithm given that $k = 1$, showing both row and column distribution to the process grid.

2.5.1 Block Size Tradeoffs

One important note is the tradeoffs that come from adjusting the block size b. Smaller values of b increase the number of outer loop iterations to k/b, where the per step latency cost is then multiplied by the same factor (essentially, as b decreases the problem becomes

```
// Assume a 2D process grid: Pr x Pc
// Each process holds local blocks A_ij, B_ij, C_ij

// Create 2D Cartesian communicator
MPI_Comm grid;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &grid);

// Determine row and column communicators
MPI_Comm row_comm, col_comm;
MPI_Comm_split(grid, row_coord, col_coord, &row_comm);
MPI_Comm_split(grid, col_coord, row_coord, &col_comm);

// Initialize local matrices
Matrix A_local, B_local, C_local;
for (k = 0; k < num_blocks; k++) {
    // Broadcast A block across process row
    if (col_coord == k)
    {
        A_panel = A_local;
    }
    MPI_Bcast(A_panel, ..., root=k, row_comm);

    // Broadcast B block down process column
    if (row_coord == k)
    {
        B_panel = B_local;
    }
    MPI_Bcast(B_panel, ..., root=k, col_comm);

    // Local matrix multiply
    C_local += A_panel * B_panel;
}
```

Listing 5: Pseudocode for SUMMA matrix multiplication using OpenMPI

more latency bound). In constrast, larger values of b increase the latency per step and allows the local matrix multiply to work on larger panels of b which can offset the load to the local multiplication kernel. However, a larger b also means the amount of memory required to operate is scaled. In practice for dense matrices, the analysis of the block size for b has been studied for whatever achieves peak performance [14]. For sparse matrices,

this is complicated as blocks of size b may not always contain the same number of non zero entries based on the sparsity pattern.

2.5.2 Communication Cost

For dense matrices, the communication cost of SUMMA is well studied. Each of the k/b outer steps performs two MPI_BCast operations: one over the process row of P_c ranks and one over the process column of P_r ranks. Under the $\alpha - \beta$ model, broadcasting a message of some size n over p ranks using the MST algorithm costs $log_2 p\alpha + n\beta$. Over all of the steps, the total volume of communication can be described as $O(mn/P_c + kn/P_r)$.

> The overall communication cost given above is a simplification, and can be expanded out to the following.
>
> $$T_{comm} = (k/b) \times (\lceil log_2 P_c \rceil \alpha + (mb/P_r)\beta + \lceil log_2 P_r \rceil \alpha + (nb/P_c)\beta)$$

Adapting the SUMMA algorithm for sparse matrices complicates the algorithm at practically every step but introduces many new perspectives on the analysis of effects of sparsity patterns. The implementation of sparse SUMMA will be discussed further in Chapter 4.

2.6 SuiteSparse (GraphBLAS)

The local multiplication step in sparse SUMMA requires computing the product of two sparse panels at each broadcast. This multiplication is a sparse times sparse matrix multiplication (SpGEMM). Implementing a correct and efficient multiplication kernel for each sparse format is a fairly well studied but certainly not trivial problem. The core idea behind this thesis is not the multiplication itself, but rather the effects of sparsity variables on the multiplication. Rather than introduce a custom SpGEMM

implementation, this thesis uses the SuiteSparse:GraphBLAS as the local multiplication kernel [15].

SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS C API standard which was created to define sparse matrix and vector operations [16]. Its parallel implementation includes algorithms for sparse matrix multiply, element wise operations, and submatrix extraction, which handles multiple combinations of matrix storage formats without the requirement to convert between them by the calling function. This is a widely used library in the HPC world, and in fact, serves as the underlying graph engine in the RedisGraph database [17] and as the built in sparse matrix multiply in MATLAB R2021a [18].

At each SUMMA step, after the A and B panels have been broadcast to their relevant processes, they are wrapped in the libraries *GrB_Matrix* objects and passed to *GrB_mxm* which handles the multiplication. An example of what this looks like is shown below in Listing 6

```
matrix::CSRMatrix a = ...;
matrix::CSCMatrix b = ...;
GrBMatrix A_panel = matrix::CSRMatrix::to_grb(a);
GrBMatrix B_panel = matrix::CSRMatrix::to_grb(b);
GrBMatrix C_local;

GrB_mxm(
    C_local, // where does this output
    NULL, // defines a mask, we do not need one
    GrB_PLUS_FP64, // tells it to accumulate into the existing C matrix
    GrB_PLUS_TIMES_SEMIRING_FP64, // standard algorithm (semiring)
    A_panel,
    B_panel,
    NULL
);
```

Listing 6: Pseudocode for a sparse matrix multiplication

2.6.1 Related Work: Distributed SpGEMM Algorithms

The SpGEMM literature provides important context for understanding some of the design choices made in this thesis. Although the local SpGEMM step here is using SuiteSparse:GraphBLAS, the algorithms surrounding how distributed implementations partition and communicate the product will help inform a lot of the analysis of Sparse SUMMA in Chapter 4.

The main distributed SpGEMM algorithm used in this thesis is the Sparse SUMMA algorithm introduced by Buluç and Gilbert [19], which extends the dense SUMMA algorithm [14] to sparse matrices using a 2D block data distribution. In their work, they demonstrate that a 2D decomposition combined with sparse kernels achieves extreme scalability. This 2D decomposition is the same structure adopted here such that each process owns a 2D block of each matrix, and the outer product accumulation proceeds step by step through a series of broadcasts over row and columns. Their CombBLAS library [20] provides a core implementation of this algorithm and has been extended significantly in CombBLAS 2.0 [21] which adds communication techniques, multithreading, and GPU support.

One challenge of the 2D Sparse SUMMA problem is that its communication cost scales with the volume of nonzeros communicated at each broadcast step. Azad et al. [22] address this by introducing a 3D SpGEMM algorithm that places processes on a $\sqrt{P} \times \sqrt{P} \times c$ grid where a factor c reduces the communication volume at the cost of additional memory. Their first implementation of this 3D formulation shows significant speedups over 2D Sparse SUMMA across a range of matrix types. The key outcome here is that replicating the input matrices across the third dimension of the process grid allows each 2D problem to communicate less data per step.

Hussain et al. [23] extends the 3D approach to address another challenge of distributed SpGEMM at scale, that being the output matrix often has more nonzeros than

the inputs combined and in some applications the output does not fit in the memory of the target supercomputer. Their SpGEMM algorithm addresses this by batching the output computation, multiplying only a subset of output columns per batch, and integrating a distributed step that estimates memory requirements and determines the number of batches in advance.

Selvitopi et al. [24] develops a CPU x GPU distributed SpGEMM algorithm called pipelined Sparse SUMMA in the context of the HipMCL clustering application. This variant overlaps communication and GPU computation by having the CPU manage the internode broadcasts while the GPU accumulates partial products from prior steps. Their work is notable in that it retains the 2D Sparse SUMMA structure rather than adopting the 3D algorithm, observing that the cost of redistributing data for a 3D grid is unlikely to be useful in the sparse case. This observation aligns with the analysis in Chapter 4 where for highly sparse matrices where panels contain few nonzeros, the dominant cost is communication latency and startup overhead, not bandwidth, which means adding a redistribution phase to achieve a lower bandwidth cost may not help.

Hong and Buluç [25] take a different approach by revisiting sparsity aware 1D algorithms, which had previously been considered less competitive than 2D and 3D approaches for general sparse matrices. Their implementation uses MPI RDMA operations to mitigate the packing and unpacking overhead that makes 1D algorithms expensive in practice. For certain input configurations and sparsity regimes, their 1D implementation outperforms the state of the-art 2D and 3D implementations in CombBLAS. This result is particularly relevant to the analysis of structured matrices in this thesis as 1D sparsity aware algorithms can exploit the structure of banded or block sparse matrices to skip fetching entire row blocks that contribute no nonzeros to the output. However, the 1D approach also concentrates communication load on a smaller set of processes, which can blow up the load imbalance problem analyzed in Section 3.3.3.

Taken together, these works cover two designs that set apart this thesis from prior distributed SpGEMM research. First, the existing literature focuses almost exclusively on optimizing the SpGEMM kernel itself such as reducing communication volume, overlapping communication with computation, and tuning local kernels for specific hardware. This thesis instead treats SpGEMM as a fixed kernel and focuses on the outer communication structure, specifically how the sparsity pattern and sparsity level of the input matrices affect the broadcast operations that precede each local SpGEMM call. Second, prior work evaluates SpGEMM primarily on unstructured matrices drawn from graph applications, where nonzeros are approximately uniformly distributed. This thesis explicitly varies the sparsity pattern across uniform random, banded, block, and Kronecker product structures, providing a more controlled analysis of how structural properties of the input govern the communication and computation balance of Sparse SUMMA.

Chapter 3

Sparse Collective Communications

This chapter analyzes how the seven collective communication operations introduced in Chapter 2 behave when their payloads are sparse matrices rather than dense arrays. The goal here is not to report measurements or experiments but to build an understanding of where and why sparse data changes the cost structure of each collective.

3.1 Adapting Collectives or Sparse Data

Collective communication algorithms are inherently designed around the assumption that the message payload is some continuous array of fixed size elements, and thus every process in the communicator contributes or receives the same amount of data. Sparse matrices of course violate this assumption. Adapting collectives to sparse data therefore requires addressing three problems that are not typically present: variable message sizes, index metadata overhead, and data dependent communication patterns.

3.1.1 Variable Message Sizes

For dense collectives, the size of every message at every step of the algorithm is known statically from the matrix dimensions and the process count. For example, a broadcast

of a $m \times n$ matrix over p processes always moves exactly $m \times n \times sizeof(data)$ bytes. However, for a sparse matrix the message size depends on the number of nonzero entries in the region being communicated, which may vary between matrices, across steps of an algorithm, and across processes. This presents two main issues for sparse collectives.

First, the receiving process cannot preallocate buffers of the correct size without first being told how many nonzero values to expect. This adds an additional metadata exchange, although only typically a small integer, that must be sent ahead of the data. And then, between when it receives that metadata exchange and the full set of data it must allocate enough space for the incoming data. For short vector collectives where the main cost already comes from latency rather than bandwidth, this metadata exchange is a meaningful statistic that can impact performance. Secondly, standard MPI collective primitives such as MPI_Bcast and MPI_Scatter assume fixed message sizes across all processes. Sparse collectives must therefore use different variants that support variable counts for processes, or do additional processing before collectives can be used.

3.1.2 Metadata Overhead

A dense matrix can be transmitted simply as a single array of values because the position of each element is implicit from its offset in the buffer. A sparse matrix however must transmit additional information alongside values so that the receiver can construct the matrix. The cost of this metadata depends on the storage format.

The COO format transmits two full integer arrays alongside the values, the row_indices and col_indices, each of which are a length of nnz. The total message size for a COO matrix with nnz nonzeroes can therefore be calculated as $(2 \times nnz \times sizeof(dimension)) + (nnz \times sizeof(data))$ where $sizeof(dimension)$ is the size of the metadata arrays (typically an integer), and $sizeof(data)$ is the size of the values data type. CSR transmits a col_indices of length nnz and a row_ptrs of length $m+1$ whos total message size can be cal-

culated as $nnz \times sizeof(dimension) + (m+1) \times sizeof(dimension) + nnz \times sizeof(data)$. CSC is symmetric to CSR with col_ptrs of length $n + 1$ replacing row_ptrs.

The implication here is that COO always transmits more metadata than CSR and CSC for the same matrix, regardless of sparsity pattern or shape. However, both CSC and CSR can encounter scenarios where the matrix pattern does not allow for maximum compression to be achieved. For example, when using CSR with taller matrices where $m > nnz$ the rows_ptr overhead can be quite significant.

3.2 Data Dependent Patterns

In dense collectives, the communication schedule is determined entirely by the algorithm and the process count. The data itself has no influence on the communication pattern. In sparse collectives, the data influences the communication in two ways.

For scatter and gather operations, the amount of data sent or received to a process depends entirely on how many nonzeroes fall within that process's assigned partition. A perfectly uniform sparsity distribution would give each process an equal share, however, real sparsity patterns are rarely uniform. For example, banded matrices distribute nonzeroes evenly across rows but unevenly if the band does not align with the partition boundaries. This data dependent load imbalance means the slowest process in the communicator determines the completion time of the collective and thus the degree of imbalance is tied directly to the sparsity pattern.

On top of that, the storage format also determines how efficiently the initial process can construct the message for each recipient. For a row partitioned scatter, the CSR format can easily extract the rows assigned to each process in $O(1)$ index lookups followed by fast contiguous memory copies. COO on the other hand must scan all nnz entries to identify which belong to each partition and thus costs $O(nnz)$ per partition. Following

this row partitioned scatter, the CSC format must reconstruct these rows from a column oriented structure, requiring a full scan across all columns. So, the format of these matrices not only effects the size of a message but also the time it takes to construct it and reconstruct it, contributing to the total observed communication cost.

3.3 Communication Cost Analysis

This section is going to continue upon the cost model introduced in Chapter 2 to account for the overhead introduced by sparse data. For each collective, we will approximate a cost function based on the number of non zeroes nnz, the matrix dimensions $m \times n$, the process count p, and the storage format.

3.3.1 Cost Model for Sparse Data

From Chapter 2 we determined that under the $\alpha - \beta$ model sending a message of size n bytes costs $\alpha + n\beta$ where α is the per message startup latency and β is the per byte transmission cost. For dense matrices, n is fixed and determined entirely by the dimensions of a matrix and data type. For sparse matrices however, n depends on the number of nonzero entries and the storage format used to represent them.

For any sparse message we can define its size in bytes as $S(format, nnz, m, n)$. This size is the required number of bytes to transmit a sparse matrix with nnz nonzero entries, $m \times n$ dimensions, given a specific format. Let $s_v = sizeof(data)$ and $s_d = sizeof(dimension)$, we can determine that:

1. $S_{COO} = nnz \times (2s_d + s_v)$
2. $S_{CSR} = nnz \times (s_d + s_v) + (m + 1) \times s_d$
3. $S_{CSC} = nnz \times (s_d + s_v) + (n + 1) \times s_d$

Each of the above reflects the actual arrays that must be transmitted. COO for example requires the full two dimension arrays (row indices and column indices) each of length nnz and the values array also of length nnz. CSR and CSC both replace either the row index array or the column index array respectively with a variant that takes a length of $m+1$ or $n+1$. Applying this to a collective communication, we can determine that the per message cost of a sparse matrix for a format is: $T_{bcast}(format, nnz, m, n)\lceil log_2 p \rceil \times \alpha + S(format, nnz, m, n) \times \beta + \alpha_{meta}$. The term α_{meta} is the additional cost of the metadata exchange required for sparse matrices so a proper buffer can be allocated.

3.3.2 Sparsity Level vs Volume

The density of a matrix, defined as $\frac{nnz}{m \times n}$ is the primary determinant of raw communication volume for sparse collectives. As the density approaches one, a sparse format will approach or exceed the size of the equivalent dense representation due to the added metadata overhead. For example, a CSR matrix with some size nnz requires $nnz \times (s_{data} + s_{dimension} + (m+1) \times s_{dimension})$ bytes, compared to $m \times n \times s_{data}$ for a dense matrix. This means the crossover point at which sparse storage offers no size advantage, and in fact incurs worse communication costs, occurs at a sparsity of $1 - s_{dimension}/(s_{data} + s_{dimension})$, which for floating point values and 32 bit integers (the basis of all experiments ran in this thesis) is roughly 50 percent sparsity.

For the communication patterns studied here, sparsity operates as a continuous scaling factor on message sizes. Halving the sparsity approximately halves the communication volume for fixed matrix dimensions, which translates directly to reduced bandwidth cost. However, the latency cost is not reduced proportionally as the number of messages and communication rounds is determined by the algorithm and process count. At very low densities, sparse collectives are therefore prone to latency rather than bandwidth and thus the α term becomes the issue as we scale.

3.3.3 Load Imbalance across Processes

Load imbalance is created whenever nonzero entries are not distributed uniformly across process partitions. For scatter and gather operations in particular, the collectives wall time is bounded by the most heavily loaded process and thus all other processes must wait for the largest message to complete before the operation can finish. So, the amount of imbalance is a direct product of the sparsity pattern.

