PARTITIONING MODELS FOR SCALING DISTRIBUTED GRAPH COMPUTATIONS

A DISSERTATION SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

By
Gündüz Vehbi Demirci
August 2019

Partitioning Models for Scaling Distributed Graph Computations
By Gündüz Vehbi Demirci
August 2019

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Cevdet Aykanat(Advisor)

Uğur Doğrusöz

Muhammet Mustafa Özdal

Tayfun Küçükyılmaz

Fahreddin Şükrü Torun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

PARTITIONING MODELS FOR SCALING DISTRIBUTED GRAPH COMPUTATIONS

Gündüz Vehbi Demirci
Ph.D. in Computer Engineering
Advisor: Cevdet Aykanat
August 2019

The focus of this thesis is intelligent partitioning models and methods for scaling the performance of parallel graph computations on distributed-memory systems. Distributed databases utilize graph partitioning to provide servers with data-locality and workload-balance. Some queries performed on a database may form cascades due to the queries triggering each other. The current partitioning methods consider the graph structure and logs of query workload. We introduce the cascade-aware graph partitioning problem with the objective of minimizing the overall cost of communication operations between servers during cascade processes. We propose a randomized algorithm that integrates the graph structure and cascade processes to use as input for large-scale partitioning. Experiments on graphs representing real social networks demonstrate the effectiveness of the proposed solution in terms of the partitioning objectives.

Sparse-general-matrix-multiplication (SpGEMM) is a key computational kernel used in scientific computing and high-performance graph computations. We propose an SpGEMM algorithm for Accumulo database which enables high performance distributed parallelism through its iterator framework. The proposed algorithm provides write-locality and avoids scanning input matrices multiple times by utilizing Accumulo's batch scanning capability and node-level parallelism structures. We also propose a matrix partitioning scheme that reduces the total communication volume and provides a workload-balance among servers. Extensive experiments performed on both real-world and synthetic sparse matrices show that the proposed algorithm and matrix partitioning scheme provide significant performance improvements.

Scalability of parallel SpGEMM algorithms are heavily communication bound. Multidimensional partitioning of SpGEMM's workload is essential to achieve higher scalability. We propose hypergraph models that utilize the arrangement of

processors and also attain a multidimensional partitioning on SpGEMM's workload. Thorough experimentation performed on both realistic as well as synthetically generated SpGEMM instances demonstrates the effectiveness of the proposed partitioning models.

Keywords: Graph partitioning, propagation models, information cascade, social networks, randomized algorithms, scalability, databases, accumulo, graphulo, parallel and distributed computing, sparse matrices, sparse matrix-matrix multiplication, SpGEMM, matrix partitioning, data locality, hypergraph partitioning, numerical linear algebra.

ÖZET

TÜRKÇE BAŞLIK

Gündüz Vehbi Demirci
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Cevdet Aykanat
Ağustos 2019

Bu tezin odak noktası, dağıtık bellekli sistemlerde paralel çizge hesaplamalarının performansını ölçeklendirmek için akıllı bölümleme modelleri ve yöntemleridir. Bu kapsamda dağıtık veritabanları, sunucular arasında veri-yerelliği ve iş-yükü dengesi sağlamak için grafik bölümlemeden yararlanır. Veritabanında gerçekleştirilen bazı sorgular, birbirlerini tetikleyen sorgular nedeniyle ardışıklık gösterebilir. Mevcut bölümleme yöntemleri, grafik yapısını ve sorgu-iş yükünün kayıtlarını dikkate almaktadır. Bu çalışmada ardışıklık gösteren işlemler sırasında sunucular arasındaki toplam iletişim maliyetini en aza indirgemek amacıyla ardışıklığa duyarlı grafik bölümleme problemi ortaya konmaktadır. Bu haliyle tarafımızdan büyük ölçekli bölümleme için girdi olarak kullanmak üzere grafik yapısını ve ardışık işlemleri birlikte değerlendiren rastsal bir algoritma önerilmektedir. Gerçek sosyal ağları temsil eden çizgeler üzerinde yapılan deneyler, önerilen çözümün bölümlendirme hedefleri açısından etkinliğini göstermektedir.

Seyrek-genelleştirilmiş-matris çarpımı (SyGEMM), bilimsel hesaplamalarda ve yüksek performanslı çizge hesaplarında kullanılan anahtar bir hesaplama çekirdeğidir. Yineleyici çerçevesi sayesinde yüksek performanslı dağıtık hesaplama yapabilen Accumulo veritabanı için tarafımızdan bir SyGEMM algoritması önerilmektedir. Önerilen algoritma, Accumulo'nun toplu tarama özelliğini ve sunucu düzeyinde paralellik yapılarını kullanarak yazma-yerelliği sağlar ve matrislerini birden fazla kez taranmasına neden olmaz. Ayrıca bu çalışmada sunucular arasında toplam iletişim hacmini azaltan ve iş yükü dengesi sağlayan bir matris bölümleme şeması önerilmektedir. Konuya dönük olarak gerçek problemlerde ortaya çıkan ve sentetik olarak üretilen seyrek matrisler üzerinde yapılan kapsamlı deneyler, önerilen algoritmanın ve matris bölümleme şemasının önemli performans iyileştirmeleri sağladığını göstermektedir.

Paralel SyGEMM algoritmalarının ölçeklendirilebilirliği yoğun bir şekilde iletişim işlemlerine bağlıdır. SyGEMM'in iş yükünün çok boyutlu bölümlenmesi

daha yüksek ölçeklenebilirlik elde etmek için şarttır. Bu itibarla işlemcilerin dizilimini dikkate alarak ve aynı zamanda SyGEMM'in iş yükü üzerinde çok boyutlu bölümleme elde eden hiperçizge modelleri tarafımızdan önerilmektedir. Yapılan kapsamlı deneyler, önerilen bölümleme modellerinin etkinliğini göstermektedir.

Anahtar sözcükler: Çizge bölümleme, yayılma modelleri, sosyal ağlar, rastsal algoritmalar, ölçeklenebilirlik, veritabanları, accumulo, graphulo, paralel ve dağıtık hesaplama, seyrek matrisler, seyrek matris-matris çarpımı, SyGEMM, matris bölümleme, veri yerelliği, hiperçizge bölümleme, doğrusal cebir.

