Distributed Sparse Computing and Communication for Big Graph Analytics and Deep Learning

by

Mohammad Hasanzadeh Mofrad

Master of Science, Amirkabir University of Technology, 2013

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Mohammad Hasanzadeh Mofrad

It was defended on

October 30, 2020

and approved by

Dr. Rami Melhem, Department of Computer Science, University of Pittsburgh

Dr. Alexandros Labrinidis, Department of Computer Science, University of Pittsburgh

Dr. John Lange, Department of Computer Science, University of Pittsburgh

Dr. Balaji Palanisamy, Department of Informatics and Networked Systems, University of Pittsburgh

Dr. Mohammad Hammoud, Department of Computer Science, Carnegie Mellon University in Qatar

Dissertation Director: Dr. Rami Melhem, Department of Computer Science, University of Pittsburgh

Distributed Sparse Computing and Communication for Big Graph Analytics and Deep Learning

Mohammad Hasanzadeh Mofrad, PhD

University of Pittsburgh, 2020

Sparsity can be found in the underlying structure of many real-world computationally expensive problems including big graph analytics and large scale sparse deep neural networks. In addition, if gracefully investigated, many of these problems contain a broad substratum of parallelism suitable for parallel and distributed executions of sparse computation. However, usually, dense computation is preferred to its sparse alternative as sparse computation is not only hard to parallelize due to the irregular nature of the sparse data, but also complicated to implement in terms of rewriting a dense algorithm into a sparse one. Hence, foolproof sparse computation requires customized data structures to encode the sparsity of the sparse data and new algorithms to mask the complexity of the sparse computation. However, by carefully exploiting the sparse data structures and algorithms, sparse computation can reduce memory consumption, communication volume, and processing power and thus undoubtedly move the scalability boundaries compared to its dense equivalent.

In this dissertation, I explain how to use parallel and distributed computing techniques in the presence of sparsity to solve large scientific problems including graph analytics and deep learning. To meet this end goal, I leverage the duality between graph theory and sparse linear algebra primitives, and thus solve graph analytics and deep learning problems with the sparse matrix operations. My contributions are fourfold: (1) design and implementation of a new distributed compressed sparse matrix data structure that reduces both computation and communication volumes and is suitable for sparse matrix-vector and sparse matrix-matrix operations, (2) introducing the new MPI $*$ X parallelism model that deems threads as basic units of computing and communication, (3) optimizing sparse matrix-matrix multiplication by employing different hashing techniques, and (4) proposing the new data-then-model parallelism that mitigates the effect of stragglers in sparse deep learning by combining data and model parallelisms. Altogether, these contributions provide a set of data structures and algorithms to accelerate and scale the sparse computing and communication.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

The current disruptive state of High Performance Computing (HPC) and Cloud computing is made possible by emerging CPU and GPU architectures [18, 95], parallel processing substrates [43], and scalability techniques to name just a few. These technologies are designed to meet the processing challenges of **Big Data** produced in healthcare, government, IT, and other sectors. Presently, with the impending information explosion of Big Data, most of the raw Big Data is dominated by data produced by the web, social networks, road networks, recommendation systems, location services, and biomedical domain.

Scalable systems for Big Data analytics are invaluable tools for efficiently extracting insights from vast volumes of raw data generated mainly by billions of Internet-connected users and devices. Big Data domains that focus on the relationships between data points, (e.g., the Web, social networks, road networks, etc.) often model such data as graphs and use graph analytics tools to interpolate or extract graphs' relationships.

Scalable systems for Big Data Classification apply machine learning techniques to Big Data. Deep learning [67] has delivered promising advancement in many large-scale practical problems such as natural language processing [33, 81], speech recognition [4, 56], and computer vision [32, 101]. Emergence of commercial virtual assistants, self-driving cars, online item recommendation systems, and AI stock trading are dramatically accelerated by the research conducted in deep learning. This dramatic change in IT industry is significantly accredited to the research on the scalability of neural networks via revamping their architecture. Often, these complex architectures can simply be represented by graphs where the relational representation of graphs makes it possible to exploit graph structures for weight propagation of neural networks [107].

The goal of this dissertation is to leverage **sparse linear algebra** for graph analytics *and* deep learning. Graphs are inherently heavily sparse structures. Exploiting this property, many algebraic methods can be applied to graph problems by converting a graph into an adjacency matrix [49]. *Thus, a linear algebra approach to graph analytics is an alternative for a graph theory approach.*

Traditionally, the architecture of the majority of neural networks is dense. However, sparsity can be found in graph representation of large neural networks. Despite the conventional wisdom implying that a neural network cannot classify accurately unless the network is fully connected (dense), the recent introduction of sparse deep neural networks [42, 63, 72, 85] has shown that sparse networks can deliver comparable accuracy to their dense counterparts while consuming significantly less memory and computing power. *Hence, a linear algebra approach can be used to reformulate the computation of neural networks.*

Sparse linear algebra-based techniques are mostly designed around a set of simple yet powerful primitives. The two primary kernels widely utilized in this dissertation are generalized **Sparse Matrix-Vector (SpMV)** and **Sparse Matrix-Matrix Multiplication (SpMM)** [64, 126]. Combining these primitives with efficient sparse data structures and algorithms can potentially accelerate compute- and memory-intensive applications while having a small memory requirement. The SpMV primitive allows implementation of a wide array of iterative graph applications in the language of linear algebra including PageRank, shortest path, breadth first search and connected component. Chapter 3 introduces a new optimized sparse data structure that accelerates the SpMV primitive. Also, Chapter 4 reports a new parallelism model that scales this primitive. Furthermore, SpMM primitive -which is more complex than the SpMV- is the core kernel behind training/inference of sparse deep neural networks. Chapter 5 studies the effect of hashing on the performance of DNN inference and Chapter 6 introduces a new parallelism model that scales the DNN inference.

1.1 Distributed Sparse Computing and Communication for Graph Analytics

1.1.1 Identifying Sources of Sparsity in Graph Analytics

Graph analytics is about determining the strength or direction of a relationship between vertices of an input graph. Most traditional graph analytics systems employ vertex-centric computation schemes [50, 52, 54, 73, 75, 76, 79, 80, 111, 129]. However, many recent systems have opted alternatively for linear algebra-based computation schemes, leveraging decades of

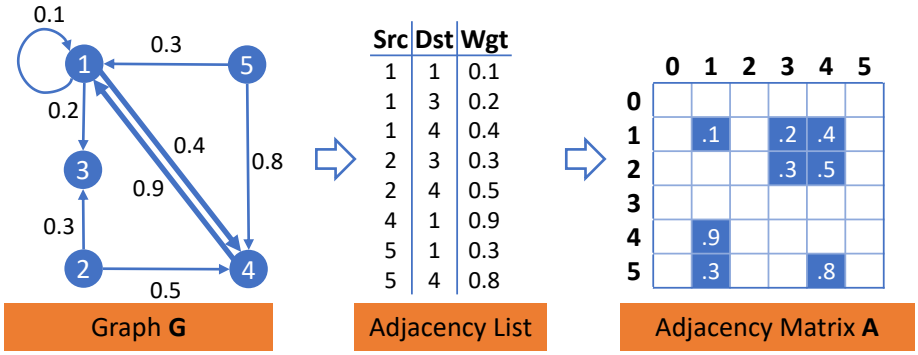


Figure 1: The transformation of a graph into its adjacency list and then its adjacency matrix. In addition to many zero entries, there are even empty rows (third row) and columns (second column) that can be ignored when executing an algorithm.

work by the HPC community on optimizing the performance and scalability of basic linear algebra operations [2, 5, 9, 21, 36, 41, 59, 64, 77, 115].

From Figure 1, we can identify multiple potential sources of sparsity. Mainly, having an $n \times n$ dense matrix with n^2 entries, a sparse representation reduces the time complexity of the matrix from $O(n^2)$ to $O(nnz)$ where nnz is the number of nonzero entries. In light of this, the majority of literature [2, 5, 115] have overlooked the dimensionality of a sparse matrix problem and leave it to be $n \times n$ whereas the true dimensionality of a sparse matrix problem is $nzr \times nzc$ where nzr and nzc are the number of nonzero rows and columns. Moreover, the SpMV primitive constitutes operating on a sparse matrix and two dense input and output vectors. Most solutions [2, 5, 115] aim to optimize the computation for the sparse matrix independent of vectors and keep them of size n; neglecting the sparsity distribution of input and output vectors completely. However, employing a sparse vector representation, sizes of input and output vectors can be reduced from $O(n)$ to $O(nzr)$ and $O(nzc)$ which can save in both storage size and the communication volume.

1.1.2 Exposing Challenges of Sparsity in Graph Analytics

Graphs are intrinsically sparse, and their sparsity is primitively encoded in their underlying structure. Hence, an adjacency matrix of a graph essentially contains many zero entries.

In linear algebra-based graph analytics, sparsity exists in two types: sparsity derived from the adjacency matrix of the input graph and sparsity produced by the semantic of the graph application. Indexing nonzero entries of a sparse matrix is among certain challenges of the former type of sparsity and identifying and exploiting the runtime sparsity characteristics of an application is among challenges of the latter type.

Big real-world graphs tend to produce highly sparse matrices and thus the data structures *and* algorithms associated with these operations need to be optimized for sparsity. Often, such optimizations are pursued independently, resulting in various algorithms that do not inherently exploit certain common data structural optimizations and vice-versa. Therefore, tightly coupling specific algorithmic and data structural optimizations can yield significant performance and scalability benefits in both centralized and distributed settings. Chapter 3 introduces a new sparse matrix format that co-compresses both matrix and vectors and provides an efficient indexing algorithm to access their compressed data.

1.1.3 Achieving scalability in Distributed Graph Analytics

Given the limited hardware resources of a single computing node, it is always of community interest to scale out the computation over a cluster of machines using partitioning techniques. MPI + X [118, 112, 11] is the defacto High Performance Computing (HPC) parallelism model that achieves scalability through partitioning. Many distributed graph theory-based systems [50, 54, 73, 75, 76, 129] rely on this model, where first a big graph is divided into multiple equal-sized subgraphs based on the number of available MPI (Message Passing Interface) [58] processes for scaling out, and then inside each machine, a subgraph is further partitioned based on the number of available threads for scaling up. Here, MPI is used to launch processes inside each machine and a threading library such as OpenMP [97] or Pthread [71] (the X part) is used to launch threads inside each process. Similarly, many distributed linear algebra-based systems [2, 5, 21, 36, 41, 59, 115] follow MPI + X model and use sparse matrix partitioning [2, 47] to break the adjacency matrix of a graph into multiple tiles for scaling out among machines, and further partition each tile based on the number of threads for scaling up within a machine. In MPI + X parallelism model,

the units of computation and communication are processes which is not resourceful. Later, Chapter 4 presents a new parallelism model called MPI $*$ X that deems threads as basic units of computation and communication and diagonally scales over a cluster of machines. This new parallelism model has less synchronization overhead and offers better overlapping of computation with communication.

The key to achieving scalability in parallel and distributed graph/matrix computing is to first partition the graph/matrix using the total number of available computing endpoints. Usually, partitioning is followed by a placement algorithm which assigns the computing endpoints to partitions. The goal of the placement algorithm is to balance the computation across machines while keeping the communication cost low. While many systems [2, 5, 19, 59, 115] run the placement algorithm independent of system configuration; an efficient placement, however, allows better utilization of hardware resources if certain architectural characteristics such as Non-uniform Memory Access (NUMA) is utilized. Chapter 4 presents a NUMA-aware placement algorithm that bridges the gap between the placement algorithm and microarchitectural characteristics.

1.2 Distributed Sparse Computing and Communication for Deep Learning

1.2.1 Identifying Sources of Sparsity in Sparse Deep Neural Networks Inference

The architecture of a neural network is defined using its number of layers (depth) and the number of neurons per layers (height/width). A layer of a neural network can be modeled using graph representation where neurons are vertices and their connections are edges [61]. This representation is adequate to model both computation of neurons *and* the state on which the neural network operates [1] as a directed acyclic graph [75]. Exploiting this property, an adjacency matrix representation is used to model layers of a neural network where connections of each layer is shown using a separate matrix. Often, these matrices are filled densely with weights due to the fully connected feature of neural networks, however, for sparse neural networks layer matrices are sparse as a result of following a predefined

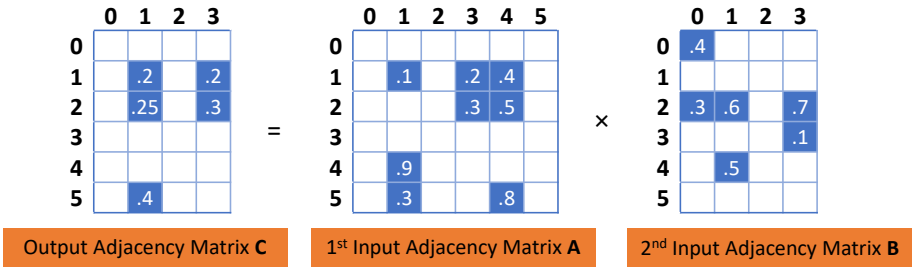


Figure 2: Matrix multiplication where the multiplication of first and second input matrices A and B produces the output matrix C. These matrices can efficiently be represented by sparse formats to reduce in both storage and computation.

architecture [74, 85, 102] or pruning a network after being trained [72].

The bulk of computation of a fully connected (dense) neural network belongs to operations applied to its weights where all weights are presented. On the other hand, in sparsely connected (sparse) neural networks with less available weights, the amount of computation is greatly reduced to the number of available weights. Hence, deep neural networks have less processor and memory requirements and are incredibly faster to train and infer.

The key linear algebra primitive for training/inference of neural networks is the SpMM primitive. In SpMM, the result of multiplying two input matrices produces the output matrix. As shown in Figure 2, sparsity can be found in the first and second input matrices, where, e.g., in the context of neural networks the first input matrix is the input dataset, the second input matrix is the first neural network layer, and the output matrix is the first input matrix for the second layer of the neural network. In addition, the resulting matrix of the SpMM is also sparse with extreme changes in its nonzero distribution compared to the input matrices.

1.2.2 Exposing Challenges of Sparsity in Deep Learning

As the key algorithm behind the training of DNNs, Backpropagation algorithm [51] comprises of two passes. In the *forward pass*, the algorithm computes the network response for an input and in the *backward pass*, it updates the weights backward using the error calculated for each neuron. Backpropagation algorithm usually uses a variant of gradient decent algorithm

[106] to calculate the error for updating the weights backward. The Inference algorithm is identical to the first pass of the DNN training where an input instance is feed to the trained network and the network outputs a prediction. Training/inference shares the same algebraic operations where an input is multiplied by the receptive weights and the summation of their inner products fans out to the connected neurons of the next layer. Therefore, SpMM turns out be to be the key primitive behind DNN training/inference. Note that DNN inference is the target application of this dissertation.

Parallel and distributed training/inference of a sparse deep neural network can essentially be reduced to the problem of parallel execution of SpMM. The theory of parallel matrix - matrix multiplication spans over decades of research with Cannon's algorithm [24] and Scalable Universal Matrix Multiplication Algorithm (SUMMA) [122] as examples of parallel dense matrix-matrix multiplication. Additionally, Gustavson's algorithm [53], Sparse Accumulator (SPA) [46], sparse Cannon [20], and Sparse SUMMA [22] are among the state-of-the-art SpMM algorithms. SpMM performance is highly sensitive to the nonzero distribution of input matrices as frequent sequential accesses to a same region of input matrices can drastically enhance the cache utilization. Moreover, due to the nature of sparsity, SpMM requires a graceful memory allocation strategy that can efficiently allocate memory for sparse operations while avoiding excessive memory operations at runtime. Chapters 6 and 5 carefully studies these challenges and propose solutions to them.

1.2.3 Achieving scalability in Distributed Deep Learning

Deep neural networks are usually trained using a variant of gradient descent optimization algorithm [106]. Until recently, due to the sequential nature of this algorithm, training of neural networks was only limited to a single machine. In such a configuration, not only training a modest neural network takes too long, but also the size of the network is limited by the computing resources of that machine. Recently, with the introduction of new distributed gradient descent algorithms [1, 39], neural networks can be scaled out to a distributed setting [1, 6, 13, 30, 29, 35, 39, 45, 57, 93, 110].

Table 1: Contributions overview

Scalability			
Distributed	Chapter 3 and 4	Chapter 5 and 6	
Multicore	Chapter 3	Chapter 5	Operator (Application)
	SpMV (Graph Analytics)	SpMM (Deep Learning)	

Often, distributed training/inference of neural networks is achieved via *data parallelism*, or *model parallelism*. Data parallelism is the partitioning of the input dataset and model parallelism is the partitioning of the network among multiple processes [45]. Chapter 6 introduces the data-then-model parallelism that combines data and model parallelisms to facilitate the parallel execution of neural network inference.

1.3 Research Contributions

In this section, I introduce the key research contributions of this dissertation. As depicted in the quadrant of Table 1, these contributions are studied in two dimensions, including operator (application) and scalability. From the X-axis of Table 1, the studied operators are SpMV which is studied in the context of graph analytics and SpMM which is studied in the context of deep learning. From the Y-axis of Table 1, the scalability issue is studied in two modes, multi-core and distributed. The Following is a more descriptive explanation of the contributions of this dissertation that will be presented in the following chapters:

1. I design and implement a new sparse matrix compression format called Triply Compressed Sparse Column (TCSC) that co-compresses a sparse matrix and the input and output vectors. This compression technique removes empty rows and columns of a sparse matrix before compressing it. Similarly, it removes empty rows/columns of a sparse 1D row/column vector. Adopting the conventional Sparse Matrix - Vector (SpMV) primitive, I redesign the SpMV primitive and devise a new primitive called Sparse Matrix -

Sparse input and output Vectors ($SpMSpV^2$). Chapter 3 shows how coupling the TCSC data structure and $SpMSpV^2$ primitive accelerates the execution of graph applications.

2. Scalability through partitioning is a common practice in distributed computing. Following MPI + X parallelism model, traditionally, scaling direction is independent of partitioning where the data is first partitioned based on the number of MPI processes for scaling out and then if possible, threads for scaling up. In contrast to this model, I couple the partitioning algorithm and scalability direction and propose MPI $*$ X parallelism model that uses thread-based data partitioning. In this model, data is partitioned based on the total number of threads and computation and communication is directly offloaded to threads. In addition, certain microarchitectural characteristics such as NUMA are leveraged to maximize the shared memory communication among threads. Chapter 4 presents these contributions.
3. Data and model parallelisms are two common techniques to parallelize the computation of neural networks where data parallelism parallelizes over that data and model parallelism parallelizes over the network. Data parallelism suffers from the straggler effect, while model parallelism undergoes frequent synchronizations. Chapter 5 studies the effects of hashing of neural networks. Furthermore, Chapter 6 introduces data-then-model parallelism which combines data and model parallelisms and mitigates their disadvantages using a lightweight thread scheduling algorithm.

2.0 Background and Related Work

In this chapter, I present the background needed for this dissertation and survey the related work. First, in Section 2.1, I introduce sparse matrix data structures and primitives which are the pillars of this dissertation. Next, in Section 2.2, I review sparse matrix partitioning. Then, in Section 2.3 and 2.4, I show the connection between graph theory and linear algebra and review the linear algebra-based approaches to graph analytics and deep learning. Then, in Section 2.5, I survey the non-algebraic (theory-based) approaches to graph analytics. Finally, in Section 2.6, I present a summary of this chapter.

2.1 Sparse Matrix Data Structures and Primitives

2.1.1 Sparse Matrix Data Structures

Sparse matrix data structures are the backbone of sparse computations. These data structures only store the nonzero entries of a 2D sparse matrix using a set of 1D arrays. In case of hypersparse matrices, sparse data structures help saving huge amounts of memory compared to their equivalent dense representations. Depending on requirements of an algorithm, sparse data structures can provide either sequential row-major access using Compressed Sparse Row (CSR) or column-major access using Compressed Sparse Column (CSC) format to consecutive nonzero entries of rows or columns of a sparse matrix [46].

CSR and CSC formats [46] are two widely used sparse data structures that compresses the nonzero entries of a matrix. Doubly Compress Sparse (DCSR) and Doubly Compress Sparse Column (DCSC) [10, 19] are descendants of CSR and CSC formats which further removes empty rows and columns of a matrix before compressing its nonzero entries. These data structures have been implemented in many programming languages and libraries. For example, Matlab's `sparse` data structure uses CSC [46, 84], the Combinatorial Basic Linear Algebra Subprogram (CombBLAS) [21] supports distributed DCSC data structure, and the SciPy library for Python [109] supports both CSR and CSC.

In the context of linear algebra-based graph analytics, column compressed sparse data structures are preferred to row compressed as there is often more empty columns than empty rows. Recent linear algebra-based systems such as CombBLAS [19] and LA3 [2] use DCSC to represent sparse matrices. DCSC removes empty columns of a matrix and hence it is asymptotically faster than CSC. Moreover, in the context of deep learning, a sparse deep neural network [3, 63, 102, 103] can encode the sparsity of its hidden layers using the compressed sparse data structures. For example, Compressed Deep Neural Network (CDNN) [55] and Sparse CNN (SCNN) accelerator [82] utilize CSC, and Sparse Evolutionary Training - Multi Layer Perceptron (SET-MLP) [74] uses CSR. Chapter 3 gives a thorough introduction on CSC and DCSC and introduces a new sparse data structure called Triply Compressed Sparse Column (TCSC) [86, 89] which is faster and more space saving than CSC and DCSC.

2.1.2 Sparse Matrix Primitives

Sparse matrix primitives are the key building blocks of many computing systems dealing with Big Data. They are small components of a larger orchestrated workflow. For example, Google Map-Reduce programming model [40] introduces `map` and `reduce` operations that can be customized by users depending on their algorithmic needs. Using these two operations many numerical problems such as counting, sorting, etc. can be implemented and solved in parallel. Similarly, a linear algebraic approach to graph algorithms and sparse deep learning boils down to a small yet powerful set of primitives including Sparse Matrix-Vector (SpMV), Sparse Matrix-Matrix Multiplication (SpMM), sparse matrix-matrix elementwise, and sparse matrix indexing primitives [21]. Among these, SpMV and SpMM are two broadly used sparse linear algebra primitives.

Utilizing the duality between a graph and its adjacency representation; a graph $G = (V, E)$ can be represented by its analogous adjacency representation A where V is a set of n vertices ($|V| = n$) and E is a set of nnz edges ($|E| = nnz$). The matrix A is an $n \times n$ matrix with n rows and columns where $A(i, j) = 1$ means there is an edge from vertex v_i to vertex v_j. A generic way of applying linear algebra to graph algorithms is to use *semiring* which is a broader definition of SpMV multiplication. In this context, the SpMV operation becomes

A op_1. op_2 v where A is a sparse matrix, v is a vector, and $op_1.op_2$ is a pair of additive and multiplicative semiring.

To elaborate, given an input graph G with n vertices, a graph algorithm on G can be translated onto an iteratively executed SpMV primitive, $y = A \oplus.\otimes x$, where A is the G's $n \times n$ adjacency matrix, x and y are input and output vectors of length n, and $\oplus.\otimes$ is a pair of overridable additive and multiplicative operations [64]. The algorithm would then iteratively apply the results from y back to x, looping until it converges or stopped after certain numbers of iterations. The sparse matrix A is commonly stored using a variant of CSC, which essentially compresses its nonzero elements into an array [19, 64]. As for x and y, they are commonly stored using dense vectors of length n [8, 64].

Similarly, SpMM, $C = A \oplus.\otimes B$ is also a widely used linear algebraic operation, where results of operating on two sparse input matrices A and B produces an output sparse matrix C. Here, A is the first $m \times n$ input sparse matrix, B is the second $n \times p$ input sparse matrix, and C is the $n \times p$ product output matrix. Moreover, $\oplus.\otimes$ is a semiring equipped with addition and multiplication. Like SpMV primitive, the sparse matrices, A, B, and C are commonly stored using a variant of CSC, which compresses the nonzero elements and offers column-major access to the data [64].

In the context of linear algebra-based graph analytics, SpMV is a widely used primitive that semantically covers a wide array of graph algorithms. For example, SpMV is used to implement PageRank (PR), Single Source Shortest Path (SSSP), Breadth First Search (BFS), Connected Component (CC), Collaborative Filtering and Between Centrality (BC) algorithms [2, 5, 115]. SpMV is implemented in CombBLAS [19], Knowledge Discovery Toolbox (KDT) [77], GraphMat [115], GraphPad [5], LA3 [2], GraphBLAS [37] ,GraphTap [86], and Graphite [89].

In the context of deep learning, the key kernel behind training of sparse neural networks is the SpMM primitive with multiplication and addition as the two operators of the SpMM semiring. For training of sparse neural networks, Liu et al. [72] implemented a sparse matrix - dense matrix multiplication kernel, Han et al. [55] decomposed the convolutional layers and used sparse matrix - dense vector multiplication, and Kepner et al. [63] used SpMM multiplication. Compared to dense matrix-matrix primitive, SpMM primitive [46, 53] has less

computation complexity and memory requirement. However, efficient application of SpMM requires addressing certain challenges such as the SpMM algorithm, SpMM accumulation strategy, and SpMM memory allocation. Later, Chapter 5 studies these challenges and introduces solutions to them.

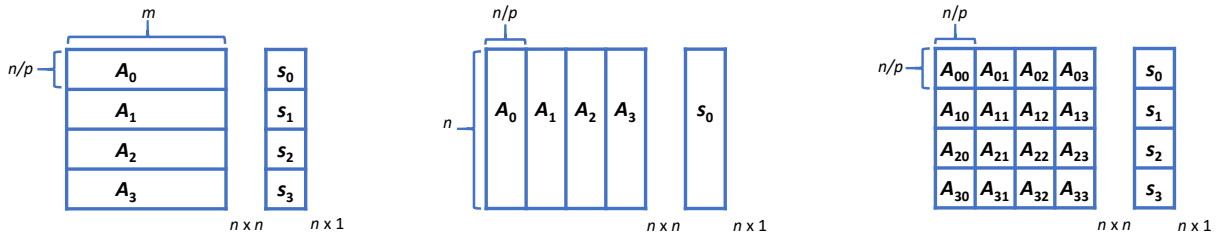
The SpMV and SpMM performances are highly sensitive to their selected sparse data structures. Chapter 3 presents Triply Compressed Sparse Column (TCSC) data structure and its designated primitive Sparse Matrix - Sparse input and output Vectors (SpMSpV^2). This data structure and primitive show how tightly coupling specific algorithmic and data structural optimizations can result in significant performance and scalability benefits in sparse computation and communication.

2.2 Sparse Matrix Partitioning

Matrix partitioning (tiling) partitions a 2D matrix into smaller partitions (sub-matrices or tiles) [47, 92, 91]. Later these tiles are distributed among machines. Here, the number of nonzero entries within each tile translates into the computational load of a partition and the number of tiles translates into the communication cost. When it comes to the distributed in-memory computing, partitioning turns out to be a sensitive task because first, imbalanced partitions cannot effectively harvest available computational resources proportional to the size of partitions and thus limit the scalability. Second, imbalanced partitions impose communication overhead on a subset of overloaded machines. Therefore, in the case of a large distributed cluster, communication time totally dominates the execution time.

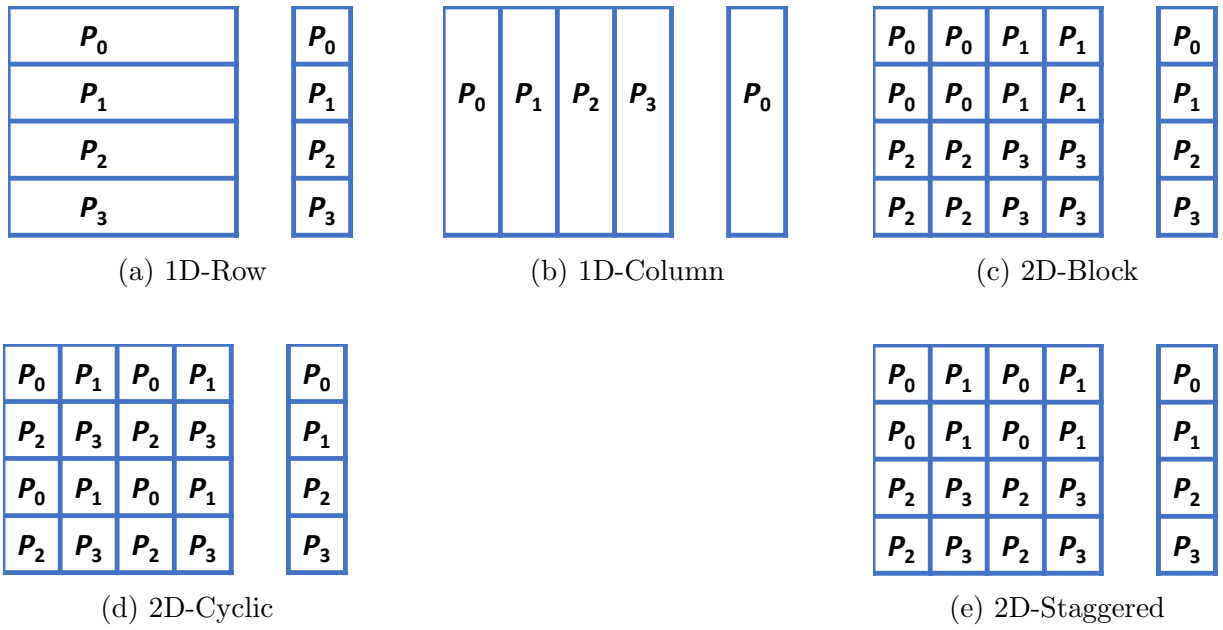
In sparse matrix partitioning, vertex-cut partitions are preferred to edge-cut. This is because a vertex-cut partitions the 2D matrix rows and/or columns into independent blocks ready to use by SpMV and SpMM operations. Furthermore, many matrix partitioning algorithms first hash and then break the 2D matrix into tiles. Hashing provides better load balance for power-law graphs since it uniformly distributes edges among partitions [2, 5].

Partitioning is applied to both matrix and vectors used in the computation. Having an $n \times n$ matrix A and a $n \times 1$ vector s, Figure 3 shows the layouts of three matrix partitioning



(a) 1D-Row (b) 1D-Column (c) 2D

Figure 3: Matrix and vector tiling for n x n matrix A and n x 1 vector s with p=4 processes.



(a) 1D-Row (b) 1D-Column (c) 2D-Block

(d) 2D-Cyclic (e) 2D-Staggered

Figure 4: Process placement for n x n matrix A and n x 1 vector s with p=4 processes.

algorithms including 1D-Row, 1D-Column, and 2D. In 1D partitioning, number of partitions is p and in 2D it is p^2 where p is the number of processes.

Partitioning is followed by a placement algorithm which assigns processes to tiles. Figure 4 shows different process placement strategies including 1D-Row, 1D-Column, 2D-Block, 2D-Cyclic [5], and 2D-Staggered [2]. 1D-Row and 1D-Column place a unique process in each row and column of tiles. 2D-Block assigns contiguous blocks of 2D matrix to each process and 2D-Cyclic assigns processes to tiles in a cyclic fashion. Finally, 2D-Staggered is like the 2D-Cyclic except for the fact it swaps rows to assign unique processes to diagonal tiles in

order to layout the communication pattern among processes.

In 1D tiling, each process will communicate with all p processes, whereas, in 2D techniques the communication is limited to $\sqrt{p}$ processes per row and column of tiles. Therefore, 2D tiling strategies are more common due to their less communication cost. Traditionally, 2D partitioning and placement algorithms are designed for processes; Chapter 4 introduces a new 2D tiling technique that is designed for threads from scratch.