For a matrix with nnz non zeroes distributed across p processes, we can note that nnz_i for some process i. The ideal case is that $nnz_i = nnz/p$ for all i, which is statistically achieved by uniform random sparsity. An imbalance ratio given by $max_i(nnz_i)/(nnz/p)$ tells us the deviation from this ideal case, where a ratio of one is a perfect balance and larger values indicate growing inefficiency. Banded matrices often produce high imbalance when only processes covering the diagonal rows hold the non zeroes. Block matrices hold a similar pattern at the block level. For Kronecker matrices, they are expected to produce the most severe imbalance as their power law degree structure concentrates non zeroes into a small fraction of rows that grows with the matrix size.

Load imbalance also compounds cost with the metadata overhead discussed in Section 3.1.2. A heavily loaded process must construct and transmit a proportionally large index array, while a lightly loaded process still pays the full startup latency α for a near empty message. Neither end of the imbalance cost benefits from additional parallelism, which is a core reason why sparse collectives exhibit worse strong scaling than the dense counterparts.

3.4 Expected Effects by Sparsity Pattern

The four sparsity patterns studied in this thesis interact with the collective cost in distinct ways, based on the previous analysis we performed.

For uniform random sparsity, each process receives approximately nnz/p non zeroes regardless of the partition boundaries, so message sizes are nearly equal across processes and steps. Collective cost here should scale linearly with density, and thus makes uniform sparsity the most predictable and useful as a baseline for the other patterns.

For banded sparsity, cost depends entirely on how the band aligns with the process partition boundaries. When the band is narrow and processes are assigned row ranges, only a small number of processes hold a significant number of non zeroes which will create a high degree of imbalance in scatter and gather operations. As bandwidth grows toward the full matrix, load imbalance improves and the resulting cost will grow closer to that of a uniform sparsity matrix. Broadcast should be less sensitive to this as all processes will achieve the same message.

For block sparsity, imbalance depends on whether block positions are spread evenly across the process grid and how blocks are distributed. Well distributed blocks should produce predictable message sizes and thus good overall balance. Clustered blocks will concentrate the load on a small number of processes. The best case here is that block sparsity is the most communication efficient pattern because it can eliminate the message size overhead if blocks align with process partitions.

For Kronecker sparsity, we expect that it will produce the worst case scenarios for every collective and operation. Because the hierarchical degree structure guarantees that a small fraction of rows account for a large share of the non zeroes, collectives like scatter and gather will be cost dominated by those processes. As the matrix size and thus number of non zeroes increase (a result of increased Kronecker iterations), the imbalance ratio and total cost will worsen dramatically.

3.5 Scaling Behavior

3.5.1 Strong Scaling

Strong scaling refers to the behavior of an algorithm when the problem size is held fixed and the number of processes is increased. In the ideal case, doubling the number of processes halves the time for a fixed workload, essentially achieving a linear speedup. For dense collective communications, there is an understood strong scaling behavior: the bandwidth cost decreases with additional processes while the latency cost increases due to growing communication rounds.

For sparse collectives, strong scaling is of course subject to a new constraint: as the number of processes increase, the average number of non zero entries per process decreases as $\frac{nnz}{p}$. Once $\frac{nnz}{p}$ drops below some threshold at which communication overhead dominates computation, further scaling will yield diminishing returns. This threshold arrives much earlier for sparse data than dense data because the effective message size decreases faster and the metadata overhead becomes a large portion of the message.

3.5.2 Weak Scaling

Weak scaling refers to the behavior of an algorithm when the problem size is scaled proportionally with the number of processes, such that each process will hold a fixed amount of work. In the ideal case, total time remains constant as both the problem size and process count increase together. For dense collectives, weak scaling is limited by the latency cost which grows even when the total data volume is fixed because more processes must be coordinated.

For sparse collectives, weak scaling has an additional complexity - if the problem is scaled by adding more rows or columns while keeping the sparsity and structure fixed,

the per process non zero count remains approximately constant. However, the metadata overhead also grows with the number of rows per process in CSR and CSC formats, even if the non zero count is constant, because the row pointer array scales with the number of rows. For matrices where the number of rows per process is large relative to the average number of non zeros per row, this can cause weak scaling to be significantly worse than the dense case.

3.6 Summary

This chapter has analyzed how the seven collective communication operations introduced in Chapter 2 are affected when operating on sparse data. The core difficulty here is that sparse formats violate all of the fixed message size assumptions that collective algorithms are often designed around and introduces a few new sources of overhead: variable message sizes, index metadata, and data dependent load imbalance.

The cost model we developed in this chapter allows us to quantify the effects in terms of nnz, matrix dimensions, and storage format. At very low densities, all formats become latency bound because the α term dominates due to the aforementioned metadata that must be communicated.

Sparsity pattern also clearly has an influence on load balance across processes. Uniform random distributes the nonzeroes evenly and is essentially equivalent to dense in terms of load balance, thus it can be used as a general baseline. Banded and blocked patterns concentrate their nonzeroes into small areas of the processes, creating imbalance for both scatter and gather operations where the total communication time is bounded by the process that takes the longest.

Chapter 4

Sparse SUMMA

This chapter extends the analysis of the SUMMA algorithm introduced in Chapter 2 to the case where one or both input matrices are sparse. The transition from dense to sparse SUMMA requires revisiting nearly every phase of the algorithm: the data distribution, the broadcast operations, and the local multiplication step. Each of these phases introduces new complexity when data is sparse and the appropriate choices for storage format, partitioning strategy, and communication primitives depend on the properties of the input matrices. The goal of this chapter is to develop a theoretical understanding of how sparsity interacts with the SUMMA structure and communication which will inform further experiments.

4.1 From Dense to Sparse SUMMA

4.1.1 Overview

Recall from Chapter 2 that the dense SUMMA algorithm computes $C = A \times B$ by distributing all three matrices across a $P_r \times P_c$ process grid and iterating $\frac{k}{b}$ panel steps. At each step s, the process in column s broadcasts its local A panel across its process

row, and the process in row s broadcasts its local panel B down its process column. Every process then computes the local outer product contribution and accumulates it into its local block of C.

In the dense case, the A panel broadcast transmits a contiguous block of m/P_r rows, each of length k/P_c, for a total message of size $(m/P_r) \times (k/P_c)$ elements. The B panel broadcast transmits a contiguous block of k/P_r rows, each of length n/P_c, for a total message size of $(k/K_r) \times (n/P_c)$ elements. Both message sizes are deterministic and equal across all steps and all processes.

When A and B are sparse, neither of those properties hold true. The A panel for step s is the submatrix of A consisting of the rows held by the broadcasting process and the columns corresponding to the s-th panel. The submatrix may be empty, partially full, or nearly dense depending on where the non zero entries of A are located relative to the panel boundaries. Similarly for the B panel. The result is that message sizes vary across steps and across processes in a pattern determined entirely by the sparsity structure of the input matrices.

4.1.2 Sparsity Challenges

The variability in message sizes creates several practical challenges for the sparse SUMMA implementation. First, the receiving processes cannot preallocate buffers of the correct size without first knowing how many nonzero entries it will receive. This requires as previously mentioned an additional metadata exchange before each broadcast, add least adding one additional latency term per step. Second, some panels may be entirely empty, in which case the broadcast carries no data but still incurs some message cost as the receiving processes do not know they will receive no data.

Third, the local multiplication that follows each broadcast is a sparse times sparse (SpGEMM) operation rather than a simple dense matrix multiply. SpGEMM is sub-

stantially more complex than dense multiplication: the output sparsity pattern is not known in advance and the computation is likely irregular in access patterns. Although not the focus of this thesis as there are many works on SpGEMM and general sparse times sparse multiplication [19], it still does present a challenge to note.

4.2 Storage Format Implications

4.2.1 CSR and A-Panel Broadcasts

The A panel broadcast in SUMMA distributes rows of A across the process row communicator. As discussed in Chapter 2, the CSR format organizes nonzero entries row by row, making it a natural choice for row oriented operations. For the A panel broadcast, the broadcasting process must extract the rows of its local A block that belong to the current panel step and pack them into a message. In CSR, this extraction is a simple slice of the values and col_indices array between row_ptrs[i] and row_ptrs[i + 1] for each relevant row i. Thus, this is a $O(nnz_{panel})$ operation where nnz_{panel} is the number of nonzeroes in the panel as it does not require a scan of the entire local matrix.

4.2.2 CSC and B-Panel Broadcasts

The B panel broadcast in SUMMA distributes columns of B across the process column communicator. The CSC format organizes non zero entries column by column, making it the natural format for these column orientated operations. For the B panel broadcast, the broadcasting process must extract the columns of its local B corresponding to the current panel step. In CSC, this is again a simple slice between col_ptrs[j] and col_ptrs[j + 1] for each relevant column j, which is $O(nnz_{panel})$.

4.2.3 Format Tradeoffs

The choice of storage format for either of the inputs as previously noted has a direct impact on both the efficiency of the broadcast and the local multiply phases. Using CSR for both A and B supports efficient A panel extraction and is compatible with standard row wise SpGEMM kernels, but makes B panel extraction inefficient because accessing a column of a CSR matrix requires scanning all rows. Using CSC for both supports efficient B panel extraction but makes A panel extraction inefficient for the same reason. A mixed strategy, storing A in CSR and B in CSC supports efficient extraction for both panels but requires the format to be specified and maintained separately. For later experiments and evaluation, both CSR/CSC only and CSR/CSC strategies are implemented and compared.

4.3 Matrix Shape and Process Grids

4.3.1 Square vs Rectangular Matrices

The SUMMA algorithm is inherently designed for arbitrary matrix shapes, but the choice of process grid dimensions P_r and P_c interacts with matrix shape in ways that affect both communication cost and load balance. For a square matrix of dimension $n \times n$ distributed over $P_r \times P_c$ grid, each process holds approximately n/P_r rows and n/P_c columns. The A panel broadcast sends messages of size proportional to $(n/P_r) \times (k/P_c)$, and the B panel sends messages of size proportional to $(k/P_r) \times (n/P_c)$. For a square process grid where $P_r = P_c = sqrt(P)$, these two message sizes are equal and the broadcast load is symmetric across row and column communicators.

For rectangular matrices, or for non square grids, the two broadcast message sizes are no longer equal. A tall and skinny matrix for example benefits from a process grid with

many rows and few columns, which reduces the per process row count and distributes the A panel broadcasts efficiently. A wide and short matrix benefits from the opposite configuration. For sparse matrices, this interaction is further complicated by the fact that the actual message sizes depend on the non zero distribution, not just the matrix dimensions. A tall matrix with banded sparsity may have very few non zeroes per row in the off diagonal region of each panel, making the A panel messages small regardless of process shape.

4.4 Sparsity Pattern Effects

4.4.1 Communication Bottlenecks vs Local Multiplication

In dense SUMMA, the relative cost of communication and local computation is bound by the ratio at which the local dense multiply can process data, which is well studied and typically favors computation at scale. For sparse SUMMA this balance shifts because the local SpGEMM performs fewer arithmetic operations but generates irregular memory access patterns that are less amenable to hardware optimization. The result is that instead, communication and computation are more closely matched in cost for sparse SUMMA, and in fact the dominant bottleneck will likely be communication based on the sparsity pattern and sparsity of the matrix.

For highly sparse matrices, the local SpGEMM in each step is fast because the A and B panels contain few non zeroes. The per step communication cost, however, still incurs a startup latency of at least α per broadcast even when panels are empty. As sparsity increases, the computation cost drops proportionally to nnz, but latency cost remains fixed. The result here is that for very low densities or high sparsity, sparse SUMMA becomes communication latency bound rather than computation bound, and scaling to more processes makes the problem worse by increasing the number of broadcasts without

reducing the startup latency.

4.4.2 Expected Behaviors by Pattern

For uniform sparsity, the A and B panels at each step statistically contain approximately the same number of non zeroes. Communication is therefore balanced across steps and processes, and sparse SUMMA cost is approximately the dense SUMMA cost scaled by the sparsity any additional communication overhead incurred by the additional metadata depending on matrix size.

For banded sparsity, the panel structure interacts strongly with the banded structure. Steps who panel range falls within the band produce large, dense panel messages, while steps outside the band produce empty or near empty panels. The total communication volume is therefore concentrated into a small number of steps, which may or may not benefit performance depending on whether those steps can be pipelined. The expected behavior is that the banded sparse SUMMA is significantly faster than uniform sparse SUMMA at the same sparsity, because most steps contribute no communication or computation.

For block sparsity, block panels are either fully dense or completely empty depending on whether the panel step intersects a dense block. When blocks align with panel boundaries, SUMMA steps are either maximally loaded or entirely skipped, so communication volume concentrates in fewer steps. This can be more efficient than uniform sparsity (fewer total messages with nonzero content) but if blocks are clustered in one region of the matrix process load becomes highly imbalanced during those steps.

For Kronecker matrices, it gets a bit more complicated. Due to the hierarchical, power law degree structure that they naturally have, Kronecker panels will be highly variable in nonzero density across steps and across networks. Steps corresponding to high degree rows and columns will produce very large panel messages while most other steps

produce nearly empty ones. This creates the worst case scenario for sparse SUMMA - an extreme load imbalance concentrated in a few steps, combined with many near empty broadcasts that still incur the full startup latency. The imbalance is expected to worsen as the Kronecker power k (and thus matrix size and nnz concentration) increases.

4.5 Summary

This chapter focused on extending the SUMMA algorithm to the sparse case and analyzing how sparsity may interact with each phase of the computation. Unlike dense summa, where message sizes are deterministic and uniform, sparse SUMMA must deal with the variable panel sizes determined purely by the nonzero structure of the input matrices. This introduces additional metadata exchanges and shifts the bottleneck from communication to communication latency within a certain range of matrix sizes we will discuss later.

We also determined that the choice of storage format likely has a direct impact on the efficiency of the lower level broadcasts. CSR for example will be optimal at doing the A panel broadcast, while CSC will be optimal for doing the B panel broadcast. A mixed strategy will likely be most optimal, but requires mixing formats which could introduce unintended consequences later down the line in the local matrix multiplication step.

Lastly there is the effect of sparsity pattern, the core analysis behind this thesis. Many of the formats will likely produce results that we expect due to their somewhat predictable nature, such as uniform sparsity or banded sparsity. Kronecker matrices are expected to produce the most interesting results, as not only do they exhibit the most imbalance but also have extremely varying patterns due to the selection of networks we will be testing against. All of the predictions made here will be evaluated in Chapters 5 and 6 by running a range of experiments across each problem type.

Chapter 5

Methodology and Experimental Setup

This chapter describes the experimental methodology used to evaluate the theoretical predictions developed in Chapters 3 and 4. The experimental design is organized around five of the previously mentioned variables: matrix shape and sparsity level, sparsity pattern, storage format, and parallelism/scaling. Each experiment holds a subset of these variables fixed while varying others, allowing us to attribute the performance differences to specific factors. All experiments are repeated multiple times to account for variability, and all statistics are reported over these repetitions.

5.1 Matrix Shape and Sparsity

Square matrices with dimensions $n \times n$ will be used as the primary test case, with n ranging in size based on the experiment. Non square matrices are also used to study the effects of a matrices aspect ratio on SUMMA and collective performance, with aspect ratios similarly ranging based on the experiment. Aspect ratios are designed to test matrices that are "tall and skinny" or "wide and short". For each of these variations,

sparsity levels that range from as low as 0.0001% sparsity to a fully dense matrix will also be tested.

Synthetic matrices are generated for each combination of shape, density, and sparsity pattern. For uniform random sparsity, entries are sampled uniformly at random. For banded sparsity, all entries within a specified bandwidth of the main diagonal are populated with uniform random values. For block sparsity, a set of block positions is randomly selected and each block is filled densely.

5.2 Parallelism

All distributed experiments are run on process counts varying counts, typically in even increments (e.g. 2, 4, 6) and in square increments (e.g. 3, 9, 16). On top of process counts varying, the amount of nodes requested on the super computer may vary as well and will be listed as such in any experiments.

Strong scaling experiments fix the total matrix size and vary the process count. Weak scaling experiments increase the matrix size proportionally with the process count such that each process holds approximately the same number of non zero entries.

5.3 Hardware and Software

All experiments that are used for statistics (e.g. tables, charts, etc) are conducted on the Oklahoma Supercomputing Center for Education and Research (OSCER) cluster at the University of Oklahoma [26]. The experiments were all designed to use the *normal* partition which contains up to 64 cores per node and up to around 250 GB per node.