Acknowledgement

I would like to express my gratitude to Prof. Dr. Cevdet Aykanat for his guidance and continuous support of my Ph.D studies. His guidance helped me in all the time of research and writing of this thesis. I would like to thank Prof. Dr. Hakan Ferhatosmanoglu for his invaluable contributions in our joint works. Many thanks to my dissertation committee members for their great feedbacks on this thesis. I would like to express my deepest gratitude to my family for their great support and understanding throughout this challenging work. I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) 1001 program for supporting me in projects EEEAG-115E512 and EEEAG-116E043.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We consider graph/hypergraph partitioning models for scaling the performance of parallel graph computations on distributed-memory systems.

Graph partitioning methods are utilized for distributed databases to provide data-locality and workload-balance among servers [5–9]. Partitioning is generally performed by considering graph structure and query workload [3, 4, 10–13]. Online social networks (OSNs) are popular graph database applications where users are represented by vertices and edges/hyperedges represent their links. Graph/Hypergraph partitioning tools (e.g., Metis [14], Patoh [15]) are used for partitioning the graph data by considering social ties and link activities. Users are assigned to servers determined by the partitioning and contents related to a user are stored by the respective server.

Some queries performed on a database may trigger the execution of further operations. For example, users in OSNs frequently share contents generated by others, and therefore; a social network application's query workload may include re-sharing operations in the form of cascades. The database needs to copy the re-shared contents to the servers containing the users that will eventually need to access this content. Moreover, a cascade process may involve users that are not necessarily the neighbors of the originator. Therefore, if the link activities

are considered independently, a database partitioning obtained through considering only the graph structure would not directly capture the underlying cascade processes (content propagation processes). For this purpose, we develop a cascade-aware graph partitioning to minimize the propagation traffic between servers while providing a workload-balance. We address the connection between the partitioning of cascade-aware and other graph partitioning objectives.

Designing efficient parallel graph algorithms is considered to be a difficult task due to the requirement of irregular data accesses and high communication-to-computation ratios in parallel graph algorithms [1, 16]. Graph Basic Linear Algebra Subprogram (GraphBLAS) specification enables a broad variety of graph algorithms to be recast in terms of sparse linear algebraic operations. Therefore, efficient parallelization of GraphBLAS kernels is beneficial for distributed graph computations. Sparse-general-matrix-multiplication (SpGEMM) forms a basis for GraphBLAS specification and enables expressing GraphBLAS primitives by replacing multiplication and addition operations with user-defined operations [27].

An SpGEMM algorithm for Accumulo NoSQL database is provided in Graphulo library [24] to implement GraphBLAS kernels and enable big data computations inside a database system [27]. We seek for alternatives to enhance SpGEMM's efficiency in the Accumulo database and propose a new SpGEMM algorithm. Additionally, we also propose a matrix partitioning scheme for the proposed SpGEMM algorithm to optimize the total communication volume and workload-balance among servers. The algorithm provided in Graphulo library has a trade-off between achieving write-locality and accessing entries of input matrices multiple times. Our solution mitigates this trade-off effectively and offers significant improvements. The efficiency of the proposed algorithm is further improved by utilizing the proposed matrix partitioning scheme.

A broad research is made for improving performance of SpGEMM on distributed and shared-memory parallel computing platforms [1,28,29]. Some of the studies are concentrated on partitioning SpGEMM's workload among processors that are logically arranged as multidimensional grids (i.e., one-dimensional (1D),

2D and 3D SpGEMM algorithms) [1, 2]. Given the problem's sparsity structure, graph/hypergraph partitioning models to provide effective data and task distribution among processors are also considered [28, 30–32]. However, these partitioning methods are not fully concerned with the dimensionality of the processor arrangements and the partitioning is performed considering 1D-parallel SpGEMM algorithms

A major drawback of the previously proposed partitioning models is that 1D SpGEMM algorithms face communication bottlenecks as the number of processors increases, which is due to the significant increase in the number of messages handled by processors. The number of messages exchanged by processors may introduce significant latency overheads, thus reducing the effectiveness and scalability of parallel algorithms. These overheads can be alleviated by considering extra dimensions in processor grids. We propose Hypergraph partitioning-based models that integrate the dimensionality available in processor grids and also attain multidimensional partitioning on SpGEMM's workload. Our partitioning models encode the communication and computational requirements of 2D and 3D SpGEMM algorithms and considerably improve the scalability and efficiency of these algorithms.

The rest of the thesis is organized as follows: Chapter 2 provides background information on graph and hypergraph partitioning problems. Chapter 3 introduces the cascade-aware graph partitioning for large graph databases. Chapter 4 discusses scaling sparse matrix-matrix multiplication algorithm for the Accumulo database. Chapter 5 presents hypergraph partitioning models for 2D and 3D SpGEMM algorithms. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background

2.1 Graph partitioning

Let $G = (V, E)$ be an undirected graph such that each vertex $v_i \in V$ has weight $w(v_i)$ and each undirected edge $e_{ij} \in E$ connecting vertices v_i and v_j has cost $cost(e_{ij})$. Generally, a K-way partition $\Pi = \{V_1, V_2 \dots V_K\}$ of G is defined as follows: Each part $V_k \in \Pi$ is a non-empty subset of V, all parts are mutually exclusive (i.e., $V_k \cap V_m = \emptyset$ for $k \neq m$) and the union of all parts is V (i.e., $\bigcup_{V_k \in \Pi} V_k = V$).

Given a partition Π, weight $W(V_k)$ of a part V_k is defined as the sum of the weights of vertices belonging to that part (i.e., $W(V_k) = \sum_{v_i \in V_k} w(v_i)$). The partition Π is said to be balanced if all parts $V_k \in \Pi$ satisfy the following balancing constraint:

$$W(V_k) \leq W_{avg}(1 + \epsilon), \quad \text{for } 1 \leq k \leq K \tag{2.1}$$

Here, W_{avg} is the average part weight (i.e., $W_{avg} = \sum_{v_i \in V} w(v_i)/K$) and ϵ is the maximum imbalance ratio of a partition.

An edge is called cut if its endpoints belong to different parts and uncut otherwise. The cut and uncut edges are also referred to as external and internal edges,

respectively. The cut size $\chi(\Pi)$ of a partition Π is defined as

$$\chi(\Pi) = \sum_{e_{ij} \in \mathcal{E}_{\text{cut}}^{\Pi}} cost(e_{ij}) \tag{2.2}$$

where $\mathcal{E}_{\text{cut}}^{\Pi}$ denotes the set of cut edges.