2.3 Linear Algebra-based Graph Analytics

2.3.1 The Case for Duality Between Graph Theory and Linear Algebra

A linear algebraic approach to graph processing consists of a fine transition from graph representation of a set of vertices to adjacency matrix representation of matrix elements [64]. Leveraging the duality between a graph and its adjacency representation, many graph algorithms can be translated into linear algebra domain effortlessly. However, matrices have not been traditionally adapted for parallel implementations of graph algorithms because a dense matrix representation is not an efficient representation for a sparse graph. With recent advances in sparse data structures *and* algorithms, practical approaches have been put forth to enable processing of large sparse graphs using their adjacency matrix representation.

A linear algebraic approach to graph algorithms offers a number of advantages: 1) many canonical graph algorithms are already developed and available to the community and with little or no effort they can be implemented in the language of sparse linear algebra [50, 52, 54, 73, 75, 76, 79, 80, 111, 129], 2) a wide range of dense linear algebra algorithms has been already developed which can simply be adopted for sparse linear algebra [24, 122], 3) over the past decades numerous parallel and distributed techniques and infrastructures on sparse computations have been introduced by the community [2, 5, 9, 21, 36, 41, 59, 64, 77, 115] and linear algebra can benefit from them, and 4) compared with graph computations, matrix computations are easier to code and optimize due to their predefined access strategy [64].

2.3.2 Linear Algebra-based Graph Analytics Systems

Pegasus [59] is one of the initial attempts to use linear algebra for solving graph problems which uses the Apache Hadoop implementation of MapReduce [40]. CombBLAS [21] is an edge-centric distributed graph analytics system that offers a rich set of primitives for operating on sparse matrices including SpMV, SpMM, SpAsgn (sparse matrix assignment) and SpRef (sparse matrix indexing). Knowledge Discovery Toolbox (KDT) [77] adapts CombBLAS to support semantic graphs (graphs with attributes on both edges and vertices). Also, GraphBLAS [37] tries to define the basic building blocks of graph algorithms in the language of linear algebra. GraphMat [115] is a multi-core graph analytics system that fills the gap between performance and productivity of graph analytics platforms. It abstracts a vertex program through a generalized iterative SpMV operation. Furthermore, GraphPad [5], the distributed version of GraphMat uses MPI for scalability and distributes the adjacency matrix of a graph among machines. Akin to GraphPad [5], LA3 [2] is a distributed linear-algebra based graph analytics system that incorporates communication and computation filtering to cut down the amount of SpMV operations.

2.4 Linear Algebra-based Sparse Deep Learning

2.4.1 Dense Deep Neural Networks

Often, (dense) deep neural networks consists of a series of fully connected layers. Stochastic Gradient Descent (SGD) [106] is the learning algorithm empowers training of deep neural networks. Due to being ill-suited for parallelism, training of neural networks using SGD was limited to single-machines for many years and often can only scale-up within a machine [34, 12, 127]. With the introduction of DistBelif [39] and its distributed SGD algorithms, Google broke the long-lasting barriers of SGD and extended it to distributed settings. Tensorflow [1], Project Adam [30], MXNet [29], SINGA [96], SparkNet [93], VGG-A [35], FireCaffe [57], TernGrad [124], and Hadoov [110] are among distributed CPU-based deep learning solutions. Also, S-Cffe [7] and [23] are distributed GPU solutions.

2.4.2 Sparse Deep Neural Networks

Conventionally, deep neural networks are fully connected. Sparse deep neural networks [87, 88, 90] are a new type of neural networks where the connections among hidden layers are sparse, however, compared with their dense peers, they can still offer comparable classification accuracy. The sparsity of these neural networks can derive from multiple sources such as: decomposing the neural network topology into sparse matrices [72], using Sparse Evolutionary Training (SET) to adaptively alter and drop the neural network connections [74, 85], sparsifying the topology of the neural network [63, 102], or using activation functions like Rectified Linear Unit (ReLU) that results in some neurons always producing zero [48].

2.4.3 Sparse Deep Neural Networks Parallelism Models

Neural network training is a computationally expensive task. To accelerate the training process, either a faster training algorithm like downpour SGD [39], sandblaster [39], or codistillation [6], or a scalable parallelism model such as data parallelism which parallelizes over the input [45] or model parallelism which parallelizes over the network [45] can be used. Like training, inference can also be scaled using data and model parallelisms. Chapter 6 combines data and model parallelisms and introduces a new parallelism model called data-then-model for neural network inference.

2.5 Traditional (Non-algebraic) Graph Analytics

2.5.1 Graph Theory-based Graph Analytics Systems

The community interest toward big graph analytics has been triggered with the introduction of Pregel [80] by Google and later Apache Giraph [31]. Both adapted the Bulk Synchronous Parallel (BSP) computing model [121] which expresses an application with multiple iterations where each iteration constitutes a sequence of computation, communication, and synchronization. Both Pregel [80] and Giraph [31] follow a vertex-oriented programming

model where a vertex maintains a partial view of the adjacency of the graph and can change state in each iteration and the new state is sent to the adjacent vertices (the ones a vertex has immediate connections/edges) at the end of each iteration.

Graph theory-based systems use neighborhood-based operations such as fan-in/fan-out explorations to implement a graph algorithm. The current *graph analytics platforms* can be examined from different angles. More precisely, in terms of computing needs they can be divided into three types: 1) **single node non-scalable solutions** such as Gunrock [123], Ligra [111], Polymer [128] and Galois [94], 2) **single node out-of-core solutions** such as GraphChi [66], X-Stream [105], GridGraph [130] and Mosaic [79], and 3) **distributed solutions** such as Apache Giraph [31], Google Pregel [80], GraphLab [75, 76], PowerGraph [50], PowerLygra [27], PowerSwitch [125], and Gemini [129].

Graph Theory-based analytics platforms can also be categorized based on programming models, namely: 1) **vertex-centric** in Pregel [80] and Giraph [31], 2) **edge-centric** in X-stream [105], 3) **sub-graph-centric** in GoFFish [113], and 4) **graph-centric** in Giraph++ [119]. In addition, they can be classified in terms of two major execution models, namely: 1) **synchronous model**, where vertices progress in a lock-step fashion such as in Giraph [31], and 2) **asynchronous model**, where vertices can change values anytime and be several steps apart during execution; an example asynchronous system is PowerGraph [50]. Last, MapGraph[41], Gunrock [123], mGPU [99], and Garaph [78] are among recent multi-GPU graph analytics systems.

2.5.2 Graph Data Structures and Operations

Graphs are mostly represented using their adjacency list where each vertex utilizes a data structure such as an array or a linked list to represent its outgoing edges [92, 91]. In addition, other systems developed more efficient data structures to represent vertices and their connections. Ligra [111] uses *vertex subset* data structure to extract subsets of vertices and update vertices that are active currently. Gemini [129] uses a dual-mode edge representation where outgoing edges are organized in CSR and ingoing edges in CSC. Also, Mosaic [79] uses CSR to represent edges.

Graph algorithms are adopted to work with fan-in and fan-out operations where fan-in operation operates on ingoing edges of vertices and fan-out operation operates on their outgoing ones. PowerGraph [50] introduces a new vertex programming abstraction called Gather, Apply, and Scatter (GAS) model. In GAS, gather operation collects information on adjacent vertices, apply changes the central vertices state, and scatter updates adjacent vertices. Ligra [111] operates on a set of vertices called *frontier vertices* which are vertices that are accessible. It uses *edge map* to operate on edges in sparse and dense modes and uses *vertex map* to operate on vertices. Gemini [129] adopts the sparse-dense representation of Ligra and introduces a push (sparse) - pull (dense) model to decouple the propagation of vertices from processing of edges.

2.5.3 Graph Partitioning

The objective of the **K-way balanced graph partitioning** is to partition a graph $G = (V, E)$ into $k > 1$ balanced partitions. A vertex-centric k-way graph partitioning uses *vertex-cut* to divide V into k distinct partitions of almost equal-sized $(|V|/k).(1 + \epsilon)$ i.e. $\epsilon > 0$ [100]. On the other hand, an edge-centric partitioning which uses *edge-cut* results into k partitions of roughly size $(|E|/k).(1 + \epsilon)$ [92, 91]. Also, other algorithms might use *hybrid-cut* to create partitions (a combination of those vertex- and edge-cut) [28, 120]. In the context of graph partitioning, a cut means extra communication. For real-world graphs that follows power-law distribution, e.g., social networks, an edge-cut can create better partitions compared to vertex-cut because a huge number of edges belong to a small subset of vertices (e.g., Twitter graph is a good example of a power-law graph) [28, 50].

Upon running a graph application in a distributed fashion, the biggest partition bounds the amount of computation and the number of inter-partition edges bounds the amount of communication under each processing step. Hence, a K-way balanced partitioning potentially lowers the runtime of a distributed graph analytics platform via enforcing near-uniform resource utilization across machines while reducing the communication volume. Kernighan-Lin [65], Spectral partitioning [26], and Metis [60] are examples of state-of-the-art balanced partitioning algorithms. Moreover, Ja-Be-Ja [100] (a local search partitioning), Fennel [120]

(a streaming balanced partitioning), Spinner [83] (a label propagation-based partitioning), and Revolver [92, 91] are four vertex-centric balanced partitioning algorithms.

2.6 Summary

In this chapter, I review the sparse matrix data structures and primitives, and partitioning algorithms. I depicted the link between graph theory and linear algebra and argued that many graph operations can be converted into basic linear algebra primitives such as SpMV and SpMM. Also, I reviewed sparse and dense deep learning and their parallelism models. Reporting the related work, in the following section, a new optimized compressed sparse format which is inspired by the CSC data structure will be introduced.

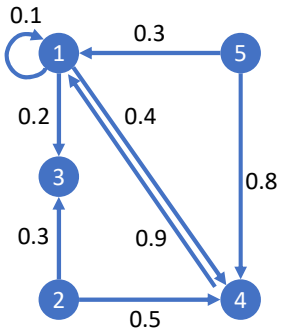
3.0 Efficient Distributed Graph Analytics using Triply Compressed Sparse Format

This chapter presents Triply Compressed Sparse Column (TCSC), a novel compression technique designed specifically for matrix-vector operations where the matrix as well as the input and output vectors are sparse. These operations are referred to as SpMSpV^2. TCSC compresses the nonzero columns and rows of a highly sparse matrix representing a large real-world graph. During the compression, TCSC encodes the sparsity patterns of the input and output vectors within the compressed representation of the sparse matrix itself. Consequently, it aligns the compressed indices of the input and output vectors with those of the compressed matrix columns and rows, thus eliminating the need for extra indirections when SpMSpV^2 operations access the vectors. This results in fewer cache misses, greater space efficiency and faster execution times. I evaluate TCSC's performance and show that it is more space and time efficient compared to Compressed Sparse Column (CSC) [64] and Doubly Compressed Sparse Column (DCSC) [19]. I integrate TCSC into GraphTap, a suggested linear algebra-based distributed graph analytics system and compare GraphTap against [5] and LA3 [2], two state-of-the-art linear algebra-based distributed graph analytics systems, using different dataset scales and numbers of processes.

This chapter is organized as follows. Section 3.1 provides an overview of two well-known compressed sparse matrix formats CSC and DCSC. Section 3.2 describes the motivation behind TCSC. Section 3.3 introduces TCSC. GraphTap -a new distributed graph analytics system that uses TCSC- is introduced in Section 3.4. Section 3.5 reports experimental results. Finally, Section 3.6 concludes this chapter.

3.1 Column Compressed Sparse Formats

Graphs are highly sparse structures. Many linear-algebra based graph processing systems use CSC or DCSC to store the adjacency matrix of a graph since they are both space efficient



(a) Input Graph G

Src	Dst	Wgt
1	1	0.1
1	3	0.2
1	4	0.4
2	3	0.3
2	4	0.5
4	1	0.9
5	1	0.3
5	4	0.8

(b) Adjacency List

	0	1	2	3	4	5
0						
1		.1		.2	.4	
2				.3	.5	
3						
4		.9				
5		.3			.8	

(c) Adjacency Matrix

	0	1	2	3	4	5
0						
1		.1			.9	.3
2						
3		.2	.3			
4		.4	.5			.8
5						

(d) Transposed A. M., A

Figure 5: (a) An input graph with 6 vertices and 8 edges. (b) The adjacency list where each entry is an edge from the source endpoint (Src) to a destination endpoint (Dst) with a weight (Wgt). (c) The adjacency matrix. (d) The transpose of the adjacency matrix denoted by A.

and fast to traverse [21, 115, 5, 2]. I next delve deeper into CSC and DCSC to set the stage for the proposed TCSC. A running example of the adjacency matrix from Figure 1 is replicated in Figure 5 for convenience, to explain the CSC and DCSC formats.

3.1.1 CSC Format

Figure 6 is the CSC format of A (Figure 5d). In CSC, JA is an array of column pointers, IA is an array of row numbers, and VA is an array that contains the nonzero values (or weights) in A. As such, $|JA| = n+1$, $|IA| = nnz$, and $|VA| = nnz$, where n is the number of vertices and nnz is the number of edges. The space requirement of CSC (without considering the space required for storing vectors) is $n + 2\ nnz + 1$.

The SpMV operation $y = A \oplus\cdot\otimes x$ is a widely used linear algebra operation. In this operation, A is highly sparse, and x and y vectors are uncompressed. For many applications, this operation is repeated multiple times with changes in input vector x. Although CSC is a common way of compressing A, it fundamentally lacks direct indexing of sparse input and output vectors. Figure 6 shows how an SpMV kernel runs on a CSC data structure. From this figure, the row and column indices retrieved by CSC essentially belong to the original number of rows and columns, n. With the presence of compressed vectors, CSC requires mappings from uncompressed to compressed vectors for converting JA and IA indices.

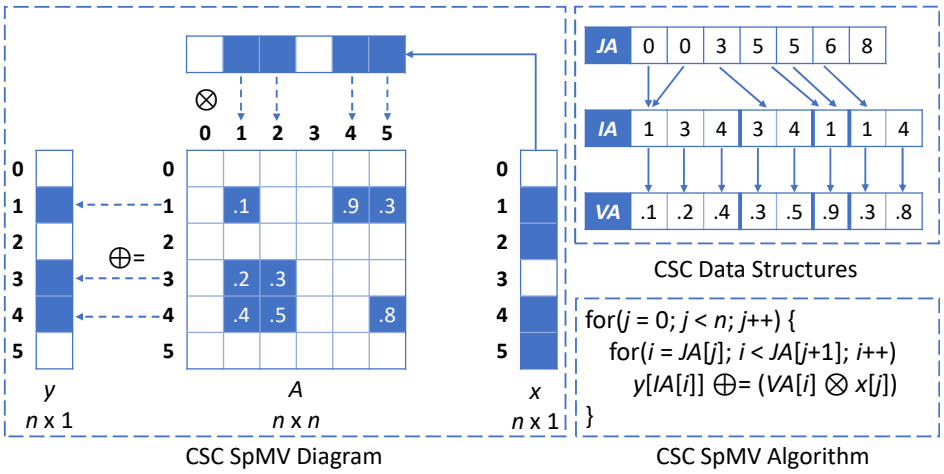


Figure 6: CSC format for Figure 5d.

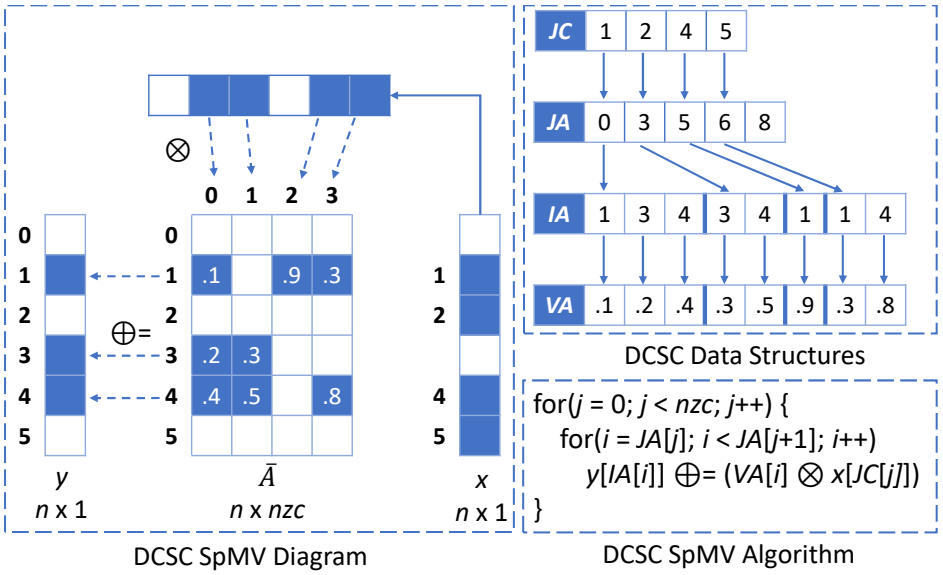


Figure 7: DCSC format for Figure 5d.

3.1.2 DCSC Format

DCSC [19] is an extension of CSC, whereby it further compresses matrix A by removing the zero (empty) columns avoiding thereby repeated elements in array JA. Since zero columns are removed, a level of indirection is required to index the retained nonzero columns. To this end, DCSC introduces an array for column indices, JC, which provides constant time access to nonzero columns (see Figure 7). In DCSC, $|JC| = nzc$, $|JA| = nzc+1$, $|IA| = nnz$, and $|VA| = nnz$, where nzc is the number of nonzero columns. Subsequently, the space re-

quirement of DCSC is $2\ nzc + \ 2nnz + 1$, without considering the space needed for vectors.

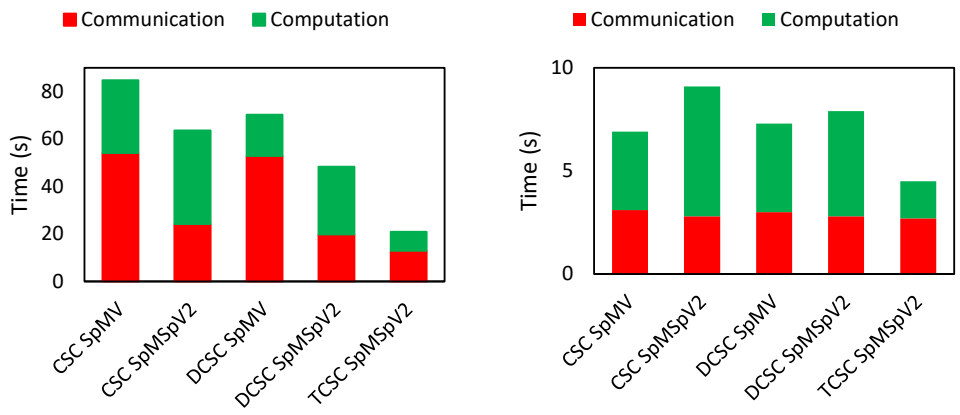
CSC can scale poorly if the number of zero columns grows significantly [129]. DCSC tackles this problem by converting A to $\bar{A}$, which does not contain zero columns. Figure 7 shows how SpMV operations are executed on top of DCSC, wherein $\bar{A}$ is multiplied by an uncompressed input vector x and the results are stored in an uncompressed output vector y. Note that Sparse Matrix - Sparse Vector (SpMSpV) operations can also be ran on top of DCSC, with x being compressed (which can be represented by (index, value) pairs) and y being uncompressed or dense [8]. Although, if x is compressed, it can be indexed through y using JC, most implementations do not exploit such an option [5, 115] in order to use the output of one SpMSpV as an input to the next SpMSpV operation [8].

3.2 Motivation

The standard CSC and DCSC runs SpMV kernels without any changes. CSC SpMV does not need any indirection to access the uncompressed input and output vectors x and y, whereas DCSC SpMV requires one indirection because it compresses the JA. Luckily, in DCSC if there are enough zero columns to remove, the cost of this indirection would not hurt the runtime.

In a distributed setting where the elements of input and output vectors are transported over the network, vector sizes become highly important because they are acting as a proxy for communication. The communication volume can be reduced by compressing the input/output vectors through removing the zero columns/rows and then adding indirection to the CSC and DCSC formats to support SpMSpV^2 kernels on the compressed vectors. To index the compressed vectors, CSC SpMSpV^2 requires two indirections (both rows and columns) and DCSC SpMSpV^2 requires only one (given it has already supported compressed column, hence it only needs one indirection for indexing rows).

To demonstrate the tradeoff between communication reduction and runtime increase due to indirection, I profile the execution of 20 iterations of PageRank (PR) on two large graphs, Twitter and Rmat29 (see Table 3 for details), running on my GraphTap distributed platform



(a) Twitter (1.46 B edges) (b) Rmat29 (8.58 B edges)

Figure 8: Comparison of different compression formats and their primitives using PR

using CSC and DCSC. As shown in Figure 8a, CSC/DCSC SpMV have roughly identical amounts of communication, whereas the computation time of DCSC SpMV is more than CSC SpMV. This is due to the DCSC SpMV's extra level of indirection. For a relatively less sparse graph like Twitter which only has a small number of empty columns, this indirection turns out to cause a computation penalty. Yet, this is not the case for a sparser graph like Rmat29 (Figure 8b), where DCSC SpMV's indirection contributes to a better runtime compared to CSC SpMV. Last, SpMV compressions are spending approximately 1/2 and 3/4 of their runtime on sending/receiving vectors, where a good portion of them are zeros.

Figure 8a shows that for Twitter graph, compressing vectors does not help CSC/DCSC SpMSpV^2 to achieve a better communication time because vectors are relatively dense. Whereas, for Rmat29 (Figure 8b) the communication time is cut in half compared to SpMV because there is a good number of zero columns/rows to remove. Finally, the computation time of SpMSpV^2 increases significantly in both Twitter and Rmat29 graphs because of the extra levels of indirections added to support SpMspV^2 kernels.

Hence, there is a trade-off between SpMV and SpMSpV^2. As communication time goes down in SpMSpV^2 due to compressing the vectors, the computation time goes up due to adding the levels of indirections (note that this trade-off is beneficial when sparsity is large and detrimental when sparsity is small). Hence, it would be desirable to compress the vectors while adding no indirection to the SpMSpV^2 kernel, which is the rationale behind TCSC.

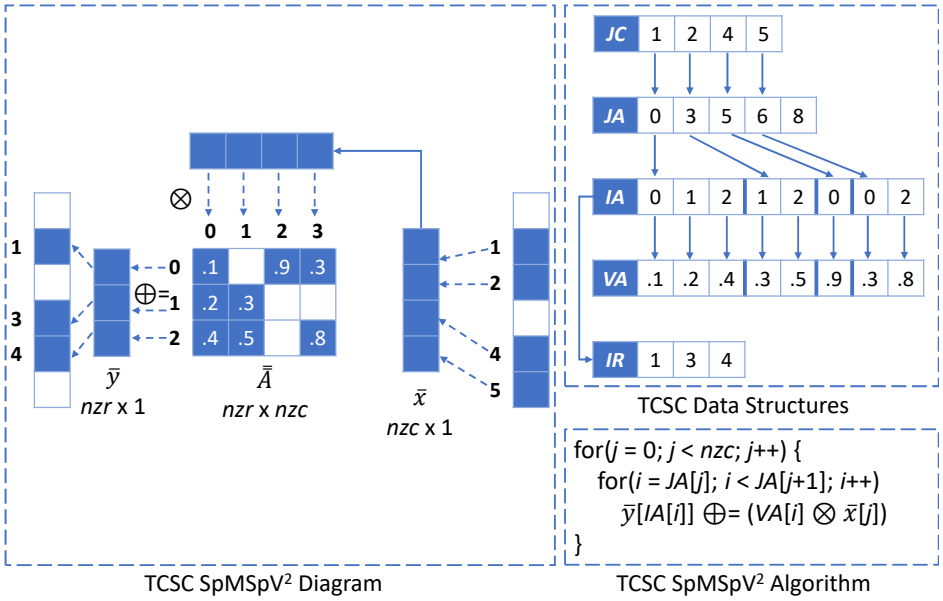


Figure 9: TCSC format for Figure 5d.

This desirable feature is shown in the last columns of Figure 8 which shows TCSC with the SpMSpV^2 kernel always decreases the computation time while decreasing (in Rmat29) or sustaining (in Twitter) the communication time.

3.3 Triply Compressed Sparse Format

In this section, I propose a simple yet highly efficient, co-compression technique called Triply Compressed Sparse Column (TCSC) (or Triply Compressed Sparse Row (TCSR) for row compressed data). By removing nonzero columns and rows of a sparse matrix, TCSC does not only store the sparse matrix in an efficient and cost-effective way, but further extends that to input and output sparse vectors. TCSC supports SpMSpV^2 operations on sparse matrix and vectors without requiring any indirection to access compressed vectors.

3.3.1 Triply Compressed Sparse Column (TCSC)

DCSC compresses matrix A by removing only its zero columns while retaining its zero and nonzero rows. TCSC capitalizes on DCSC's compression strategy via removing A's

Table 2: Space required for storing matrix, vector, and row and column indirections of different compression schemes.

	Array	CSC SpMV	DCSC SpMV	CSC SpMSpV^2	DCSC SpMSpV^2	TCSC SpMSpV^2
Matrix	JC		nzc		nzc	nzc
	JA	$n+1$	$nzc+1$	$n+1$	$nzc+1$	$nzc+1$
	IA	nnz	nnz	nnz	nnz	nnz
	VA	nnz	nnz	nnz	nnz	nnz
	IR					nzr
Vector	$x/\bar{x}$	n	n	$2\ nzc$	nzc	nzc
	$y/\bar{y}$	n	n	$2\ nzr$	$2\ nzr$	nzr
Indices	c		$nzc \rightarrow n$	$nzc \rightarrow n$		
	r			$nzr \rightarrow n$	$nzr \rightarrow n$	

zero rows as well. Like array JC for indexing nonzero columns, TCSC introduces array IR, the row indices array for indexing nonzero rows, where $|IR| = nzr$. As illustrated in Figure 9, TCSC utilizes IR to populate IA with values within the range of nonzero rows. This eliminates the problem of row indexing upon executing SpMSpV^2 operations. Figure 9 shows how an SpMSpV^2 kernel can run on top of TCSC with fully compressed matrix $\bar{\bar{A}}$ and fully compressed input and output vectors $\bar{x}$ and $\bar{y}$, without requiring any additional support from a bitvector or a list of (index, value) pairs. More precisely, by using JC and IR together, TCSC provides direct accesses to $\bar{x}$ and $\bar{y}$. Lastly, the space requirement of TCSC is $2\ nzc + nzr + 2\ nnz + 1$.

TCSC consolidates the sparsity of matrix and vectors in a co-designed data structure to enable efficient executions of SpMSpV^2 operations. CSC and DCSC can also be used to run SpMSpV^2. However, to support SpMSpV^2, CSC requires two levels of indirections for indexing compressed input and output vectors, while DCSC requires only one indirection for indexing the compressed output vector. Last, given these extra levels of indirections are already incorporated in TCSC data structure, TCSC is the right choice to execute SpMSpV^2 operations.

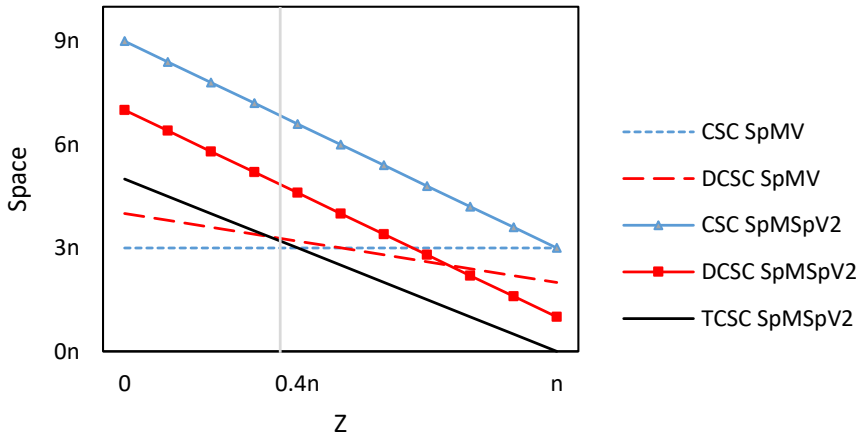


Figure 10: Space of different compressions using (3.1).

3.3.2 Comparison of Space Requirements

Table 2 shows a comparison between different compression techniques. For SpMSpV^2 operations, CSC requires using data structures like two lists of (index, value) pairs for input and output vectors. In addition, it needs to store metadata for column and row indirections. Therefore, its space requirement evaluates to $3\,n+3\,nzc+3\,nzr+2\,nnz+1$. DCSC requires maintaining information on input and output vectors and metadata for row indirection. Thus, its space requirement for SpMSpV^2 operations is $n+3\,nzc+3\,nzr+2\,nnz+1$. TCSC total space requirement is $3\,nzc+2\,nzr+2\,nnz+1$.

In comparing space requirements for SpMSpV^2 operations, TCSC demands the least space due to uniquely addressing the sparsity of vectors in conjunction with the sparsity of the matrix. It can be proved that under certain conditions TCSC can save space when at least 40% of rows/columns of the matrix are empty compared to CSC and DCSC with SpMV (see Figure 10; more on this shortly). Alongside space savings, TCSC provides faster SpMSpV^2 operations because: 1) it averts two levels of indirections compared to CSC and one level of indirection compared to DCSC, 2) it requires sending/receiving only values of compressed vectors (especially in distributed settings) without exchanging any metadata since it retains internally the nonzero indices, 3) it results in smaller vectors, which can potentially fit in cache, and 4) it exhibits sequential access patterns on the input vector (like DCSC), thus exploiting more cache locality (as compared to CSC).

Given the information reported in Table 2, a relaxed space formulas for all the compression schemes can be derived by ignoring the IA and VA arrays and the plus one in JA array, which are equivalent across all the schemes. Thus, the term $2\ nnz + 1$ can be eliminated. Furthermore, assuming $nzc \approx nzr \approx nz$, then $nz = n - z$, where nz is the number of nonzero elements and z is the number of zeros agnostic to rows and columns. Finally, by removing $2\ nnz + 1$ and, subsequently, substituting nzc and nzr with $n - z$, the following approximate space formulas are obtained:

$$\begin{aligned}
&\text{CSC SpMV} \rightarrow 3\ n \\
&\text{DCSC SpMV} \rightarrow 2\ n + 2\ (n - z) = 4\ n - 2\ z \\
&\text{CSC SpMSpV}^2 \rightarrow 3\ n + 3\ (n - z) + 3\ (n - z) = 9\ n - 6\ z \\
&\text{DCSC SpMSpV}^2 \rightarrow n + 3\ (n - z) + 3\ (n - z) = 7\ n - 6\ z \\
&\text{TCSC SpMSpV}^2 \rightarrow 3\ (n - z) + 2\ (n - z) = 5\ n - 5\ z
\end{aligned} \tag{3.1}$$

By varying the value of z in equations (3.1) over the range $[0, n]$, the space of each compression can be computed in terms of n. As demonstrated in Figure 10, from $z = 0.4\ n$ onward (marked by the vertical gray line), TCSC will require less space as opposed to other schemes. Section 3.5 presents experimental results that corroborate this observation.

3.3.3 Translating Graph Algorithms onto SpMSpV^2 Operations

Leveraging the duality between graphs and sparse matrices, many graph theory operations can be mapped onto certain linear algebra primitives on sparse matrices [21]. As a brand new yet simple linear algebra primitive, SpMSpV^2 primitive, $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$ can be formalized as follows:

- $\bar{\bar{A}}$ is the $nzr \times nzc$ sparse matrix with nnz entries (edges), where nzr and nzc are the number of nonzero rows and columns, respectively.
- $\bar{x}$ is the $nzc \times 1$ sparse input vector with nzc entries (columns), which is multiplied in $\bar{\bar{A}}$ using the multiplication and addition operators.
- $\bar{y}$ is the $nzr \times 1$ sparse output vector with nzr entries (rows), which stores the results of multiplying $\bar{\bar{A}}$ and $\bar{x}$.