The implementation is written in C++ and compiled with MpiCC using standard optimization flags. The following table describes specific versions used for OpenMPI, MPI, and any other software. Sparse matrix storage and generation utilities are imple-

mented from scratch without dependence on external sparse linear algebra libraries, in order to maintain full control over the communication patterns and conversions. However, other implementations of sparse multiplication libraries such as CombBLAS may be used as baselines and comparisons [20, 21].

Software	**Version**
C++ Standard	C++17
Compiler	GCC 14.2
OpenMPI	5.0.7
MPI Standard	3.1
SuiteSparse/GraphBLAS	10.3.1
CombBLAS	2.0

Table 5.1: Software versions used across all experiments.

5.4 Metrics and Instrumentation

5.4.1 Custom Profiling Tool

In order to facilitate the testing required for future experiments, a custom profiling tool was built that can measure a variety of variables and automatically generate per rank graphs and tables based on the outputs. Because many of the algorithms implemented in this thesis are complex and involve multiple levels of communication, the ability to profile sub scopes of the code was required. Listing 7 demonstrates an example of the SUMMA algorithm scoped by individual phases, specifically: a k iteration of the algorithm, the a and b panel broadcasts, and the local computation. This allows us to create statistics that represent the algorithm as a whole, but also view data like communication vs computation comparisons.

```
MatA summa(const MatA& a_local, const MatB& b_local, const SummaGrid& g)
{
    ...
    for (int k = 0; k < g.grid_rows; ++k)
    {
        perf::Scope summa_scope("summa_iteration" + std::to_string(k));
        MatA a_panel = (g.col_coord == k) ? a_local : MatA{};
        {
            perf::Scope scope("summa_apanel_bcast");
            summa_bcast(a_panel, k, g.row_comm);
        }

        MatB b_panel = (g.row_coord == k) ? b_local : MatB{};
        {
            perf::Scope scope("summa_bpanel_bcast");
            summa_bcast(b_panel, k, g.col_comm);
        }

        {
            perf::Scope scope("summa_computation");
            summa_accumulate(c_local, a_panel, b_panel);
        }
    }
    ...
}
```

Listing 7: A snippet of the sparse SUMMA implementation with profiling for individual scopes.

5.4.1.1 PMPI

PMPI is a small library used to intercept MPI calls used for creating more in-depth profiling tools that can individually measure each call [27]. Our custom profiling tool uses this to fully breakdown functions not only by the custom wrappers, but also by MPI calls. Whenever a stacks runtime is calculated, the total MPI runtime within that stack is also calculated so that the remainder of the code within a stack can be determined. An example of this behavior is shown below.

5.4.1.2 Output

The output of the profiling tool is a folder that is uniquely named by the current timestamp at the time of initialization. All ranks are given this time via an MPI call and thus all log to the same directory. The directory itself is then split into a few different files that all contain varying data.

1. /ranks/x.json - Contains information about that ranks individual profiling such as MPI call statistics, stack information, etc.

2. /metadata.json - Contains information about the current state of the project whenever the profiling is initialized, such as the current git commit/branch, arguments, etc.

3. /system.json - Contains information about the system the test was ran on such as total memory, the CPU, the GPU, etc. In the event that more than 1 node was used (as determined by how the experiment was created), this information may contain an array of systems.

The information in metadata.json and system.json are not particularly interesting outside of creating hardware tables used to ensure these experiments can be closely

created. The interesting data here comes from the individual files output by each rank. For each scope declared via the profiling tool, information including the total execution time, any children (scopes executed within a scope), mpi calls and mpi bytes, etc. are recorded. Listing 8 contains an example of a ranks JSON from a small MST Scatter experiment.

```
{
    "name": "experiment_iteration",
    "wall_time_s": "123.333",
    "values": {
        "rank": 2,
        "world_size": 8,
        "iteration": 0,
        "experiment": "mst_scatter_banded"
    },
    "children": [
        {
            "name": "mst_scatter",
            "mpi_calls": {
                "MPI_Barrier": 990,
                "MPI_Recv": 3960
            },
            "mpi_bytes": {
                "MPI_Barrier": 0,
                "MPI_Recv": 3189019200
            }
        }
    ]
}
```

Listing 8: A portion of a ranks output data from an experiment.

Chapter 6

Results and Analysis

This chapter presents and interprets the experimental results collected using the methodology described in Chapter 5. The results are organized into four sections. Section 6.1 discusses the experimental design choices and the characterization of the variability in the measurements. Section 6.2 establishes baselines for both the collective communication and SUMMA experiments. Section 6.3 and 6.4 present results for sparse collective communication and sparse SUMMA respectively, organized by the primary variables: sparsity level, sparsity pattern, storage format, and process count.

6.1 Experiment Design and Variability

As described in Chapter 5, each experimental configuration is run multiple times and the median results are used as the primary performance statistic. The choice of median over mean is chosen to filter out occasional outliers caused by operating system jitter, network contention, or memory allocation delays that inflate the mean without reflecting the real behavior of the experiments.

6.2 Collective Communication Results

This section will focus on running all of the collective communications discussed in Chapter 2 and analyzing the effects of sparse matrices on them. Because the introduction of sparsity creates so many different knobs we can adjust, we will take a look at all of them individually and then view the effects when combined. Unfortunately, it would be incredibly difficult to view all of them together (think five or six dimensional graphs), so the appendices attached to this thesis will contain any figures that do not fit anywhere specific in the main content.

6.2.1 Impact of Sparsity Level

6.2.1.1 MST Collectives

This section will analyze the impact of matrix sparsity on all of the MST collectives with a focus on a fixed matrix size of $n = m = 4096$ and $p = 8$ where p is the number of processes. Figure 6.1 has all four MST collectives in one visualization. This section will continue to analyze each individually.

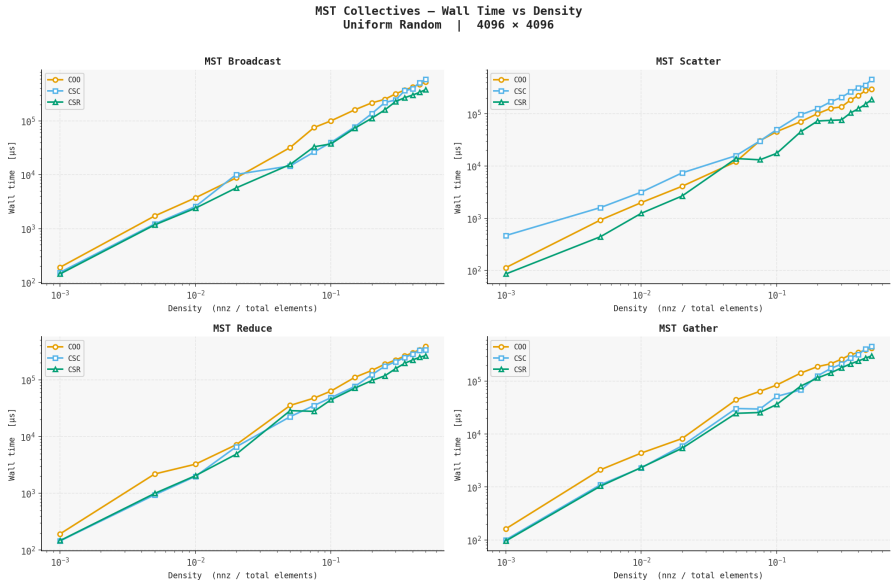


Figure 6.1: Sparse MST Collectives wall time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.2 shows the broadcast time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. As expected based on our analysis in Chapter 2, COO is the slowest out of the three formats in terms of wall time whilst CSR and CSC are nearly identical in performance as density increases.

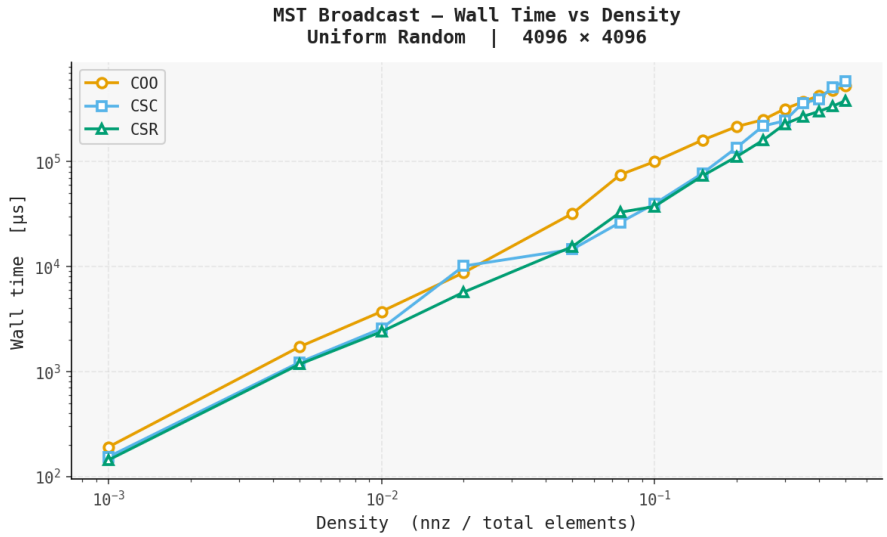


Figure 6.2: Sparse MST Broadcast time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.3 shows the gather time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. Similar to the MST Broadcast collective, COO is the slowest whilst CSC and CSR are nearly identical in performance as density increases.

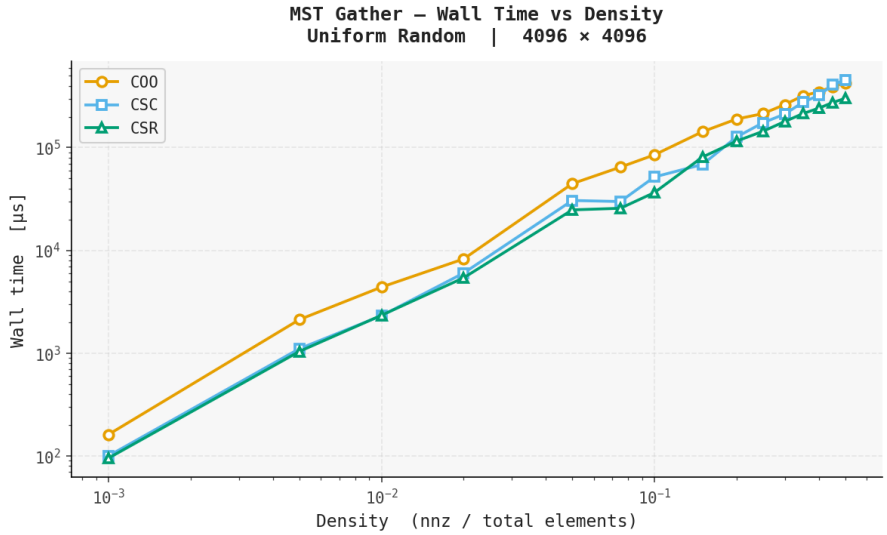


Figure 6.3: Sparse MST Gather time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.4 shows the reduce time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. Again, this shows nearly identical behavior to both MST Broadcast and MST Gather.

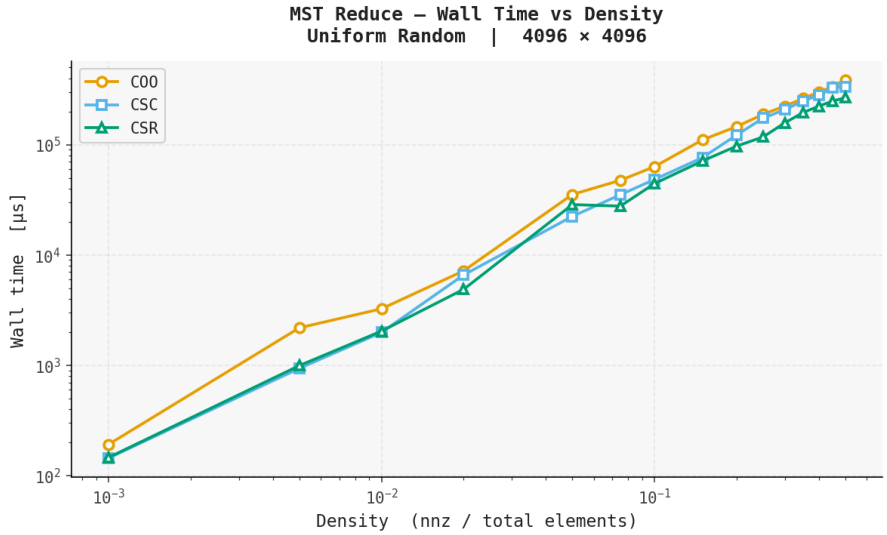


Figure 6.4: Sparse MST Reduce time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.5 shows the scatter time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. Unlike the broadcast, gather, and reduce collectives the scatter operation reveals a notable change between CSC and the other two formats. CSC is consistently the slowest across the full density range, whilst CSR is the fastest and COO is between. This arises from the directional nature of the scatter we implemented, where the root process partitions the matrix in matrices by row. This appears to be the most common implementation of scatter, especially for algorithms like SUMMA. COO avoids having to do any reorganization like CSC, but still is costly due to the larger metadata footprint it carries.

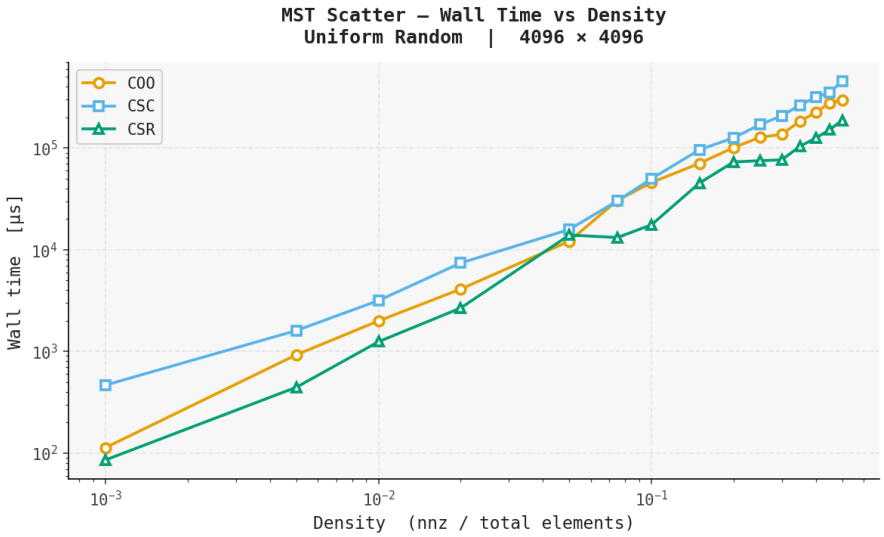


Figure 6.5: Sparse MST Scatter time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

6.2.1.2 Bucket Collectives

This section will analyze the impact of matrix sparsity on all of the Bucket collectives with the same matrix configuration as described in the MST collectives section. Figure 6.6 has all three Bucket collectives in one visualization. This section will continue to analyze each individually.

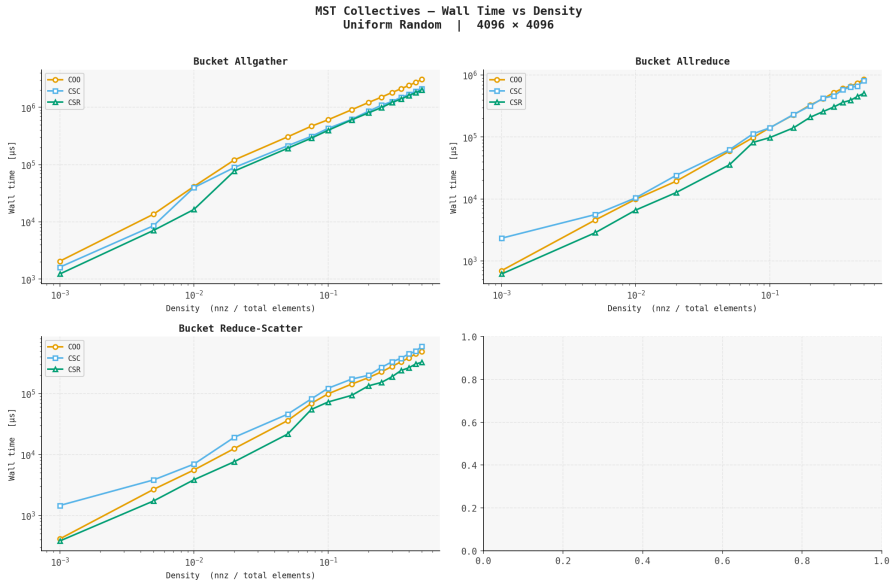


Figure 6.6: Sparse Bucket Collectives wall time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.7 shows the bucket all gather time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. The results here are quite similar to the MST counterpart, however, one important statistic to note is that the walls times are substantially larger. This is of course a direct result of all gathers communication, where every process must receive data from all other processes. As expected, CSR and CSC are very similar in performance while COO is slower overall due to the metadata overhead.