In the multi-constraint extension of the graph partitioning problem, each vertex v_i is associated with multiple weights $w^c(v_i)$ for $c = 1, \ldots, C$. For a given partition Π, $W^c(V_k)$ denotes the weight of part V_k on constraint c (i.e., $W^c(V_k) = \sum_{v_i \in V_k} w^c(v_i)$). Then, Π is said to be balanced if each part V_k satisfies $W^c(V_k) \leq W_{avg}^c(1+\epsilon)$, where W_{avg}^c denotes the average part weight on constraint c.

The graph partitioning problem, which is an NP-Hard problem [33], seeks to compute a partition Π^* of G that minimizes the cut size $\chi(\cdot)$ in Eq. (2.2) while satisfying the balancing constraint in Eq. (2.1) defined on part weights.

2.2 Hypergraph partitioning

A hypergraph $H = (V, N)$ is defined as a two-tuple of vertex set V and net set N. A hypergraph is the generalization of a graph, where each net connects possibly more than two vertices and the set of vertices connected by a net n_j is represented by $Pins(n_j)$. Each vertex $v_i \in V$ is associated with C weights $w^c(v_i)$ for $c = 1, \ldots, C$, and each net $n_j \in N$ is associated with $cost(n_j)$.

$\Pi = \{V_1, V_2 \cdots V_K\}$ is a K-way partition of H if vertex parts are mutually disjoint and exhaustive. In Π, a net n_j connecting at least one vertex in a part is said to connect that part. The connectivity set $\Lambda(n_j)$ and the connectivity $\lambda(n_j) = |\Lambda(n_j)|$ of net n_j are respectively defined as the set of parts and the number of parts connected by n_j. A net n_j that connects more than one part (i.e., $\lambda(n_j) > 1$) is said to be cut and uncut otherwise. Then the connectivity cut size of Π is defined as

$$\text{CutSize}(\Pi) = \sum_{n_j \in N} cost(n_j) \times (\lambda(n_j) - 1) \tag{2.3}$$

In a given partition Π, the cth weight $W^c(V_k)$ of a part V_k is

$$W^c(V_k) = \sum_{v_i \in V_k} w^c(v_i) \tag{2.4}$$

Π is said to be balanced if it satisfies

$$W^c(V_k) \leq W^c_{avg}(1 + \epsilon^c), \quad \forall\, V_k \in \Pi \text{ and } c = 1 \ldots C, \tag{2.5}$$

where $W^c_{avg} = \sum_{v_i \in V} w^c(v_i)/K$ is the average part weight on the cth vertex weights and ϵ is the maximum allowed imbalance ratio on part weights.

The multi-constraint hypergraph partitioning problem is defined as finding a K-way partition with the objective of minimizing the cut size given in (2.3) and the constraint of maintaining balance(s) on the part weights given in (2.5).

Chapter 3

Cascade-Aware Partitioning of Large Graph Databases

Distributed graph databases employ partitioning methods to provide data locality for queries and to keep the load balanced among servers [5–9]. Online social networks (OSNs) are common applications of graph databases where users are represented by vertices and their connections are represented by edges/hyperedges. Partitioning tools (e.g., Metis [14], Patoh [15]) and community detection algorithms (e.g., [35]) are used for assigning users to servers. The contents generated by a user are typically stored on the server that the user is assigned.

Graph partitioning methods are designed using the graph structure, and the query workload (i.e., logs of queries executed on the database), if available [3,4,10–13]. Some queries performed on the database may trigger further operations. For example, users in OSNs frequently share contents generated by others, which leads to a propagation/cascade of re-shares (cascades occur when users are influenced by others and then perform the same acts) [36–38]. The database needs to copy the re-shared contents to the servers that contain the users who will eventually need to access this content (i.e., at least a record id of the original content need to be transferred).

see [34] for the original work

Many users in a cascade process are not necessarily the neighbors of the originator. Hence, the graph structure, even with the influence probabilities, would not directly capture the underlying cascading behavior, if the link activities are considered independently. We first aim to estimate the cascade traffic on the edges. For this purpose, we present the concept of random propagation trees/forests that encodes the information of propagation traces through users. We then develop a cascade-aware partitioning that aims to optimize the load balance and reduce the amount of propagation traffic between servers. We discuss the relationship between the cascade-aware partitioning and other graph partitioning objectives.

To get insights on the cascade traffic, we analyzed a query workload from `Digg`, a news sharing-based social network [39]. The data include cascades with a depth of up to six links, i.e., the maximum path length from the initiator of the content to the users who eventually get the content. When we partitioned the graph by just minimizing the number of links straddling between 32 balanced partitions (using Metis [14]), the majority of the traffic remained between the servers, as opposed to staying local. This traffic goes over a relatively small fraction of the links. Only 0.01% of the links occur in 20% of the cascades, and these links carry 80% of the traffic observed in these cascades. It is important to identify the highly active edges and avoid placing them crossing the partitions.

We draw an equivalence between minimizing the expected number of cut edges in a random propagation tree/forest and minimizing communication during a random propagation process starting from any subset of users. A probability distribution is defined over the edges of a graph, which corresponds to the frequency of these edges being involved in a random propagation process. #P-Hardness of the computation of this distribution is discussed and a sampling-based method, which enables estimation of this distribution within a desired level of accuracy and confidence interval, is proposed along with its theoretical analysis.

Experimentation has been performed both with theoretical cascade models and with real logs of user interactions. The experimental results show that the proposed solution performs significantly better than the alternatives in reducing the amount of communication between servers during a cascade process. While

the propagation of content was studied from the perspective of data modeling, to the best of our knowledge, these models have not been integrated into database partitioning for efficiency and scalability.

3.1 Related Work

Graph partitioning and replication. Graph partitioning has been studied to improve scalability and query processing performances of the distributed data management systems. It has been widely used in the context of social networks. Pujol et al. [3] propose a social network partitioning solution that reduces the number of edges crossing different parts and provides a balanced distribution of vertices. They aim to reduce the amount of communication operations between servers. It is later extended in [10] by considering replication of some users across different parts. SPAR [11] is developed as a social network partitioning and replication middle-ware.

Yuan et al. [4] propose a partitioning scheme to process time-dependent social network queries more efficiently. The proposed scheme considers not only the spatial network of social relations but also the time dimension in such a way that users that have communicated in a time window are tried to be grouped together. Additionally, the social graph is partitioned by considering two-hop neighborhoods of users instead of just considering directly connected users. Turk et al. [13] propose a hypergraph model built from logs of temporal user interactions. The proposed hypergraph model correctly encapsulates multi-user queries and is partitioned under load balance and replication constraints. Partitions obtained by this approach effectively reduces the number of communications operations needed during executions of multicast and gather type of queries.