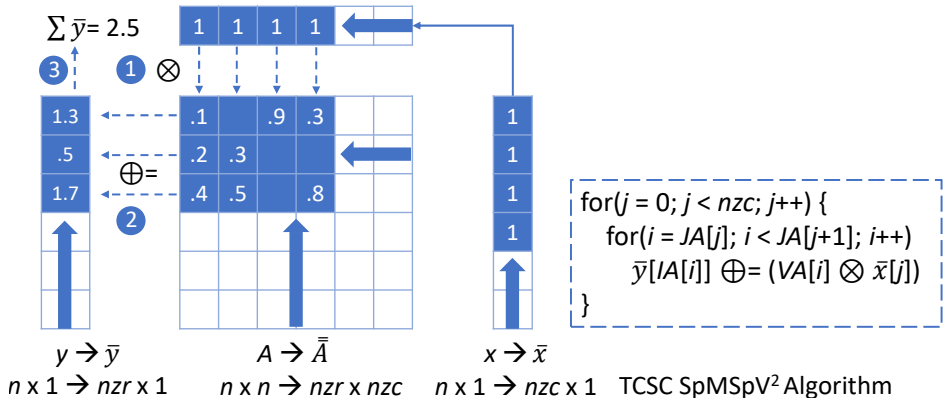
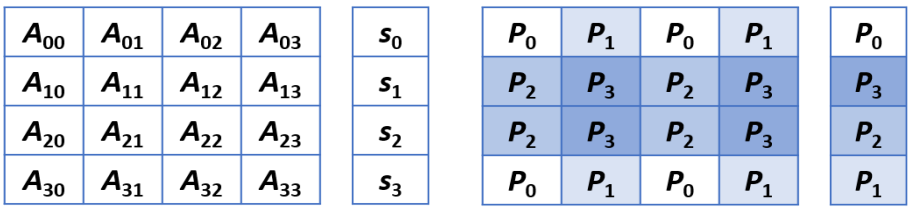


Figure 11: Calculating weighted outgoing degree of Figure 5d.

- $\oplus.\otimes$ is a semiring equipped with $(+, \times)$ operators.

SpMSpV^2 requires a way of encoding the sparsity for both $\bar{x}$ and $\bar{y}$ vectors. Previous works have used bitvectors [115, 5] or lists of (index, value) pairs [2] to encode this information. In contrast, TCSC coalesces this information in the compressed sparse matrix format and assumes that sparse input and output vectors are of sizes nzc and nzr, respectively. Hence, TCSC cuts down memory usage while eliminating unnecessary computation resulted from compressing the vectors.

To exemplify, consider the *weighted degree calculation* which calculates the outgoing degree of a graph ponderated by the weight of each edge (Figure 11). This problem can be solved via multiplying the outgoing edges of each vertex by one and summing up the results. Using SpMSpV^2 operations, first $\bar{x}$ is initialized with ones. Second, the weighted outgoing degree of each vertex is calculated by multiplying each entry of $\bar{x}$ to its corresponding column of $\bar{\bar{A}}$. Third, the result of each row is stored in the respective entry of $\bar{y}$, which will eventually hold the weighted outgoing degrees of all vertices.



(a) (b)

Figure 12: (a) Partitioning a matrix into a p x p grid of tiles and a vector into p segments where p=4 is the number of processes. (b) Assigning processes to tiles and segments where.

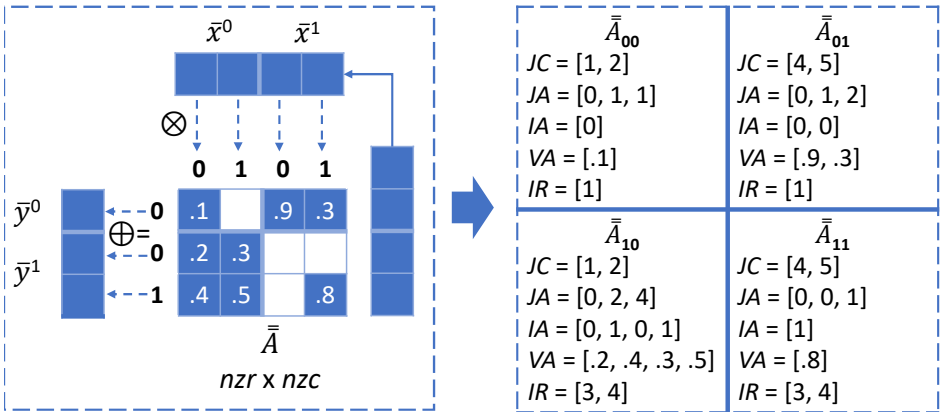


Figure 13: Figure 5d matrix partitioned into four TCSC tiles.

3.4 GraphTap: Distributed Graph Analytics using Triply Compressed Sparse Format

This section introduces GraphTap, a new distributed system for scalable graph analytics that features a TCSC-based SpMSpV^2 system mated with a vertex-centric programming interface. As such, GraphTap can execute any user-defined vertex program on any input graph. This is done in two steps. First, GraphTap loads and partitions the input graph into TCSC tiles distributed across multiple processes. Next, it executes the user's vertex program in an iterative fashion via its distributed SpMSpV^2 core. The followings describe these steps in details.

3.4.1 Matrix Partitioning

GraphTap can read graphs given in an edge-list format. It loads edges into an adjacency matrix representation that is partitioned in two dimensions and distributed for scalability [17, 5, 2, 47]. To elaborate, given p processes, GraphTap partitions the matrix into p^2 *tiles* and any associated vector into p *segments*, as exemplified in Figure 12a.

GraphTap assigns tiles and segments to processes while accounting for both load balancing and locality [2]. As Figure 12b shows, each process is assigned p tiles and one of p vector segments. In particular, the process owning the i^{th} diagonal tile, A_{ii}, also owns the i^{th} vector segment, s_i. This process is called the *leader* of the i^{th} *row group* (i.e., the set of processes that own tiles in the i^{th} row) and *column group* (i.e., the set of processes that own tiles in the i^{th} column). During distributed SpMSpV^2 execution, each leader communicates with its row and column group *followers* (members) via MPI. For example, in Figure 12b, process P_0 owns tiles A_{00}, A_{02}, A_{30}, and A_{32}. Also, P_0 is the leader of the first row and column groups; thus, P_1 is P_0's follower in the first row group and P_2 is P_0's follower in the first column group.

GraphTap stores each tile using the TCSC format. The compressed height of any given tile, $\bar{\bar{A}}_{ij}$, is the number of nonzero rows across the entire i^{th} row of tiles. Similarly, the compressed width of any given tile, $\bar{\bar{A}}_{ij}$, is the number of nonzero columns across the entire j^{th} column of tiles. Moreover, in order to eliminate indirections during SpMSpV^2, the compressed sizes of the i^{th} input (or output) vector segments are equal to the compressed width (or height) of the i^{th} column (or row) of tiles. For example, in Figure 13, tiles $\bar{\bar{A}}_{00}$ and $\bar{\bar{A}}_{10}$ both have a compressed width of two, as does input segment $\bar{x}^0$. Similarly, tiles $\bar{\bar{A}}_{00}$ and $\bar{\bar{A}}_{01}$ both have a compressed height of one, as does output segment $\bar{y}^0$.

3.4.2 Vertex Program Execution

Similar to other recent graph analytics systems [2, 5, 115], GraphTap translates a user-defined vertex-centric program into iteratively-executed SpMSpV^2 operations. Like these systems, GraphTap applies a variant of the *Gather*, *Apply*, *Scatter* (GAS) model [50]. To be more precise, GraphTap involves three method calls per SpMSpV^2 iteration: *Scatter-Gather*,

Combine, and *Apply*, which will be elaborated upon shortly.

In order to map a vertex-centric program to its SpMSpV^2 system, GraphTap maintains – in addition to the compressed input and output vectors, $\bar{x}$ and $\bar{y}$ – a *state* vector, v, which stores vertex states. The state vector is not compressed, and its size equals the number of vertices, because all vertices have states, even if some states may remain unchanged. The state vector is partitioned into p segments, each assigned to its corresponding leader process. Thus, each process initializes the states of its own vertices. Thereafter, GraphTap launches its iterative SpMSpV^2 execution. Per iteration, each process calls the following methods.

3.4.2.1 Scatter-Gather To begin an iteration, each i^{th} leader, in parallel, prepares its new $\bar{x}^i$ and *scatters* it to its column group followers. $\bar{x}^i$ is essentially an interpolation of the old state, v^i (i.e., resulting from the previous iteration).

Consequently, TCSC offers the following advantages during Scatter-Gather: 1) Since $|\bar{x}| = nzc < |x| = n$, less communication is required (per column group). 2) Given that TCSC already incorporates the sparsity information inside its data structures, there is no need to send the indices of the nonzero elements. Therefore, the communication volume is only limited to sending the values themselves, which is less compared to sending a list of (index, value) pairs in [5, 2]. 3) When calculating the new $\bar{x}$ from v, TCSC's JC array efficiently enables direct indexing on both $\bar{x}$ and v (i.e., without requiring any extra levels of indirection).

3.4.2.2 Combine After the scattered $\bar{x}$ is gathered at all processes, each process starts processing the tiles that it owns in a row-wise fashion. For each tile, T_{ij}, in the i^{th} row, the SpMSpV^2 operation is called on its TCSC value array, VA, and the $\bar{x}^j$ belonging to the j^{th} column group. The result is *combined* (accumulated) locally in $\bar{y}^i$, which is indexed directly using the IA array. After processing all its tiles belonging to the i^{th} row, each follower sends its $\bar{y}^i$ to its row group leader, which combines it into its own $\bar{y}^i$. Given GraphTap uses asynchronous communication, leaders/followers post their receives/sends and move on to their next row of tiles.

Thus, TCSC offers the following advantages during Combine: 1) No indirections are

needed while running SpMSpV^2 operations on $\bar{x}$, VA, or $\bar{y}$ (for storing the results). This is because, $\forall\, T_{ij}$, $|\bar{x}^j| = |JA|$ and $|\bar{y}^i| = |IA|$. 2) Since $|\bar{y}| = nzr < |y| = n$, less communication is required (per row group). 3) When followers send $\bar{y}^i$s to their leader, only the actual values are sent without their indices, further reducing communication volume. 4) Asynchronous communication allows GraphTap to overlap communication with computation.

3.4.2.3 Apply To complete an iteration, each i^{th} leader, in parallel, waits until its $\bar{y}^i$ is finalized, and then uses it to update its v^i (to be used in the next iteration). Although $|\bar{y}| = nzr \neq |v| = n$, TCSC's IR array circumvents an undesired indirection when computing v from $\bar{y}$ since it contains the original row ids of the nonzero indices of v.

GraphTap continues iterating until v converges or a specified maximum number of iterations is reached.

3.4.2.4 Activity Filtering and Computation Filtering Graph applications may be classified as *stationary* or *non-stationary* [5, 2]. In a stationary application, all vertices remain active over all iterations. In a non-stationary application, only a subset of vertices is active during each iteration and this subset can change dynamically. GraphTap skips the communication and computation of inactive vertices in non-stationary applications. *Activity filtering* is implemented by communicating (index, value) pairs of active vertices only.

For directed graphs, it is possible to make the SpMV more efficient via *computation filtering* [2]. This firstly requires classifying vertices into regular vertices (have both incoming and outgoing edges), source vertices (have only outgoing edges), sink vertices (have only incoming edges), and isolated vertices (have no edges). Subsequently, processing only regular and source vertices in the first iteration, only regular vertices in the middle iterations, and only regular and sink vertices in the final iteration. Computation filtering is implemented for stationary applications on directed graphs only.

3.5 Results

Experiments are conducted in two settings: single node processing and distributed processing, both written in C/C++. The single node implementation is a single thread PageRank application which basically compares CSC, DCSC, and TCSC SpMSpV[2]. The distributed implementation uses GraphTap[1], the proposed distributed graph analytics system which utilizes TCSC as its default compression technique and MPI for both inter and intra-node communication. GraphTap's experiments include both weak scaling comparison where graph size is scaled alongside the cluster size, and strong scaling where graph size is fixed, and the cluster size is varied.

3.5.1 Experimental Setup

3.5.1.1 Hardware and Software Configurations Experiments are ran on a cluster of machines that uses Slurm workload manager for batch job queuing [108]. Intel MPI [58] is used to compile programs on the cluster. Moreover, for single node experiments, a machine with 12-core Xeon processor (@ 3.40 GHz speed) and 512 GB RAM is used. For the distributed experiments, a sub-cluster of 32 machines each with 28-core Broadwell Processor (@ 2.60 GHz speed), 192 GB RAM, and Intel Omni-path network (10 Gbps transfer speed) is used. At largest scale, all these 32 machines are utilized and 16 processes (cores) per machine are launched without over subscription of cores (512 cores in total). Finally, any data point reported here is the average of multiple individual runs.

3.5.1.2 Counterpart Systems GraphTap has been tested against GraphPad [5], a linear algebra-based system developed by Intel, and LA3 [2], a linear-algebra based system with sophisticated communication and computation optimizations. After a careful assessment, I noticed that GraphPad works best when launched with two threads per MPI process and LA3 with one thread per MPI process (without multithreading). Furthermore, 16 cores are allocated per machine and thus GraphPad is launched with 8 processes and two threads

[1] GraphTap source code is online at https://github.com/hmofrad/GraphTap

(cores) per process, and LA3 and GraphTap are launched with 16 processes (cores) per machine.

3.5.1.3 Graph Datasets Table 3 shows the collection of six real-world graphs and five synthesized graphs alongside their characteristics and the number of processes allocated to process them. This collection includes multiple web crawls and social network from LAW [16], and RMAT 26 - 30 graphs from the Graph 500 challenge [25].

3.5.1.4 Graph Applications To evaluate TCSC, two types of graph applications are implemented: 1) stationary applications including Degree, and PageRank (PR) on unweighted directed graphs, and 2) non-stationary applications including Single Source Shortest Path (SSSP) on weighted directed graphs, and Breadth First Search (BFS) and Connected Component (CC) on unweighted undirected graphs. Note that similar to the setting used in [2, 5], I ran PR for 20 iterations and SSSP, BFS, and CC until convergence and report the average execution.

3.5.2 Single Node Results

To experimentally measure the performance of TCSC, I implemented a single thread PageRank application and reported its space, number of L1 cache misses, and speedup in Figure 14. PageRank is chosen as it is a compute-intensive application and the focus in this section is more on identifying the computational characteristics of TCSC.

3.5.2.1 Space Utilization Figure 14a shows the space utilization measured for different compressions. Similar to the TCSC space analysis (Section 3.3.2), only the space required for vectors and indirections is reported for this comparison as the amount of storage required for storing the graph edges is the same across all compressions (see Table 2).

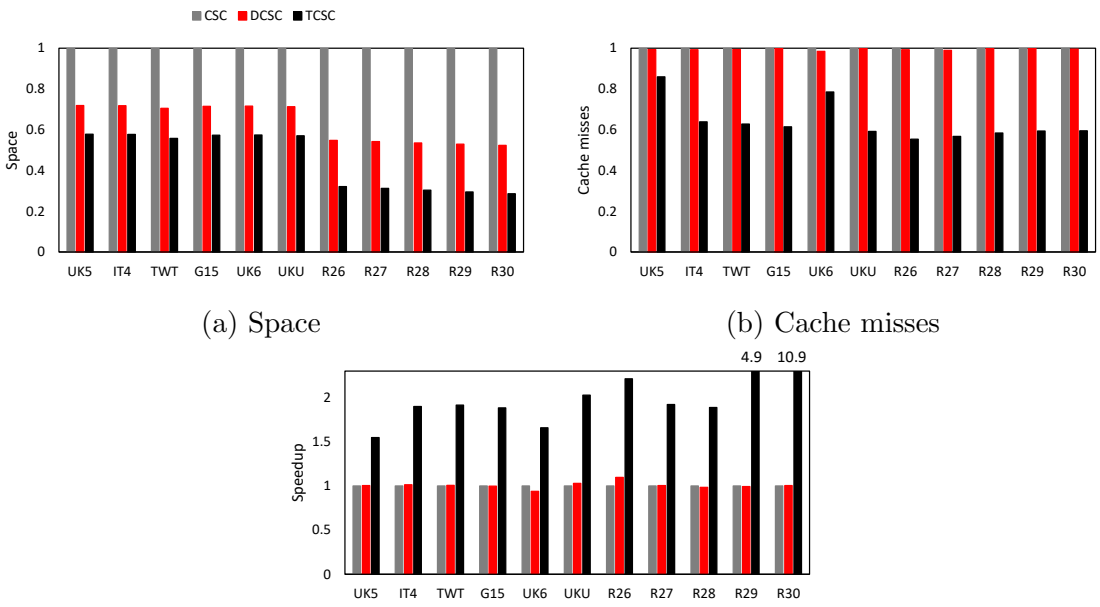
From Figure 14a, note that CSC and DCSC have approximately similar space utilization and TCSC has the least space requirement in both real-world and synthetic graphs. Compared to CSC, on average TCSC requires 45% and 70% less space in real-world and synthetic

Table 3: Datasets used for experiments. *Zc* and *Zr* are the percentage of zero columns and rows. T is the type (including web crawl, social network and synthetic graphs). N is the number of machines used to process the graph.

Graph	$\|V\|$	$\|E\|$	Z_c	Z_r	T	N
UK'05 (UK5) [16]	39.4 M	0.93 B	0	0.12	Web	4
IT'04 (IT4) [16]	41.2 M	1.15 B	0	0.13	Web	4
Twitter (TWT) [16]	41.6 M	1.46 B	0.09	0.14	Soc	8
GSH'15 (G15) [16]	68.6 M	1.80 B	0	0.19	Web	8
UK'06 (UK6) [16]	80.6 M	2.48 B	0.01	0.14	Web	16
UK Union (UKU) [16]	133 M	5.50 B	0.05	0.09	Web	24
Rmat26 (R26) [25]	67.1 M	1.07 B	0.55	0.72	Syn	4
Rmat27 (R27) [25]	134 M	2.14 B	0.57	0.73	Syn	8
Rmat28 (R28) [25]	268 M	4.29 B	0.59	0.74	Syn	16
Rmat29 (R29) [25]	536 M	8.58 B	0.61	0.75	Syn	24
Rmat30 (R30) [25]	1.07 B	17.1 B	0.62	0.76	Syn	32

graphs. Also, compared to DCSC, on average TCSC requires 15% and 25% less space in real-world and synthetic graphs. This space efficiency roots in the indexing algorithm of TCSC where it stores the sparsity of vectors while constructing the compressed matrix data structure by renumbering its column and row indices and removing zero (empty) columns and rows. This successfully allows TCSC to trivially expand or compress the input and output vectors and at the same time consumes the least space.

3.5.2.2 Cache Analysis CPU performance counters are used to collect data on L1 cache misses. Figure 14b shows the number of cache misses of different compressions. Comparing CSC and DCSC with TCSC, on average TCSC has 20% to 40% less cache misses across all real-world and synthetic graphs. TCSC is a cache friendly compression since it can access the compressed input and output vectors without requiring any level of indirection while avoid



(a) Space

(b) Cache misses

(c) Speedup

Figure 14: Normalized space, speedup, and cache misses of different compression techniques on a single node for PR with CSC as baseline.

trashing the L1 cache. TCSC sequentially indexes the input vector. This avoids unnecessary cache invalidations of the input vector and provides more cache locality. Moreover, TCSC can access the output vector with no level of indirection compared to CSC and DCSC, providing faster access to output vector entries. Last, given the compressed input and output vectors are essentially smaller than the original SpMV vectors, they can possibly fit in L2 cache which further yields better cache utilization.

3.5.2.3 Time Analysis Figure 14c compares the speedup for different compressions. From this figure, compared to CSC and DCSC, TCSC is up to 2.2× and 11× faster in real-world and synthetic graphs, respectively. This performance gain is mainly due to the direct indexing algorithm of TCSC which offers a better cache locality. CSC and DCSC underperform compared to TCSC because they suffer from access indirections and poor cache locality.

In Figure 14c, DCSC is slightly faster than CSC on average because it collapses the nonzero columns and skips the computation for nonzero columns. Furthermore, TCSC is faster than both CSC and DCSC because it additionally collapses the nonzero rows which further reduces the chances of L2 cache and memory thrashing. Moving to larger scales synthetic graphs such as RMAT30, the cache thrashing effect becomes more prominent and TCSC is 11× faster than CSC and DCSC.

There are two levels of indirection while running the SpMSpV^2 kernel: 1) indirection used for the input vector while accessing column data using pairs of (index, value), and 2) indirection used for sparse output vector while writing the result of executing the operation. Although CSC and DCSC are adapted to work with sparse vectors, CSC requires both levels of indirections and DCSC requires the latter one. TCSC, on the other hand does not need these levels of indirections because for the former one, like DCSC the number of columns in the sparse matrix are aligned with the size of input vector. For the latter indirection, since TCSC's row indices array is populated using values derived from the number of nonzero rows, the row indices stored in the compressed matrix are essentially able to directly index the output vector.

3.5.3 Distributed Processing Results

This section discusses the experimental results of GraphTap. In the first and second experiments different compression techniques implemented inside GraphTap are compared and in the third experiment, GraphTap is compared with GraphPad [5] and LA3 [2], two state-of-the-art linear algebra-based graph analytics systems. The graphs and cluster sizes used for these experiments are reported in Table 3.

3.5.3.1 Speedup Comparison of CSC, DCSC, and TCSC in GraphTap CSC, DCSC, and TCSC SpMSpV^2 are implemented in GraphTap and benchmarked them using PR (a stationary application). As shown in Figure 15, on real-world and synthetic graphs, CSC and DCSC perform comparatively with DCSC performing slightly better. Also, TCSC performs the best compared to CSC and DCSC with up 3.5× and 5.7× speedup in real-

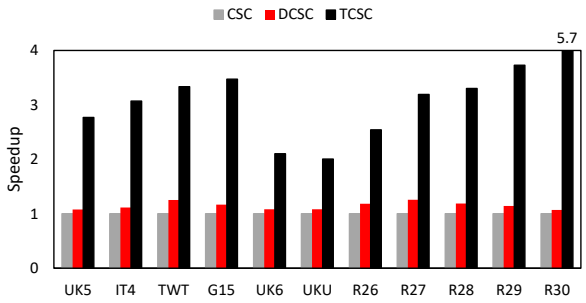
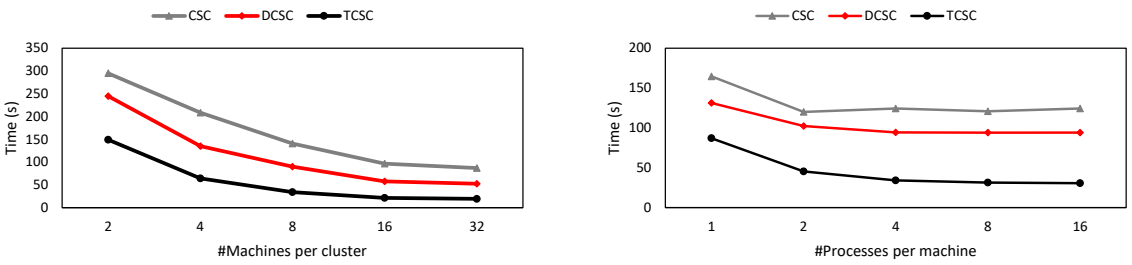


Figure 15: Normalized speedup of compressions on GraphTap for PR with CSC as baseline

wold and synthetic graphs, respectively. From the results, CSC and DCSC are not scaling well compared to TCSC as while solving PR they become slower as dataset size increases (especially in synthetics). TCSC on the other hand is scalable because as dataset size increases, the runtime also improves in both real-world and synthetic graphs. This is because TCSC not only compresses vectors leading to less communication, but also has a better indexing algorithm, leading to more efficient computation.

3.5.3.2 Scalability Comparison of CSC, DCSC, and TCSC in GraphTap Figure 16a shows the results of *cluster scalability test*. In this experiment, the number of processes per machine is kept to 16 but change the number of machines from 2 to 32 and run PR on R29. Here, TCSC improves the runtime as more machines (or processes) are added to solve the problem because TCSC's communication volume is smaller compared to CSC and DCSC. Thus, increasing the communication volume by having more machines does not hurt its performance. It is worth noting that in a distributed computing scenario, communication volume is a factor of intermediate vectors. Given TCSC efficiently compresses the vectors, it is being resilient to the size of cluster.

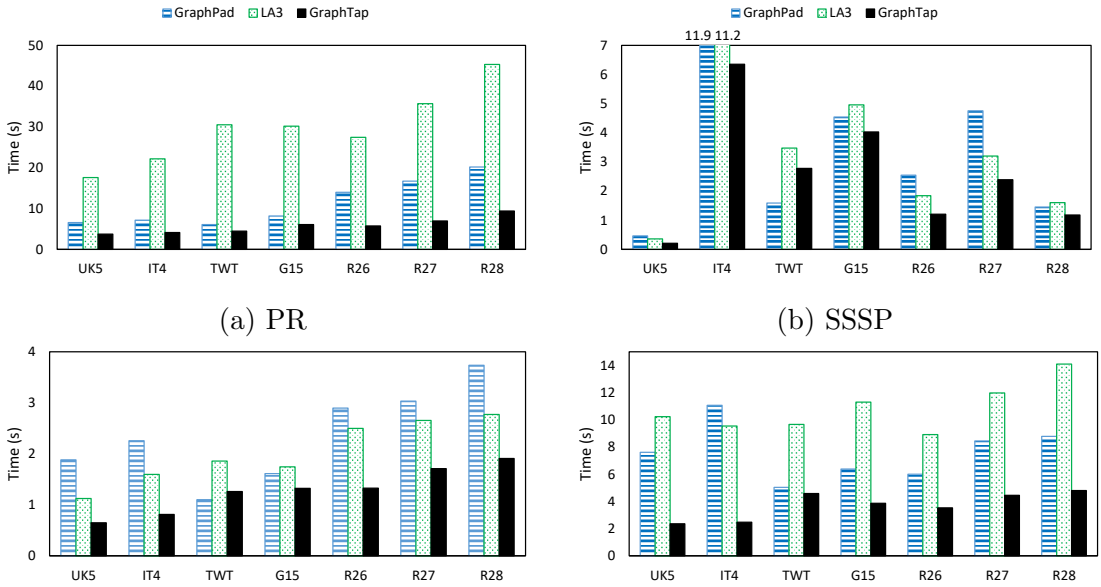
Figure 16b shows the *process scalability test* of different compressions. In this experiment, PR is ran on R30 using 32 machines while changing the number of processes per machine from 1 to 16. From this figure, TCSC is scalable because it can efficiently harvest the added processes to achieve a better runtime while maintaining a decent gap with other compressions. Moreover, CSC is not scalable because it achieves worse or comparable runtimes with more



(a) Cluster scalability test using PR on R29 with 8.58 B edges.

(b) Process scalability test using PR on R30 with 17.1 B edges.

Figure 16: Scalability tests for different compressions.



(a) PR

(b) SSSP

(c) BFS

(d) CC

Figure 17: Runtime of GraphPad, LA3, and GraphTap

than two processes, whereas TCSC even with 16 processes can still improve the runtime.

3.5.4 Runtime Comparison of GraphPad, LA3, and GraphTap

In this experiment, GraphTap is compared with GraphPad and LA3 using PR, SSSP, BFS, and CC applications on selected datasets from Table 3. GraphPad [5] uses DCSC for compressing the sparse matrix and bitvectors for representing the sparse vectors. Similarly,

LA3 [2] uses DCSC for sparse matrices, but uses lists of (index, value) pairs for representing sparse vectors. On the other hand, GraphTap uses TCSC that compress both matrix and vectors simultaneously and uses lists of (index, value) pairs for representing sparse vectors.

Figure 17a reports the results for PR. Based on this figure, GraphTap is up to 1.5× faster than GraphPad and 7× faster than LA3 in real-world datasets. Also, GraphTap is up to 2× faster than GraphPad and 4× faster than LA3 in synthetic datasets. LA3 uses aggressive communication optimizations that tailor the communication per tile while sending the input vectors. The overhead of this optimization becomes a bottleneck when running on a cluster with a fast communication infrastructure. Specifically, LA3 spends a significant amount of time on constructing these tailored input vectors. GraphTap, on the other hand, tailors input vectors for each column group of tiles so that it can skip the overhead of constructing individual input vectors per a receiver process, while still efficiently utilizing the network bandwidth. Also, GraphPad is better than LA3 because of its efficient communication.

Figure 17b and Figure 17c show the results for the non-stationary applications SSSP and BFS. From these figures, GraphTap is 2–3× faster than GraphPad and LA3. SSSP runs on weighted directed graphs, it starts from a source vertex and converges when it finds the shortest path to all vertices inside the connected component the source vertex is belonged to. Clearly, executing vertices which are not at the same component with source is unnecessary. Thus, activity filtering removes them from the main loop of computation. Moreover, vertices that have converged already are also factored out of the computation. For non-stationary applications, activity filtering significantly reduces the volume of communication compared to stationary applications like PR. Therefore, having less communication is the reason that LA3 performs better than GraphPad while running BFS on synthetic graphs. Also, GraphTap performs worse than GraphPad in SSSP and BFS on TWT; this is because TWT is among the relatively high-density real-world graphs where there is a small number of zero rows and columns to filter for TCSC.

Figure 17d shows the result for CC. GraphTap is $1.2 - 4.5\times$ faster than GraphPad and $2 - 4\times$ faster than LA3 in real-world graphs. Also, GraphTap is 2× faster than GraphPad and 3× faster than LA3 in synthetic graphs. From Figure 17d, GraphPad performs better than LA3 because CC deals with significant amount of messaging to identify the connected

components and the communication optimizations of LA3 are extremely expensive for such an application. Also, comparing GraphTap's TCSC with DCSC used in GraphPad, DCSC uses a bitvector to locate the nonzero entries of output vectors, whereas TCSC can directly index the output vectors.

Last, in Figure 17 on average GraphTap is $2-4\times$ faster than others on all scales which is due to the proposed TCSC format. Moreover, GraphTap scales better compared to GraphPad and LA3 because while adding more processes for larger graphs, it can efficiently utilize the additional processes with a negligible increase in runtime (this trend is more visible in Rmat synthesized graphs).

3.5.5 Discussion of Results

TCSC has significant space and indexing advantages over CSC and DCSC. Moreover, GraphTap which uses TCSC as its core compression format, outperforms GraphPad and LA3 distributed systems with DCSC compression scheme. The following are a summary of TCSC and GraphTap results:

1. TCSC is more cache friendly than CSC and DCSC. The input and output vectors are intrinsically smaller for TCSC and are accessed directly without indirection. The smaller vector sizes and the locality of access patterns cause fewer cache misses and less cache pollution in TCSC.
2. GraphTap communication volume is less than GraphPad and LA3 because the sizes of its vectors are equal to the number of nonzero columns/rows. There is no need for an auxiliary mechanism to index input and output vectors as they are aligned to the number of nonzero columns/rows. Therefore, input vectors are scattered without any change in their size and partial output vectors are aggregated without requiring any extra indexing metadata.
3. The proposed triple compression can be applied to row major compression resulting in a TCSR scheme. However, TCSC is picked for the same reason CSC and DCSC are preferred over CSR and DCSR. Specifically, in column major compressions, like CSC, DCSC or TCSC, access to the input vector is sequential and infrequent and access to the

output vector is random and frequent providing better cache locality for input vectors. This flips for row major compressions like CSR, DCSR and TCSR. In non-stationary applications, given that input vector only carries information about active vertices, a column compression can immediately locate the active columns and runs the SpMV kernel, whereas in row compression, the algorithm first needs to scan all rows and locates the active vertices and then runs the SpMV which requires more effort. Thus, column compressions have been shown to perform better for graph applications.