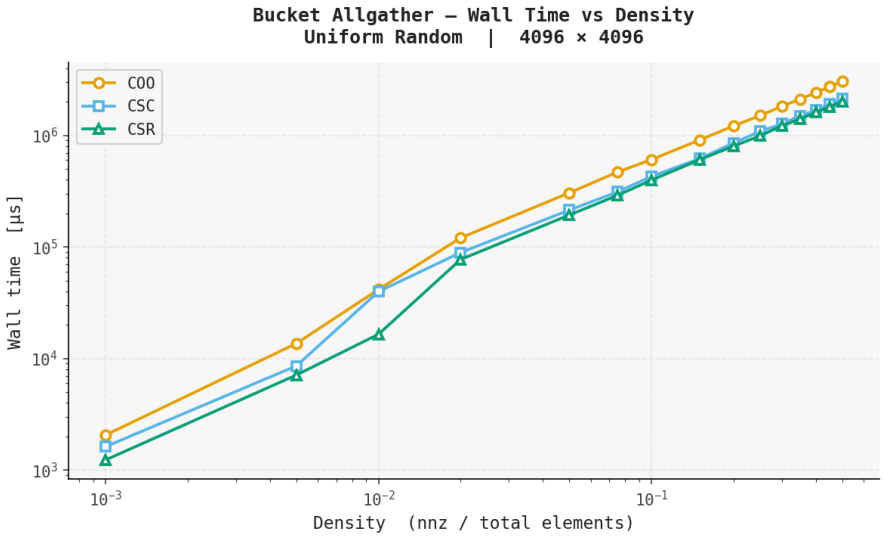


Figure 6.7: Sparse Bucket All Gather time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.8 shows the bucket all reduce time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes.

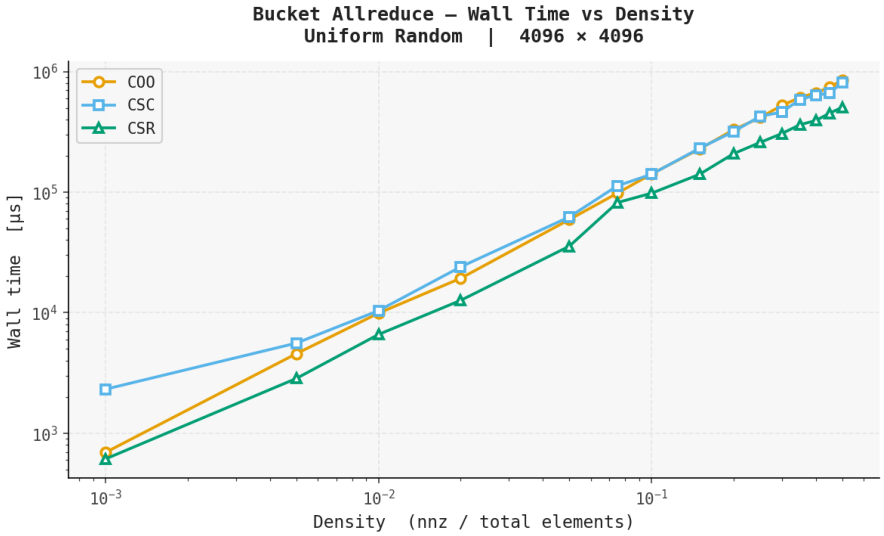


Figure 6.8: Sparse Bucket All Reduce time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

Figure 6.9 shows the bucket reduce scatter time as a function of matrix sparsity for a uniform random sparsity pattern with a fixed matrix dimension and 8 processes. CSC

appears to be the slowest format across the range, whilst CSR is the fastest and COO is between. This of course mirrors the behavior we observed in the MST Scatter operation, in which the CSC format must perform additional work because the data is not indexed in a form that is most efficient for CSC.

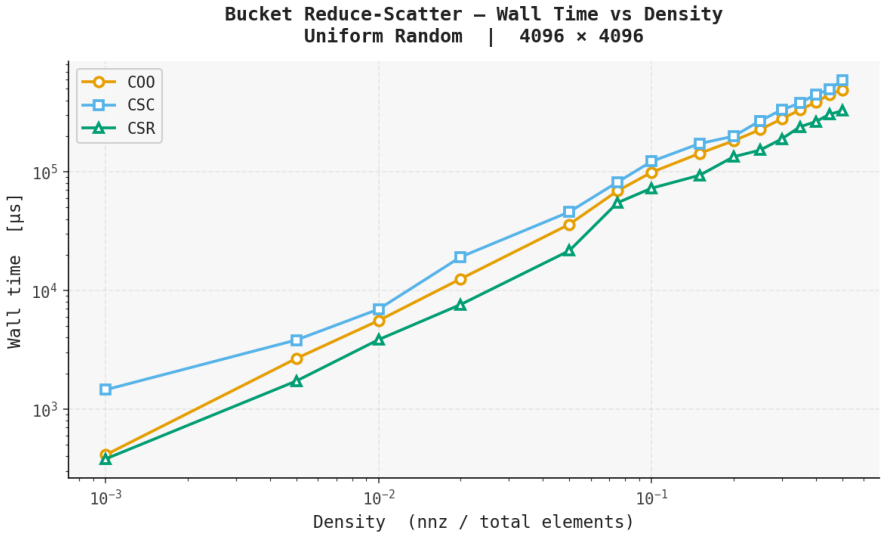


Figure 6.9: Sparse Bucket Reduce Scatter time as a function of matrix sparsity for a uniform random sparsity pattern ($n = 4096$, $p = 8$)

6.2.2 Impact of Sparsity Pattern

This section focuses on the Sparsity Patterns we have implemented in this thesis such as blocked or banded and how they effect the performance of our collectives. Each experiment was ran using a similar configuration as described before.

6.2.2.1 MST Collectives

Figures 6.10 through 6.13 visualize wall time versus three different sparsity patterns and storage formats at a fixed density, matrix size, and number of processes. The most obvious result here for all of the figure is that the uniform random pattern clearly under performs compared to the rest of the results. This is expected based on all of our

previous analysis, as even at a 5% density almost every process evenly holds a roughly equal share of the data.

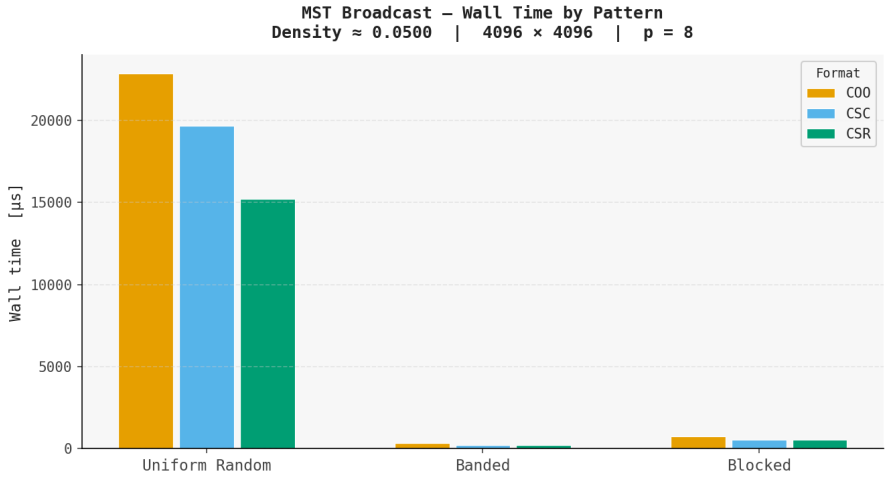


Figure 6.10: Sparse MST Broadcast wall time as a function of sparsity pattern and storage format (n = 4096, density 0.05, p = 8)

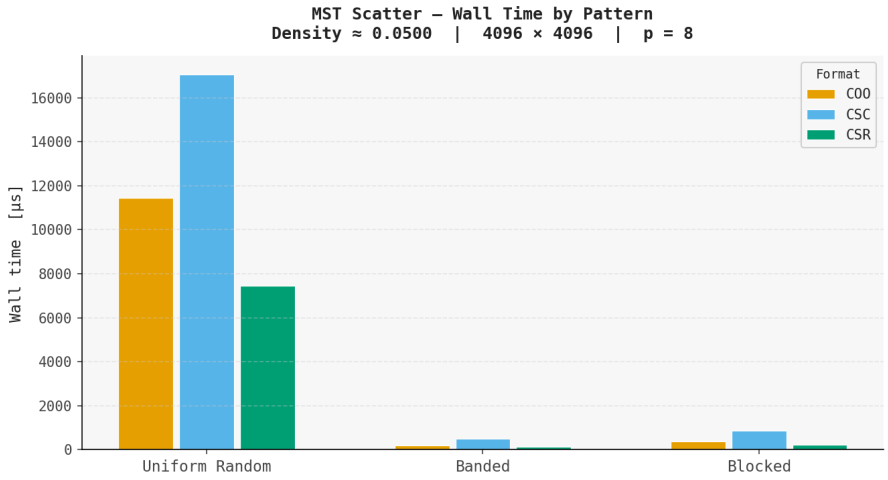


Figure 6.11: Sparse MST Scatter wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

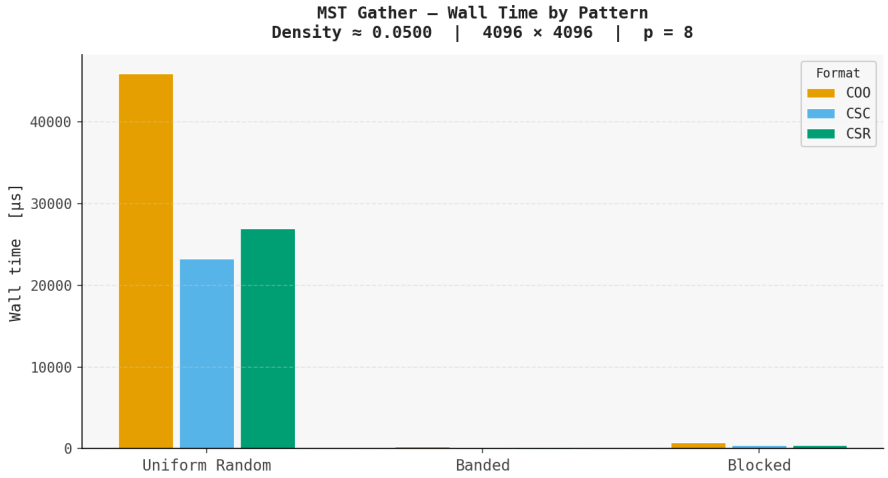


Figure 6.12: Sparse MST Gather wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

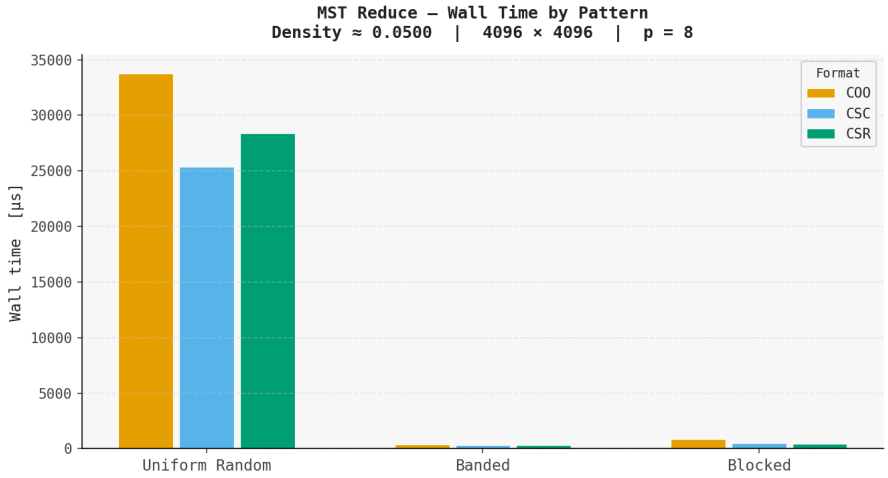


Figure 6.13: Sparse MST Reduce wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

6.2.2.2 Bucket Collectives

Figures 6.14 through 6.16 visualize the Bucket All Gather, All Reduce, and Reduce Scatter wall times across the three sparsity patterns at a fixed density, matrix size, and number of processes. The pattern of results here mirrors that of the MST collectives where uniform random sparsity consistently produces the highest wall times. Banded

and blocked, however, concentrate their data into smaller subsets and thus reduces the amount of data that the majority of processes must handle.

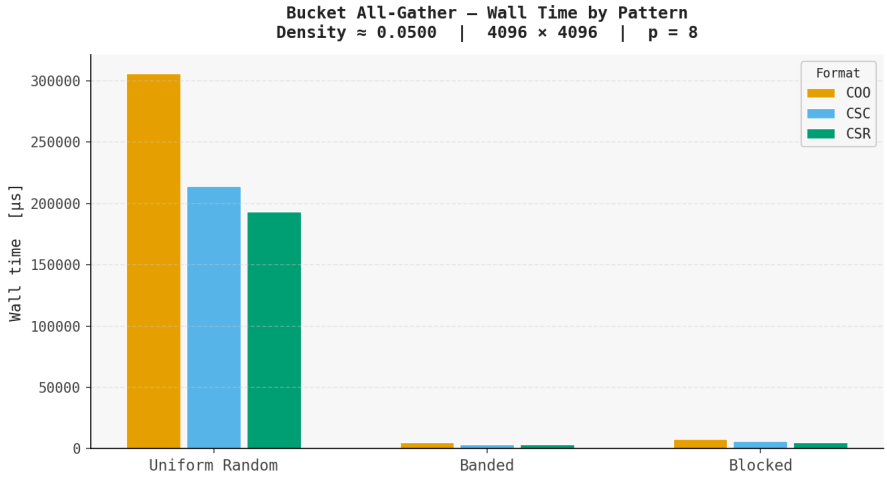


Figure 6.14: Sparse Bucket All Gather wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

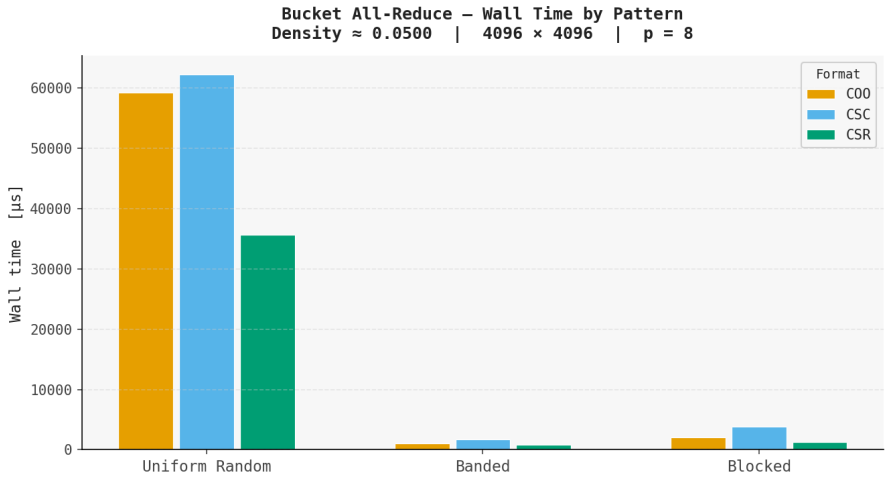


Figure 6.15: Sparse Bucket All Reduce wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

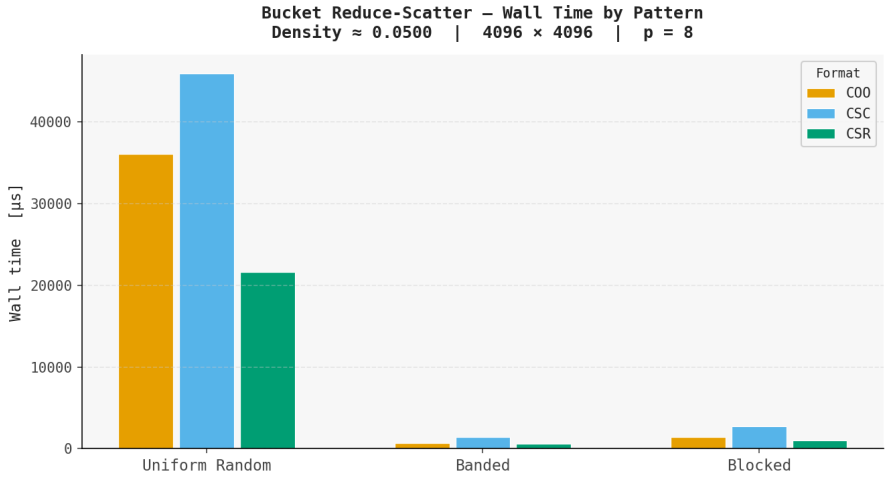


Figure 6.16: Sparse Bucket Reduce Scatter wall time as a function of sparsity pattern and storage format (n = 4096, density ≈ 0.05, p = 8)

6.3 SUMMA Results

6.3.1 Impact of Sparsity Level

Figure 6.17 shows sparse SUMMA wall time as a function of matrix density across the three different sparsity patterns, using CSR for both inputs on a 4096×4096 matrix with 8 processes. The results do confirm the prediction we made back in Chapter 4, where wall time scales roughly with density. However, the relationship as seen in the graph is not perfectly linear due to the per step latency cost regardless of the number of non zeroes.