Sedge [7] is a distributed graph management environment based on Pregel [40] and designed to minimize communication among servers during graph query processing. Sedge adopts a two-level partition management system: In the first level, complementary graph partitions are computed via the graph partitioning tool

Metis [14]. In the second level, on-demand partitioning and replication strategies are employed. To determine cross-partition hotspots in the second level, the ratio of number of cut edges to uncut edges of each part is computed. This ratio approximates the probability of observing a cross-partition query and later is compared against the ratio of the number of cross-partition queries to internal queries in a workload. This estimation technique differs from our approach, since we estimate an edge being included in a cascade process whereas this approach estimates the probability of observing a cross-partition query in a part and does not consider propagation processes.

Leopard is a graph partitioning and replication algorithm to manage large-scale dynamic graphs [5]. This algorithm incrementally maintains the quality of an initial partition via dynamically replicating and reassigning vertices. Nicoara et al. [41] propose Hermes, a lightweight graph repartitioner algorithm for dynamic social network graphs. In this approach the initial partitioning is obtained via Metis and as the graph structure changes in time, an incremental algorithm is executed to maintain the quality of the partitions.

For efficient processing of distributed transactions, Curino et al. [8] propose SCHISM, which is a workload-aware graph model that makes use of past query patterns. In this model, data items are represented by vertices and if two items are accessed by the same transaction, an edge is put between the respective pair of vertices. In order to reduce the number of distributed transactions, the proposed model is split into balanced partitions using a replication strategy in such a way that the number of cut edges is minimized.

Hash-based graph partitioning and selective replication schemes are also employed for managing large-scale dynamic graphs [6]. Instead of utilizing graph partitioning techniques, a replication strategy is used to perform cross-partition graph queries locally on servers. This method makes use of past query workloads in order to decide which vertices should be replicated among servers.

Multi-query optimization. Le et al. [42] propose a multi-query optimization algorithm which partitions a set of graph queries into groups where queries in the same group have similar query patterns. Their partitioning algorithm is based on k-means clustering algorithm. Queries assigned to each cluster are rewritten to their cost-efficient versions. Our work diverges from this approach, since we make use of propagation traces to estimate a probability distribution over edges in a graph and partition this graph, whereas this approach clusters queries based on their similarities.

Influence propagation. Propagation of influence [36] is commonly modeled using a probabilistic model [43, 44] learnt over user interactions [45, 46]. Influence maximization problem is first studied by Domingos and Richardson [47]. Kempe et al. [48] proved that the influence maximization problem is NP-Hard under two influence propagation models such as Independent Cascade (IC) and Linear Threshold (LT) models. The Influence spread function defined in [48] has an important property called submodularity, which enables a greedy algorithm to achieve $(1 - 1/e)$ approximation guarantee for the influence maximization problem. However, computing this influence spread function is proven to be #P-Hard [38], which makes the greedy approximation algorithm proposed in [48] infeasible for larger social networks. Therefore, more efficient heuristic algorithms are targeted in the literature [38, 49–52]. More recently, algorithms that run nearly in optimal linear time and provide $(1 - 1/e)$ approximation guarantee for the influence maximization problem are proposed in [53–55].

The notion of influence and its propagation processes have also been used to detect communities in social networks. Zhou et al. [56] find community structure of a social network by grouping users that have high influence-based similarity scores. Similarly, Lu et al. [57] and Ghosh et al. [58] consider finding community partition of a social network that maximizes different influence-based metrics within communities. Barbieri et al. [59] propose a network-oblivious algorithm making use of influence propagation traces available in their datasets to detect community structures.

3.2 Problem Definition

In this section, we present the graph partitioning problem within the context of content propagation in a social network where the link structure and the propagation probability values associated with these links are given. Let an edge-weighted directed graph $G = (V, E, w)$ represent a social network where each user is represented by a vertex $v_i \in V$, each directed edge $e_{ij} \in E$ represents the direction of content propagation from user v_i to v_j and each edge e_{ij} is associated with a content propagation probability $w_{ij} \in [0, 1]$. We assume that the w_{ij} probabilities associated with the edges are known beforehand as in the case of Influence Maximization domain [48, 49, 54]. Methods for learning the influence/content propagation probabilities between users in a social network are previously studied in the literature [45, 46]. In this setting, a partition Π of G refers to a user-to-server assignment in such a way that a vertex v_i assigned to a part $V_k \in \Pi$ denotes that the user represented by v_i is stored in the server represented by part V_k.

We adopt a widely used propagation model, the IC model, with propagation processes starting from a single user for ease of exposition. As we discuss later, this can be extended to other popular models such as the LT model and propagation processes starting from multiple users as well. Under the IC model, a content propagation process proceeds in discrete time steps as follows: Let a subset $S \subseteq V$ consists of initially active users who share a specific content for the first time in a social network (we assume $|S| = 1$ for ease of exposition). For each discrete time step t, let set S_t consists of users that are activated in time step $t \geq 0$, which indicates that $S_0 = S$ (i.e., a user becomes activated meaning that this user has just received the content). Once activated in time step t, each user $v_i \in S_t$ is given a single chance of activating each of its inactive neighbor v_j with a probability w_{ij} (i.e., user v_i activates user v_j meaning that the content propagates from v_i to v_j). If an inactive neighbor v_j is activated in time step t (i.e., v_j has received the content), then it becomes active in the next time step $t + 1$ and added to the set S_{t+1}. The same process continues until there are no new activations (i.e., until $S_t = \emptyset$).

Kempe et al. [48] define an equivalent process for the IC model by generating an unweighted directed graph g from G by independently realizing each edge $e_{ij} \in E$ with probability w_{ij}. In the realized graph g, vertices reachable by a directed path from the vertices in S can be considered as active at the end of an execution of the IC model propagation process starting with the initially active users in S. As a result of the equivalent process of the IC model, the original graph G induces a distribution over unweighted directed graphs. Therefore, we use the notation $g \sim G$ to indicate that we draw an unweighted directed graph g from the distribution induced by G by using the equivalent process of IC model. That is, we generate a directed graph g via realizing each edge $e_{ij} \in G$ with probability w_{ij}.