4. TCSC is a scalable compression format. It has been used to process big graphs as large as 17.1 B edge on up 32 machines with 16 processes per machine (512 processes in total). From the experiments, by adding more machines per cluster or more processes per machine, TCSC can harvest additional processes efficiently because it compresses empty rows/columns and reduces the problem space.

3.6 Conclusion

In this chapter, I propose Triply Compressed Sparse Column (TCSC), a novel compression technique which leads to efficient Sparse Matrix - Sparse input and output Vectors ($SpMSpV^2$) operations. TCSC logically compresses both columns and rows of a sparse matrix and hence integrates the sparsity of input and output vectors within the sparse matrix. The performance of TCSC on real-world and synthetic graphs with different sizes is analyzed and demonstrated that TCSC has less space requirement while offering up to 11× speedup compared to common CSC and DCSC. TCSC is implemented in GraphTap, a new linear algebra-based distributed graph analytics system introduced in this chapter. GraphTap is compared with GraphPad and LA3, two state-of-the-art linear algebra-based distributed graph analytics systems on different graph sizes and numbers of machines and cores. Experiments showed that GraphTap is up to 7× faster than these distributed systems due to its efficient sparse matrix compression format, faster $SpMSpV^2$ algorithm, and smaller communication volume.

4.0 Graphite: A NUMA-aware HPC System for Graph Analytics Based on a new MPI ∗ X Parallelism Model

In the previous chapter, the details of a new compressed sparse data structure called TCSC which saves in memory requirement, and computation and communication time were discussed. TCSC is utilized in a MPI + X distributed system called Graphite and showed promising results. However, having an efficient data structure is not enough to achieve linear scalability as the scalability of a distributed system is limited to the its parallelism model. In this chapter, I propose a new parallelism model denoted as $MPI * X$ and suggest a linear algebra-based graph analytics system, namely, Graphite, which effectively employs this new parallelism model. The MPI ∗ X promotes *thread-based partitioning* to distribute computation and communication across threads on a cluster of machines, while eliminating the need for unnecessary thread synchronizations. Consequently, it contrasts with the traditional *MPI + X parallelism model*, which utilizes *process-based partitioning* to distribute data among processes as a way to *scale out* on a cluster of machines (the MPI part), then splits each partition into subpartitions among the threads of each process as a method to *scale up* within a machine (the X part). Besides adopting MPI ∗ X, Graphite is NUMA-aware. In particular, it assigns threads to partitions in a way that exploits CPU and memory affinity, alongside leveraging faster MPI shared memory transport. Moreover, it adopts a variant of the popular GAS (Gather, Apply, and Scatter) computing model, thus decoupling the computation of partitions from the communication of partial results. Lastly, it supports thread-level asynchrony, which does not only overlap the computation with communication, but further interleaves multiple communications. Graphite is compared against GraphPad, Gemini, and LA3 graph analytics systems in an HPC setting using different graph applications. Results show that Graphite is roughly up to 3× faster than these systems.

The rest of this chapter is organized as follows. Section 4.1 discusses the classical 2D-process-based matrix partitioning and placement approaches. Section 4.2 proposes the new 2D-thread-based matrix partitioning and placement paradigms. Section 4.3 discusses NUMA-aware thread placement and Section 4.4 summarizes features of the new MPI ∗ X

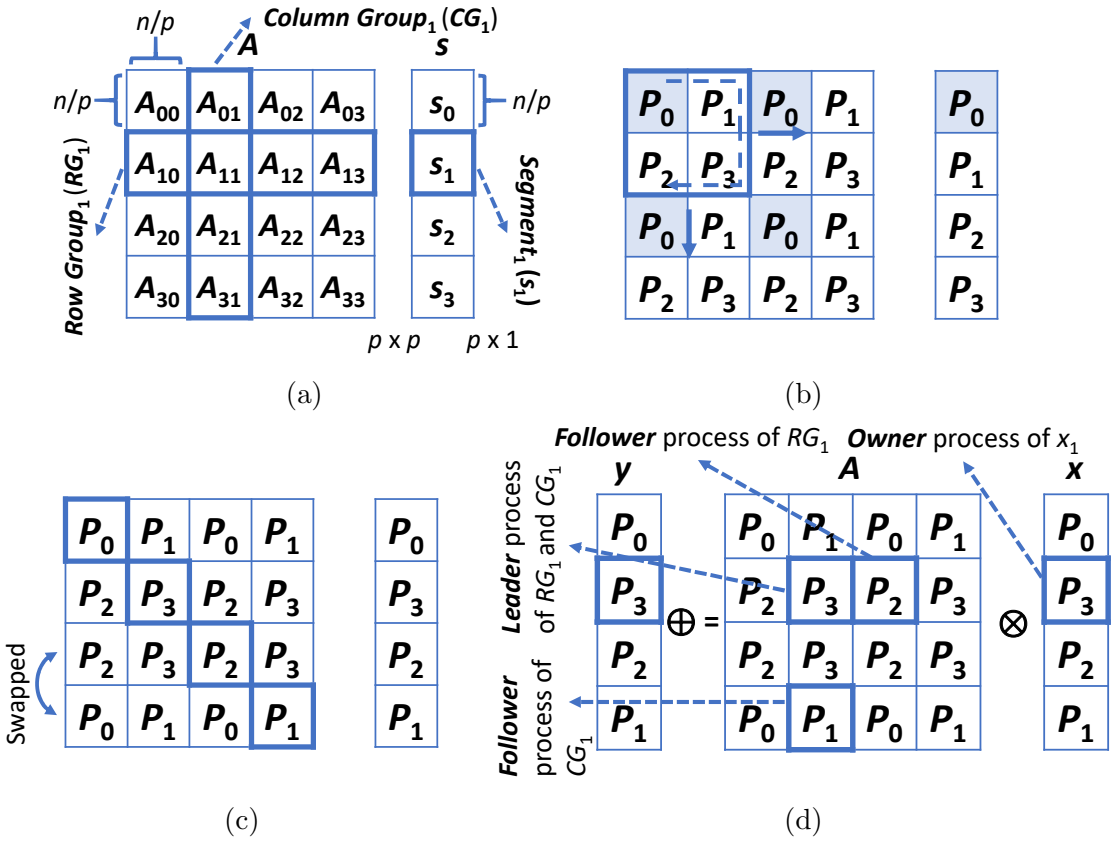


Figure 18: Matrix and vector 2D layouts ($p = 4$). (a) 2D-process-based partitioning of matrix and vector, (b) 2D-Cyclic process placement (e.g. the shaded tiles are assigned to $P0$), (c) 2D-Staggered process placement, and (d) 2D-Staggered leader/follower configuration for distributed SpMV.

model. Section 4.5 puts it altogether and introduces Graphite. Results are reported in Section 4.6 and Section 4.7 concludes with some remarks.

4.1 2D-process-based Matrix Tiling & Placement

A graph can be represented by an adjacency matrix, in which an edge is denoted by a non-zero element in the matrix. Many real-world graphs consist of billions of vertices and tens of billions of edges. Typical linear-algebra based systems use 2D-process-based partitioning

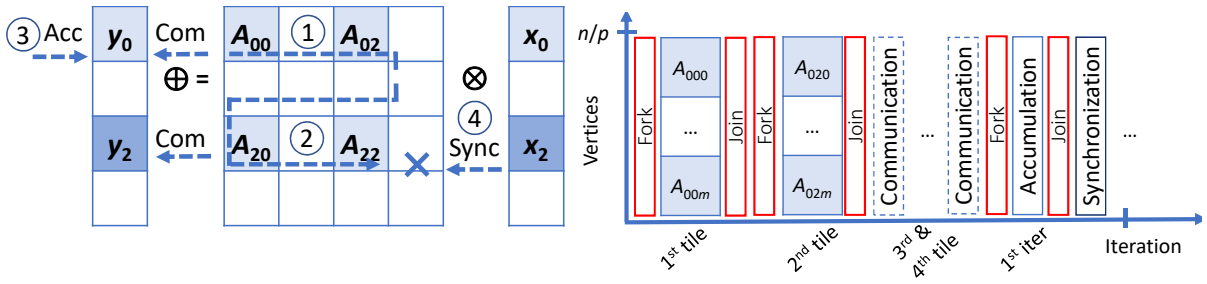
(tiling) to decompose a matrix into a 2D grid of tiles and achieve load balancing among processes [2, 5, 21, 115]. Having p processes, 2D tiling partitions the adjacency matrix of a graph G (say, A) with n vertices into a 2D p by p grid of tiles, producing p^2 tiles where each tile covers n/p rows and columns. This tiling creates a 2D layout of p *Row Groups* (RGs) and *Column Groups* (CGs) of tiles, where A_{ij} denotes the tile placed at i^{th} row group (RG_i) and j^{th} column group (CG_j). Similarly, a vector that can be involved in computation with the matrix (more on this shortly) is also partitioned into p *segments*, where each segment contains n/p elements.

Figure 18a shows the 2D-process-based partitioning of an exemplified matrix and a vector using $p = 4$. After partitioning, a process placement is pursued. To assign p processes to the 2D grid, many 2D-process-based placements put $\sqrt{p}$ processes per row/column group to limit the communication between processes [21]. Examples of this are 2D-Cyclic [5], which assigns processes in a cyclic order (Figure 18b), and 2D-Staggered [2], which further reorders row groups of the 2D-Cyclic to guarantee that each diagonal tile is assigned to a unique process, and thus aligns the assignment of row/column groups to processes (Figure 18c).

Many graph operations can be converted into simple linear algebra primitives. A common linear algebra primitive is SpMV operation $y = A_{\oplus\cdot\otimes}x$, where A is the $n \times n$ adjacency matrix of G, x and y are $n \times 1$ input and output vectors, and $\oplus\cdot\otimes$ is a semiring equipped with $(+, \times)$ operators. Overloading the semiring with operators specific to the application allows SpMV to run different applications. The iterative SpMV algorithm repeatedly uses the result vector, y, from one iteration to compute the input vector, x, for the next iteration until convergence. Often, y is transformed to an intermediate vector v, which is then used to compute x.

Figure 18d shows the assignment of tiles to processes in the 2D-Staggered placement. Processes are classified into two distinct categories, *leaders* and *followers*. The leaders (or processes assigned to diagonal tiles) are responsible for aggregating/broadcasting data from/to followers (or processes assigned to off-diagonal tiles) of their row/column group of tiles. In other words, leaders are the *owners* of their corresponding row/column group of tiles. Also, the leader of a row/column group of tile is the owner of the associated x and y vector segments and is responsible for maintaining/updating those segments.

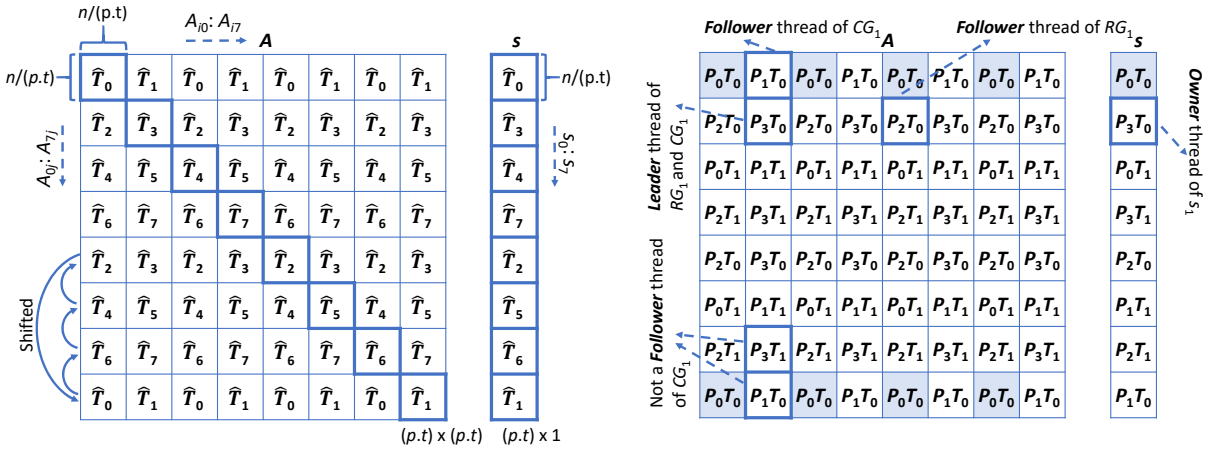
2D-Cyclic and 2D-Staggered are classified as 2D-process-based placements, which can be



(a) (b)

Figure 19: GraphPad tile processing (MPI + X) with $p = 4$ processes and $t = 2$ threads. Tiles are processed in a row-wise order where each tile is split into m smaller sub-tiles where m is much bigger than t for balancing load among threads. (a) Steps taken to process tiles/segments by process zero: (1) and (2) are row group SpMVs followed by their communication episodes, (3) is the accumulation of results for the row group owned by process zero, and (4) is $P0$'s synchronization with other processes. (b) Compulsory forks/joins of t threads while processing each tile.

used by the MPI + X model. Hence, systems relying on this parallelism model such as GraphPad [5], LA3 [2], Gemini [129], and CombBLAS [21] are not well suited for multithreading. As an example, Figure 19 shows how tiles are processed in GraphPad [5]. Specifically, Figure 19a is the sequence of processing tiles executing SpMV and accumulating the results for P_0 instructed by 2D-Cyclic (Figure 18b), while Figure 19b shows the steps taken by P_0 for tile processing. From these figures, this scheme comes with multiple caveats, namely (1) tiles have to be further split based on m which is a multiple of number of threads per process t (m sub-tiles in Figure 19b), (2) there are unnecessary compulsory thread forks and joins before and after the processing of each tile, (3) the main MPI process is responsible for the entire row group communication, (4) there are mandatory thread forks and joins for partial accumulation of results, and (5) the final synchronization point for checking the convergence is offloaded to the main process. These issues are potential performance bottlenecks and are applicable to other state-of-the-art graph processing systems such as Gemini [129] and LA3 [2]. Next section shows how thread-based tiling can fix these issues.



(a) 2DT-Staggered global thread assignment (b) 2DT-S. process and local thread assignment

Figure 20: Tile layout for $p = 4$ and $t = 2$. The 2D grid has $(p\ .\ t)(p\ .\ t) = 64$ tiles with $p\ .\ t = 8$ tiles per thread and $(p\ .\ t)(p\ .\ t) = 16$ tiles per process. In (a), rows marked as *shifted* are shifted to guarantee having t diagonal tiles per process. In (b), $Pi\ Tj$ denotes thread j of process i. Leader threads are at diagonal tiles, and follower threads have the same ids as their leader. Note that each thread is responsible for 8 tiles (e.g., the 8 tiles and 1 segment processed by thread $P0\ T0$ are shaded in (b).

4.2 2D-Thread-based Matrix Tiling & Placement

To address the problems of process-based partitioning and placement, including compulsory synchronization points for threads and heavy communication load for MPI processes, I propose **2D-thread-based matrix partitioning and placement**, a scheme implied by MPI $*$ X parallelism, which intrinsically deems threads as the basic units of computation and communication and reduces synchronization points.

Let A be the n by n adjacency matrix of a graph G with n vertices. To distribute the computation of A to p processes each with t threads, 2D-thread-based partitioning divides A into a grid of $(p \cdot t)$ by $(p \cdot t)$ tiles, each with height/width of $n/(p \cdot t)$. Afterwards, it assigns $p \cdot t$ tiles to each thread and, subsequently, $p \cdot t^2$ tiles to each process. To this end, each row group has $\sqrt{p}$ threads/processes and each column group has $\sqrt{p \cdot t}$ threads and $\sqrt{p/t}$ processes. Also, each thread/process has tiles in $\sqrt{p}$ row groups and each thread has tiles

Algorithm 1 2D-thread-based Staggered *tile to process/thread assignment* (See Table 4).

1: Input: # of processes p and # of threads per process t
2: Output: Assignment of Tiles$[p \cdot t][p \cdot t]$ to global thread ids $\hat{T}_k,\ k = 0, ..., (p \cdot t) - 1$ (Figure 20a)
3: Derivation of process id $P_k,\ k \in [0, p-1]$, and local thread id $T_k, k \in [0, t-1]$ from global thread id (Figure 20b).
4: $gcd\ =\ \texttt{GCD}(t_r, t_c)$
5: **for** $i = 0$ **to** $p.t$ **do**
6: **for** $j = 0$ **to** $p.t$ **do**
7: Tiles$[i][j].\hat{T} = ((i\ \mod t_c) \cdot t_r) + (j\ \mod t_r)$
8: Tiles$[i][j].\hat{T} += (\lfloor i/(p.t/gcd) \rfloor \cdot r_t)\ \mod (p \cdot t)$ ▷ Assignment of tiles to global threads
9: Tiles$[i][j].P$ = Tiles$[i][j].\hat{T}\ \mod p$ ▷ Grouping of threads into processes
10: Tiles$[i][j].T$ = Tiles$[i][j].\hat{T}/p$ ▷ Derivation of local thread ids

in $\sqrt{p/t}$ column groups. Alongside, each process has tiles in $\sqrt{p \cdot t}$ column groups. These values only hold when both p and $p.t$ are square numbers. In general, however, an integer factorization method [5] shall be used to determine the number of processes and threads per row/column group of tiles (p_r/p_c and t_r/t_c) (Algorithm 1).

Partitioning and placement are two intertwined concepts, whereby partitioning produces the tiles, and placement assigns threads (or processes) to tiles. This chapter extends the process-based 2D-Staggered placement [2] to a **thread-based 2D-Staggered (2DT-Staggered) placement**. The input to the 2DT-Staggered is the 2D grid of $(p \cdot t)^2$ tiles produced by 2D-thread-based partitioning.

In 2DT-Staggered, if a diagonal tile, $A_{i,i},\ i = 0, ..., (p \cdot t) - 1$, is assigned to a thread, then that thread becomes the leader of the i^{th} row group RG_i and i^{th} column group CG_i. Also, that thread renders the owner of the i^{th} segments of the input and output vectors, y_i and x_i. So, before executing the iterative SpMV $y_i = A_{ij} \oplus . \otimes x_j$ (in a right-multiplication fashion), the leader thread of CG_j sends x_j to its follower threads (threads that have tile(s) in CG_j). Later, after executing the SpMV of RG_i by all threads, the leader thread of RG_i receives partial results from the follower threads of that row group and accumulates them in y_i. Next, y_i is used to produce x_i via v_i (a segment of an intermediate vector v that stores results permanently) for the next iteration.

Figure 20a shows the 2DT-Staggered placement for eight global threads ($p = 4$, $t = 2$). From Algorithm 1: line 7 - 8, the 2DT-Staggered is materialized in two steps: (1) $p \cdot t$ thread ids are cyclically assigned to tiles, and (2) these ids are shifted so that each global thread $\hat{T}_k$

Table 4: 2D-process-based tiling versus 2D-thread-based tiling. The utilized function `Factorize(p)` returns pr and pc such that pr . $pc = p$ and `abs`(pr - pc) is minimized.

	2D-process-based	2D-thread-based
# of row/ column group of tiles	p	$p \cdot t$
# of tiles	$p \cdot p = p^2$	$(p \cdot t) \cdot (p \cdot t) = (p \cdot t)^2$
Tile height/ width	n/p	$n/(p \cdot t)$
Tile area	$(n/p)^2$	$(n/(p \cdot t))^2$
# of processes per row/column group (p_r / p_c)	$(p_r, p_c) =$ `Factorize`(p)	$(p_r, p_c) =$ `Factorize`(p)
# of threads per row/column group (t_r/ t_c)		$t_r = p_r$, $t_c = (p \cdot t)/t_r$
# of row/column groups per process (r_p/ c_p)	$r_p = p/p_c$, $c_p = p/p_r$	$r_p = (p \cdot t)/p_c$, $c_p = (p \cdot t)/p_r$
# of row/column groups per thread (r_t/ c_t)		$r_t = (p \cdot t)/t_c$, $r_c = (p \cdot t)/t_r$

is assigned to exactly one diagonal tile. Figure 20b shows the process ids and local thread ids derived from Algorithm 1: lines 9 - 10. Each process P_k is assigned to exactly t diagonal tiles, and each local thread in P_k receives one diagonal tile. RGs are distributed among processes in a staggered way, and then among their local threads in a row-wise way. In 2DT-Staggered, the staggered property balances the computation and communication among threads, while the row-wise property eliminates concurrent writes onto similar segments of y by multiple threads [1].

Generally, quick bursts of computation are interleaved with bursts of communication as a result of overlapping computation with communication and, accordingly, achieving scalability. As summarized in Table 4, the area of tiles in 2D-thread-based partitioning is t^2

[1]The uniqueness of diagonal processes/threads ids can be proved by derangement, which is a permutation of elements of a set i.e. no element appears in its original position [44] (Algorithm 1: lines 7-8 creates deranged id permutations).

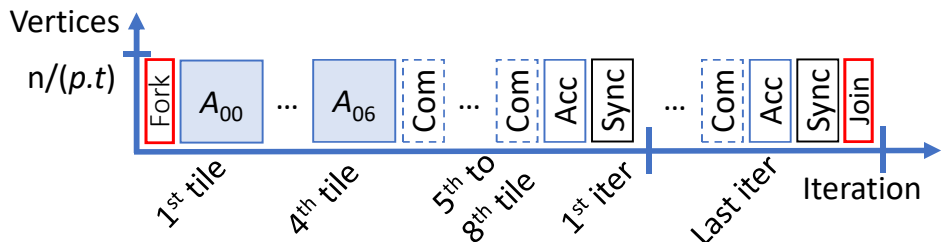


Figure 21: Tiles processed by thread $P0\,T0$; shaded tiles in Figure 20b (MPI * X). $P0\,T0$ has a single fork/join, and the synchronization is delayed till the end of an iteration to maximize the overlapping of computation and communication.

times smaller than in 2D-process-based partitioning (which is reasonably small and suitable for overlapping). Moreover, the MPI + X model, which uses 2D-process-based partitioning, considers p processes for carrying out communication. Conversely, MPI $*$ X, which uses 2D-thread-based partitioning, utilizes $p.t$ threads to pursue communication. Hence, the MPI $*$ X has t times more communication endpoints and, as such, a better degree of overlapping computation with communication. In summary, MPI $*$ X is a cost-effective parallelism since it overlaps the computation of reasonably smaller tiles with the communication of fairly smaller messages per thread. Also, by considering threads as basic units of computation and communication, this parallelism delivers better scalability.

Figure 21 demonstrates the advantages of 2D-thread-based over 2D-process-based: (1) 2D-thread-based inherently distributes the computation of tiles among threads, resulting in each thread being only forked/joined before/after the first/last iteration. Clearly, this avoids the overhead of frequent thread creation/termination and enables cooperative thread synchronization at the end of each iteration. (2) 2D-thread-based evenly splits the row/column group communication among threads, eliminating thereby the communication bottleneck caused by offloading communication to only MPI processes in the process-based variant. (3) 2D-thread-based offers a higher degree of overlapping between computation and communication because of its smaller tiles and larger number of MPI endpoints.

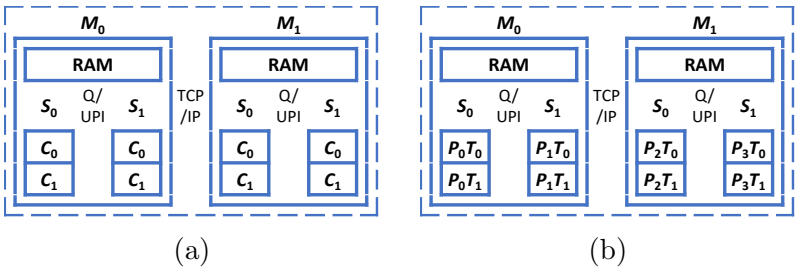


Figure 22: (a) A cluster with two dual-core dual-socket NUMA machines, and (b) NUMA-aware assignment of threads to cores with $p = 4$ and $t = 2$.

4.3 NUMA-aware placement in 2D-thread-based Tiling

The 2DT-Staggered placement assigns $P_0, ..., P_{p-1}$ process ids to tiles in a staggered way. Also, it assigns local thread ids of a process, e.g., $P_0T_0, ..., P_0T_{t-1}$ for P_0 to different rows. In Figure 20b, threads placed in the same row/column group, but belong to different processes (e.g., P_0T_0 and P_1T_0), use MPI to communicate with each other. Also, threads placed in the same row/column group, but belong to the same process (e.g., P_0T_0 and P_0T_1 in P_0), use shared memory to communicate. MPI has two transports, TCP/IP transport with 4 GB/s speed [104] used for inter-machine communication, and shared memory transport with 60 GB/s speed [104] used for intra-machine communication. Thus, a NUMA-aware assignment of MPI endpoints to tiles will benefit from the faster shared memory transport.

The linear order of process/thread ids does not necessarily imply assignments of processes to machines/NUMA sockets and threads to cores. These assignments are done by the MPI/threading environments before launching an MPI application and do not necessarily follow an expected order such as a sequential assignment order. However, knowing these assignments, processes/threads can be reordered before populating the 2D grid to efficiently exploit MPI's shared memory transports. To reorder processes, at first five pieces of information are gathered, namely, the topology of the cluster (using MPI [58]), the microarchitecture of machines (using NUMActl [70]), the number of processes per machine (using MPI [58]), the number of threads per process (using OpenMP [97]), and the number of processes/threads per row/column group (using the integer factorize method [5]). Finally, tapping into these, processes are reordered to maximize the MPI shared memory communication.

For instance, consider a simple cluster consisting of two NUMA machines as shown in Figure 22a, and its NUMA-aware assignment of processes/threads to machines/sockets/ cores as shown in Figure 22b. Combining the information of thread assignment to tiles (from Figure 20b) with the information of thread assignment to cores from Figure 22b), a NUMA-aware assignment of threads to cores can maximize the usage of the MPI shared memory transport for inter-socket communication among different MPI endpoints. Furthermore, experiments show that assigning one MPI process per socket provides faster MPI communication as the usage of integrated memory controller of NUMA sockets like Intel's QuickPath Interconnect (QPI) or Ultra Path Interconnect (UPI) with 16 GB/s speed [104] is only limited to the threads of the two processes placed in different sockets of a machine. Alongside, the controller is not used for shared memory accesses of threads inside the same process (because the shared memory communication of threads is limited to a single socket.)

4.4 Summary of MPI $*$ X Features

Table 5 outlined the main characteristics of the MPI $*$ X parallelism model and contrast them with those of the classical MPI + X model. The key advantages of MPI $*$ X stem from the 2D-thread-based partitioning which elevates threads to first-class citizens across the computation, communication, and synchronization. Specifically partitioning the input matrix into smaller tiles based on the total number of threads, thus evenly balancing the computation load among threads in a fine-granular way (1 in Table 5). Moreover, threads are communicating endpoints which provides a finer degree of computation and communication overlapping when using MPI asynchronous primitives (2 in Table 5). In addition, in MPI $*$ X, threads are persistent throughout computation, and synchronization is performed directly among threads at the end of each iteration. Thus, MPI $*$ X has less synchronization overhead (3 in Table 5). Conversely, in MPI + X, threads are *forked* and *joined* at every iteration and synchronization is applied between threads in each MPI process *then* among MPI processes.

MPI $*$ X leverages NUMA where this micro-architectural property provides the following benefits (4 through 6 in Table 5). Specifically, by launching one process per socket, MPI $*$ X's

Table 5: The traditional MPI + X versus the new MPI * X parallelism models.

	MPI + X parallelism	MPI * X parallelism	MPI * X advantages
1. Partitioning & placement	Process-based	Thread-based	More overlapping of computation and communication
2. Computation & communication units	Processes	Threads	Front-loading the computation & communication patterns
Synchronization	3. Process-based barriers	Thread-based barriers	Avoids compulsory forks/joins, & minimizes synchronization
4. Process layout	One process per machine	One process per socket	Enabling NUMA-aware computation & communication, & cache locality
5. Communication among multiple processes	Process-based inter-machine MPI-TCP/IP	Thread-based inter-socket MPI-SH.-MEM./ Thread-based inter-machine MPI-TCP/IP	Enabling faster MPI shared memory transport via Q/UPI interconnect
6. Communication among threads within a process	Intra-/Inter-socket shared memory	Intra-socket shared memory	Avoid inter-socket communication
7. Scaling	Horizontal **then** vertical scaling	Horizontal **&** vertical (diagonal) scaling	Removing the boundaries between processes & threads

threads enjoy processor/memory affinity where threads are bound to unique processors and, subsequently exploit L1 cache locality. Also, threads' memory accesses are local to their host sockets, restricting the shared memory communication of threads to those sockets which also avoids overloading the QPI/UPI interconnect. Moreover, combining the 2D-thread-based tiling with the micro-architectural information allows MPI $*$ X to take advantage of the MPI shared memory transport for inter-socket communication within a machine. Accordingly, MPI $*$ X offers fast inter-socket communication using the QPI/UPI interconnect.

Finally, MPI $*$ X incorporates diagonal scaling (7 in Table 5), which blurs the boundaries between processes and threads, and front-loads computation, communication, and synchronization among threads. The diagonal scaling is possible because the abstraction model, the library specification, and hardware properties are seamlessly integrated.

4.5 The Graphite

This section discusses Graphite, a new linear algebra-based distributed graph analytics system that employs the MPI $*$ X parallelism model. Graphite uses 2D-thread-based partitioning and placement (,i.e., 2DT-Staggered placement) to equally break the computation and communication of a sparse matrix among threads, while avoiding non-compulsory synchronizations. It scales diagonally and treats threads as basic units of computation and communication. Internally, Graphite utilizes MPI's `MPI_THREAD_MULTIPLE` option in conjunction with splitting the MPI communicator to enable collective and point-to-point communication between computing threads.

4.5.1 Multithreaded MPI Input Processing

Graphite supports distributed reading of plaintext and binary unweighted/weighted edge lists (which represent input graphs). For unweighted edge lists, it only stores the source and destination of each edge without a weight. Graphite has a built-in graph converter to manipulate an input graph based on problem constraints such as transposing it, making it acyclic, removing self-loops, or removing parallel edges. The 2D-thread-based partitioning used in Graphite instructs threads to collectively read edges from an edge list and insert them in their associated tiles. Tiles are compressed using Triply Compressed Sparse Column (TCSC) [86], a new sparse matrix compression format offering a compact representation of given sparse matrix and vectors. In addition, TCSC supports a new optimized variant of the SpMV primitive that takes advantage of the sparsity distribution of the matrix and vectors. This variant is called SpMSpV^2 (Sparse Matrix - Sparse input and output Vectors) which filters the empty rows and columns of a sparse matrix and vector.

4.5.2 Distributed SpMSpV^2 using 2D-thread-based Tiling & Placement

As pointed out earlier, in Graphite, tiles are compressed using TCSC [86], which enables distributed execution of SpMSpV^2 at scale. Rendering an n by n matrix A that represents a graph G with n vertices, a graph operation can be translated into an SpMSpV^2 primitive $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$; where $\bar{\bar{A}}$ is a $nzr \times nzc$ compressed matrix holding no empty rows and columns, and $\bar{x}$ and $\bar{y}$ are $nzc \times 1$ and $nzr \times 1$ compressed input and output vectors, with nzc and nzr standing for the numbers of nonzero columns and rows, respectively. Graphite is a *vertex-centric* system that abstracts the iterative computation of a large graph from the standpoint of a vertex. A vertex has a *value* (or state) containing some information about the problem being solved. Hence, there is a value (state) vector v of length n, which is divided into multiple segments like the $\bar{x}$ and $\bar{y}$ vectors. To run an application, first, v is interpolated to construct the new compressed input vector $\bar{x}$. Second, the SpMSpV^2 primitive $\bar{y} = \bar{\bar{A}} \oplus.\otimes \bar{x}$ is executed to produce the compressed vector $\bar{y}$. Finally, $\bar{y}$ is expanded to an uncompressed value vector v to interpolate and store the results permanently; as it will be used to construct the new $\bar{x}$ in the next iteration. In the following, this sequence is formalized as an iterative matrix computing model which closely works with the new 2D-thread-based partitioning.