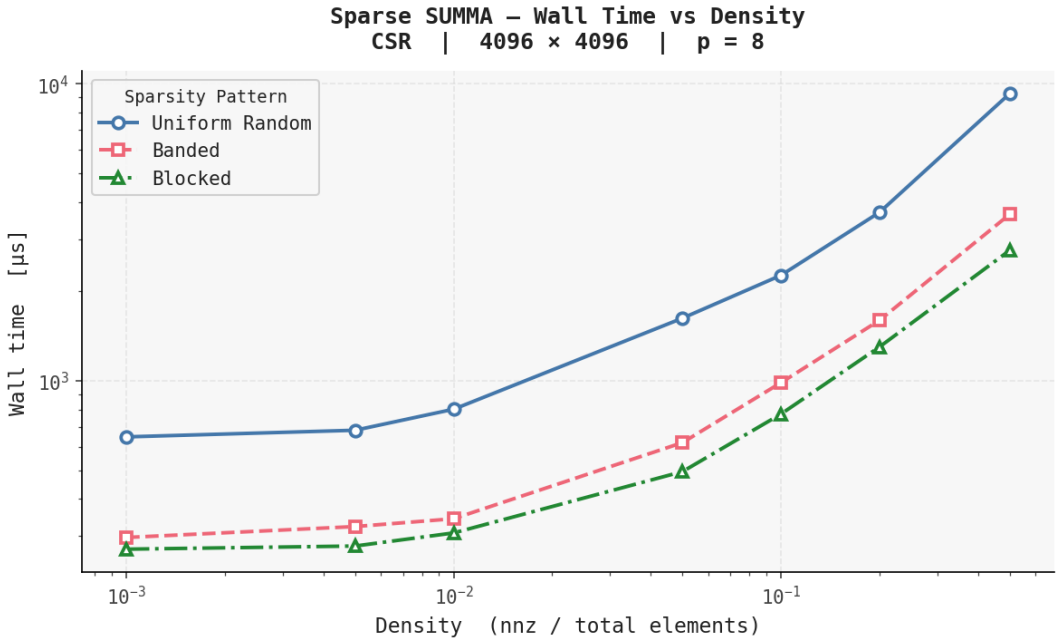


Figure 6.17: Sparse SUMMA as a function of matrix density and wall time given three different sparsity patterns.

Uniform random sparsity consistently produces the highest wall times at any given density level. This is of course expected because the panels at every step of the algorithm carry non zero data proportional to the overall density, meaning that all k steps contribute both meaningful communication and computation. Banded and blocked sparsity patterns perform much better at equivalent densities because their non zeros are concentrated, leaving many panels empty or near empty. However, as the density increases all three patterns begin to converge since the panels become dense enough that structured patterns begin to lose their advantage. At very low densities, the curve flattens because of the communication latency discussed earlier. Because even when the panels are nearly empty, each broadcast step still incurs some startup cost α so lower densities do not yield better results.

6.3.2 Impact of Sparsity Pattern

Figure 6.18 compares sparse SUMMA wall time across three sparsity patterns at a fixed density of around 0.05, using all of the available format strategies: CSR only, CSC only, and a mixed CSR + CSC approach.

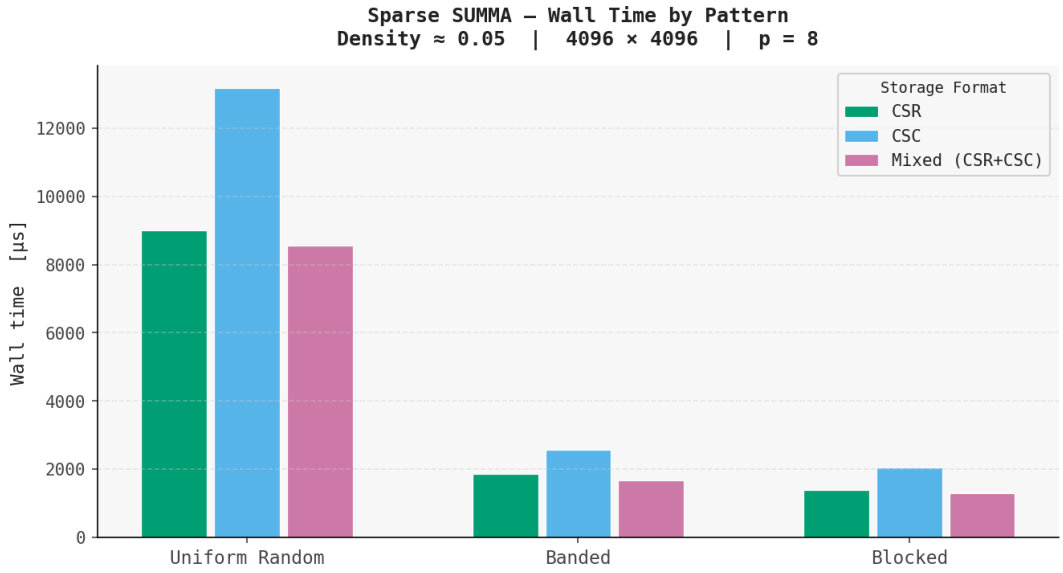


Figure 6.18: Sparse SUMMA as a function of wall time and sparsity type, given the storage format.

The most interesting result here is that there is quite a large gap between uniform random and the structured pattern, although perhaps not all that surprising. At this density, both banded and blocked matrices complete in roughly a fifth of the time required as by the uniform random case. This matches the prediction we made earlier, that structured patterns concentrate non zeroes into a subset of panel steps, leaving the remaining steps with low communication and computation cost. The uniform random pattern distributes non zeroes evenly, so no step is cheap.

6.3.3 Impact of Matrix Dimensions

Figure 6.19 shows sparse SUMMA wall time as a function of total matrix elements for three different matrix shapes: square, tall and skinny, and wide and short. All of these are using the CSR only format at around 5% density and 8 processes.

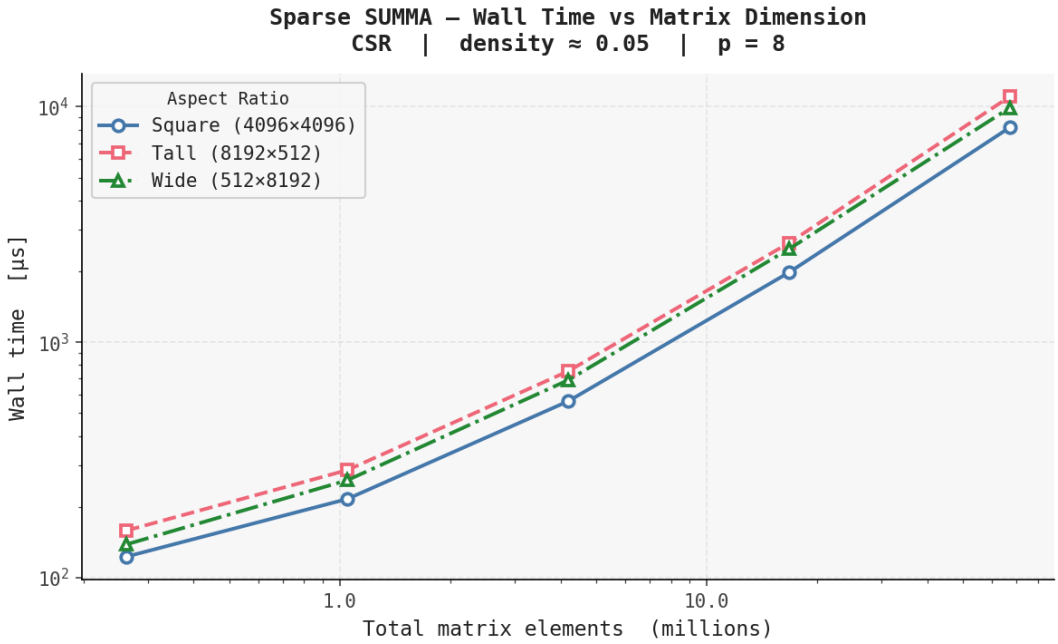


Figure 6.19: Sparse SUMMA as a function of total nnz and wall time, as a function of the matrices aspect ratio (e.g. tall, wide, square).

All three of these configurations scale similarly in total wall time as the number of elements increases. It should however be noted that the tall and wide configurations consistently run slightly slower than the square configuration at equivalent element counts. This can best be explained by our earlier discussion on how the process grid interacts with the matrix shape.

As the problem size grows it appears that the three shapes converge in their scaling, suggesting that the per element cost of sparse SUMMA is approximately agnostic at this configuration. The offset between square and non square configurations is certainly because of the process grid alignment, such that square matrices map evenly to square

process grids, and wide or tall matrices produce unequal panel sizes.

6.3.4 Impact of Storage Format

Figure 6.20 shows parse SUMMA wall time as a function of density for a uniform random matrix, with similar configuration to before. At low densities, it is clear that CSR is the fastest single format option which is consistent with how we expected it to perform. A mixed strategy when aligned properly, which uses CSR for the A matrix and CSC for the B matrix, is by far the fastest.

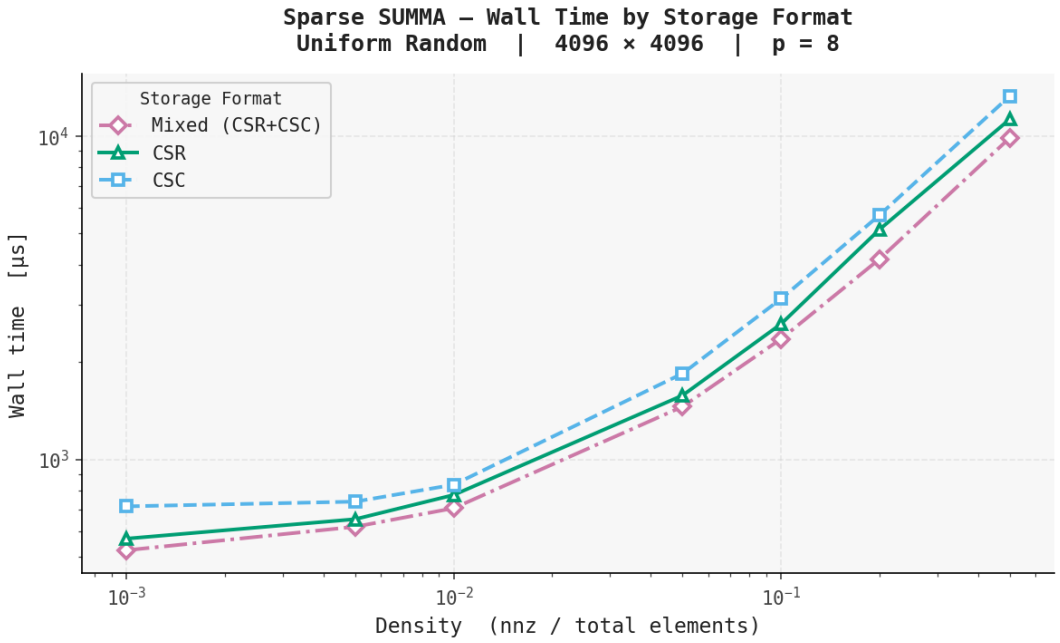


Figure 6.20: Sparse SUMMA as a function of matrix density and wall time, given the storage format.

6.3.5 Communication vs Computation Bottlenecks

As noted in Chapter 2, an important topic to consider is whether or not our sparse SUMMA algorithm is actually being limited by the communication speeds or if we are still running into the dense problem of computation. Via our earlier cost analysis

in Chapter 3, the expected result should be that our algorithm is almost completely bottle necked by communication overhead. Figure 6.21 breaks down the sparse SUMMA wall time into three components: data communication, metadata overhead, and local SpGEMM computation, across a range of densities.

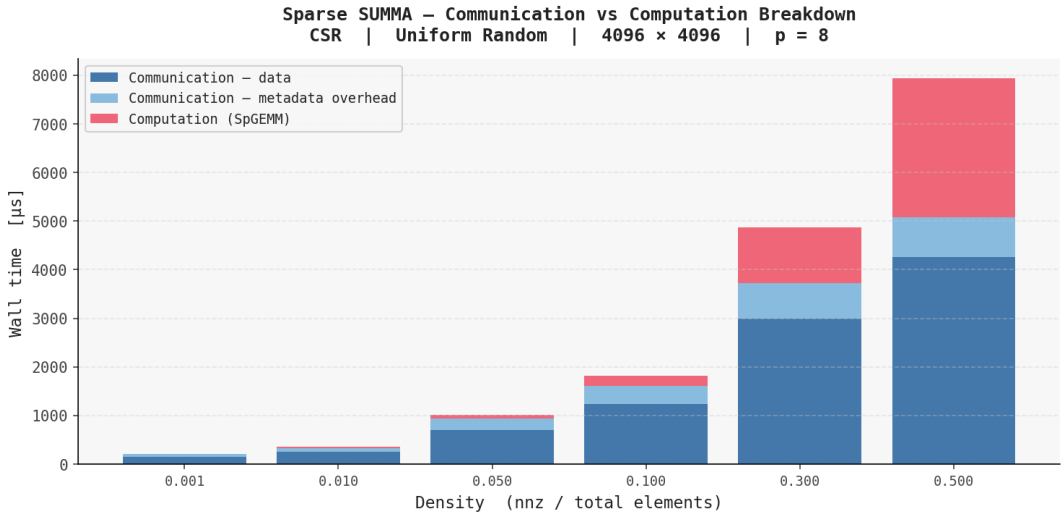


Figure 6.21: Sparse SUMMA as a function of density and wall time, demonstrating a breakdown of communication and computation costs.

The results confirm that the communication dominates all of the density levels we tested. At around 10% density, data communication already accounts for a majority of the total runtime with computation contributing a small fraction of it. As the density increases towards 50% it becomes clear that computation is starting to become a bottleneck because the panels are becoming dense enough for SpGEMM to perform lots of work. The metadata overhead does remain small overall compared to the data communication, but at these sizes and communication costs is not a significant bottleneck.

6.3.6 Load Imbalance Observations

Figure 6.22 shows the distribution of non zero entries per rank across the 8 processes for three sparsity patterns. The dashed line marks the mean non zeroes per rank, which is

held constant across all of the patterns.