Propagation trees. Given a vertex v, we define the propagation tree $I_g(v)$ to denote a directed tree rooted on vertex v in graph g. The tree $I_g(v)$ corresponds to an IC model propagation process, when v is used as the initially active vertex, in such a way that each edge $e_{ij} \in I_g(v)$ encodes the information that the content propagated to v_j from v_i during this process. Here, there can be more than one possible propagation trees for v on g, since g may not be a tree itself. One of the possible trees can be computed by performing a breadth-first-search (BFS) on g starting from vertex v, since IC model does not prescribe an order for activating inactive neighbors of the newly activated vertices. Note that generating a graph g and performing a BFS on a vertex v is equivalent to performing a randomized BFS algorithm starting from the vertex v. The difference between the randomized BFS algorithm and usual BFS algorithm is that each edge $e_{ij} \in E$ is searched with probability w_{ij} in the randomized case. That is, during an iteration of BFS, if a vertex v_i is extracted from the queue, each of its outgoing edge(s) e_{ij} to an unvisited vertex v_j is examined and added to the queue with a probability w_{ij}.

Here we also define a fundamental concept called random propagation tree which is used throughout the text. A random propagation tree is a propagation tree that is generated by two level of randomness: First, a graph g is drawn from the distribution induced by G, then a vertex $v \in V$ is chosen randomly and its propagation tree $I_g(v)$ on g is computed. It is important to note that a random

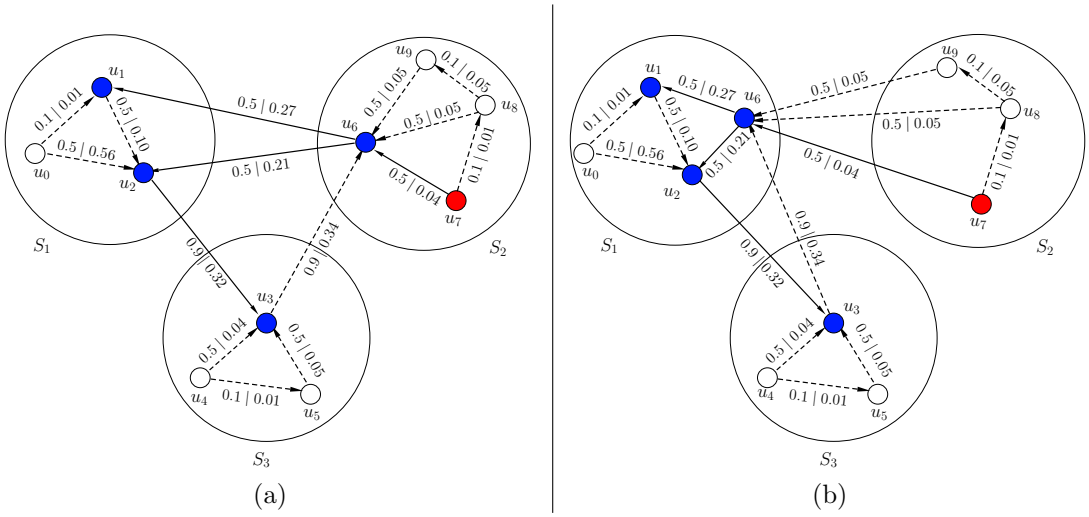


Figure 3.1: An IC model propagation instance starting with initially active user u_7. Dotted lines denote edges that are not involved in the propagation process, straight lines denote edges activated in the propagation process. (a) and (b) display the same social network under two different partitions $\{S_1 = \{u_0, u_1, u_2\}, S_2 = \{u_6, u_7, u_8, u_9\}, S_3 = \{u_3, u_4, u_5\}\}$ and $\{S_1 = \{u_0, u_1, u_2, u_6\}, S_2 = \{u_7, u_8, u_9\}, S_3 = \{u_3, u_4, u_5\}\}$, respectively.

propagation tree is equivalent to an IC model propagation process starting from a randomly chosen vertex. Here, the concept of random propagation trees has similarities to reverse-reachable sets previously proposed in [53, 54]. Reverse-reachable sets are built on transpose G^T of directed graph G by performing a randomized-BFS starting from a vertex v and including all BFS edges. Hence, reverse-reachable sets are different from propagation trees in the ways that they do not constitute directed trees and they are built on the structure of G^T instead of G.

From a systems perspective, if a content propagation occurs between two users located on different servers, we assume this causes a communication operation. This is depicted in Figure 3.1 which displays a graph with its edges denoting directions of content propagations between users. In this figure, two different partitionings of the same social network are given in Figures 3.1a and 3.1b. In Figure 3.1a, users are partitioned among three servers as $S_1 = \{u_0, u_1, u_2\}$, $S_2 = \{u_6, u_7, u_8, u_9\}$ and $S_3 = \{u_3, u_4, u_5\}$. In Figure 3.1b, user u_6 is moved from S_2 to S_1 where S_3 remains the same. In the figure, a content shared by user u_7

propagates through four users u_6, u_1, u_2 and u_3 under the IC model. Here, the straight lines denote the edges along which propagation events occurred and these lines constitute the propagation tree formed by this propagation process (probability values associated with the edges will be discussed later in the next section). The dotted lines denote the edges that are not involved in this propagation process. Therefore, in accordance with our assumption, straight lines crossing different parts necessitate communication operations. For instance, in Figure 3.1a, the propagation of the content from u_7 to u_6 does not incur any communication operation, whereas the propagation of the same content from u_6 to u_1 and u_2 incurs two communication operations. For the whole propagation process initiated by user u_7, the total number of communication operations are equal to 3 and 2 under the partitions in Figures 3.1a and 3.1b, respectively.

Given a partition Π of G and a propagation tree $I_g(v)$ of vertex v on a directed graph $g \sim G$, we define the number of communication operations $\lambda_g^{\Pi}(v)$ induced by the propagation tree $I_g(v)$ under the partition Π as

$$\lambda_g^{\Pi}(v) = |\{e_{ij} \in I_g(v) \mid e_{ij} \in \mathcal{E}_{\text{cut}}^{\Pi}\}|. \tag{3.1}$$

That is, the number of communication operations performed is equal to the number of edges in $I_g(v)$ that are crossing different parts in Π. It can be observed that each different partition Π of G induces a different communication pattern between servers for the same propagation process.