4.5.3 Matrix Computing Model

Many Vertex-centric systems [2, 5, 31, 76, 80] assume graph-parallel abstractions such as *vertex programs* for encapsulating the operations executed on vertices of a graph. To collect and disseminate information, the GAS (Gather, Apply, and Scatter) model [50] adds fan-in/fan-out operations to a vertex program and characterizes the differences between vertex and edge computations. The Gather operation collects information about adjacent vertices and edges via a centralized sum. The Scatter operation propagates the new value of a central vertex through its adjacent edges. Finally, the Apply operation updates the value of the central vertex. Graphite adopts a similar model and iterates through *Broadcast* (Scatter in GAS), *Combine* (Gather in GAS), and *Apply* operations.

Graphite's computing model suggests a vertex program that can be overloaded with the desired code for the Broadcast, Combine, and Apply operations. Before running the

Algorithm 2 Matrix Computing Model

```
1: Input: Tiles of matrix Ā and x̄, ȳ and v vectors
2: Input: Overloaded functions to implement the operators for combine (⊗, ⊕), apply (←_a) and broadcast (←_b)
3: Temporary vector: ŷ
4: for k = 0 to t do fork(T_k)                                   ▷ Pin T_k to a unique core
5: Initialize v_k                                                ▷ Every thread T_k executes the following:
6: do
7:     x̄_k ←_b v_k                                               ▷ Broadcast
8:     for ∀j ∈ CG do
9:         MPI_Ibcast(x̄_j, leader_j, MPI_COMM_COL_j)
10:
11:    for i, j ∈ Ā do                                           ▷ Combine
12:        ȳ_i ⊕= (Ā_ij ⊗ x̄_j)
13:        if RG_i tiles are processed then
14:            if T_k is leader_i then
15:                for T_l ∈ RG_i followers do
16:                    MPI_Irecv(ŷ_il, T_l, MPI_COMM_ROW_i)
17:            else
18:                MPI_Isend(ȳ_i, T_k, MPI_COMM_ROW_i)
19:    ȳ_i ⊕= Σ_l ŷ_il
20:
21:    v_k ←_a ȳ_k                                               ▷ Apply
22: while Not CONVERGED(T_k)                                     ▷ Check convergence
```

vertex program, tiles of $\bar{\bar{A}}$ and segments of $\bar{x}$, $\bar{y}$ and v are distributed among threads using 2DT-Staggered, where $\bar{\bar{A}}_{ij}$ is the tile placed at the intersection of i^{th} and j^{th} row and column groups of tiles. In 2DT-Staggered, each thread is the leader of a unique row/column group of tiles and their corresponding segments of $\bar{y}/\bar{x}$ vectors (although it may have tiles in multiple row/column groups). Therefore, the k^{th} thread $T_k \mid k \in [0, t-1]$ of a process is the leader of the k^{th} uniquely owned row/column group and the associated vectors segments.

Algorithm 2 demonstrates the pseudocode of Graphite's GAS-like matrix computing model. Also, Figure 23 sketches the operations of the matrix computing model of Graphite which is used in conjunction with 2D-thread-based partitioning and placement for matrix parallel computations. In the following, I delve deeper into Graphite's computing model and discuss Broadcast, Combine, and Apply operations in details.

4.5.3.1 Broadcast Operation At the beginning of each iteration, each k^{th} leader thread (the leader of the k^{th} owned column group) calls the *Broadcast* operation (Algorithm 2: lines

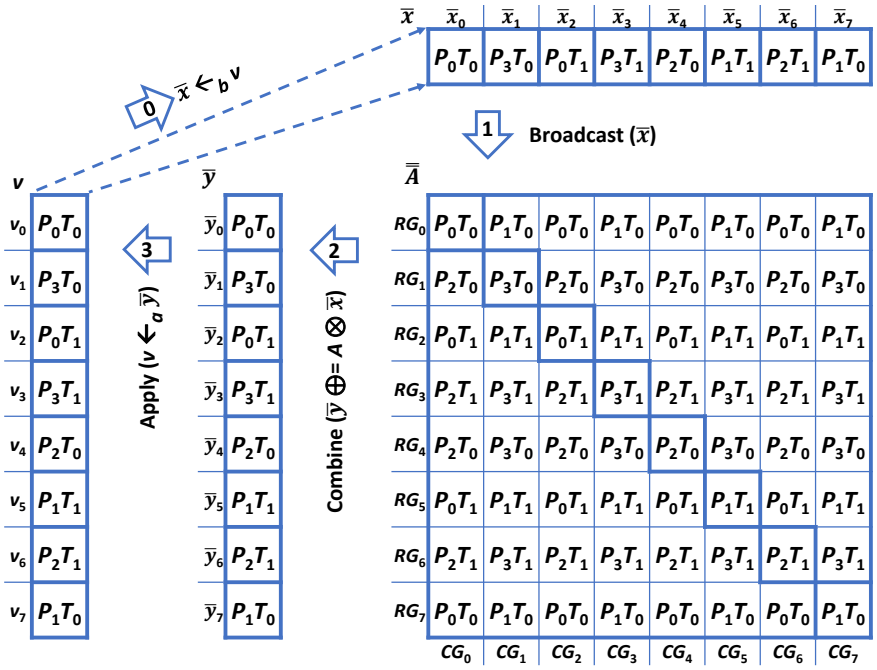


Figure 23: Integrating the matrix computing model (Broadcast, Combine, and Apply) with 2D-thread-based tiling to run SpMSpV2 ($p = 4$ and $t = 2$).

7 - 9, where $\leftarrow_b$ is the broadcast operator) to produce the new input segment $\bar{x}_k$ from an interpolation of v_k values (e.g., new ranks in PageRank). This transformation is marked by arrow 0 in Figure 23. Later, as signified by arrow 1, each T_k thread broadcasts the new input segment $\bar{x}_k$ to its followers in its column group, which enables every thread to receive new inputs required for the Combine operation. This is equivalent to the GAS's Scatter operation, which fans out on outgoing edges and sends new inputs to neighboring vertices.

In systems like GraphPad [5] and LA3 [2], GAS's Scatter operation is implemented using point-to-point primitives (`MPI_Isend()`/`MPI_Irecv()`) on the global MPI communicator. In Graphite, the two exemplified MPI primitives are merged into a single `MPI_Ibcast()` function (which is faster than point-to-point primitives due to using a tree-based communication algorithm [69]) via splitting the MPI communicator. To enable broadcasting messages inside a column group, first independent MPI column group communicators $\texttt{MPI\_COMM\_COL}_j$ are created for the threads in the same column group, CG_j, of tiles. While `MPI_COMM_WORLD` enables broadcast and collective communication among *all* processes, splitting the communi-

cator into subgroups enables broadcast and collective communication among threads hosted by the same column group. Broadcasting across column group communicators mitigates the pressure on the global communicator and avoids potential delays and contentions. Also, since each thread T_k is the leader in only one of its column groups (the root process of the `MPI_Ibcast()`) and a follower in the rest, a nonblocking broadcast can overlap communication among threads. So, threads can simultaneously send/receive different input segments associated with different column groups and interleave the communication of *sends/receives*.

4.5.3.2 Combine Operation As marked by arrow 2 in Figure 23, after broadcasting the new input $\bar{x}$, the *Combine* operation runs the SpMSpV^2 kernel. This is similar to GAS's Gather operation, which fan-ins and calculates a generalized sum over a neighborhood of a vertex. In the Combine operation (Algorithm 2: lines 11 - 19), each thread T_k iterates over its tiles in a *row order fashion* and executes the SpMSpV^2 kernel on its edges (i.e., $\bar{y}_i \oplus = (\bar{\bar{A}}_{ij} \otimes \bar{x}_j)$, where i and j are the indices of i^{th}/j^{th} row/column group of tiles, or i^{th}/j^{th} segments of $\bar{y}_i/\bar{x}_j$). After consuming tiles related to i^{th} row group, follower threads post their sends to the leader thread of the i^{th} row group, while leader threads post receives for partial output segments $\hat{\bar{y}}_i$'s from their followers. Afterwards, all threads move on to their next row group of tiles asynchronously. Once, all tiles are consumed, each T_k adds the partial result segments of $\hat{\bar{y}}_i$ to its $\bar{y}_i$ segment, which later will be used to update the i^{th} segment of value vector v_i.

To expedite the communication of the Combine operation, the global communicator is split into row group communicators, whereby threads inside the same row group of tiles, RG_i, use the same row group communicator $\texttt{MPI\_COMM\_ROW}_i$ for sending/receiving partial results. Combine uses the row group communicator to send/receive partial accumulation results. Moreover, for row group communication, the number of communicators is set equal to the number of row groups per process in order to provide concurrent race-free communication for all threads. Furthermore, Combine uses the MPI asynchronous communication routines, including `MPI_Isend()` and `MPI_Irecv()` to overlap the computation of tiles with the communication of partial $\hat{\bar{y}}$ segments. Hence, at the end of each row group, follower threads post their sends and leader threads post their receives. Subsequently, all threads

carry on independently with processing their next row group of tiles, while MPI buffers are still being sent/received in the background. Since the communication is only performed when the last tile of a row group is consumed, only a single pair of send/receive is required to transfer the partial results from a follower to the leader thread of that row group. Lastly, the accumulation of the segment owned by each leader, T_k, is done when all receives are completed as T_k's receives are sufficient for accumulation.

4.5.3.3 Apply Operation Marked by arrow 3 in Figure 23, in the *Apply* operation, each leader thread T_k interpolates its owned output segment $\bar{y}_k$ and constructs the new vertex values v_k (Algorithm 2: line 21, where $\leftarrow_a$ is the apply operator). This is similar to the GAS's Apply operation, which updates the state of the central vertex.

Finally, the matrix computing model operations, including Broadcast, Combine, and Apply are followed by a check for convergence, which is also run concurrently by all threads. Depending on the application requirements, this sequence repeats until executing a certain number of iterations or reaching convergence.

4.5.4 Leveraging NUMA in Graphite

4.5.4.1 NUMA-aware Shared Memory Communication As discussed in Section 4.5.3.2, the proposed computing model relies on point-to-point primitives for the Combine operation, which can be effectively accelerated using the MPI shared memory transport. Guided by the MPI $*$ X model, which suggests launching one MPI process per socket, threads that belong to two processes launched at the same machine are placed in the same row group of tiles in the 2D grid of tiles. Therefore, the inter-socket communication can be highly optimized using the MPI shared memory transport (see Section 4.3). Having this setting, the communication of the Combine operation is overlapped with its computation of tiles, which further alleviates the use of point-to-point MPI primitives. Contrarily, column group communication cannot benefit from the MPI shared memory transport because column group processes run mostly on different machines. However, `MPI_Ibcast()` already offers swift TCP/IP communication which mitigates the lack of having a better transport.

4.5.4.2 Processor & Memory Affinity Processor/memory affinity avoids excessive migrations of processes/threads, thus allowing them to benefit from hot caches and NUMA. GraphPad [5] and Gemini [129] leverage MPI [58, 98] and OpenMP [97] to control CPU and memory affinity at runtime. In contrast, Graphite explicitly controls affinity by launching one MPI process per socket and pinning threads to cores. In particular, the *processor affinity* forces threads to be launched at the same NUMA socket as the MPI process. This translates to fewer context switches, TLB flushes, and L1 cache invalidations. Also, it offers efficient L2/L3 cache accesses because an access to L2 is limited to the working thread pinned on a core and an access to L3 is limited to the working threads running on that cores' socket. Moreover, *memory affinity* enforces contiguous allocation of memory for matrix tiles and vector segments on a NUMA node when the MPI process uses `numa_alloc_onnode` [70]. Memory affinity avoids overloading the memory interconnect across sockets such as QPI/UPI and offers faster main memory accesses. Also, binding a core to a thread allows all data structures of tiles and segments to be allocated at the same NUMA socket of the core. All in all, threads can subsequently enjoy hot caches while running SpMSpV^{2}s on $\bar{x}$ and $\bar{y}$ segments. Moreover, within a process, there is only one $\bar{x}$ segment per column group from which all threads can safely read in parallel.

4.5.5 Enabling Compiler Optimization

Based on my experience with multithreaded programming, compiler optimizations are not fully supported (or are degraded– e.g., from `-O3` to `-O2`) while developing programs with cross-function and/or cross-file invocations by threads. The loss from the absence of compiler optimizations and the presence of sandboxing (where programs are sandboxed in multithreading runtimes, thus inducing overhead) may completely offset the gain from multithreaded programming [15, 116, 117]. As such, I keep the iterative compute-intensive SpMSpV^2 kernels concise and overload them locally with basic mathematical operators instead of using expensive cross-function calls. In addition, I avoid using `virtual` methods because they enforce each function call to go through a virtual table to look up and invoke the callee method. To this end, I utilize `inline` methods, which allow the compiler to see

the majority of code in advance and, accordingly, exploit vectorization and loop unrolling. Lastly, to effectively break the code and computation among threads, I use Pthread instead of OpenMP, which is about 20% faster [114].

4.5.6 Activity & Computation Filtering

Graph applications are divided into *stationary applications*, where *all* vertices remain active during execution, and *non-stationary applications*, where the number of active vertices varies during runtime.

Activity filtering is a technique used in non-stationary applications to remove unnecessary computation and communication of inactive vertices [2, 5, 129]. In Graphite, if less than 60% of vertices render inactive, only a list of (index, value) pairs representing active vertices are used for communication, precluding thereby any traffic data that pertains to inactive vertices. Otherwise, Graphite falls back to sending the original arrays of nonzero elements, which encompass actual values for active vertices and dummy values for inactive ones. When activity filtering is enabled, a SpMSpV^2 kernel only executes the received list of (index, value) pairs, skipping naturally the computations of inactive vertices. When activity filtering is disabled (i.e., when Graphite falls back to sending original arrays), a SpMSpV^2 kernel skips the computations of inactive vertices using the dummy placeholders.

Besides activity filtering, *computation filtering* is used in stationary applications [2] to skip the computation of unnecessary edges of a sparse matrix. First, computation filtering classifies vertices into four categories: 1) *regular vertices*, which are vertices with both ingoing and outgoing edges, 2) *source vertices*, which are vertices with only outgoing edges, 3) *sink vertices*, which are vertices with only ingoing edges, and 4) *isolated vertices*, which are vertices with no edges. Next, it leverages these types of vertices to avert unnecessary computations as follows: 1) regular vertices are executed in all iterations because their values are used by other vertices via the input vector, 2) source vertices are only executed in the first iteration because their values will not be changed afterwards, 3) sink vertices are only executed in the last iteration because their values are not used in earlier iterations, and 4) isolated vertices are discarded completely from the execution loop because their values are never used in

Table 6: Datasets used for experiments, and the number of nodes used to process them.

Graph	$\|V\|$	$\|E\|$	Type	Nodes
UK'05 (UK5) [16]	39.4 M	0.93 B	Web Crawl	4
IT'04 (IT4) [16]	41.2 M	1.15 B	Web Crawl	4
Twitter (TWT) [16]	41.6 M	1.46 B	Social	8
GSH'15 (G15) [16]	68.6 M	1.80 B	Web Crawl	8
UK'06 (UK6) [16]	80.6 M	2.48 B	Web Crawl	16
UK Union (UKU) [16]	133 M	5.50 B	Web Crawl	20
Rmat26 (R26) [25]	67.1 M	1.07 B	Synthetic	4
Rmat27 (R27) [25]	134 M	2.14 B	Synthetic	8
Rmat28 (R28) [25]	268 M	4.29 B	Synthetic	16
Rmat29 (R29) [25]	536 M	8.58 B	Synthetic	20

any iteration. Graphite adopts computation filtering for directed graphs, only since these four types of vertices exist only in them. For undirected graphs, all vertices are regular or isolated, rendering computation filtering less effective.

4.6 Results

4.6.1 Experimental Settings

4.6.1.1 Cluster Configuration Experiments are conducted on a HPC cluster of 20 nodes, each with 28-core (14 cores per socket) Broadwell Processor (2.60GHz) and 192GB RAM. The cluster has Intel Omni-path interconnect (10 Gb/s speed) and all nodes are connected to an OFA network fabric. Nodes run Red Hat Enterprise Linux Server 7.6. The cluster uses Slurm workload manager for batch job queuing [108]. Intel MPI [58] with multithreading is used to support for communication across machines and Pthread [71] for launching threads inside an MPI process. OpenMP [97] is utilized to collect information of allocated cores within a process, Pthread to provide CPU affinity, NUMActl [70] to enable memory affinity, and Linux `sysconf` to get the cache information at runtime.

Experiments on the cluster follow two settings. *Weak scaling* where the number of machines (and cores) used for processing is proportional to the size of the graphs and *Strong scaling* where the graph size is fixed and the number of machines (cores) is varied. At scale, all 20 nodes of the cluster (560 cores) are used. Finally, any reported number is the average of multiple individual runs.

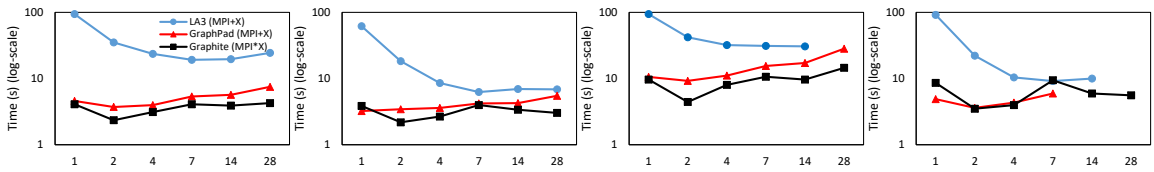
4.6.1.2 Counterpart Systems Graphite[2] has been tested against two linear algebra-based systems GraphPad [5] and LA3 [2], and one graph theory-based system Gemini [129]. For all systems, the number of processes per machine π and the number of threads per process t are fine-tuned and the configuration that demonstrates the best runtime is picked. Thus, GraphPad is ran with $\pi = 2$ and $t = 14$, LA3 with $\pi = 14$ and $t = 2$, and Gemini with $\pi = 1$ and $t = 28$. Similarly, Graphite is ran with different combinations of π and t and use $\pi = 2$ and $t = 14$ as it delivers the best results. Note GraphPad, LA3, and Gemini crashed for some experimental settings because of limitation memory or number of processes.

4.6.1.3 Graph Datasets Table 6 shows graphs used in the experiments including six real-world graphs (web crawls and social network from LAW [16]), and four synthesized graphs (RMAT 26 - 30 graphs from the Graph 500 challenge [25].[3]). In Table 6, the last column, *Node* reports the number of nodes used to process a graph dataset in the experiments (unless otherwise stated). To provide the weak scaling property, starting from four nodes up to 20 (the cluster size), the number of nodes are increased relative to the graph size.

4.6.1.4 Graph Applications Graphite has an extensible API supporting different graph applications. Graphite supports PageRank (PR) for unweighted directed graphs as a prime stationary application. PR includes degree application as well. In addition, Graphites supports three non-stationary applications including Single Source Shortest Path (SSSP) for weighted directed graphs, Breadth First Search (BFS) and Connected Component (CC) for unweighted undirected graphs. Similar to the setting used in GraphPad [5], LA3 [2], and

[2]Graphite's source code is available at https://github.com/hmofrad/Graphite

[3]RMATs follow power of two growth rate, i.e., RMATn has 2^n vertices and 2^{n+4} edges.



(a) PR on TWT (8 nodes) (b) CC on TWT (c) PR on R28 (16 nodes) (d) CC on R28

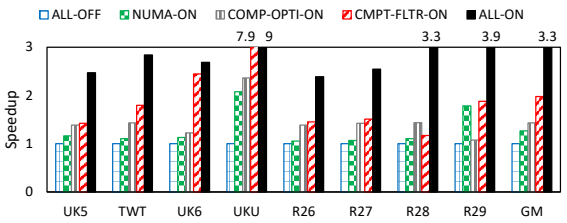
Figure 24: Runtime of Graphite and others with (# processes per machine, # of threads per process) = (1, 28), (2, 14), (4, 7), (7, 4), (14, 2), and (28, 1).

Gemini [129] PR is ran for 20 iterations and SSSP, BFS, and CC until convergence.

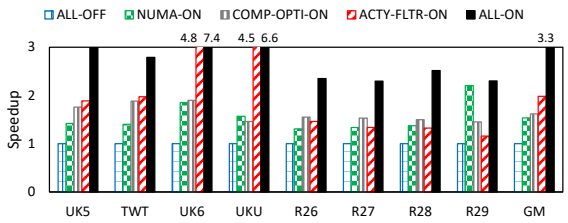
4.6.2 Multithreading Spectrum

This experiment shows how the number of processes per machine π and the number of threads per process t affect the scalability of GraphPad [5] and LA3 [2] (two MPI + X systems), and Graphite (an MPI $*$ X system). Figure 24 shows the results of GraphPad, LA3, and Graphite with different configurations of π and t (x-axis), i.e., $\pi \cdot t = 28$ (total number of cores per machine) using PageRank (PR) and Connected Component (CC). For Graphite, certain observations can be made from its Double-u (W) shaped trends of Figure 24. The optimal configuration for Graphite is $\pi = 2$ and $t = 14$ where one process per socket is launched and faster inter-socket communication is leveraged. Also, there is a spike at runtime for $\pi = 7$ which is due to having an odd number of processes; with this configuration there is a process in each machine that has threads on both sockets, therefore stressing the QPI/UPI interconnect for shared memory communication among threads. Moreover, neither the pure multithreading ($\pi = 1$) nor the pure multi-processing ($\pi = 28$) per machine produces good results. From the viewpoint of a single machine, pure multithreading imposes communication overhead on QPI/UPI for accessing input vector segments across sockets, and pure multi-processing imposes communication overhead on QPI/UPI for inter-process communication.

From Figure 24, GraphPad has comparable performance when launched with one or two processes per machine and its performance drops as it moves to more processes per machine (perfect multiprocessing). Also, LA3 cannot utilize threads effectively, and therefore as it utilizes more processes than threads its performance first improves (up to 14 processes



(a) PR



(b) CC

Figure 25: Normalized speedup (weak scaling) of NUMA, COMP-OPTI, CMPT-FLTR, and ACTY-FLTR with ALL-OFF/ALL-ON as baseline/headline. GM (grand geometric mean).

per machine) and then drops (for 28 processes) which roots in LA3's poor work distribution among threads. Comparing with GraphPad and LA3, Graphite has a decent runtime difference across the majority of configurations.

4.6.3 Sensitivity to Different Optimizations

Graphite offers a set of features for scalable graph processing including NUMA-aware shared memory MPI communication (NUMA), *compiler optimization* (COMP-OPTI), *computation filtering* (CMPT-FLTR), and *activity filtering* (ACTY-FLTR). From Figure 25a, on PageRank (PR) (a stationary application), NUMA, compiler optimization, computation filtering, and the combination of these features give 27%, 43%, 2×, and 3.3× speedups, respectively. From this figure, smaller graphs benefit more from compiler optimization whereas larger graphs benefit more from NUMA and computation filtering. Also, from Figure 25b, on Connected Component (CC) (a non-stationary application), NUMA, compiler optimization, activity filtering, and the combination of them give 53%, 62%, 2×, and 3.3× speedups, respectively. From this figure, NUMA and compiler optimization are more effective in synthetic graphs (which has uniform distribution with constant edge factor), whereas activity filtering is more effective in real-world graphs (which follows power-law distribution with high variance in number of edges per vertex). From the last bars of each dataset of Figure 25, enabling all features results in a better speedup which shows their effects are cumulative.

Figure 25 shows that **NUMA** is more effective for larger graphs which stems from leveraging memory and processor architecture to maximize the usage of MPI shared memory

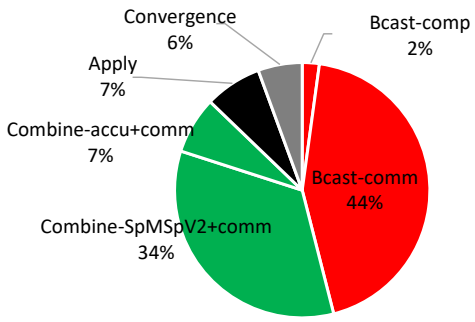
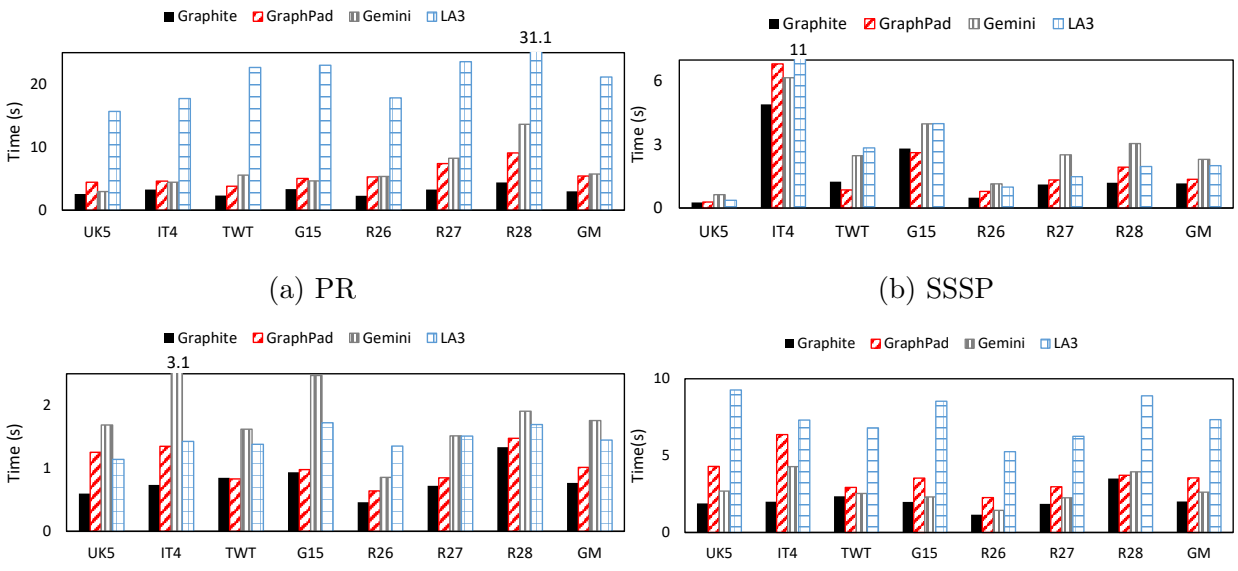


Figure 26: Graphite Execution time breakdown (s) from running PR on R28 using 16 nodes.

transport. Also, enabling the **compiler optimization** to its fullest extent is vital for running an iterative compute-intensive SpMSpV^2 kernel, because this kernel includes the bulk of computation done by threads, and any optimization that can slightly improve on this kernel, will largely improve the overall runtime. Finally, the **computation filtering** advantage comes from passing over the computation of subsets of unnecessary vertices in stationary applications (e.g., PR), and the **activity filtering** advantage comes from skipping the communication and computation of inactive vertices in non-stationary applications (e.g. CC).

4.6.4 Execution Time Analysis

Graphite's matrix computing model iterates over Broadcast, Combine, and Apply operations. In addition, Graphite checks for convergence and enforces synchronization among threads at the end of each iteration. Figure 26 shows the breakdown of execution time of 20 iterations of PR on R28. It is clear that Broadcast and Combine operations are both computation and communication intensive, and constitutes about 90% of the runtime. **Broadcast time** (46%) consists of the time for preparing the new input segments (Bcast-comp), plus the overlapped communication time (Bcast-comm). **Combine time** (41%) constitutes the time spent for running the SpMSpV^2 (Combine-SpMSpV^2+comm), and the accumulation time of the partial output segments which is overlapped with background communication (Combine-accu+comm). **Apply time** is the time for interpolating and updating the values of the vertices. Combine-accu+comm and Apply times are roughly equal as both are operating on similar segments. Finally, **Convergence time** is the total synchronization time of threads at the end of each iteration which includes the time for checking the convergence.



(a) PR

(b) SSSP

(c) BFS

(d) CC

Figure 27: Runtime of Graphite and other systems (weak Scaling). GM is the grand geometric mean over all datasets.

4.6.5 Comparisons with other Systems

4.6.5.1 Weak Scaling Comparison Weak scaling of Graphite versus GraphPad, Gemini, and LA3 are reported in Figure 27. Based on the grand geometric mean of results (geometric mean of geometric mean of each subfigure), Graphite acheives superior speedup and is 2.9×, 60%, 80%, and 2.1× faster than these systems in PR, SSSP, BFS, and CC applications.

From Figure 27a, in **PageRank (PR)**, Graphite is on average (geometric mean) 81%, 91%, and 7.1× faster than GraphPad, Gemini, and LA3. PR is a computation- and communication-intensive non-stationary application which needs to visit all vertices and their associated edges in order to rank them. For PR, computation filtering helps Graphite to skip the computation of subsets of vertices[4]. Also, compared to others, LA3 does not perform good on PR because it has rigorous communication optimizations which are not effective in an HPC cluster with fast interconnect.

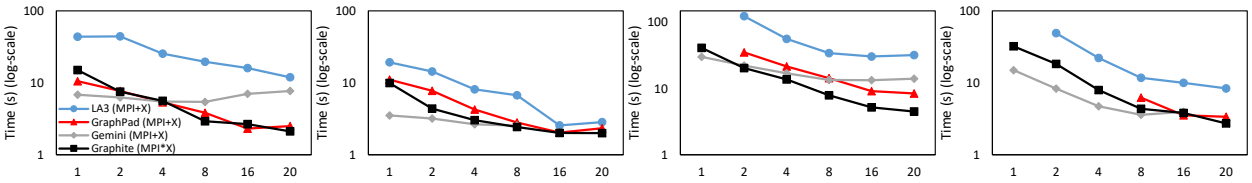
[4]In R29, computation filtering skips 5% of SpMSpV^2 ops.

Having a look at Figure 27b, Graphite is 18%, 2×, and 73% faster than GraphPad, Gemini, and LA3 on average in **Single Source Shortest Path (SSSP)**. Running SSSP on a directed graph, the source vertex is an important factor regardless of the size of graph. Starting from the source, SSSP traverses all vertices connected to the source with an incoming link from the source. So, if source is sampled from a small connected component, all vertices of that component will be visited quickly and that is why for some graphs like UK5 or R26 the runtime is small compared to other graphs. Graphite performs the best in SSSP except for TWT because the complex structure of the largest component of TWT causes a huge load imbalance among threads[5]. In addition, GraphPad outperforms Gemini and LA3 because of its better communication and compression optimizations.

On BFS (Figure 27c), Graphite outperforms GraphPad, Gemini, and LA3 with 33%, 2.3×, and 90% better runtime on average. Given **Breadth First Search (BFS)** uses undirected graphs, unlike SSSP, it eventually visits all vertices of the connected component where the source is chosen from. Therefore, BFS deals with more communication and computation than SSSP. Gemini is relatively slow because it does not have a good communication strategy and relies on a single thread per process to communicate messages of a row of tiles which works fine only for small number of nodes e.g. 8 nodes. LA3's communication optimizations work better in BFS (and SSSP) because communication pattern of BFS (and SSSP) include(s) small bursts of data transfer which can quickly be compressed in LA3.

As shown in Figure 27d, on average Graphite performs 77%, 31%, and 3.7× faster than GraphPad, Gemini, and LA3 for **Connected Component (CC)**. CC tries to find a set of vertices that are connected to each other by paths (a strongly connected subgraph) and accomplishes this task by iteratively visiting all vertices inside components. Gemini outperforms GraphPad and LA3 because it uses NUMA-aware partitioning which offers faster local memory access and higher cache utilization. Both GraphPad and Gemini use a pair of dense vectors accompanied by a bitvector for fast random access of compressed vectors. However, GraphPad is slower than Gemini in CC because for this application Gemini can effectively switches between its sparse and dense representations using its push/pull model, whereas, GraphPad compression threshold is ineffective here. On the other hand, Graphite's

[5]The largest component of TWT includes 80% of its edges.