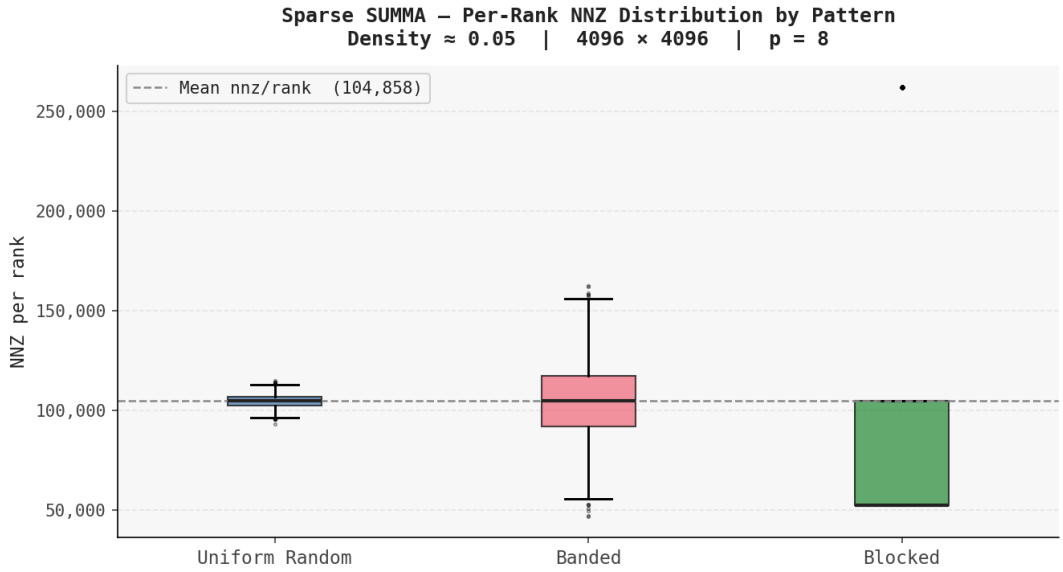


Figure 6.22: Sparse SUMMA as a function of nnz per rank and sparsity pattern, showing the nnz distribution per rank (load imbalance).

As expected, the uniform sparsity pattern produces a tight distribution with minimal spread around the mean. This confirms that the balanced baseline as described in Chapter 3 is correct, where nearly every rank holds approximately the same number of non zeroes.

The banded sparsity pattern shows a substantially higher variance. Because the band concentrates non zeroes into a contiguous row, the processes covering that region receive a disproportionate share of the work. This corresponds directly into the per step communication imbalance referred to in Section 3.3.3, where the slowest process determines the wall time of each collective.

The blocked pattern shows the widest distribution and the heaviest outliers. Because non zero entries are concentrated in these dense rectangular blocks, a rank that happens to own rows intersecting one or more blocks receives a disproportionate share of the total non zeroes. This produces the distribution we see in the diagram, some ranks are

heavily loaded and others are nearly idle.

Taken together, these distributions explain a problem that appears throughout the SUMMA results: a sparsity pattern can have a low average communication cost yet still performs poorly due to load imbalance. Conversely, a pattern with a higher average cost but uniform distribution can achieve more consistent performance when scaling.

6.3.7 Combined Effects

Figure 6.23 presents a grid of wall time heatmaps combining the effects of sparsity pattern, density, and process count for CSR storage on a 4096×4096 matrix. Each sub grid corresponds to a fixed process count where $p = 2, 4, 6, 8$, with rows representing the patterns and columns representing the density levels.

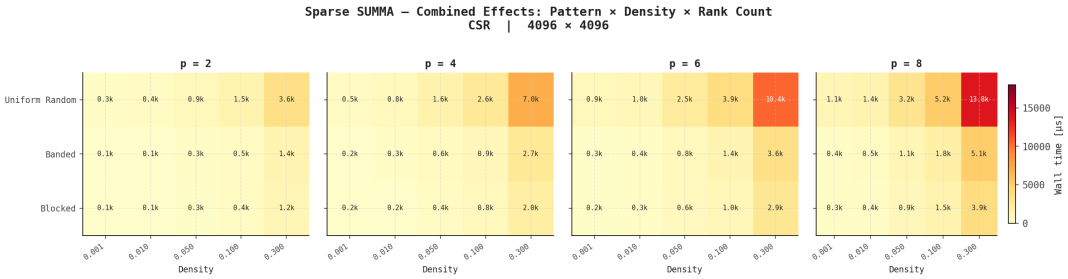


Figure 6.23: Sparse SUMMA a function of density, matrix sparsity, number of ranks, and wall time shown as a heatmap for different number of processes.

This figure demonstrates a few trends that occur from sparse SUMMA. First, the sparsity pattern dominates at low process counts and low densities. At $p = 2$, the gap between uniform random and structured patterns is quite large across all densities. Second, as the process count increases the gap between the patterns narrow at low densities because the per step startup latency α becomes a larger fraction of the total cost. Third, the hottest cells in the grid, or notably the higher wall times, consistently appear in the uniform patterns. The high density, high process count corners essentially represent the worst case combination of maximum communication volume per step and

maximum latency.

6.4 Discussion

The experimental results we presented largely confirm the predictions we created earlier in this thesis. This section takes all of that information and creates rough guidance for selecting storage formats and communication strategies based on known properties like matrix size and density.

6.4.1 Storage Format Selection

Across both the collective communication experiments and the SUMMA results, the relative performance of COO, CSR, and CSC follow consistent patterns. COO is the slowest format for communication heavy workloads due to its higher metadata overhead, transmitting two full integer arrays of length nnz alongside the values. The performance gap between COO and the compressed formats widens as the density increases, which matches the cost models we developed earlier. At very low densities, the formats converse due to the total message size being small enough that the α term dominates over bandwidth.

CSR and CSC perform nearly identically for symmetric operations like broadcast and gather when the matrix is square and uniform. The main distinction here is only when the operation being performed has a directional preference. For example, in the SUMMA context we discussed the A panel extraction benefits from CSR and the B panel extraction benefits from CSC. The mixed CSR and CSC strategy provides the best efficiency, for SUMMA at low densities its advantage over single format (CSC or CSR only) starts wearing off around 5% to 10%.

The main outcome here is fairly clear - for general collective operations on sparse data

CSR or CSC should be preferred over COO except for when construction simplicity is a main concern. For distributed sparse matrix multiplication specifically, a mixed strategy is definitely worth the additional complexity when operating in specific sparse patterns, but can be safely simplified to a single format at higher densities without meaningful loss.

6.4.2 Summary

This chapter focused on evaluating the sparse collective communications and sparse SUMMA across a variety of sparsity levels, sparsity patterns, storage formats, and process counts. The results confirmed many of the predictions we made back in Chapters 3 and 4. For the collectives, sparsity level scales communication volume almost linearly and load imbalance from structured patterns ends up being more significant due to the small pockets of dense data only some processes deal with. The MST collectives generally outperform the Bucket collectives at lower densities however, the inverse is also true where as the density increases Bucket starts to outperform MST.

For sparse SUMMA, the storage format has a direct impact on performance. The mixed CSR x CSC strategy provides the best efficiency at lower densities as it aligns the A and B panel extractions with the most natural format (e.g. column or row extractions for CSC and CSR respectively). Structured sparsity benefits from the fact that SUMMA panels may contain little or no data which reduces the communication cost and computation cost. All of this considered, it will provide a good baseline for the results we will see in Chapter 7 where we use Kronecker matrices which most closely mirror many real life networks.

Chapter 7

Kronecker Matrix Results

All of the previous chapters established the relationship between sparsity level, sparsity pattern, storage format, and communication algorithm performance across synthetic matrices. Kronecker matrices however offer a unique perspective, they are generated by not a one or two parameters like before but rather the structure of an underlying source network. Each Kronecker matrix inherits properties from its source matrix such as the degree distribution or clustering behavior which means that two Kronecker matrices of the same power k and the same density can behave quite differently at runtime depending on the source network.

This chapter focuses on the direct analysis of Kronecker matrices and a suite of source networks provided by Leskovec et al. [9]. Section 7.1 presents a quick summary of the predicted behavior of Kronecker matrices discussed in Chapters 3 and 4. Section 7.2 discusses collective communication results across all of the source networks. Section 7.3 talks about the sparse SUMMA results and Section 7.4 contains one of the core contributions of this thesis; a set of tables and scatter plots designed to serve as a way to lookup the optimal broadcast algorithm and storage format most likely to minimize runtime given another matrix.

7.1 Predicted Behavior

Based on all of the analysis we performed in Chapters 3 and 4, Kronecker matrices are expected to present the most challenging characteristics. Their degree structure which is modeled from real world source networks will concentrate a uneven share of the nonzero entries into a small fraction of the rows and columns.

For sparse SUMMA, the Kronecker structure produces a worst case scenario for the algorithm, a small number of the panels will carry a large portion of the total non zeroes. So, the total runtime of the SUMMA algorithm in sparse land when using Kronecker matrices is dominated by just a few steps or nodes rather than evenly distributed. The storage format will likely be similar to those of the synthetic matrices, where CSR is preferred for A panel broadcasts and CSC is preferred for B panel broadcasts. In general, CSR x CSC should provide the best results at lower densities.

7.2 Collective Communication Results

This section presents the wall time results for the broadcast, scatter, gather, and reduce collectives applied to Kronecker matrices generated at $k = 20$ across all source networks using $p = 16$ processes.

7.2.0.1 Broadcast

Figure 7.1 plots the ratio of MST to BKT broadcast wall time against the imbalance ratio for all of the available source networks at $k = 20$ and $p = 16$. Values above the dashed line indicate that the bucket algorithm is faster, and below the line indicates MST is faster. It is clear that almost every network with an imbalance above roughly 20 favors the bucket algorithm, although it should be noted that the ANSWERS and DELICIOUS

networks do not follow that pattern. Similarly, the BIO-PROTEINS network also does not follow that pattern and performs better using the Bucket algorithm at an imbalance ratio of around 8. Overall this is consistent with our prior analysis that higher imbalance will favor the bucket algorithm due to the large size of the messages preventing the startup cost from dominating.

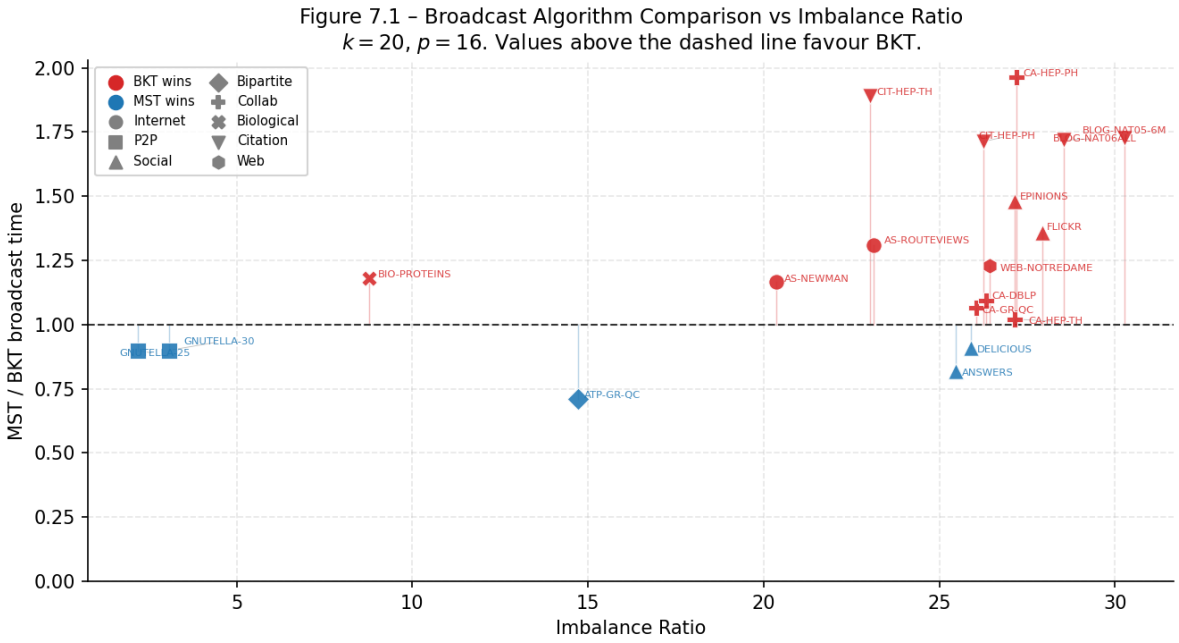
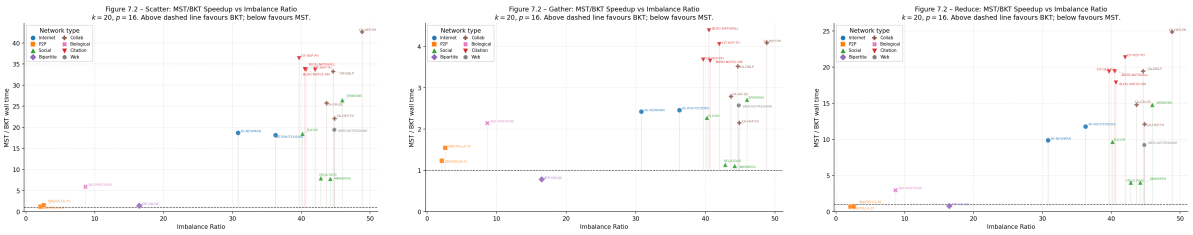


Figure 7.1: A scatter plot of broadcast wall time vs matrix density for Kronecker matrices at $k = 20$. Colors determine the broadcast algorithm.

7.2.0.2 Scatter, Gather, and Reduce



(a) Scatter (b) Gather (c) Reduce

Figure 7.2: MST/BKT speedup ratio for scatter, gather, and reduce collectives on Kronecker matrices at $k = 20$, $p = 16$ processes. Point color indicates network type; values above the dashed line favour BKT, below favour MST.

7.3 Sparse SUMMA Results

Table 7.1 presents the main SUMMA results across all networks at $k = 20$ and $p = 16$. Test harness parameters such as the median of five trials are the same as mentioned previously. Density is expressed in units of nonzeros per million matrix entries ($/M$).

Table 7.1: Sparse SUMMA dispatch summary for Kronecker matrices at $k = 20$ and $p = 16$ processes. *Best fmt*: lowest total wall time among CSR, CSC, and mixed $CSR \times CSC$. *Best bcast*: lowest broadcast-step time between MST and bucket. Density in units of $/M$ (nonzeros per million entries).

Network	**nnz**	**Dens**	**Imbal**	**Total**	**Bcast**	**Fmt**	**Bcast**
AS-ROUTEVIEWS	5,824,940	5.30	23.14	38.80	0.661	CSR	BKT
AS-NEWMAN	4,916,153	4.47	19.10	43.65	0.807	CSR	BKT
GNUTELLA-25	2,784,297	2.53	2.06	2.78	0.134	CSC	MST
GNUTELLA-30	2,073,584	1.89	2.87	2.05	0.113	CSC	MST
EPINIONS	13,329,197	12.12	25.26	21.08	0.745	CSC	BKT
ANSWERS	2,484,615	2.26	26.97	2.56	0.117	CSR	MST
DELICIOUS	2,041,919	1.86	26.87	5.14	0.242	CSR	MST
FLICKR	8,533,583	7.76	29.69	12.23	0.501	CSR	BKT
ATP-GR-QC	705,529	0.64	15.07	2.41	0.220	CSR	MST
CA-GR-QC	5,651,871	5.14	27.69	7.57	0.277	CSR	BKT
CA-HEP-PH	28,502,769	25.92	25.53	94.82	0.584	CSR	BKT
CA-HEP-TH	3,693,202	3.36	27.59	5.89	0.305	CSR	BKT
CA-DBLP	5,938,722	5.40	24.94	9.91	0.440	CSR	BKT
BIO-PROTEINS	6,851,892	6.23	9.15	24.27	0.405	CSR	BKT
BLOG-NAT05-6M	15,476,466	14.08	24.17	93.44	1.867	CSC	BKT
BLOG-NAT06ALL	18,739,157	17.04	27.08	133.22	1.155	CSC	BKT
CIT-HEP-PH	18,876,833	17.17	24.33	44.08	0.441	CSC	BKT
CIT-HEP-TH	19,777,866	17.99	26.88	37.65	0.390	CSC	BKT
WEB-NOTREDAME	5,901,094	5.37	25.98	7.84	0.306	CSR	BKT

Table 7.2 pulls apart of the timing in more detail by separating the broadcast, compute, and total times. For broadcasts, there is a specific column which notes the fraction of the total runtime the broadcast step took under the best algorithm which indicates

whether communication or computation is the main bottleneck for each network. Figure 7.3 demonstrates this table visually.