Cascade-aware graph partitioning. In the cascade-aware graph partitioning problem, we seek to compute a partition Π^* of G that achieves the following two objectives:

(i) Under the IC model, the expected number of communication operations to be performed between servers during a propagation process starting from a randomly chosen user should be minimized.

(ii) The partition should distribute the users to servers as evenly as possible in order to ensure a balance of workload among them.

The first objective reflects the fact that many different content propagations, starting from different users or subsets of users, may simultaneously occur during any time interval in a social network and in order to minimize the total communication between servers, the expected number of communication operations in a random propagation process can be minimized. It is worth to mention that, due to the equivalence between random propagation trees and randomized BFS algorithm, the first objective is also equivalent to minimizing the expected number of cross-partition edges traversed during a randomized BFS execution starting from a randomly chosen vertex.

To give a formal definition for the proposed problem, we redefine the first objective in terms of the equivalent process of the IC model. For a given partition Π of G, we write the expected number of communication operations to be performed during a propagation process starting from a randomly chosen user as $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$. Here, subscripts v and $g \sim G$ of the expectation function denote the two level of randomness in the process of generating a random propagation tree. As mentioned above, a random propagation tree $I_g(v)$ is equivalent to a propagation process that starts from a randomly chosen user in the network. Therefore, the expected value of $\lambda_g^{\Pi}(v)$, which denotes the expected number of cut edges included in a random propagation tree, is equivalent to the expected number of communication operations to be performed. Due to this correspondence, computing a partition Π^* that minimizes the expectation $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$ achieves the first objective (i) of the proposed problem. Consequently, the proposed problem can be defined as a special type of graph partitioning in which the objective is to compute a K-way partition Π^* of G that minimizes the expectation $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$ subject to the balancing constraint in Eq. (2.1). That is,

$$\Pi^* = \arg\min_{\Pi} \mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)] \tag{3.2}$$

subject to $W(V_k) \leq W_{avg}(1+\epsilon)$ for all $V_k \in \Pi$. Here, we designate weight $w(v_i) = 1$ of each vertex $v_i \in V$ and define the weight $W(V_k)$ of a partition $V_k \in \Pi$ as the number of vertices assigned to that part (i.e., $W(V_k) = |V_k|$). Therefore, this balancing constraint ensures that the objective (ii) is achieved by the partition Π^*.

Table 3.1: Notations used

Variable	Description
$\Pi = \{V_1, \ldots, V_k\}$	A K-way partition of a graph $G = (V, E)$
V_k	Part k of partition Π
$\chi(\Pi)$	Cut size under partition Π
$I_g(v)$	Random propagation tree
$\lambda_g^{\Pi}(v)$	Communication operations induced by propagation tree $I_g(v)$ under Π
$g \sim G$	Unweighted directed graph g drawn from the distribution induced by G
$\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$	Expected number of communication operations during a propagation process
w_{ij}	Propagation probability along edge e_{ij}
p_{ij}	Probability of edge e_{ij} being involved in a random propagation process
I	The set of random propagation trees generated by the estimation technique
$F_I(e_{ij})$	The number of trees in I that contains edge e_{ij}
N	The size of set I (i.e., $N = \|I\|$)

3.3 Solution

The proposed approach is to first estimate a probability distribution for modeling the propagation and use it as an input to map the problem into a graph partitioning problem. Given an edge-weighted directed graph $G = (V, E, w)$ representing an underlying social network, the first stage of the proposed solution consists of estimating a probability distribution defined over all edges of G. For that purpose, we define a probability value p_{ij} for each edge $e_{ij} \in E$ apart from its content propagation probability w_{ij}. The value p_{ij} of an edge e_{ij} is defined to be the probability that the edge e_{ij} is involved in a propagation process that

starts from a randomly selected user. Equivalently, when a random propagation tree $I_g(v)$ is generated by the process described in Section 3.2, the probability that the edge e_{ij} is included in the propagation tree $I_g(v)$ is equal to p_{ij}. It is important to note that the value w_{ij} of an edge e_{ij} corresponds to the probability that the edge e_{ij} is included in a graph $g \sim G$, whereas the value p_{ij} is defined to be the probability that e_{ij} is included in a random propagation tree $I_g(v)$ rooted on a randomly selected vertex v in graph g. For now, we delay the discussion on the computation of p_{ij} values for ease of exposition, and assume that we are provided with the p_{ij} values. Later in this section, we provide an efficient method that estimates these values.

The function $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$ corresponds to the expected number of cut edges in a random propagation tree $I_g(v)$ under a partition Π. In other words, if we draw a graph g from the distribution induced by G and randomly choose a vertex v and compute its propagation tree $I_g(v)$, then the expected number of cut edges included in $I_g(v)$ is equal to $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$. On the other hand, the value p_{ij} of an edge e_{ij} is defined to be the probability that the edge e_{ij} is included in a random propagation tree $I_g(v)$. Therefore, given a partition Π of G, the function $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)]$ can be written in terms of p_{ij} values of all cut edges in $\mathcal{E}_{\text{cut}}^{\Pi}$ as follows:

$$\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)] = \sum_{e_{ij}\in\mathcal{E}_{\text{cut}}^{\Pi}} p_{ij} \tag{3.3}$$

In Eq. (3.3), the expected number of cut edges in a random propagation tree is computed by summing the p_{ij} value of each edge $e_{ij} \in \mathcal{E}_{\text{cut}}^{\Pi}$, where the value p_{ij} of an edge e_{ij} is defined to be the probability that the edge e_{ij} is included in a random propagation tree. Hence, the main objective becomes to compute a partition Π^* that minimizes the total p_{ij} values of edges crossing different parts in Π^* and satisfies the balancing constraint defined over the part weights. That is,

$$\Pi^* = \arg\min_{\Pi} \sum_{e_{ij}\in\mathcal{E}_{\text{cut}}^{\Pi}} p_{ij} \tag{3.4}$$

subject to the balancing constraint defined in the original problem. As mentioned earlier, each vertex v_i is associated with a weight $w(v_i) = 1$ and part weight

$W(V_k)$ of a part V_k is defined to be the number of vertices assigned to V_k (i.e., $W(V_k) = |V_k|$).

As a result of Eq. (3.4), the problem can be formulated as a graph partitioning problem for which successful tools exist [14, 60]. However, the graph partitioning problem is usually defined for undirected graphs, whereas G is a directed graph and p_{ij} values are associated with the directed edges of G. To that end, we build an undirected graph $G' = (V, E')$ by symmetrizing directed graph G through computing the cost of each edge $e'_{ij} \in E'$ as $c_{ij} = p_{ij} + p_{ji}$.