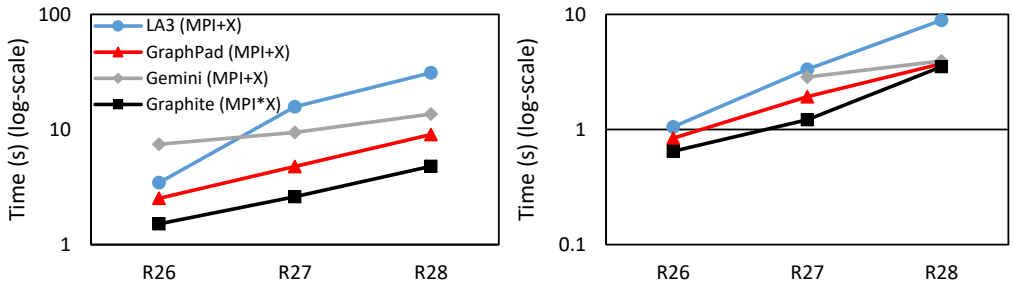
(a) PR on TWT (b) CC on TWT (c) PR on R28 (d) CC on R28

Figure 28: Strong cluster scaling of different systems on R28. X-axis is the number of nodes.

decision for switching between sparse and dense representations is made in Broadcast operation and reused in Combine operation. Although this approach poses a small overhead to the Broadcast operation, altogether it results in a better performance for CC as it skips the computation of activities in the Combine operation.

From Figure 27, Graphite outperforms GraphPad, Gemini, and LA3 systems, where this outperformance is largely due to the usage of MPI $*$ X parallelism model and 2D-thread-based partitioning and placement. Conversely, other systems follow MPI + X parallelism and process-based partitioning that underperform in iterative applications. For example, GraphPad and Gemini use process-based 2D-Cyclic and 1D-Row placements, which are less scalable than thread-based 2D-Staggered placement used in Graphite.

4.6.5.2 Strong Cluster Scaling Comparison Figure 28 shows the runtime of different systems for different number of machines for PR and CC on TWT and R28. Overall, Graphite scales very well on both TWT (real-world) and R28 (synthetic). It can effectively leverage the added processing power and improve the runtime. This scalability is highly due to MPI $*$ X parallelism model which balances the computation and communication of tiles among threads. Moreover, GraphPad which follows the MPI + X parallelism model exhibits comparable scalability on TWT and poorer scalability on R28. Next, Gemini starts with a good performance, but fails to scale for larger clusters due to the limitations of MPI + X parallelism, e.g., only MPI processes carry out communication. Last, LA3 does not scale well as its communication strategy is not suitable for HPC clusters.



(a) PR (b) CC

Figure 29: Strong data scaling (R26-28 with 16 nodes)

.

4.6.5.3 Strong Data Scaling Comparison Figure 29 shows the runtimes of different systems on R26 - R28 using 16 machines for PR and CC. From Figure 29a, Graphite, which follows MPI $*$ X exhibits a better data scaling with minimal changes in the runtime for PR. In this figure, Graphite outperforms GraphPad, Gemini and LA3 which all follow MPI + X. Additionally, a similar trend for CC can also be seen in Figure 29b.

4.6.5.4 Discussion of Evaluated Systems Table 7 thoroughly reports different features of the studied systems. From this table, GraphPad [5], Gemini [129], and LA3 [2] use MPI + X model with processes being the basic units of computation and communication, whereas Graphite uses MPI $*$ X model where threads are the basic units of computation and communication. GraphPad, Gemini, and LA3 use process-based 2D-Staggered, 2D-Cyclic, and 1D-Row placements, whereas Graphite uses 2DT-Staggered that is devised for threads. Moreover, although all these systems utilize GAS-like computing models, Graphite carefully incorporates asynchronous collective MPI primitives in its model enabling faster communication. Also, Graphite leverages NUMA for both computation (CPU/memory affinity) and communication (MPI shared memory communication) purposes, whereas Gemini internally supports memory affinity but relies on OpenMP for processor affinity. In addition, Graphite carefully follows strict programming guidelines to completely enable compiler optimizations for multithreaded SpMSpV^2 kernels. Last, all systems use activity filtering, but only Graphite and LA3 use computation filtering.

Table 7: Summary of features of the studied systems.

Feature	Graphite	GraphPad	LA3	Gemini
1. Parallelism Model	MPI*X	MP+X	MP+X	MP+X
2. Unit	Thread	Process	Process	Process
3. Tiling	2DT-Staggered	2D-Cyclic	2D-Staggered	1D-Row
4. Computational Model	GAS	GAS	GAS	Push/pull
5. NUMA	Full	No	No	Memory
6. Computation Optimization	Targeted	Default	Default	Default
7. Computation Filtering	Yes	No	Yes	No
8. Activity Filtering	Yes	Yes	Yes	Yes

The performance difference between Graphite and LA3 is due to three design decisions made in LA3: (1) Communication strategy: LA3 is designed for *cloud environments* (not HPC) with low-bandwidth and high-latency interconnection networks. It has an extensive communication optimization that tailors input messages per tile to reduce the communication volume at the expense of more computation overhead. In a cloud environment, this strategy works well because the communication delay is more expensive than the time spent for constructing the optimized messages. In contrast, in an HPC environment, this strategy is not productive because of fast interconnects. (2) Parallelism model: LA3 follows the MPI + X model. It relies on OpenMP runtime to distribute the computation of tiles across threads while bounding the communication to only MPI processes. Thus, compared to an MPI $*$ X system like Graphite, LA3 has less MPI communication endpoints and larger tiles, which reduces the overlapping of computation with communication. (3) Matrix compression: LA3 uses Doubly Compressed Sparse Column (DCSC) [19], whereas Graphite uses Triply Compressed Sparse Column (TCSC) [86], which is more cache friendly.

4.7 Conclusions

In this chapter, I introduced Graphite, a new linear algebra based graph analytics system that uses the MPI $*$ X parallelism model with 2D-thread-based partitioning and placement.

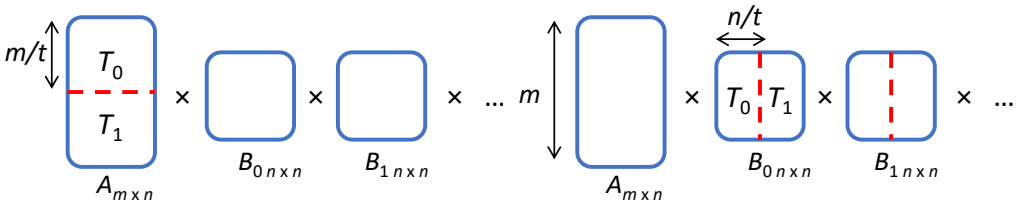
In Graphite, threads are treated as first-class citizens of a distributed system where computation and communication are fairly distributed among all threads while minimizing the synchronization points. Graphite utilizes a GAS-like matrix computing model for fast execution of iterative analytics that takes advantage of MPI and distributed shared memory capabilities. It exploits NUMA for both computation (CPU/memory affinity) and communication (MPI shared memory communication). Compared against GraphPad, Gemini and LA3 analytics systems, the proposed Graphite achieves a speedup of roughly up to 3× due to its thread-level asynchronous communication and computation, high degree of concurrent communications, and NUMA-ware computation and communication.

5.0 Studying the Effects of Hashing of Sparse Deep Neural Networks on Data and Model Parallelisms

Thus far, SpMV primitive and its diverse application in graph analytics is discussed. In this chapter, SpMM primitive and its application in sparse neural network inference will be discussed in details.

Deep Neural Network (DNN) training and inference are two resource-intensive tasks that are usually scaled out using data or model parallelism where *data parallelism* parallelizes over the *input data* and *model parallelism* parallelizes over the *network*. Also, *dense matrix-matrix multiplication* is the key primitive behind training/inference of *dense DNNs*. On the contrary, *sparse DNNs* are less resource-intensive compared to their dense counterparts while offering comparable accuracy. Similarly, they can be parallelized using data or model parallelism with *Sparse Matrix-Matrix Multiplication (SpMM)* as the key primitive. To scale out, both data and model parallelisms initially use data parallelism to partition the input data among multiple machines. This initial partitioning of the input makes data and model parallelisms performance prone to load imbalance as partitions may be imbalanced. As part of this chapter, I take a deeper look into data and model parallelisms and closely study the mechanics of the SpMM used for each. Moreover, to intuitively remedy their load imbalance problem, I incorporate hashing as a simple yet powerful method to address load imabalance. Results suggest that with hashing, data and model parallelisms achieve super-linear speedup due to better load balance and cache utilization.

The rest of this chapter is organized as follows. Section 5.1 presents the background and surveys the related work. Section 5.2 investigates data and model parallelisms. Section 5.3 studies the effect of neural network hashing. Section 5.4 reports the results. Finally, Section 5.5 concludes this chapter.



(a) Data parallelism (b) Model parallelism

Figure 30: Data and model parallelisms for two threads (t=2).

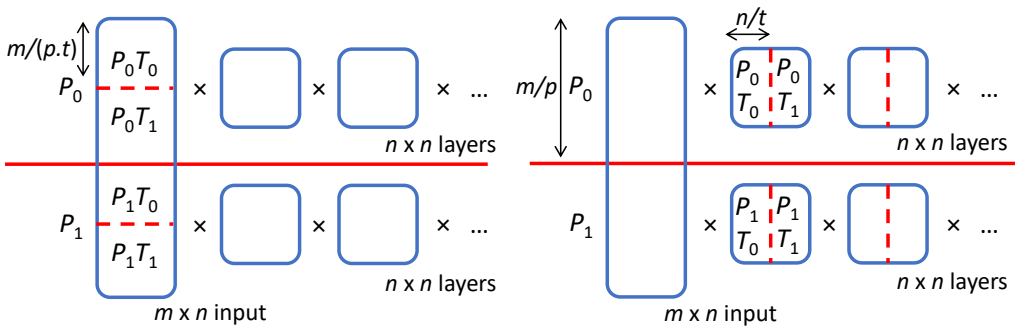
5.1 Background

5.1.1 Inference using Sparse Matrix-Matrix Multiplication

Neural network connections can be represented using the *triplet format* [87], where a triplet (i, j, w) implies a connection from neuron i of layer l to neuron j of the following layer $l+1$ with w being the weight of their connection. Hence, inference can be boiled down to the iterative SpMM of $C_{l+1} = h((A_l \times B_l) + b_l)$, where l is the index of the hidden layer, A_l is the l^{th} $m \times n$ input sparse matrix with A_0 being the input layer, B_l is the l^{th} $n \times n$ hidden layer, and C_{l+1} is the $m \times n$ sparse matrix resulting from the l^{th} layer SpMM which will be copied to A_{l+1}. Also, the function h is a nonlinear mapping function such as the ReLU activation function $h(y) = max(y, 0)$ and b_l is the bias vector of the l^{th} layer.

5.1.2 Data and Model Parallelisms

Inference can be parallelized in different ways including input size, DNN's breadth (height/width), and depth. Having t processes, **model parallelism** uses **1D-Column partitioning** (vertical stripes) to divide the **network** of breadth n (height/width) into t partitions of size n/t neurons. Here, all threads are synchronized at the end of each layer since the output of a current layer becomes the input of the following layer (see Figure 30b). **Data parallelism** uses **1D-Row partitioning** (horizontal stripes) to divide the **input** with m input instances into t partitions of size m/t instances. Then, each thread independently processes its partition without requiring any synchronization (see Figure 30a). Both parallelisms do not require concurrency control as threads execute on separate partitions.



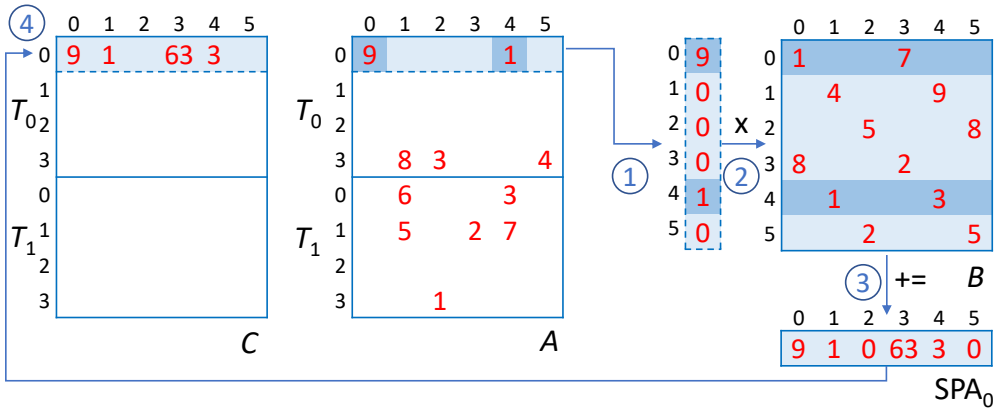
(a) Data*data parallelism (b) Data*model parallelism

Figure 31: Data*data and data*model parallelisms for two processes and two threads per process (p=2, t=2).

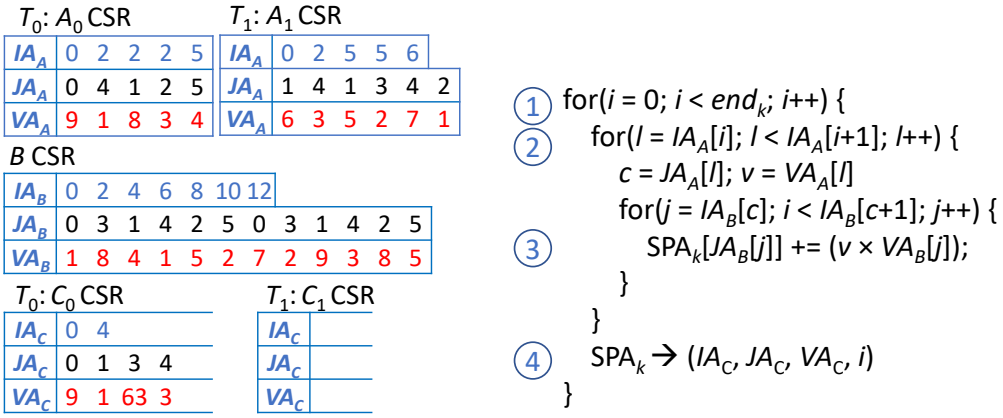
In a distributed setting with p processes and t threads per process, **data∗model** and **data∗data** parallelisms are applied where the network is replicated for each process to avoid unnecessary network communications and the input is partitioned into p partitions to provide a load balance among processes. Afterward, in data∗model the network is broken into t partitions of size n/t, and in data∗data each input partition is further broken into t subpartitions of size $m/(p \cdot t)$ (see Figure 31a and Figure 31b).

5.2 The Duality Between Left and Right SpMM

Gustavson's algorithm [53] is a widely used SpMM algorithm. This algorithm is often combined with other data structures such as Sparse Accumulator (SPA) [46], heap, or hash to produce a row/column of the output matrix C. In the following of this section, Gustavson's left and right SpMMs are described in the context of data and model parallelisms. Note that a symbolic SpMM step to pre-allocate C precedes these SpMM algorithms. Hence, enough memory for C is already allocated.



(a) Data parallelism row-by-row left SPMM using CSR

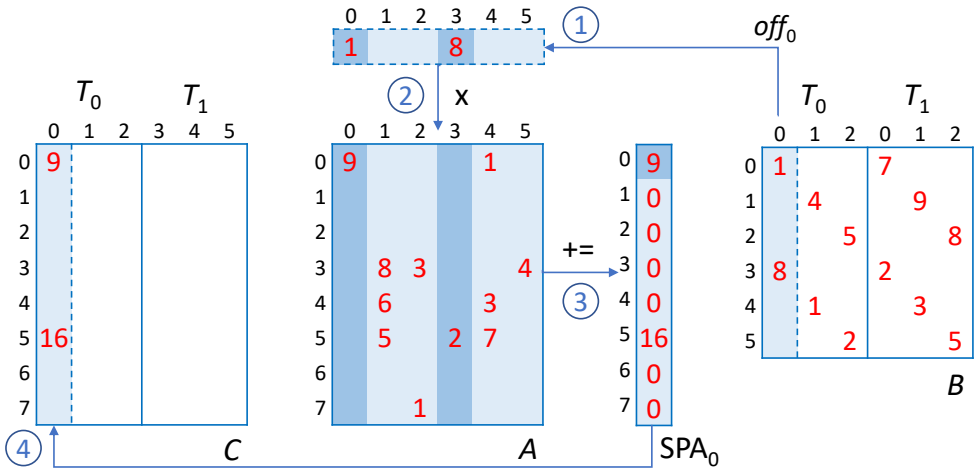


(b) CSR data structure (c) CSR SpMM pseudocode

Figure 32: Parallel Left SpMM C=A x B for **data parallelism** using two threads (t=2, i.e., Tk is the kth thread). (a) In data parallelism matrices are stored in CSR and each thread multiplies a row of Ak by the entire B to produce a row of Ck. (b) CSR storage for matrices A, B, and C. (c) pseudocode of the left SpMM algorithm.

5.2.1 Data Parallelism with Left SpMM

Data parallelism partitions the input A into t partitions where each thread processes a separate partition independently. Since data parallelism horizontally partitions the input instances, a row-major format like CSR perfectly fits this parallelism. Figure 32a depicts the **SPA-based Gustavson's left SpMM algorithm with CSR** for **data parallelism**. In this algorithm, (1) each thread T_k extracts a row from A_k (its partition in A), and (2) multiplies it by the entire B, (3) while accumulating in SPA_k, and (4) finally outputs a row



(a) Model parallelism column-by-column right SpMM using CSC

A CSC

JA_A	0	1	4	6	7	10	11				
IA_A	0	3	4	5	3	7	5	0	4	5	3
VA_A	9	8	6	5	3	1	2	1	3	7	4

T_0: B_0 CSC

JA_B	0	2	4	6		
IA_B	0	3	1	4	2	5
VA_B	1	8	4	1	5	2

T_1: B_1 CSC

JA_B	0	2	4	6		
IA_B	0	3	1	4	2	5
VA_B	7	2	9	3	8	5

C CSC

JA_C	0	2	
IA_C	0	5	
VA_C	9	16	

(b) CSC data structure

```
1 for(j = 0; j < end_k; j++) {
2   for(l = JA_B[j]; l < JA_B[j+1]; l++) {
      r = IA_B[l]; v = VA_B[l]
      for(i = JA_A[r]; i < JA_A[r+1]; i++) {
3       SPA_k[IA_A[i]] += (v x VA_A[i]);
      }
    }
4   SPA_k → (JA_C, IA_C, VA_C, j + off_k)
}
```

(c) CSC SpMM pseudocode

Figure 33: Parallel right SpMM $C{=}A * B$ using two threads ($t{=}2$, i.e., Tk is the kth thread). (a) In **model parallelism** matrices are stored in CSC and each thread multiplies a column of Bk by the entire A to produce a column of C. (b) CSC storage for matrices A, B, and C. (c) pseudocode of the left SpMM algorithm.

of C_k (its partition in C) by storing the nonzero values of SPA_k. Note that C_k acts as the input to the next iteration A_k. Figure 32b shows the CSR representations of A, B, and C where each thread T_k has a separate CSR for its A_k and C_k partitions. Also, B has a single CSR that is shared among threads. Finally, Figure 32c depicts the row-by-row left SpMM algorithm used for data parallelism where end_k is the number of rows in A_k. Note that rows of A_k are re-indexed from 0 to end_k since each partition is allocated separately per thread.

Data parallelism can also be implemented using right SpMM with CSC. However, *at scale* this algorithm is not as efficient as the left SpMM with CSR as it cannot exploit the locality existed in horizontal partitions of data parallelism. Especially, if the partition is balanced and each row is receiving roughly equal number of nonzeros, a row compressed data parallelism like CSR is more efficient. In the experiments, these two variants of data parallelism will be thoroughly compared.

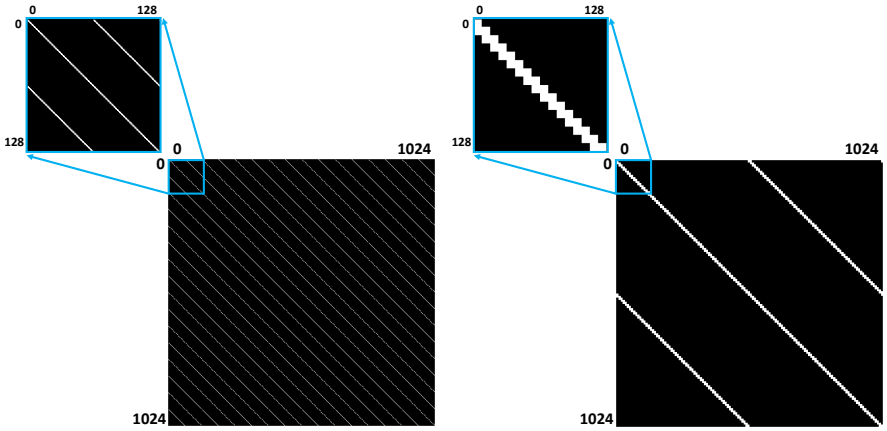
5.2.2 Model Parallelism with Right SpMM

Model parallelism partitions the network B into t partitions where each thread is responsible to execute on a sub-range of columns. The CSC data structure is suitable for model parallelism since this parallelism vertically partitions the network. Figure 33a shows the **SPA-based Gustavson's right SpMM algorithm with CSC** for **model parallelism**. In this algorithm, (1) each thread T_k extracts a column of B_k, and (2) multiplies it by the entire A_k, (3) while accumulating in SPA_k, and (4) finally outputs a column of C_k by storing the nonzeros of the SPA_k. Figure 33b shows the CSC format of A, B, and C with B being vertically partitioned among threads. Note that to allow threads randomly access A and C, a single CSC is allocated for each. Figure 33c shows the column-by-column right SpMM algorithm where end_k is the number of columns in B_k and off_k is the offset of B_k from the beginning of B.

Model parallelism can also be implemented using left SpMM with CSR. However, such an implementation requires an extra step to accumulate partial SPAs per row of A which is extremely expensive. So, the discussion on model parallelism is tailored around right SpMM with CSC and the left SpMM variant of model parallelism is not explored.

5.3 Neural Network Hashing

A common approach to balance nonzero distribution of a matrix is to hash its rows and columns. Considering an input matrix A and a DNN layer B, hashing can be applied to



(a) An unhashed Radix-Net Layer (b) A hashed Radix-Net Layer

Figure 34: First layer of $A0$ of Table 8 with white dots as weights. (a) E.g., column 1 is only connected to rows 1,2, 64, and 65. (b) E.g., column 1 is connected to rows 1-15.

these matrices in different ways including 1) **Input hashing** which hashes the rows of A. 2) **Layers hashing** which hashes columns of A, and rows and columns of Bs in order to achieve locality in accessing DNN. 3) **Input & layers hashing** which hashes rows and columns of both A and B. Input hashing benefits data*data and data*model parallelisms since it produces balanced input partitions by reordering the input rows. Also, it is a cheap way to mitigate the straggler effect. Furthermore, layer hashing may benefit the SpMM algorithm itself when it yields an optimal access pattern. Hence, a hashing function that provides localized access can effectively benefit the cache hierarchy.

Figure 34a shows the first layer of A_0 DNN of Table 8 where each column (neuron) has 32 connections. These connections are spread over the entire column where, e.g., first column has connections in row IDs 1, 2, 64, 65, ..., and second column has connections in row IDs 2, 3, 66, 67, ..., etc. Considering model parallelism, this layout leads to an extremely poor access pattern for its right SpMM algorithm because: 1) Those 32 connections are scattered throughout the columns and thus it forces the SpMM algorithm to almost traverse the entire A for each column of B which is expensive. 2) Connections that are placed in each column are different from the ones placed in its next column. Hence, per column the SpMM algorithm should index a completely different set of columns in A. Based on these two characteristics, the original layout of the DNNs generated by Radix-Net [102] is not

Table 8: Sparse DNNs dataset. *m, n, nnz* & *L* are numbers of instances, features/ neurons, nonzeros, and layers, respectively. First column is used as an ID for DNN scale.

	Input		Network			
			Each Layer		All Layers	
ID	Size ($m \times n$)	nnz	Size ($n \times n$)	nnz	L	nnz
A_0	60 K × 1 K	6.3 M	1 K × 1 K	32 K	120	3.9 M
A_1	60 K × 1 K	6.3 M	1 K × 1 K	32 K	480	15.7 M
A_2	60 K × 1 K	6.3 M	1 K × 1 K	32 K	1920	62.9 M
B_0	60 K × 4 K	25 M	4 K × 4 K	131 K	120	15.7 M
B_1	60 K × 4 K	25 M	4 K × 4 K	131 K	480	62.9 M
B_2	60 K × 4 K	25 M	4 K × 4 K	131 K	1920	251 M
C_0	60 K × 16 K	98.8 M	16 K × 16 K	524 K	120	62.9 M
C_1	60 K × 16 K	98.8 M	16 K × 16 K	524 K	480	251 M
C_2	60 K × 16 K	98.8 M	16 K × 16 K	524 K	1920	1 B
D_0	60 K × 65 K	392 M	65 K × 65 K	209 K	120	251 M
D_1	60 K × 65 K	392 M	65 K × 65 K	209 K	480	1 B
D_2	60 K × 65 K	392 M	65 K × 65 K	209 K	1920	4 B

cache efficient. To address this disadvantage, a 2D bucket hashing algorithm [2] is used to hash rows and columns of the DNN. Figure 34a shows the first layer of A_0 DNN of Table 8 after its rows and columns are hashed. From this figure, e.g., the 32 connections of column IDs 1-6 are to row IDs 1-15, 512-527, and 1024. So, hashing congregate the connections around the diagonal of the matrix instead of being dispersed within the matrix. This layout is extremely in favor of cache hierarchy because same subsets of contiguous rows of A are recurrently being accessed.

5.4 Results

5.4.1 Experimental Settings

5.4.1.1 Datasets Table 8 illustrates the IEEE HPEC sparse DNN challenge dataset [62]. This dataset is generated by RadiX-Net sparse DNN generator [102] with 120, 480, and 1,920 layers; 1,024, 4,096, 16,384, and 65,536 neurons per layer, and 32 connections per neuron.

The input to these DNNs is MNIST dataset [68] with 60,000 instances and respective number of features (equals to the number of neurons).

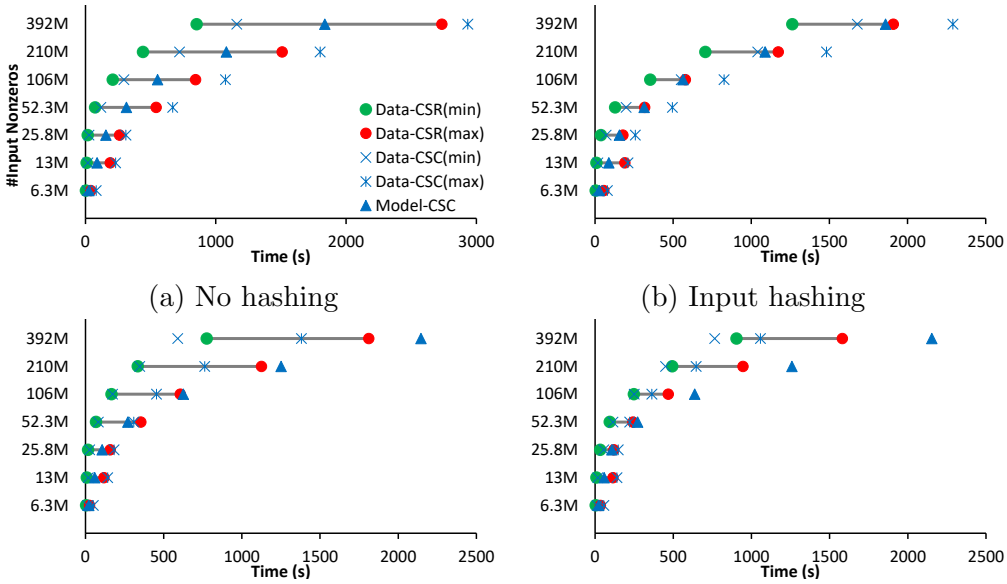
5.4.1.2 Hardware Specifications A cluster of **32 machines (896 cores)** is used to run experiments. Each machine has 28-core Intel Xeon CPU @ 2.60GHz and 192 GB memory. Intel MPI [58] is used for building and executing binaries as well as distributing input partitions among machines. Two MPI processes are launched for each machine (one per socket) and Pthread [71] is used to launch threads inside MPI processes.

5.4.1.3 Software Specifications I developed a new DNN inference engine in C++[1] that supports *SPA-based left and right SpMM kernels* which are backed by *CSR, and CSC formats*. These SpMMs consist of two steps including, *symbolic SpMM step* that estimates the size of the output matrix and allocates memory for it, and the *real SpMM step* that runs the SpMM algorithm and generates the output matrix. Leveraging these kernels, I implement *data parallelism* in two flavors of left and right SpMM and *model parallelism* in right SpMM flavor only. At scale, *data*data* and *data*model* parallelisms are used where data parallelism is first used to distribute the input among multiple processes and then data or model parallelism used inside each process. Last, **2D bucket hashing** [2] with 128 buckets is used to uniformly hash rows/columns of input, network, or both of them. Note if hashing applied to the rows of network, columns of the input should also be hashed.

5.4.2 Single Machine Benchmarking

Figure 35 and 36 are the results for left and right SpMM data parallelism with CSR and CSC, and right SpMM model parallelism with CSC on D_2 DNN of Table 8 using a 28 core machine with $p = 1$ and $t = 28$. The y-axis represents different input sizes from the set of 6.3 M, 13 M, 25.8 M, 53.3 M, 106 M, 210 M, 392 M nonzeros (associated with 1,000, 2,000, 4,000, 8,000, 16,000, 32,000, 60,000 input samples).

[1]The source code is available at https://github.com/hmofrad/DistSparseDNN



(a) No hashing

(b) Input hashing

(c) Layers hashing

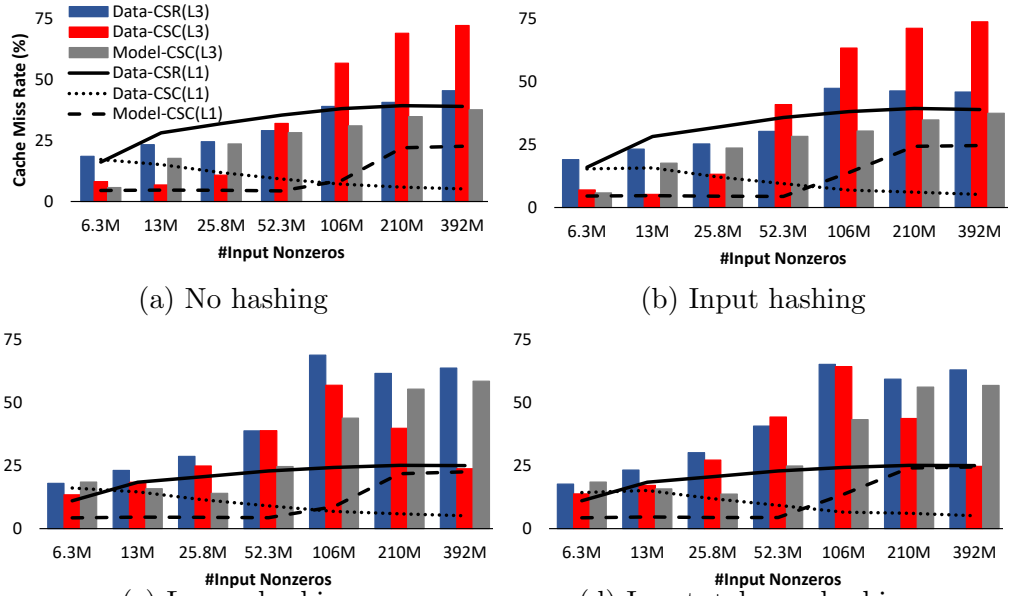
(d) Input + layers hashing

Figure 35: Runtime comparison of different parallelisms processing D2 of Table 8 on a 28 core machine with p=1 and t=28. (a) - (d) are different hashing types with y-axis as the input size varying from 6.3 M (1,000 sample) to 392 M nonzeros (60,000)

.