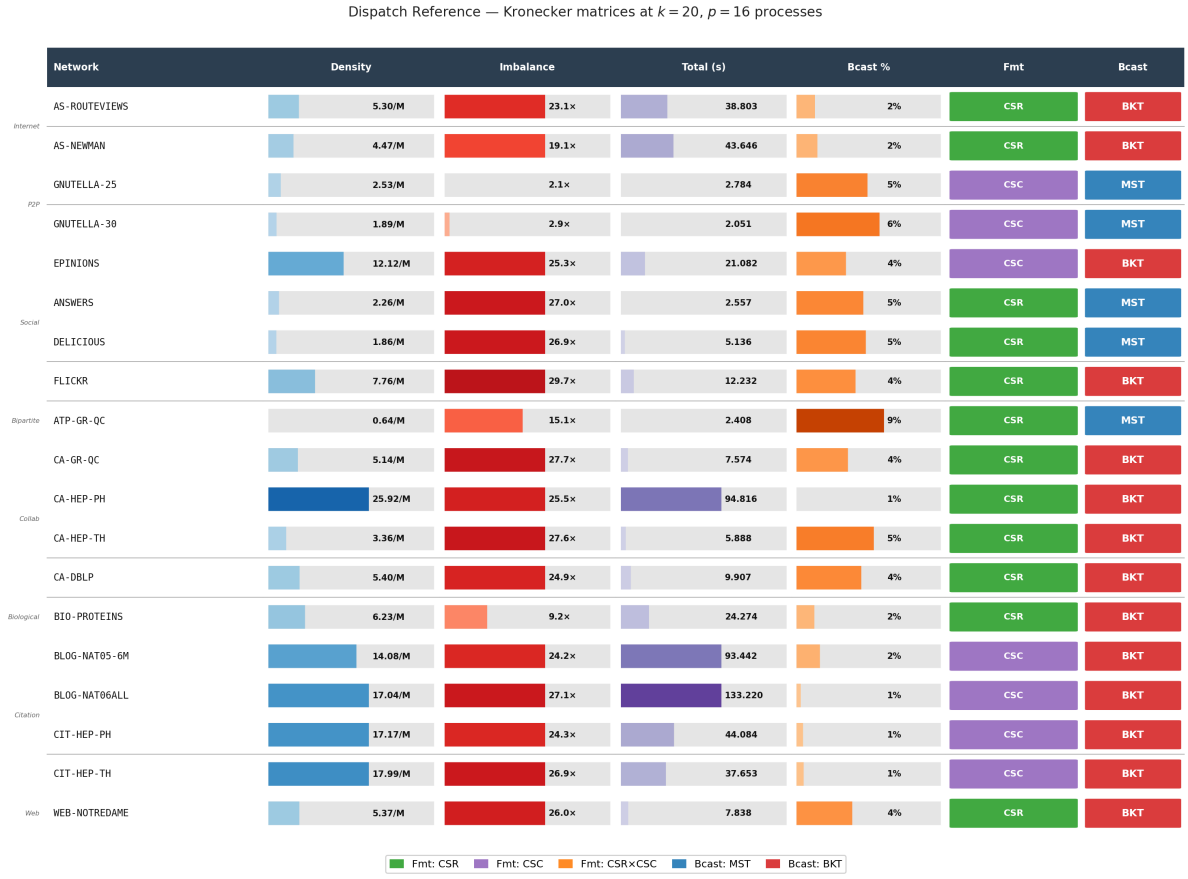


Figure 7.3: Dispatch reference table for Sparse SUMMA on Kronecker matrices. Each row corresponds to a source network associated with the data in Table 7.2

Table 7.2: Timing breakdown for Sparse SUMMA on Kronecker matrices at $k = 20$, $p = 16$ processes. MST_Bcast and BKT_Bcast are the broadcast step wall times for each algorithm independently. Bcast is the fraction of total runtime consumed by the broadcast step under the best-performing algorithm.

Network	**Total (s)**	**MST (s)**	**BKT (s)**	**Compute (s)**	**Bcast (%)**	**Imbalance**
AS-ROUTEVIEWS	38.80	0.719	0.548	8.29	1.7%	23.14

(Table 7.2 continued)

Network	Total (s)	MST (s)	BKT (s)	Compute (s)	Bcast (%)	Imbalance
AS-NEWMAN	43.65	0.856	0.765	8.18	1.8%	19.10
GNUTELLA-25	2.78	0.182	0.193	0.69	4.8%	2.06
GNUTELLA-30	2.05	0.152	0.175	0.50	5.5%	2.87
EPINIONS	21.08	0.484	0.304	5.55	3.5%	25.26
ANSWERS	2.56	0.154	0.169	0.65	4.6%	26.97
DELICIOUS	5.14	0.262	0.272	2.67	4.7%	26.87
FLICKR	12.23	0.331	0.227	3.20	4.1%	29.69
ATP-GR-QC	2.41	0.230	0.351	1.32	9.1%	15.07
CA-GR-QC	7.57	0.276	0.232	3.60	3.7%	27.69
CA-HEP-PH	94.82	0.580	0.262	23.02	0.6%	25.53
CA-HEP-TH	5.89	0.355	0.349	2.67	5.2%	27.59
CA-DBLP	9.91	0.441	0.397	4.22	4.4%	24.94
BIO-PROTEINS	24.27	0.365	0.325	5.97	1.7%	9.15
BLOG-NAT05-6M	93.44	1.632	1.067	20.70	2.0%	24.17
BLOG-NAT06ALL	133.22	1.131	0.638	28.34	0.9%	27.08
CIT-HEP-PH	44.08	0.446	0.285	9.60	1.0%	24.33
CIT-HEP-TH	37.65	0.591	0.295	9.77	1.0%	26.88
WEB-NOTREDAME	7.84	0.256	0.239	1.80	3.9%	25.98

7.3.1 Broadcast vs Compute

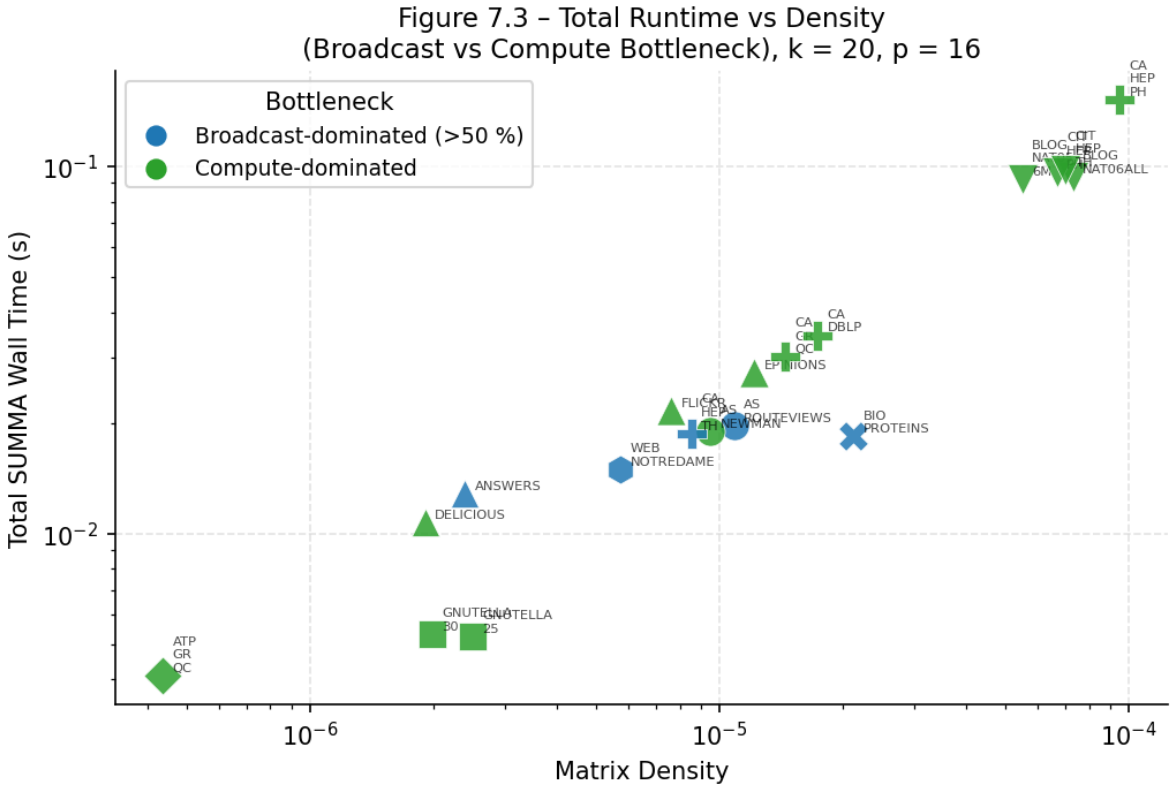


Figure 7.4: A scatter plot of runtime and density for each network measuring the broadcast time.

7.3.2 Effects of Kronecker Power

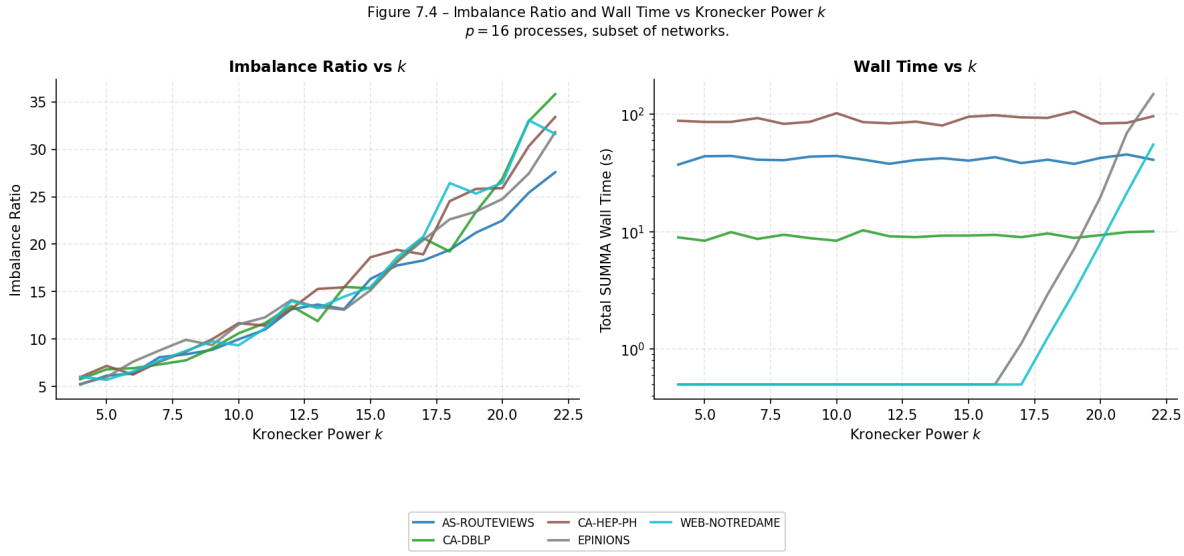


Figure 7.5: A line plot showing the imbalance ratio and wall time as a function of the Kronecker power k for a small subset of networks.

7.4 Dispatch Reference

These results support what is essentially a dispatch guide for researchers that are working with Kronecker structured matrices. The main question we are proposing is: given a matrix whose structure resembled a Kronecker product, what algorithm and storage format should be selected, and what measurements are needed to determine that?

7.4.1 Needed Measurements

The experimental results note three main measurements that should be sufficient to select the optimal algorithm and storage format for a Kronecker structured matrix.

1. Matrix density - this is the primary indicator of the dispatch table. Low density matrices favor MST and single storage formats, while high density matrices favor

the bucket algorithm and a mixed CSR x CSC format.

2. Imbalance ratio - this determines whether the density thresholds that separates MST vs BKT should be adjusted. Networks with low imbalance ratios (around 15 or less) may continue to favor MST despite their high density. High imbalance ratios (20 or more) typically favor BKT regardless of the density.

3. Broadcast fraction - this is the hardest measurement to obtain as it requires running the source network through a profiler. It indicates whether the bottleneck is communication computation. Networks with a broadcast time of over 50% benefit from specific tuning (choosing MST or BKT), while below that threshold will instead benefit from tuning the local SpGEMM operation.

In reality, the density and imbalance ratio can both be computed using a single pass over the matrix without having to do any additional communication or multiplication and thus is an inexpensive and simple way to squeeze out early performance.

7.4.2 CombBLAS

In an effort to show the use case of these decision tables in comparison to state of the art implementations, we have tested our implementation against CombBLAS [20].

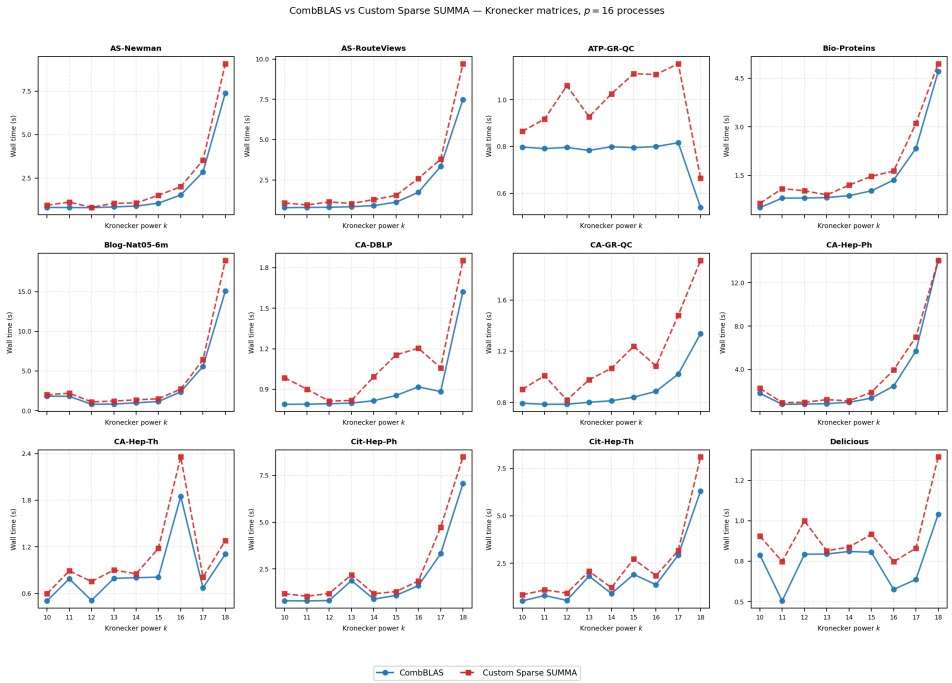


Figure 7.6: A series of plots showing the comparison between our implementation and CombBLAS.

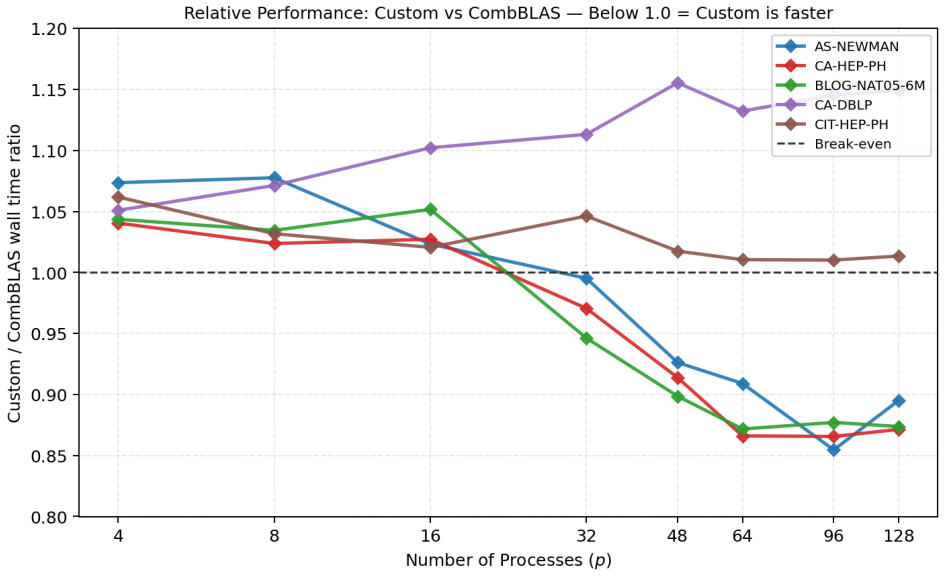


Figure 7.7: A plot showing the ratio between CombBLAS performance and our implementation.