Let Π be a partition of G'. Since G' and G consist of the same vertex set V, Π induces a set $\mathcal{E}^{\Pi}_{\text{cut}}$ of cut edges for the original graph G. Due to the cost definitions of edges in E', the cut size $\chi(\Pi)$ of G' under partition Π is equal to the sum of the p_{ij} values of cut edges in $\mathcal{E}^{\Pi}_{\text{cut}}$ which is shown to be equal to the value of the main objective function in Eq. (3.2). That is,

$$\chi(\Pi) = \sum_{e_{ij} \in \mathcal{E}^{\Pi}_{\text{cut}}} p_{ij} = \mathbb{E}_{v,g\sim G}[\lambda^{\Pi}_{g}(v)] \tag{3.5}$$

Hence, a partition Π^* that minimizes the cut size $\chi(\cdot)$ of G' also minimizes the expectation $E_{v,g\sim G}[\lambda^{\Pi}_{g}(v)]$ in the original social network partitioning problem. In other words, if the partition Π^* for G' is an optimal solution for the partitioning of G', it is also an optimal solution for Eq. (3.2) in the original problem. Additionally, the equivalence drawn between the graph partitioning problem and the cascade-aware graph partitioning problem also proves that the proposed problem is NP-Hard even the p_{ij} values were given beforehand.

In Figure 3.1, the main objective of cascade-aware graph partitioning is depicted as follows: Each edge in the figure is associated with a content propagation probability along with its computed p_{ij} value (i.e., each edge e_{ij} is associated with "$w_{ij} \mid p_{ij}$"). The partitioning in Figure 3.1a provides a better cut size both in terms of number of cut edges and the total propagation probabilities of edges crossing different parts. However, we assert that the partitioning in Figure 3.1b provides a better partition for objective function 3.2, at the expense of providing worse cut size in terms of other cut size metrics (i.e., the sum of p_{ij} values of cut edges is less in the second partition).

Computation of the p_{ij} values. We now return to the discussion on the computation of the p_{ij} values defined over all edges of G and start with the following theorem indicating the hardness of this computation:

Theorem 1. *Computation of the p_{ij} value for an edge e_{ij} of G is a #P-Hard problem.*

Proof. Let function $\sigma(v_k, v_i)$ denote the probability of there being a directed path from vertex v_k to vertex v_i on a directed graph g drawn from the distribution induced by G. Assume that the only path goes from v_k to v_j is through v_i on each possible g. That is v_j is only connected to v_i in G (This simplifying assumption does not affect the conclusion we draw for the theorem). Hence, the probability of v_i included in a propagation tree $I_g(v_k)$ is $\sigma(v_k, v_i)$. Let p_{ij}^k denote the probability of e_{ij} is included in $I_g(v_k)$. We can compute p_{ij}^k as

$$p_{ij}^k = w_{ij} \cdot \sigma(v_k, v_i) \tag{3.6}$$

since inclusion of e_{ij} in g and formation of a directed path from v_k to v_i on g are two independent events; therefore, their respective probability values w_{ij} and $\sigma(v_k, v_i)$ can be multiplied. As mentioned earlier, the value p_{ij} of an edge e_{ij} is defined to be the probability of edge e_{ij} included in a random propagation tree. Therefore, we can compute the value p_{ij} of an edge e_{ij} as follows:

$$p_{ij} = \frac{1}{|V|} \sum_{v_k \in V} p_{ij}^k \tag{3.7}$$

Here, to compute the p_{ij} value of edge e_{ij}, we sum the conditional probability $\frac{1}{|V|} \cdot p_{ij}^k$ for all $v_k \in V$. Due to the definition of random propagation trees, selections of v_k in a graph $g \sim G$ are all mutually exclusive events with equal probability $\frac{1}{|V|}$. Therefore, we can sum the terms $\frac{1}{|V|} \cdot p_{ij}^k$ for each $v_k \in V$ to compute the total probability p_{ij}.

In order to prove the theorem, we present an equivalence between the computation of function $\sigma(\cdot, \cdot)$ and the s,t-connectedness problem [61], since the p_{ij}

value of an edge e_{ij} depends on the computation of $\sigma(v_k, v_i)$ for all $v_k \in V$. The input to the s,t-connectedness problem is a directed graph $G = (V, E)$, where each edge $e_{ij} \in E$ may fail randomly and independently from each other. The problem asks to compute the total probability of there being an operational path from a specified source vertex s to a target vertex t on the input graph. However, computing this probability value is proven to be a #P-Hard problem [61]. On the other hand, the function $\sigma(v_k, v_i)$ denotes the probability of there being a directed path from v_k to v_i in a $g \sim G$, where each edge in g is realized with probability w_{ij} randomly and independently from other edges. It is obvious to see that the computation of function $\sigma(v_k, v_i)$ is equivalent to the computation of the s,t-connectedness probability (We refer the reader to [38] for a more formal description for the reduction of $\sigma(v_k, v_i)$ to s,t-connectedness problem). This equivalence implies that the computation of function $\sigma(v_k, v_i)$ is #P-Hard even for a single vertex v_k, and therefore implies that the computation of p_{ij} value for any edge e_{ij} is also #P-Hard. □

Theorem 1 states that it is unlikely to devise a polynomial time algorithm which exactly computes p_{ij} values for all edges in G. Therefore, we employ an efficient method that can estimate these p_{ij} values for all edges in G at once. These estimations can be made within a desired level of accuracy and confidence interval; but there is a trade-off between the runtime and the estimation accuracy of the proposed approach. On the other hand, the quality of the results produced by the overall solution is expected to increase with increasing accuracy of the p_{ij} values.