5.4.2.1 Runtime Variability Figure 35 reports the effect of different hashing types on runtime of different parallelisms. It presents the runtime variation of data parallelism by showing the min and max runtime associated with the fastest and slowest threads. The variation only exists in data parallelisms as threads can progress independently. According to this figure, the variation escalates when inputs are larger which is due to the load imbalance among threads. Although this property allows some thread to finish early, it creates the undesirable effect of stragglers. On the other hand, the end-to-end runtime does not have any variation in model parallelism since threads should strictly abide synchronization barriers to correctly accumulate the results for each layer.

Comparing Figure 35a (no hashing is applied) with Figure 35b (input data is hashed), hashing of the input mitigates the straggler effect by balancing the partitions and hence reducing the variation of runtime in data parallelisms. Input hashing does not affect model parallelism because hashing of the input only reorders the computation of its right SpMM.



(a) No hashing

(b) Input hashing

(c) Layers hashing

(d) Input + layers hashing

Figure 36: Cache utilization of different parallelisms processing D2 of Table 8 on a 28 core machine with p=1 and t=28. (a) - (d) are different hashing types with x-axis as the input size varying from 6.3 M (1,000 sample) to 392 M nonzeros (60,000).

Moreover, comparing 35a (no hashing) with Figure 35c (layers are hashed), hashing of DNN significantly reduces the end-to-end runtime of CSC-based data parallelism along with its runtime variability. By reordering the rows, layer hashing renders rows together which turns out to be exceptionally suited the right SpMM (see Figure 34b). Last, as shown in Figure 35d, if hashing is applied on both input and layers, the runtime for both CSR and CSC data parallelisms improve.

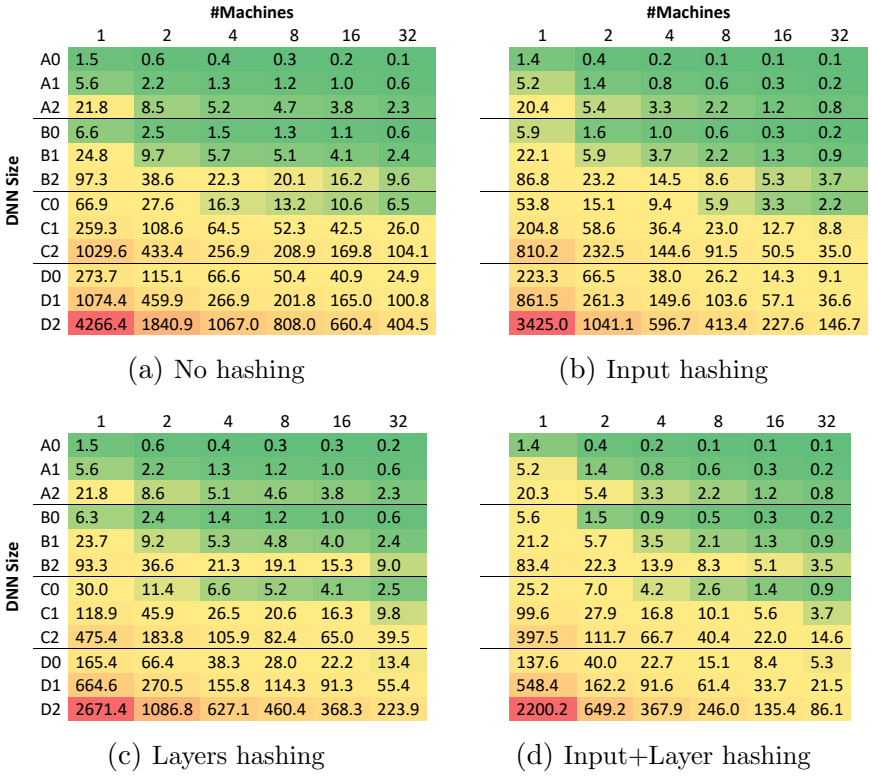
5.4.2.2 Cache Utilization Figure 36 shows L1 and L3 cache miss rate of different parallelisms. As a rule of thumb, increasing the number of input instances from left to right should cause cache miss rate to increase due to putting more stress on the cache hierarchy. However, data parallelism with CSC does not conform to this observation when the input data is large enough. The reason behind this will be discussed shortly.

Comparing Figure 36a with Figure 36b, hashing of the input does not affect the cache utilization. To retrace this, let us take a deeper look into the left and right SpMMs. In

left SpMM (data parallelism with CSR), hashing of the input only reordered the input rows and hence it essentially does not alter the nature of the SpMM algorithm. Moreover, in right SpMM (data and model parallelisms with CSC), input hashing does not provide any advantage as it does not change the overall nonzero distribution (count) of the input columns. Finally, a typical use case of input hashing is for data*data and data*model to achieve load balance when scaling out which will be discussed in the next section.

Inherently, w/ or w/o input hashing data parallelism does not have a good L3 performance because each thread can progress independently. Therefore, at any point of time copies of different layers sits in L3 that may be invalidated/evicted shortly by any thread. However, based on Figure 36a and Figure 36b model parallelism has a decent L3 utilization since all threads are accessing a single shared layer matrix. Oddly enough, when layers (Figure 36c) or both input and layers (Figure 36d) are hashed data parallelism with CSC offers superior L3 utilization with a peak utilization at 106 M nonzeros. This phenomenon is highly accredited to its right SpMM that multiplies L1-friendly hashed DNNs by a smaller input partition that fits into L3.

5.4.2.3 Implications of hashing The left multiplication of data parallelism accesses input rows sequentially and layers rows randomly. This parallelism can benefit from having balanced partitions since balanced partitions (created by hashing) uniformly distribute the input among threads while amortizing the access latency to the DNN rows. Hence, this parallelism has a decent cache utilization. On the other hand, the right multiplication of data and model parallelisms with CSC can highly exploit the underlying structure of DNN (if existed or created by hashing) and boost the cache performance. These parallelisms access the DNN sequentially and the input randomly. Hence, a cache-friendly DNN architecture can perfectly elevate their input's random access pattern to a pseudo-sequential pattern. Last, model parallelism offers better cache utilization for smaller input sizes. This indicates model parallelism would perform better in a distributed setting where many threads process small input partitions. Next section studies the scalability of these parallelisms.



(a) No hashing

DNN Size \ #Machines	1	2	4	8	16	32
A0	1.5	0.6	0.4	0.3	0.2	0.1
A1	5.6	2.2	1.3	1.2	1.0	0.6
A2	21.8	8.5	5.2	4.7	3.8	2.3
B0	6.6	2.5	1.5	1.3	1.1	0.6
B1	24.8	9.7	5.7	5.1	4.1	2.4
B2	97.3	38.6	22.3	20.1	16.2	9.6
C0	66.9	27.6	16.3	13.2	10.6	6.5
C1	259.3	108.6	64.5	52.3	42.5	26.0
C2	1029.6	433.4	256.9	208.9	169.8	104.1
D0	273.7	115.1	66.6	50.4	40.9	24.9
D1	1074.4	459.9	266.9	201.8	165.0	100.8
D2	4266.4	1840.9	1067.0	808.0	660.4	404.5

(b) Input hashing

DNN Size \ #Machines	1	2	4	8	16	32
A0	1.4	0.4	0.2	0.1	0.1	0.1
A1	5.2	1.4	0.8	0.6	0.3	0.2
A2	20.4	5.4	3.3	2.2	1.2	0.8
B0	5.9	1.6	1.0	0.6	0.3	0.2
B1	22.1	5.9	3.7	2.2	1.3	0.9
B2	86.8	23.2	14.5	8.6	5.3	3.7
C0	53.8	15.1	9.4	5.9	3.3	2.2
C1	204.8	58.6	36.4	23.0	12.7	8.8
C2	810.2	232.5	144.6	91.5	50.5	35.0
D0	223.3	66.5	38.0	26.2	14.3	9.1
D1	861.5	261.3	149.6	103.6	57.1	36.6
D2	3425.0	1041.1	596.7	413.4	227.6	146.7

(c) Layers hashing

DNN Size \ #Machines	1	2	4	8	16	32
A0	1.5	0.6	0.4	0.3	0.3	0.2
A1	5.6	2.2	1.3	1.2	1.0	0.6
A2	21.8	8.6	5.1	4.6	3.8	2.3
B0	6.3	2.4	1.4	1.2	1.0	0.6
B1	23.7	9.2	5.3	4.8	4.0	2.4
B2	93.3	36.6	21.3	19.1	15.3	9.0
C0	30.0	11.4	6.6	5.2	4.1	2.5
C1	118.9	45.9	26.5	20.6	16.3	9.8
C2	475.4	183.8	105.9	82.4	65.0	39.5
D0	165.4	66.4	38.3	28.0	22.2	13.4
D1	664.6	270.5	155.8	114.3	91.3	55.4
D2	2671.4	1086.8	627.1	460.4	368.3	223.9

(d) Input+Layer hashing

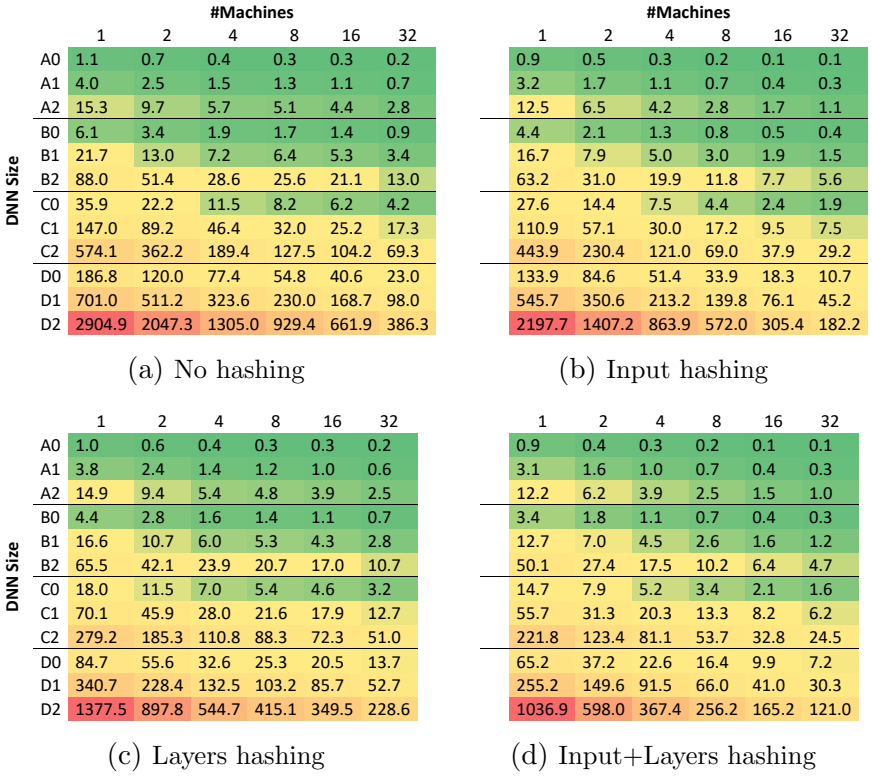
DNN Size \ #Machines	1	2	4	8	16	32
A0	1.4	0.4	0.2	0.1	0.1	0.1
A1	5.2	1.4	0.8	0.6	0.3	0.2
A2	20.3	5.4	3.3	2.2	1.2	0.8
B0	5.6	1.5	0.9	0.5	0.3	0.2
B1	21.2	5.7	3.5	2.1	1.3	0.9
B2	83.4	22.3	13.9	8.3	5.1	3.5
C0	25.2	7.0	4.2	2.6	1.4	0.9
C1	99.6	27.9	16.8	10.1	5.6	3.7
C2	397.5	111.7	66.7	40.4	22.0	14.6
D0	137.6	40.0	22.7	15.1	8.4	5.3
D1	548.4	162.2	91.6	61.4	33.7	21.5
D2	2200.2	649.2	367.9	246.0	135.4	86.1

Figure 37: Runtime (s) of data*data parallelism with CSR for different hashings (1-32 nodes).

5.4.3 Wide-scale Benchmarking

This section gives a comprehensive overview of the performance of data*data parallelism with CSR and CSC (left and right SpMM), and data*model parallelism with CSC (right SpMM) using different types of hashing.

Figures 37 - 39 show the results of data*data parallelism (CSR & CSC) and data*model parallelism (CSC) on DNNs reported in Table 8. X-axis represents the number of machines (cluster scalability) and y-axis represents the DNN size (data scalability). Results are shown using heatmaps to improve data visualization. From this figure, data*data with CSR performs best for smaller DNNs (A_0 to B_2), whereas, data*model with CSC produces the best results for larger DNNs (C_0 to D_2). Overall, from this numbers, input hashing improves data*data parallelism a lot by balancing the input partitions to the data parallelism. furthermore, network (layers) hashing helps model parallelism since it hashes the network nonzeros to a distribution highly in favor of right SpMM used in model parallelism.



(a) No hashing (b) Input hashing

(c) Layers hashing (d) Input+Layers hashing

Figure 38: Runtime (s) of data*data parallelism with CSC for different hashings (1-32 nodes).

Figures 37a - 37d shows the result for data*data parallelism with CSR. For D_2 with 32 machines, input, layers, and input & layers hashing offer 2.6×, 1.7×, and 4.7× speedups over the unhashed results, respectively. These results suggest input hashing improves the runtime significantly and its improvement is even reinforced further if combined with layers hashing.

Figures 38a - 38d are the results for data*data parallelism with CSC compression format. From these figures, for D_2 with 32 machines, input, layers, and input & layers hashing offer 2.1×, 1.7×, and 3.2× speedups over the unhashed results, respectively. This parallelism (data*data with CSC) is not as scalable as the CSR variant due to its poor cache efficiency when input partitions are small.

Figures 39a - 39d shows the results obtained from data*model parallelism with CSC format. From these figures, for D_2 with 32 machines, input, layers, and input & layers hashing offer 1.9×, 1.9×, and 3× speedups over the unhashed results, respectively. Both input and DNN hashing can improve the runtime of this parallelism, however, if combined

(a) No hashing (b) Input hashing

(c) Layers hashing (d) (Input+Layers hashing

Figure 39: Runtime (s) of data*model parallelism with CSC for different hashings (1-32 n.).

they can offer a significant runtime improvement.

Different speedup trends can be observed if input and/or DNN are hashed. These effects can be explained in terms of cache performance and load imabalance. A **super-linear speedup** occurs when the number of machines is small and hence cache subsystem is under severe pressure. In this case doubling the number of machines results in more than doubling of the speedup since more cache is available. Conversely, when the number of machines is large, the cache conflict is less hence doubling the number of machines does not have a significant effect on the cache conflict and speedup. Therefore, a **sub-linear speedup** happens when the number of machines is large and the effect of load imbalance kicks in and become dominant (whilst cache conflict is no longer dominant).

5.5 Conclusion

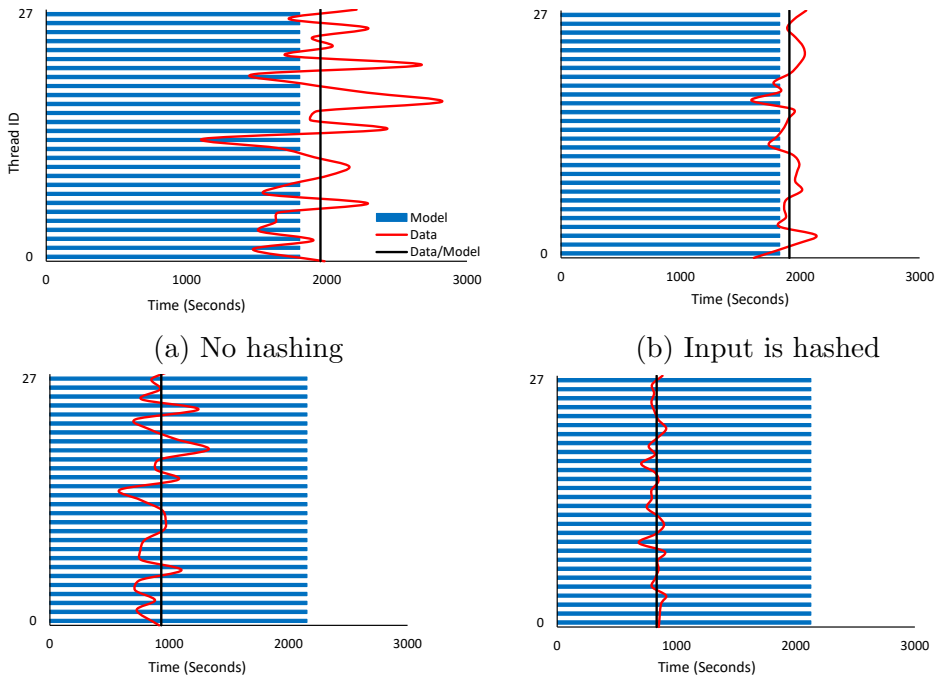
DNN inference is an embarrassingly parallel compute- and memory-intensive task. Data and model parallelisms can be leveraged to run the inference at scale. In this chapter, I thoroughly investigated the internals of data and model parallelisms by focusing on their core SpMM kernels. In addition, I studied the effects of hashing on the performance of these parallelisms. A cluster of 32 machines (896 cores) are used to run inference on sparse neural networks of different sizes. Results suggest data parallelism is suitable for smaller DNNs and model parallelism for larger ones. Also, I found out input hashing improves load balance and network hashing improves cache utilization. Lastly, I observed that these parallelisms can achieve super-linear speedup by hashing the DNN layers.

6.0 Accelerating Distributed Inference of Sparse Deep Neural Networks via Mitigating the Straggler Effect

Once a Deep Neural Network (DNN) is trained, an inference algorithm retains the learning and applies it to batches of data. The trained DNN can be sparse because of pruning or following a preset sparse connectivity pattern. Inference in such sparse networks requires less space and time complexities compared to dense ones. Data **and** model parallelisms are two common parallelism models used for parallelizing training/inference of dense/sparse neural networks, where data parallelism partitions the **input** among multiple threads and data parallelism partitions the **network** among multiple threads. In the previous chapter, I showed that hashing of the input matrix helps data parallelism to balance its computation and hashing of the the DNN provides a better access pattern to model parallelism. However, hashing can not completely remove the imabalance in data parallelism, or data*data and data*model parallelisms. To remedy this imabalance problem, in this chapter, a new hybrid parallelism model for DNN inference is proposed.

Model parallelism efficiently utilizes the Last Level Cache (LLC) but has a heavy synchronization cost because of compulsory reductions per layer. In contrast, data parallelism allows independent execution of partitions but suffers from a straggler effect due to a load imbalance among partitions. I combine data and model parallelisms through a new type of parallelism that I denote **data-then-model**. In data-then-model, each thread starts with data parallelism, thus mitigating the per-layer synchronization cost of model parallelism. After it finishes its partition, it switches to model parallelism to support a slower active thread, hence, alleviating the straggler effect of data parallelism. I compare data-then-model parallelism with data and model parallelisms as well as task-based parallelisms on a HPC cluster. On average, up to 65% speedup is achieved compared to these parallelisms.

The rest of this chapter is organized as follows. Section 6.1 motivates the straggler effect in data parallelism and introduces workarounds to mitigate it. Section 6.2 proposes the new data-then-model parallelism. Section 6.3 reports the results. Finally, Section 6.4 concludes this chapter.



(a) No hashing

(b) Input is hashed

(c) DNN layers are hashed

(d) Input & DNN are hashed

Figure 40: Performance of different parallelisms running different hashing types on D2 DNN of Table 8 using a 28-core CPU (0 to 27 thread IDs). Horizontal bars, zig-zag line, and vertical line show model, data, and data-then-model parallelisms, respectively.

6.1 Motivation

Model parallelism enforces strict synchronization among threads before progressing to the next layer. In this parallelism, each thread takes a separate partition of the layer matrix, multiplies it by the input matrix, and emits a portion of the output. Finally, the coalescing of these partial results makes the input to the next layer. Model parallelism is more sensitive to the sparsity distribution of nonzeros in the layers than in the input. However, DNN layers are usually architected to be balanced [63] and thus threads will not typically linger behind barriers placed at the end of each layer. Contrarily, each thread in data parallelism processes a separate partition of the input matrix and can independently progress. Not tying threads to synchronization barriers causes data parallelism performance to be extremely sensitive to the sparsity distribution of nonzeros in the input partitions which can lead to straggler threads.

Figure 40 demonstrates the runtimes of data and model parallelisms using the D_2 DNN of Table 8 over a 28-core processor sharing the same memory. The horizontal bars of Figure 40a are runtimes of different threads of model parallelism, which finish together because they are synchronized. The zig-zag line represents data parallelism, which shows significant runtime variations due to threads receiving imbalanced partitions. A remedy to these variations is to **hash** the input and/or DNN with the objective of creating balanced partitions. This idea is studied using **2D bucket hashing** [2] in three ways, including hashing of input (Figure 40b), DNN layers (Figure 40c), and both input & DNN layers (Figure 40d).

In Figure 40b, although data parallelism still cannot perform better than model parallelism, input hashing reduces the runtime imbalance significantly by creating uniform input partitions. Next, if the layers are only hashed, data parallelism improves significantly. This is because D_2 belongs to the family of X-Nets [103] and follows a deterministic topology, hence hashing serves in rearranging the topology to a cache-friendly configuration that boosts data parallelism performance. On the contrary, hashing does not help model parallelism, as in model parallelism threads compute balanced vertical chunks of the network which are often designed symmetrically to evenly propagate the weights along the network. Moreover, model parallelism asymptotically does not have runtime variation, let alone that any variation will be amortized over the runtime due to threads being synchronized at each layer. Nonetheless, when only layers are hashed, the variation of data parallelism appears again and the key to virtually remove this variation is to hash both input&DNN as shown in Figure 40d.

From Figures 40a - 40d, the runtime ratios of fastest to slowest thread in data parallelism are 2.5×, 1.4×, 2.3×, and 1.4×. This suggests a full rectification of straggler threads is not possible through hashing and there is still room for other mitigation strategies. As such, I suggest addressing the runtime variation through a scalable hybrid parallelism with minimal overhead. To elaborate, I allow data parallelism to create the runtime difference where some threads finish earlier than others and become idle due to load imbalance. Afterwards, idle threads are identified and directed to support other non-idle (or active) threads, thus gradually and eventually switching altogether into model parallelism. The solid vertical lines in Figure 40 represent this new parallelism which I coin **data-then-model (data/model)** which improves on data parallelism by 1.45×, 1.12×, 1.45×, and 1.10× on different hashings.

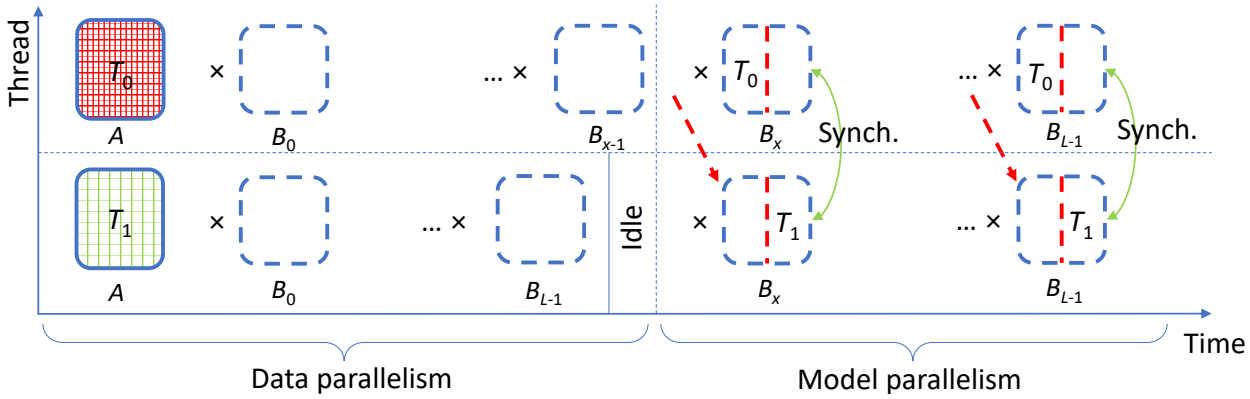


Figure 41: In data-then-model parallelism, all threads start off with data parallelism. Once a thread becames idle, it gets recruited by an active thread and only those threads collectively switch onto model parallelism. Here, $T0$ gets recruited by $T1$.

6.2 Inference using Data-then-Model Parallelism

As discussed earlier, data parallelism suffers from straggler threads due to load imbalance. Typically, task-based parallelisms such as work-sharing [38] or work-stealing [14] are utilized to remedy this problem. In task-based parallelisms, one or multiple queues of tasks protected by lock(s) are created per process or thread where a task is a small input subpartition. On one hand, this involves no data movement and thread migrations since all threads often have data to process. On the other hand, for this to be efficient, it requires scalable queue and locking mechanisms alongside inputting balanced subpartitions, which makes it quite expensive. To mitigate the straggler effect of data parallelism and not completely discount the benefits of model parallelism, **data-then-model parallelism** is introduced which is a new lazy load balancing technique. This new parallelism allows stragglers to be created but enables them to recruit faster threads that finish earlier.

Figure 41 depicts this process where faster thread T_0 gets recruited by slower thread T_1. Initially, forked threads, both T_0 and T_1 execute data parallelism while also looking for idle threads. After T_0 becomes idle, it gets recruited by T_1. Finally, T_1 divides DNN columns into two partitions and delegates half of its SpMM computation to T_0. This shift from data to model parallelism requires careful concurrency control for retaining idle threads.

Algorithm 3 Data-then-model parallelism (vanilla)

1: Input: A_k input partition, B layers and C_k output partition i.e. k is the thread ID. L is the number of layers. *list* is the global shared list of idle threads, and *mutex* & *cond* are the shared lock and condition variable protecting the *list*. $lists_\kappa$ is the list of idle threads recruited by leader T_κ including itself, and $mutexes_\kappa$ & $conds_\kappa$ are T_κ's shared mutex and condition variable.
2: **for** $k = 0$ **to** t **do** `fork`(T_k) ▷ Fork thread T_k
3: DATA(T_k)
4: **if** $l < L$ **then** MODEL(T_κ)
5: **while** ENLIST(T_k) **do** MODEL(T_k)
6: VALIDATE(C_k) ▷ validate T_k's partition

Algorithm 4 Data method

1: **function** DATA(T_k) ▷ Data parallelism
2: **for** $l = 0$ **to** L **do**
3: **if** `not` RECRUIT(T_k) **then** $C_k = A_k \times B_l$

In the next section, I describe my solution to this dynamic problem, which is captured in Algorithm 3. Note that although the following discussion is on data-then-model parallelism, it also extends to *data∗data-then-model parallelism* when having multiple processes, whereby each process will run data-then-model on a separate input partition with no communication. Moreover, thread recruiting strategy can have different flavors which will be discussed in the next section. Lastly, I refer henceforth to any active recruiting thread as a **leader**, to idle threads as **helpers**, and to a list including both as a **group** of threads.

6.2.1 Elastic Locking Mechanism

Data-then-model parallelism utilizes an elastic locking system that encompasses two levels. **The first level** is a global list that helpers (idle threads) enlist in (*list*). This list is protected by a mutex lock (*mutex*) to guarantee mutual execution. Moreover, threads in this list are synchronized using a condition variable (*cond*). Here, after forking each thread T_k (k^{th} thread), it proceeds to **data parallelism** (DATA(T_k) in Algorithm 4), whilst looking for idle threads. Concurrently, if a thread finishes its computation (either data and/or model parallelisms) it enlists in the list of helpers *list* and goes to sleep until it receives a wake-up signal from a leader thread (ENLIST(T_k) in Algorithm 5).

Algorithm 5 Enlist method

```
1: function ENLIST(T_k)                          ▷ Enlisting as an idle thread (helper)
2:     lock(mutex)
3:     list.add(T_k)
4:     if list.size() == t then
5:         broadcast(cond, mutex), unlock(mutex)
6:         return false
7:     else
8:         wait(cond, mutex), unlock(mutex)
9:         if list.size() == t then return false
10:        else return true
```

Algorithm 6 Recruit method

```
1: function RECRUIT(T_κ)                         ▷ Recruiting idle threads (helpers)
2:     lock(mutex)
3:     if list.size() > 0 then
4:         lists_k.insert(list), list.clear()
5:         repartition(B's column indices)
6:         reinit(mutexes_κ, conds_κ, barriers_κ)
7:         broadcast(cond, mutex), unlock(mutex)
8:         return true
9:     unlock(mutex)
10:    return false
```

The second level is an array of lists (*lists*) along with associated arrays of synchronization primitives (*mutexes* and *conds*) that provide decentralized independent synchronization channels to groups of threads. These arrays are of size t (the number of threads per process). Here, $lists_\kappa$ is used to maintain the group information of each T_κ leader (κ^{th} leader). Once helpers are inserted in the designated list for T_κ, they are removed from the global list. Subsequently, columns of the network are partitioned based on the number of threads inside T_κ's group ($lists_\kappa$). Given layers are already compressed using the CSC format (a column major compression), this process is fairly lightweight and only involves redistributing indices of columns of the remaining network layers among the threads of the group (`repartition()` in Algorithm 6). As mentioned, alongside the global lock and condition variable, there is an array of mutexes and condition variables for each leader ($mutexes_\kappa$ and $conds_\kappa$ in Algorithm 6). Since each leader T_κ periodically probes for idle threads, member threads of its group may monotonically grow. Hence, an elastic yet efficient way of group synchronization is necessary.

Algorithm 7 Model method

```
1: function MODEL(T_k)                                   ▷ Model parallelism
2:     for l to L do
3:         RECRUIT(T_κ)                                  ▷ Leader thread T_κ, n = 0
4:         lock(mutexes_κ), n++                          ▷ Sync idle threads (helpers)
5:         if n == lists_κ.size() then
6:             broadcast(conds_κ, mutexes_κ)
7:         else wait(mutexes_κ)
8:         unlock(mutexes_κ)
9:         C_κ = A_κ × B_lk
10:        barrier(barriers_κ)
```

In order to have an elastic locking mechanism, a pair of $mutexes_\kappa$ and $conds_\kappa$ is used to implement a counting semaphore for T_κ's group. In addition, an array of barriers is employed where $barrier_\kappa$ is used to synchronize T_κ's group within and at the end of each layer. In the implementation, these synchronization constructs are (re)initialized on demand and administered independently by each leader T_κ (`reinit(`T_κ`)` in Algorithm 6). Also, allocating these constructs inside arrays allows helpers to easily index them using T_κ thread ID. Finally, after recruiting helpers, all threads in T_κ's group proceed to model parallelism.

Model parallelism (MODEL(T_κ) in Algorithm 7) is executed by threads within a group, however, these threads converged to this method through different ways, either via ENLIST or DATA functions. Also, the leader iteratively probes the global list of helpers to seek more help, and thus threads might be added to the group at any time. To meet synchronization requirements of these situations, $mutexes_\kappa$ and $conds_\kappa$ are used to implement a counting semaphore right before executing the SpMM kernel for model parallelism. It is worth noting that SpMM execution does not need any concurrency control among threads because each thread produces a disjoint partition of the output. Lastly, $barriers_\kappa$ is used within and at the end of each layer to protect memory allocation and synchronize the accumulation.