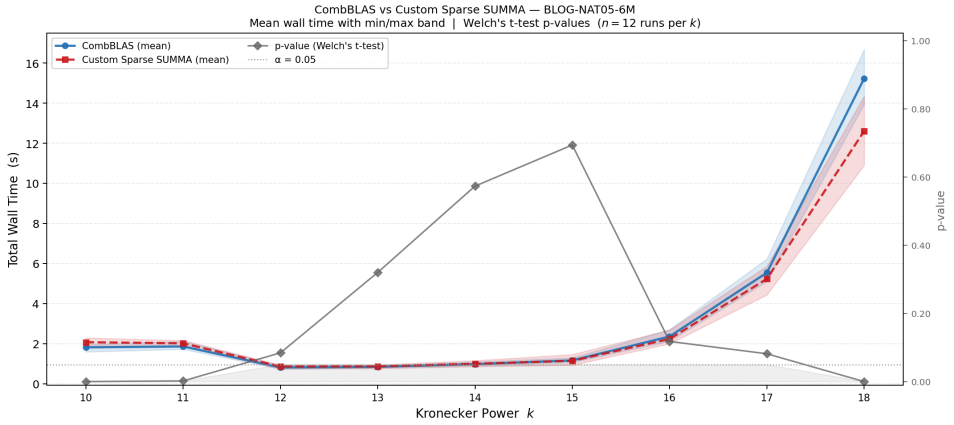


Figure 7.8: A plot showing a small statistical analysis of CombBLAS performance and our implementation.

The main outcome for these three graphs is that our implementation, as mostly expected, does not beat CombBLAS in terms of performance. However, there are situations where we can use the reference table discussed earlier to select a more optimal set of communication and storage patterns to gain additional performance. With these performance gains, we are able to beat CombBLAS by a small amount which provides a means for future work in this area in both communication tuning and local kernel improvements. The statistical analysis demonstrates that for a set of samples, the majority of the time the results are the same.

7.5 Summary

This chapter focused on the performance of the collective communications and sparse SUMMA across 19 Kronecker source networks. The results confirmed the predictions we developed in Chapters 3 and 4.

For collective communications, the Kronecker matrices confirmed that imbalance ratio is the dominant predictor of algorithm selection. Networks with an imbalance ratio above roughly 20 consistently favored the bucket broadcast, while lower imbalance

networks favored MST. The scatter, gather, and reduce collectives followed the same pattern, consistent with the theoretical analysis in Chapter 3.

For sparse SUMMA, the results showed that density and imbalance ratio together explain the best performing storage format and broadcast algorithm for the majority of networks. High density networks (above roughly 10 nonzeros per million entries) with high imbalance ratios favored BKT with CSC storage, while lower density networks favored MST with CSR storage.

Finally, a dispatch reference table was presented as a way to combine all of these findings into a guide of sorts. Given a Kronecker structured matrix, only three main measurements are needed to select an algorithm and storage format most likely to minimize wall time. This is the main and practical contribution of this thesis and is intended to be applicable to other researchers working on distributed sparse graph computations using real world networks.

Chapter 8

Visualization and STEM Communication

Outside of the implementation of sparse collective communications and algorithms, another problem that often arises is the ability to visually communicate how these algorithms work. Because they often work over multiple steps, as is seen in the SUMMA algorithm, it's not uncommon for a problem to require either multiple figures or an animated figure of some sort to understand it. This chapter focuses on the work we did throughout this thesis to create visualizations and animations related to sparse algorithms. Although there are not any clear cut experiments or results outside of the visualizations and code to create them, it does create an interesting research topic for future work.

8.1 Communicating Sparse Structure

Visualizing sparse matrices presents a unique challenge compared to visualizing dense ones. A dense matrix can be rendered in a fairly straightforward fashion with heatmaps or grids where each cell carries a value. The structure of the data here is immediately

apparent from the color of each entry. A sparse matrix, however, has the most meaning from its positions of non zero entries rather than their values. Here, the most informative visualization we can use is called a spy plot, which renders only the locations of non zero entries as filled cells against a white background. This can be seen throughout the thesis, specifically in Chapter 2, for sparsity pattern visualizations.

While these spy plots are extremely effective at communicating the structure of a matrix, it becomes much harder to use them as the matrix size grows. At extremely large dimensions commonly used for sparse matrices, it can be difficult to view the patterns like banding or blocking.

8.2 Communicating Sparse Algorithms

Communicating how sparse algorithms execute is substantially harder than communicating the dense counterparts. A dense matrix multiplication or broadcast can be visualized as a small number of diagrams because the data layout and message sizes are uniform and predictable. For sparse algorithms, the data layout varies by step, the message sizes depend on the structure, and some steps may carry no data at all while other steps carry very large messages. A single diagram can not capture all of this variability.

The SUMMA algorithm is probably the most clear example of this. At the level of the outer loop, sparse SUMMA looks identical so dense SUMMA such that we broadcast an A panel across rows, broadcast a B panel across columns, and accumulate the local product. The interesting behavior is in the inner details, where are empty broadcasts, how do panel sizes vary, how does sparsity pattern determine communication cost, etc. Communicating this in a meaningful and factual way requires not just one diagram but a sequence of diagrams that essentially animate over the steps of the algorithm. In a non-static scenario we could create a actual visualization in the form of a GIF or video that

walks through these steps, but of course that is not possible within a PDF (practically that is).

8.3 Figures in this Thesis

Many of the diagrams in this thesis were created using a custom tool designed to visualize sparse data but also visualize multiple the effects and contribution of multiple ranks. In general, there are not many preexisting tools that are able to easily visualize multi rank and sparse data easily. The following sections all demonstrate how the tool we created works by presenting examples of the code used to generate a variety of figures.

8.3.1 Tool Implementation

The tool itself that is used to generate many of the figures in this thesis was created using a mixture of Svelte and TailwindCSS [28, 29]. While not typical for creating diagrams, especially in the field of High Performance Computing, I have quite a background of web development experience especially in the context of creating visualizations for realtime or static data and decided it would be a fun challenge.

8.3.2 Matrix Structure Diagrams

For generating matrix structure diagrams, a set of scripts were added to the tool that reads in either a MatrixMarket/HDF5 file, or Zarr directory, and creates a spy plot based on the positions of non zero entries in the data. Optionally, you can also create static diagrams that are not generated by user input as shown in Listing 9.

```
<script>
    // A list of entries and their row/column positions
    const entries = []
</script>

<MatrixVisualization
    title="Diagonal Matrix"
    subtitle=""
    gradient={["#000000", "#FFFFFF"]}
    grid={true}
    {entries}
/>
```

Listing 9: An example of the code to generate a static matrix structure diagram.

At the heart of actually generating these is the standard HTML canvas element, which allows a developer to draw individual pixels. This was chosen over creating dozens of elements in an attempt to support thousands of more elements, although, limitations are still imposed by the browser due to both the processing speed of Javascript and its ability to load large files. Theoretically this tool could be improved by the use of WebAssembly [30] and having it handle the drawing of the canvas, but that was not in the scope of this thesis.

Whenever a matrix is loaded, a minimum pixel size is decided which determines if groups of non zeroes need to be condensed or not. When a group (or block) of non zeroes is condensed, the size of that inner block is calculated and then the density of it is determined. Essentially, this means on matrices that are extremely large some resolution will be lost because it becomes infeasible to display every pixel. After this

is determined, every pixel is drawn based on the input parameters such as an optional gradient based on the value of an element, grids, etc.

8.3.3 Collective Communication Diagrams

For generating collective communications, a set of scripts similar to those for the matrix structure diagrams were added to the tool that can take any number of states and any number of processes and visualize the way that data flows between each state. Listing 10 demonstrates a subset of the code required to generate Figure 2.7.

```
const states = [
    mk([[0, x]]),
    mk([[0, x], [3, x]]),
    mk([
        [0, x], [1, x],
        [2, x], [3, x],
    ]),
];
const stepLabels = ["Step 1", "Step 2"];
const messages = [
    [{ from: 0, to: 3, label: "x" }],
    [
        { from: 0, to: 1, label: "x" },
        { from: 3, to: 2, label: "x" },
    ],
];

<CollectiveDiagram
    title="MST Broadcast"
    subtitle="root = 0"
    p={4}
    {states}
    {stepLabels}
    {messages}
/>
```

Listing 10: An example of the code for a MST Broadcast visualization.

One of the key considerations in developing this model of generating collective visualizations was the ability to automate them if desired. Creating a tool to automatically generate the output for a collective would be fairly trivial.

8.3.4 Summary

As previously mentioned, outside of the figures used to create this thesis and the individuals reading this thesis, there were no experiments or studies ran to determine the effectiveness of these figures. I implemented this tool as a way to easily regenerate new figures and graphs over time without having to mangle with *matplotlib* or other Python libraries which are most commonly used for this purpose. Over the duration of my thesis I spent a lot of time improving this tool, adding features like importing matrix files from disk, etc.

Outside of the tool, the whole concept of generating visualizations for sparse data is still evolving. The usage of sparse matrices and data continues to grow day by day and so does the requirement to teach it and how it operates, even though it may be initially complex.

Chapter 9

Discussion and Future Work

9.1 Summary of Contributions

This thesis has made four primary contributions to the understanding of sparse collective communication and distributed sparse matrix multiplication. First, it has provided a systematic analysis of how four distinct sparsity patterns: uniform random, banded, block, and Kronecker product affect the communication cost, load balance, and scaling behavior of standard collective communication operations adapted for sparse data. Prior work on sparse distributed computing has frequently focused on a single application or a single class of sparsity patterns, and this thesis was created to span multiple patterns under a shared cost model.

Second, it extended the SUMMA algorithm to the sparse case and analyzed how the choice of storage format interacts with the two phase broadcast and multiply structure of the algorithm. Our analysis showed that format selection is a critical detail in getting performance out of the algorithm, particularly at lower densities where we saw the $CSR \times CSC$ strategy perform well as it avoids extractions that are not most efficient for the storage type.

Third, it demonstrated a need for and produced some small tooling for the visual communication of sparse algorithms and matrices. The custom visualization tool we created creates a rough foundation for generating spy plots, multi rank collective diagrams, and step wise algorithm diagrams that are not typically easily produced by standard scientific plotting libraries.

9.2 Limitations

Several limitations of this thesis should be noted.

First, the evaluation of this thesis was purely conducted on the OU OSCER Super Computer with its specific configurations and hardware. The statistics reported in this thesis are therefore specific to that hardware and may not convert perfectly to other systems.

Second, the synthetic matrix generation used for uniform random, banded, and block sparse matrices produce perfect structures and may not always match real world structures. For example, a real banded matrix rarely have perfectly formed uniform bands and block sparsity may not align as nicely with process boundaries as the ones in this thesis did. The experiments here served as a way to isolate the effects of patterns and may perform different on real matrices. However, the introduction and evaluation of Kronecker matrices was implemented to help mitigate this limitation.

9.3 Future Work

9.3.1 Extending to Other Sparse Formats

The analysis in this thesis focuses on the COO, CSR, and CSC formats, which are among the most commonly used formats in high performance sparse computing. However, a

number of specialized formats offer performance advantages for specific sparsity patterns. For example, the ELLPACK format stores a fixed number of nonzeroes per row in a dense array, which would be efficient for matrices with uniform row density but wastes space on skewed distributions [31]. We also discussed optimized Block Compressed Sparse Row earlier in Chapter 2.

9.3.2 Topology

All experiments in this thesis treat the communication network as relatively flat and do not take into account any sort of hierarchical organization. In the real world, HPC clusters may be organized into specific topologies that could be taken advantage of in some scenarios. An interesting expansion of this work could be evaluating how topology aware collective algorithms perform on the sparse workloads we studied here and whether exploiting that reduces the communication cost for sparsity patterns that produce highly unequal message sizes.

9.3.3 Adaptive Algorithm Selection

One of the practical outcomes for this thesis is that no single collective algorithm or storage format is optimal across all sparsity patterns and densities. The MST collectives for example perform well at low densities while the bucket collectives are preferable when message sizes are large and bandwidth ends up being the bottleneck. Given this, a dispatch system that measures a small set of matrix properties at load time such as density or imbalance ratio and selects the appropriate communication algorithm and storage format accordingly could be feasible. The outcome of this is that researchers would not need any deep knowledge of the tradeoffs mentioned here and the dispatch would handle a large subset of the work. Of course, there are tradeoffs to this method such as the time it takes to analyze a small section of the matrix and whether or not

that small section is representative of the total matrix.

Bibliography

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003.

[2] SciPy Developers, *SciPy sparse matrix documentation*, 2023. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/sparse.html.

[3] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-purpose sparse matrix–matrix multiplication on GPUs for graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, 2020. DOI: 10.1109/SC41405.2020.00076.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep. 1999-66, 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/.

[5] G. Strang, *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007, ISBN: 9780961408817.

[6] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," 1, vol. 18, 2004, pp. 135–158. DOI: 10.1177/1094342004041296.

[7] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proceedings of the International Conference on High Performance Computing and Communications (HPCC)*, ser. Lec-

ture Notes in Computer Science, vol. 3726, Springer, 2005, pp. 807–816. DOI: 10.1007/11557654_91.

[8] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication," in *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, Springer, 2005, pp. 133–145. DOI: 10.1007/11564096_9.

[9] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.

[10] Graph500 Steering Committee, *Graph500 benchmark specification*, 2010. [Online]. Available: https://graph500.org.

[11] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 4.1," University of Tennessee, Tech. Rep., 2023. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[12] E. Gabriel et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 3241, Budapest, Hungary: Springer, 2004, pp. 97–104.

[13] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. DOI: 10.1002/cpe.1206.

[14] R. A. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997. DOI: 10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2.

[15] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software*, vol. 45, no. 4, 44:1–44:25, 2019. DOI: 10.1145/3322125.

[16] J. Kepner et al., "Mathematical foundations of the GraphBLAS," *2016 IEEE High Performance Extreme Computing Conference*, HPEC 2016, pp. 1–9, 2016. DOI: 10.1109/HPEC.2016.7761646.

[17] Redis Labs, *RedisGraph: A graph database module for Redis*, https://redis.io/docs/latest/operate/oss_and_stack/stack-with-enterprise/deprecated-features/redisgraph/, Accessed: 2024, 2019.

[18] The MathWorks, Inc., *MATLAB r2021a release notes*, https://www.mathworks.com/help/matlab/release-notes.html, Accessed: 2024, 2021.

[19] A. Buluç and J. R. Gilbert, "Parallel sparse matrix–matrix multiplication and indexing: Implementation and experiments," 4, vol. 34, 2012, pp. C170–C191. DOI: 10.1137/110848244.

[20] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011. DOI: 10.1177/1094342011403516.

[21] 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*, 2021. DOI: 10.1109/TPDS.2021.3094895.

[22] A. Azad et al., "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016. DOI: 10.1137/15M104253X.

[23] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, "Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale," in *Proceedings of the 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 89–100. DOI: 10.1109/IPDPS49936.2021.00018.

[24] O. Selvitopi, M. T. Hussain, A. Azad, and A. Buluç, "Optimizing high performance Markov clustering for pre-exascale architectures," in *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020. DOI: 10.1109/IPDPS47924.2020.00029.

[25] Y. Hong and A. Buluç, "A sparsity-aware distributed-memory algorithm for sparse-sparse matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24)*, 2024. DOI: 10.1109/SC41406.2024.00053.

[26] OU Supercomputing Center for Education and Research (OSCER), *OU supercomputing center for education and research*, https://www.ou.edu/oscer, Norman, OK.

[27] B. Mohr, "PMPI tools," in *Encyclopedia of Parallel Computing*, D. Padua, Ed., Boston, MA: Springer, 2011. DOI: 10.1007/978-0-387-09766-4_57.

[28] R. Harris and Svelte Contributors, *Svelte*, https://svelte.dev, Accessed: 2024, 2024.

[29] A. Wathan and Tailwind Labs, *Tailwind CSS*, https://tailwindcss.com, Accessed: 2024, 2024.

[30] A. Haas et al., "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[31] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'09)*, 2009. DOI: 10.1145/1654059.1654078.