Task-based parallelisms [38, 14] rely on a central unit to distribute tasks, whereas the introduced locking system makes data-then-model parallelism **decentralized**. Hence, this parallelism converts the execution imbalance of data parallelism into a leverage for model parallelism without requiring any central unit. The locking system is **elastic** because leaders dynamically recruit helpers as they appear in the list of idle threads. Lastly, the system's capacity to help slower threads keeps growing as current leaders turn into future helpers.

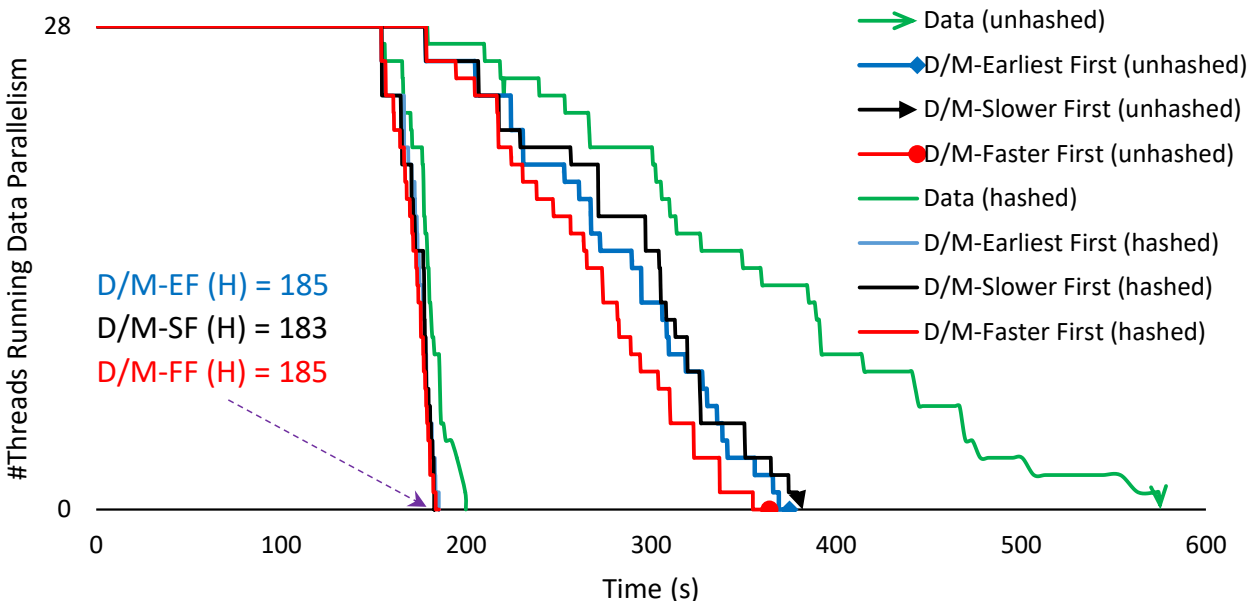


Figure 42: #threads running different parallelisms on C2 DNN of Table 8 on a 28-c. CPU.

6.2.2 Thread Scheduling Algorithms

In data-then-model parallelism, a recruiting algorithm is a scheduling policy that distributes helpers among leaders. Due to the largely unpredictable nature of the inference problem, designing an optimal scheduling strategy is a non-trivial job. Therefore, the following simple scheduling algorithms are explored: 1) **Earliest first**: The first leader that acquires the shared lock will recruit all helpers. In the previous section, this vanilla version is used to describe the locking system. 2) **Slower first**: The first *slow* thread that acquires the shared lock is able to recruit all helpers. Here, the *speed* of a leader T_κ is defined by a score $scores_\kappa$ which is the number of layers that are processed by that thread so far. Also, any thread that satisfies $(scores_\kappa - scores.\texttt{min()}) < \eta$ is considered as a slow thread, where η is the scheduling threshold. The threshold η helps reduce the shared lock contention as the earliest leader that gets the lock and is within the window of the threshold can recruit all threads. 3) **Faster first**: The first *fast* thread to acquire the shared lock recruits all helpers, where any thread that satisfies $(scores.\texttt{max()} - scores_\kappa) < \eta$ is a fast thread.

Figure 42 shows the number of threads executing data parallelism where finished threads become completely idle in data parallelism but turn into helpers in data-then-model parallelism. Here, data parallelism tends to have less runtime variation when the input & DNN

are hashed. Also, for data-then-model the recruiting algorithm gets triggered in the middle or at the end of the execution for unhashed and hashed DNNs. This suggests utilizing a probing threshold θ to allow threads to skip calling the recruiting strategy until reaching a certain layer that makes the probing useful. Hence, this simple tweak increases the volume of useful work by avoiding unnecessary lock contentions when there is no helper to recruit. Note that this tweak is already demonstrated in Figure 42. Lastly, the slower first tends to perform better than other strategies especially when both input & DNN are hashed.

6.3 Results

6.3.1 Experimental Settings

6.3.1.1 Hardware Specifications A cluster of 16 machines is used to run the experiments. Each machine has 28-core Intel Xeon CPU @ 2.60GHz and 192 GB memory. Intel MPI [58] is used for building and executing binaries as well as distributing input partitions among machines. One MPI process is launched for each machine. Lastly, Pthread [71] is used to launch threads inside MPI processes. Experiments are conducted in two scales: single machine and distributed with up to 16 machines (448 cores). Distributed experiments consist of weak scaling (the number of machines scales proportionally to the DNN size), strong cluster scaling (DNN size is fixed and the number of machines is varied), and strong data scaling (the number of machines is fixed and DNN size is varied). Note that the average of maximum execution time is reported here.

6.3.1.2 Implementation Details The sparse DNN inference implementation is open source and freely available[1]. It is written in C/C++ and supports different weight types via template metaprogramming. It allows CPU affinity via Pthread [71] and memory affinity via NUMActl [70]. CPU affinity is strictly implemented for thread scheduling algorithms where each socket has a separate list for idle threads. In addition, threads cannot recruit

[1]The source code is available at https://github.com/hmofrad/DistSparseDNN

idle threads from the other socket *unless* all the remote socket's threads are idle. The NUMA-aware scheduling provides 70% improvement over the NUMA-oblivious version due to prioritizing the remote memory accesses across sockets. The implementation uses CSC [64] along with a SPA-based right multiplication SpMM algorithm [64]. The CSC data structure and the SpMM algorithm are scaled using data parallelism via partitioning. Finally, a custom 4 KB page-aligned memory allocator backed by `mmap()`/`mremap()` is also utilized.

6.3.1.3 Parallelism Models Having p processes and t threads per process, experiments are conducted in two settings of single machine (single process) and distributed (multiple processes). For **single machine experiments**, I study: (1) *Model parallelism* that breaks the network's layers into t partitions, (2) *Data parallelism* that splits the input into t partitions, (3) *Data-then-model* that starts with data parallelism and gradually switches to model parallelism, (4) *Work-sharing* [38] that breaks the input into $t \cdot \sigma$ small partitions and places them in a central queue protected by a single shared lock; σ is the split factor, (5) *Work-stealing* [14] which is similar to the work-sharing, but for each thread employs a separate queue of σ partitions protected by a unique lock; if a thread finishes its partitions it starts probing queues owned by other threads in a circular fashion. Also, I compare against LAGraph [38] which employs manager-worker (a.k.a. work-sharing) strategy. For **distributed experiments**, the input is first partitioned using the number of processes p to scale out and then on each process, the requested parallelism is ran as mentioned above to scale up. For example, data*data parallelism, first divides the input into p partitions, then runs data parallelism with t threads on each partition. In distributed, no communication happens during the inference since partitions are disjoint and the only communication is for final validation of inference.

6.3.1.4 Parameter Settings For data-then-model, the results for slower-first scheduling with NUMA-aware lists of idle threads per socket are reported as it offers the best results for the majority of DNNs. Also, scheduling threshold η and probing threshold θ are set to 4 and 0.3 as they produce the best results. Finally, for work-sharing and work-stealing split factor σ is set as 8.

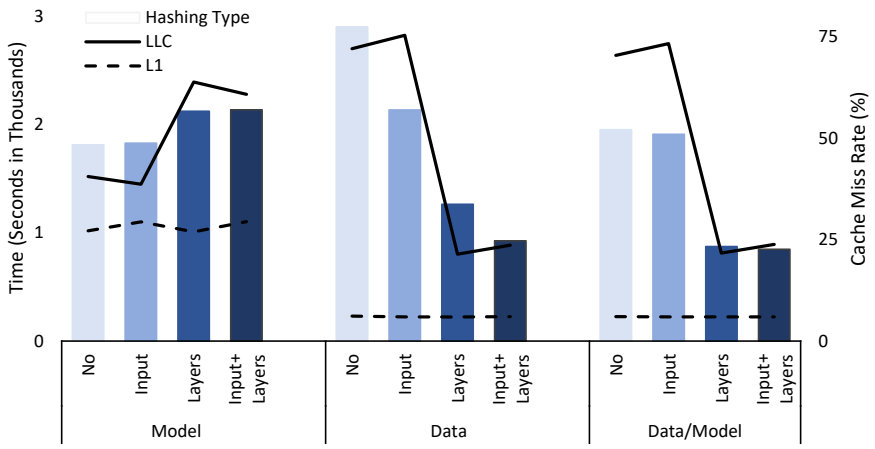


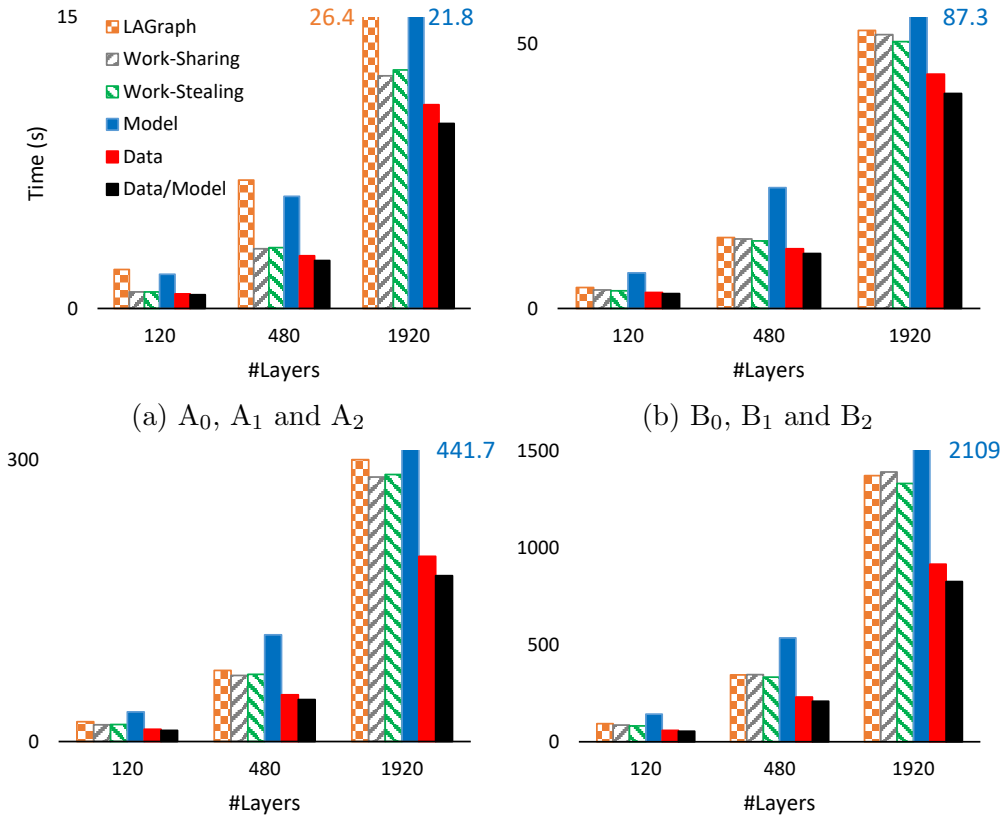
Figure 43: Effect of hashing on runtime (left y-axis) and cache performance (right y-axis) for different parallelisms on D2 (Hashing Type: No = no hashing; Input = input hashing; Layers = layer hashing; and Input + Layers = input & layer hashing).

6.3.1.5 Datasets For the experiments, the IEEE HPEC sparse DNN challenge dataset [62] is utilized. Please refer to Table 8 for more information on datasets.

6.3.2 Studying the Impact of Neural Network Hashing

Figure 43 shows the effects of different types of hashing. For model parallelism hashing does not help (No or Input in the figure) or even hurt (Input and/or Layers) as it already exploits cache locality (particularly LLC) due to all threads accessing a single shared copy of the layer matrix simultaneously. In addition, when dealing with a large DNN which does not fit in cache, model parallelism incurs a huge L1 cache miss rate as threads are randomly accessing the input matrix and continuously invalidating/overwriting L1 entries. On the other hand, data parallelism benefits from hashing a great deal since hashing the input and/or layers produces balanced partitions which alleviate the straggler problem. Finally, like data parallelism, data-then-model also improves with hashing.

From Figure 43, compared to model parallelism, data and data-then-model parallelisms have better L1 cache utilization because their input partition is smaller. Moreover, with No and Input hashing these parallelisms tend to have poor LLC performance which stems from having multiple copies of layers in memory due to threads progressing and accessing different layers individually. Conversely, Layers and Input & Layers hashing significantly improves



(a) A_0, A_1 and A_2 (b) B_0, B_1 and B_2

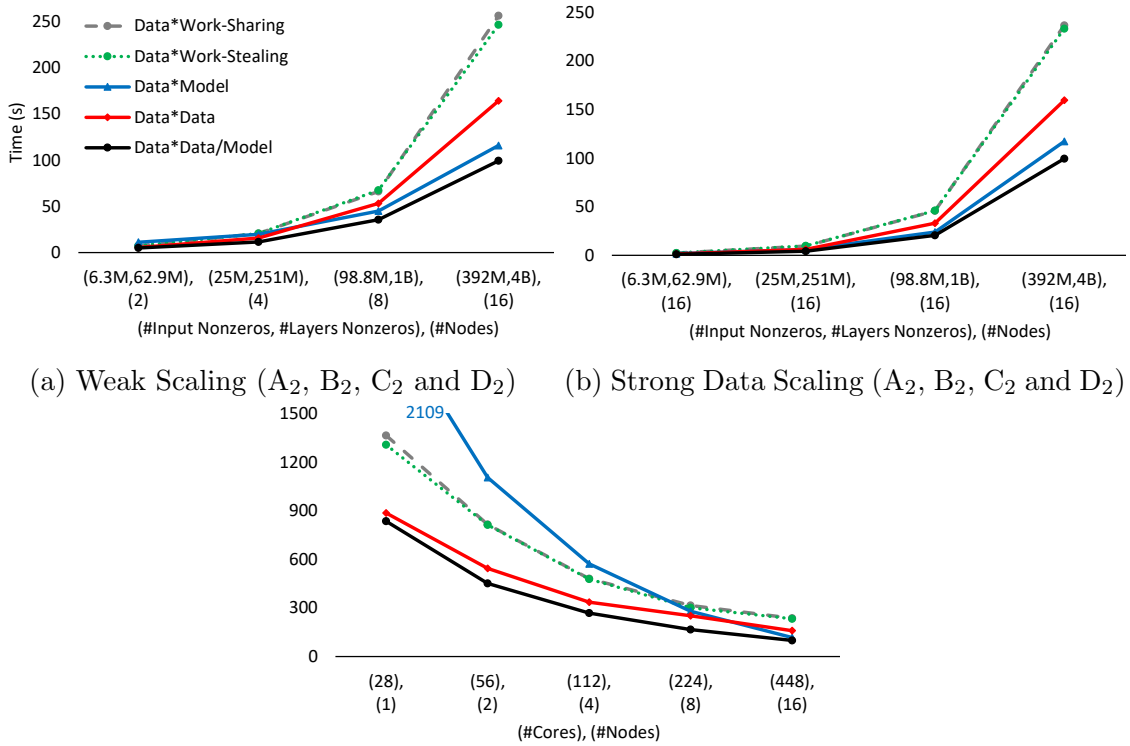
(c) C_0, C_1 and C_2 (d) D_0, D_1 and D_2

Figure 44: Runtime of different parallelisms on a single machine for different DNN sizes.

LLC utilization for these parallelisms. This is because when a DNN generated by RadiX-Net topology [102] is hashed, it offers a pseudo-sequential access pattern to the input matrix.

6.3.3 Single Node Comparison with other Parallelisms

Figure 44 shows the single machine results of different parallelisms on DNNs described in Table 8. Overall, based on the geometric mean of all reported numbers, data-then-model parallelism is about 10% to 65% faster than data parallelism. As threads become idle, data-then-model utilizes idle threads to support slower threads, switching thereby to model parallelism and suppressing pure data parallelism. Here, data parallelism is faster than work-sharing and work-stealing as their threads waste time contending on locks. Model parallelism tends to produce poor results because if the DNN is large it will quickly pollute the cache. Also, from this figure LAGraph [38] demonstrates decent results, but still trails



(a) Weak Scaling (A_2, B_2, C_2 and D_2) (b) Strong Data Scaling (A_2, B_2, C_2 and D_2)

(c) Strong Cluster Scaling (D_2)

Figure 45: Scalability of different parallelism models.

behind data-then-model. Finally, two general observations are: 1) As the number of layers is increased from 120 to 1920, e.g., A_0 to A_2 (deeper DNNs), the runtime increases due to the added depth, 2) As the number of neurons is increased, e.g., from 1k to 4K, scale A to B (wider DNNs), the runtime increases due to the added breadth.

6.3.4 Distributed DNN Inference Performance Analysis

This experiment reports the distributed results of different parallelisms on some selected DNNs from Table 8. Figure 45a shows the **weak scaling results** where the number of machines is increased along with the DNN size. Overall, data-then-model offers superior weak scaling and delivers the best results on different scales. It is followed by model and data parallelisms, and lastly by work-sharing and work-stealing, which perform comparably. Figure 45b illustrates the **strong data scaling results** where the DNN size is increased while the number of machines is fixed. Also, Figure 45c shows the **strong cluster scaling**

results where the number of machines is increased while the DNN size is fixed. In these figures, data-then-model offers the best strong data and cluster scalings. One interesting observation is that in Figure 45c model parallelism under cluster scaling improves significantly as more machines are added. The reason behind this is that smaller partitions consumed by each thread result in less cache thrashing.

6.4 Conclusion

In this chapter, I propose data-then-model parallelism, a lightweight scheme which capitalizes on performance variation exhibited by data parallelism. Threads in data-then-model parallelism start with data parallelism, where faster threads can progress and finish early. Then, instead of terminating these fast threads, data-then-model recruits them to assist stragglers, switching all threads eventually and dynamically to model parallelism. Experiments over single and distributed machines with DNNs as large as 4B nonzeros show that on average data-then-model can deliver up to 65% speedup versus data parallelisms.

7.0 Conclusions and Future Work

7.1 Conclusions

The future of Big Data analytics and classification is shifting toward sparse Big Data which is an undercurrent of Big Data. I do believe that, with correct designs, sparse Big Data can offer agile and cost-effective analytics. Recently, sparse Big Data is becoming more and more important as IT companies want to build/update their data models in faster and cheaper ways on Cloud servers, HPC clusters or lightweight edge devices. However, designing scalable solutions for rendering analytics on sparse Big Data is a challenging task. In this dissertation, I investigated different dimensions of parallel and distributed sparse computing and communication including distributed sparse data structures and primitives. I showed how simple yet efficient changes in partitioning and parallelism models can help accelerate or scale an existing distributed computing strategy.

My first major contribution is that I could identify the gap between sparse matrix and sparse vectors compressions and introduced a new co-compression technique called Triply Compressed Sparse Column (TCSC). TCSC bundles up with the Sparse Matrix - Sparse input and output Vector (SpMSpV^2) primitive offering faster memory accesses, less cache pollution, and asymptotically less space compared to the state-of-the-art compression techniques. Specifically, TCSC reduces the time and space complexities of both sparse matrix and vectors where in a distributed setting reducing the vector sizes yields less accumulation and communication volumes.

My second major contribution is that I combined the partitioning with the scalability direction and introduced the MPI $*$ X parallelism model. The proposed MPI $*$ X leverages a new 2D-thread-based sparse matrix partitioning and placement that deems threads as basic units of computing instead of processes and can scale diagonally over a cluster of machines. The MPI $*$ X model offers balanced computation and communication per thread while reducing the cost of accumulation and synchronization. It leverages the topology and microarchitectural information including Non-uniform Memory Access (NUMA) to enable

faster main memory access and hot caches, and maximize the usage of MPI shared memory transport for communication among threads.

My third contribution is to thoroughly study, data and model parallelisms, two well-known parallelism models for parallel and distributed training/inference of sparse neural networks, where data parallelism parallelizes over the input and model parallelism parallelizes over the DNN. I motivated SpMM as the key primitive behind training/inference of sparse neural networks using either of data or model parallelism and experimentally showed that picking a right compression format for the SpMM algorithm can significantly impact the runtime. I further investigated the effects of hashing in sparse DNN inference and showed input hashing helps reducing the input data imbalance of data parallelism and DNN hashing improves cache utilization of model parallelism.

My fourth contribution is data-then-model parallelism which is a new parallelism for parallel and distributed inference of sparse DNNs. As the core kernel behind the DNN training/inference, SpMM is usually parallelized using data (input partitioning) or model (network partitioning) parallelism. Data parallelism allows threads to progress independently whilst suffering from straggler threads due to load imbalance. To the contrary, Model parallelism incurs a heavy synchronization cost due to compulsory reductions at the end of each layer. To address these limitations, I proposed data-then-model parallelism where all threads start with data parallelism and incrementally switch to model parallelism. When a thread becomes idle, it gets recruited by an active thread which has not yet finished processing its partition and progressively all threads switch into model parallelism. This upcycling of threads removes the straggler effect inherent in data parallelism while postponing the synchronization overhead of model parallelism to the last layers.

Altogether, the contributions of this dissertation provide a set of highly efficient and scalable data structures and algorithms to accelerate and scale big graph analytics and deep learning applications. Additionally, my proposed SpMV and SpMM primitives can be applied to a wide class of sparse linear algebra problems.

7.2 Future Work

In this dissertation, I propose the TCSC matrix compression format in the context of graph analytics. TCSC co-compresses both matrix and vectors to the computation and reduces time and space complexities of the SpMV operations. SpMM is the core kernel behind many scientific applications such as sparse DNN inference or training. Here, CSC is used as the key compression format behind the SpMM kernel implemented for sparse DNN inference. However, with small changes in the SpMM primitive, TCSC can also be utilized as a sparse matrix compression technique for SpMM. When used for SpMM, TCSC co-compresses first and second input matrices and hence can potentially save in both memory and processing power.

Distributed sparse DNN inference is a compute-intensive application that consists of a single forward pass that propagates the weights in the network to infer an instance. SpMM is the key kernel behind inference of sparse DNNs. Although my contributions are mainly tested against inference, all of them are applicable to the training of sparse DNNs as well. Distributed sparse DNN training is a compute- and communication-intensive application which comprises of the iterative execution of a forward pass that propagates the weights and a backward pass that updates the weights. Data-then-model parallelism can be effectively utilized in both forward/backward passes of training where faster threads help slower straggler threads to propagate/update weights faster.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[2] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. La3: a scalable link-and locality-aware linear algebra-based graph analytics system. *Proceedings of the VLDB Endowment*, 11(8):920–933, 2018.

[3] Simon Alford, Ryan Robinett, Lauren Milechin, and Jeremy Kepner. Pruned and structurally sparse neural networks. *arXiv preprint arXiv:1810.00299*, 2018.

[4] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[5] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 313–322. IEEE, 2016.

[6] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl, and Geoffrey E Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[7] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Acm Sigplan Notices*, volume 52(8), pages 193–205. ACM, 2017.

[8] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 688–697. IEEE, 2017.

[9] David Bader, Aydın Buluç, John Gilbert, Joseph Gonzalez, Jeremy Kepner, and Timothy Mattson. The graph blas effort and its implications for exascale. In *SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14)*, 2014.

[10] Satish Balay, Kris Buschelman, Victor Eijkhout, William D Gropp, Dinesh Kaushik, Matthew G Knepley, Lois Curfman McInnes, Barry F Smith, and Hong Zhang. Petsc users manual. Technical report, Technical Report ANL-95/11-Revision 2.1. 5, Argonne National Laboratory, 2004.

[11] Richard F Barrett, Dylan T Stark, Courtenay T Vaughan, Ryan E Grant, Stephen L Olivier, and Kevin T Pedretti. Toward an evolutionary task parallel integrated mpi+ x programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 30–39. ACM, 2015.

[12] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[13] Saman Biookaghazadeh, Yitao Chen, Kaiqi Zhao, and Ming Zhao. Knowledgenet: Disaggregated and distributed training and serving of deep neural networks. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pages 47–49, 2019.

[14] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[15] Hans-J Boehm. Threads cannot be implemented as a library. In *ACM Sigplan Notices*, volume 40, pages 261–268. ACM, 2005.

[16] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *13th ACM WWW*, pages 595–601, 2004.

[17] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[18] Pat Allen Buckland. Numa system with redundant main memory architecture, August 31 2004. US Patent 6,785,783.

[19] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[20] Aydin Buluc and John R Gilbert. *Linear algebraic primitives for parallel computing on large graphs*. University of California, Santa Barbara, 2010.

[21] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

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

[23] Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-i Nieto, and Jordi Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed gpu cluster. *Procedia Computer Science*, 108:315–324, 2017.

[24] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University-Bozeman, College of Engineering, 1969.

[25] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[26] Pak K Chan, Martine DF Schlag, and Jason Y Zien. Spectral k-way ratio-cut partitioning and clustering. *ransactions on computer-aided design of integrated circuits and systems*, 13(9):1088–1096, 1994.

[27] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *10th EuroSys*, 2015.

[28] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):13, 2019.

[29] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[30] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[31] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[32] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[33] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[34] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[35] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[36] Timothy Davis. Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra. *submitted to ACM Trans on Mathematical Software*, 2018.

[37] Timothy A Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[38] Timothy A Davis, Mohsen Aznaveh, and Scott Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse: Graphblas. In *2019 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2019.

[39] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[40] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[41] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[42] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[43] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[44] Ira M Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *Journal of Combinatorial Theory, Series A*, 64(2):189–215, 1993.

[45] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86. ACM, 2018.

[46] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[47] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proceedings of the VLDB Endowment*, 12(4):321–334, 2018.

[48] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[49] Chris Godsil and Gordon F Royle. *Algebraic graph theory*, volume 207. Springer Science & Business Media, 2013.

[50] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2. Usenix, 2012.

[51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[52] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *ACM SIGPLAN Notices*, volume 53(1), pages 246–260. ACM, 2018.

[53] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[54] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[55] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[56] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[57] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[58] Intel. Intel mpi library. `https://software.intel.com/en-us/mpi-library`.

[59] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[60] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[61] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. *MIT Graphchallenge (`http://graphchallenge.mit.edu`)* , 2019.

[62] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[63] Jeremy Kepner, Vikalo Gadepally, Hayden Jananthan, Lauren Milechin, and Sid Samsi. Sparse deep neural network exact solutions. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2018.

[64] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[65] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[66] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[67] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[68] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[69] Shigang Li, Torsten Hoefler, and Marc Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.

[70] Linux. Numa - numa policy library. `http://man7.org/linux/man-pages/man3/numa.3.html`.

[71] Linux. Posix thread (pthread) library. `http://man7.org/linux/man-pages/man7/pthreads.7.html`.

[72] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814, 2015.

[73] Hang Liu and H Howie Huang. Graphene: Fine-grained {IO} management for graph computing. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 285–300, 2017.

[74] Shiwei Liu, Decebal Constantin Mocanu, Amarsagar Reddy Ramapuram Matavalam, Yulong Pei, and Mykola Pechenizkiy. Sparse evolutionary deep learning with over one million artificial neurons on commodity hardware. *arXiv preprint arXiv:1901.09181*, 2019.

[75] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[76] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, pages 1–10, 2014.

[77] Adam Lugowski, Aydın Buluç, John R Gilbert, and Steve Reinhardt. Scalable complex graph analysis with the knowledge discovery toolbox. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5345–5348. IEEE, 2012.

[78] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 195–207, 2017.

[79] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *12th ACM EuroSys*, pages 527–543, 2017.

[80] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[81] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[82] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[83] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1083–1094. Ieee, 2017.

[84] Matlab. Create sparse matrix in matlab. `https://www.mathworks.com/help/matlab/ref/sparse.html`.

[85] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):2383, 2018.

[86] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Efficient distributed graph analytics using triply compressed sparse format. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[87] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2019.

[88] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Accelerating distributed inference of sparse deep neural networks via mitigating the straggler effect. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2020.

[89] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Graphite: a numa-aware hpc system for graph analytics based on a new mpi* x parallelism model. *Proceedings of the VLDB Endowment*, 13(6):783–797, 2020.

[90] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Studying the effects of hashing of sparse deep neural networks on data and model parallelisms. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2020.

[91] Mohammad Hasanzadeh Mofrad, Rami Melhem, and Mohammad Hammoud. Partitioning graphs for the cloud using reinforcement learning. *arXiv preprint arXiv:1907.06768*, 2019.

[92] Mohammad Hasanzadeh Mofrad, Rami Melhem, and Mohammad Hammoud. Revolver: vertex-centric graph partitioning using reinforcement learning. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 818–821. IEEE, 2018.

[93] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[94] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[95] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[96] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony KH Tung, Yuan Wang, et al. Singa: A distributed deep learning platform. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 685–688. ACM, 2015.

[97] OpenMP. The openmp api specification for parallel programming. `https://www.openmp.org/`.

[98] OpenMPI. Open mpi: Open source high performance computing. https://www.open-mpi.org/.

[99] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. Multi-gpu graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 479–490. IEEE, 2017.

[100] Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *7th IEEE SASO*, pages 51–60, 2013.

[101] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[102] Ryan Robinett and Jeremy Kepner. Radix-net: Structured sparse matrices for deep neural networks. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2019.

[103] Ryan A Robinett and Jeremy Kepner. Neural network topologies for sparse training. *arXiv preprint arXiv:1809.05242*, 2018.

[104] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[105] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, page 472, 2013.

[106] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[107] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[108] Schedmd. Slurm workload manager. https://slurm.schedmd.com/.

[109] SciPy. Scipy: open-source software for mathematics, science, and engineering. https://docs.scipy.org/doc/scipy/reference/sparse.html.

[110] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[111] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48(8), pages 135–146. ACM, 2013.

[112] Min Si, Antonio J Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Mt-mpi: multithreaded mpi for many-core environments. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 125–134. ACM, 2014.

[113] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[114] Alexandros Stamatakis and Michael Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with mpi, pthreads, and openmp: A performance study. In *IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 424–435. Springer, 2008.

[115] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[116] Saeed Taheri, Ian Briggs, Martin Burtscher, and Ganesh Gopalakrishnan. Difftrace: Efficient whole-program trace analysis and diffing for debugging. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[117] Saeed Taheri, Sindhu Devale, Ganesh Gopalakrishnan, and Martin Burtscher. Parlot: Efficient whole-program call tracing for hpc applications. In *Programming and Performance Visualization Tools*, pages 162–184. Springer, 2017.

[118] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and J Larsson Träff. Mpi at exascale. *Procceedings of SciDAC*, 2:14–35, 2010.

[119] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[120] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *7th ACM WSDM*, pages 333–342, 2014.

[121] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[122] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[123] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51(8), page 11. ACM, 2016.

[124] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[125] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

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

[127] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[128] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50(8), pages 183–193, 2015.

[129] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[130] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.