The proposed estimation technique employs a sampling approach that starts with generating a certain number of random propagation trees. Recall that a random propagation tree is generated by first drawing a directed graph $g \sim G$ and then computing a propagation tree $I_g(v)$ on g for a randomly selected vertex $v \in V$. Let I be the set of all random propagation trees generated for estimation and let N be the size of this set (i.e., $N = |I|$). After forming the set I, the value p_{ij} of an edge e_{ij} can be estimated by the frequency of that edge's appearance in random propagation trees in I as follows: Let function $F_I(e_{ij})$ denote the number

of random propagation trees in I that contains edge e_{ij}. That is,

$$F_I(e_{ij}) = |\{I_g(v) \in I \mid e_{ij} \in I_g(v)\}| \tag{3.8}$$

Due to the definition of p_{ij}, the appearance of edge e_{ij} in a random propagation tree $I_g(v) \in I$ can be considered as a Bernoulli trial with success probability p_{ij}. Hence, the function $F_I(e_{ij})$ can be considered as the number of total successes in N Bernoulli trials with success probability p_{ij}, which implies that $F_I(e_{ij})$ is Binomially distributed with parameters N and p_{ij} (i.e., $F_I(e_{ij}) \sim \text{Binomial}(p_{ij}, N)$). Therefore, the expected value of the function $F_I(e_{ij})$ is equal to $p_{ij}N$, which also implies that

$$p_{ij} = \mathbb{E}[F_I(e_{ij})/N] \tag{3.9}$$

As a result of Eq. (3.9), if an adequate number of random propagation trees are generated to form the set I, the value $F_I(e_{ij})/N$ can be an estimation for the value of p_{ij}. Therefore, the estimation method consists of generating N random propagation trees that together form the set I, and computing the function $F_I(e_{ij})$ according to Eq. (3.8) for each edge $e_{ij} \in E$. After computing the function $F_I(e_{ij})$ for each edge e_{ij}, we use $F_I(e_{ij})/N$ as an estimation for the p_{ij} value.

Implementation of the estimation method. We seek an efficient implementation for the proposed estimation method. The main computation of the method consists of generating N random propagation trees. A random propagation tree can be efficiently generated by performing a randomized BFS, starting from a randomly chosen vertex, in G. It is important to note that the randomized BFS algorithm starting from a vertex v is equivalent to drawing a graph $g \sim G$ and performing a BFS starting from the vertex v on g. That is, the randomized BFS algorithm is equivalent to the method introduced in Section 3.2 to generate a propagation tree $I_g(v)$ rooted on v. Therefore, forming the set I can be accomplished by performing N randomized BFS algorithms on G starting from randomly chosen vertices. Moreover, the computation of the function $F_I(\cdot)$ for all edges in E can be performed while forming the set I with a slight modification to the randomized BFS algorithm. For this purpose, a counter for each $e_{ij} \in E$ can be kept in such a way that its value is incremented each time the corresponding

edge is traversed during a randomized BFS execution. This counter denotes the number of times an edge is traversed during the performance of all randomized BFS algorithms. Therefore, after N randomized BFS executions, the function $F_I(e_{ij})$ for an edge e_{ij} is equal to the value of the counter maintained for that edge.

Algorithm. The overall cascade-aware graph partitioning algorithm is described in Algorithm 3.1. In the first line, the set I is formed by performing N randomized BFS algorithms, where the function $F_I(e_{ij})$ is computed for each edge $e_{ij} \in E$ during these randomized BFS executions. In lines 2 and 3, an undirected graph $G' = (V, E')$ is built via composing a new set E' of undirected edges, where each undirected edge $e'_{ij} \in E'$ is associated with a cost of c_{ij} using the estimations computed in the first step. In line 4, each vertex $v_i \in V$ is associated with a weight $w(v_i) = 1$ in order to ensure that the weight of a part is equal to the number of vertices assigned to that part. Lastly, a K-way partition Π of the undirected graph G' is obtained using an existing graph partitioning algorithm and returned as a solution for the original problem. Here, the graph partitioning algorithm is executed with the same imbalance ratio as with the original problem.

Determining the size of set I. As mentioned earlier, the accurate estimation of the p_{ij} values is a crucial step to compute "good" solutions for the proposed problem, since the graph partitioning algorithm used in the second step makes use of these p_{ij} values to compute the costs of edges in G'. The total cost of cut edges in G' represents the value of the objective function in Eq. (3.2). Therefore, the p_{ij} values need to be accurately estimated so that the graph partitioning algorithm correctly optimizes the objective function.

Estimation accuracies of the p_{ij} values depend on the number of random propagation trees forming the set I. As the size of the set I increases, more accurate estimations can be obtained. However, we want to compute the minimum value of N to get a specific accuracy within a specific confidence interval. More formally, let $\hat{p}_{ij}$ be the estimation computed for the p_{ij} value of an edge $e_{ij} \in E$ (i.e.

Algorithm 3.1 Cascade-Aware Graph Partitioning

Input: $G = (V, E, w)$, K, ϵ
Output: Π

1: Form a set I of N random propagation trees by performing randomized BFS algorithms on G and compute $F_I(e_{ij})$ for each $e_{ij} \in E$ according to Eq. (3.8)
2: Build an undirected graph $G' = (V, E')$ where edge set E' is composed as follows:

$$E' = \{e'_{ij} \mid e_{ij} \in E \vee e_{ji} \in E\}$$

3: Associate a cost c_{ij} with each $e'_{ij} \in E'$ as follows:

$$c_{ij} = \begin{cases} F_I(e_{ij})/N + F_I(e_{ji})/N, & \text{if } e_{ij} \in E \wedge e_{ji} \in E \\ F_I(e_{ij})/N, & \text{if } e_{ij} \in E \wedge e_{ji} \notin E \\ F_I(e_{ji})/N, & \text{if } e_{ij} \notin E \wedge e_{ji} \in E \end{cases}$$

4: Associate each vertex $v_i \in V$ with weight $w(v_i) = 1$.
5: Compute a K-Way partition Π of G' using an existing graph partitioning algorithm (using the same imbalance ration ϵ).
6: **return** Π

$\hat{p}_{ij} = F_I(e_{ij})/N$), we want to compute the minimum value of N to achieve the following inequality:

$$Pr[|\hat{p}_{ij} - p_{ij}| \leq \theta \,, \forall e_{ij} \in E] \geq 1 - \delta. \tag{3.10}$$

That is, with a probability of at least $1 - \delta$, we want the estimation $\hat{p}_{ij}$ to be within θ of p_{ij} for each edge $e_{ij} \in E$. For that purpose, we make use of well-known Chernoff [62] and Union bounds from probability theory. Chernoff bound can be used to find an upper bound for the probability that a sum of many independent random variables deviates a certain amount from their expected mean. In this regard, due to the function $F_I(\cdot)$ being Binomial, Chernoff bound can guarantee the following inequality:

$$Pr\left[|F_I(e_{ij}) - p_{ij}N| \geq \xi \, p_{ij}N\right] \leq 2 \, exp(-\frac{\xi^2}{2+\xi} \cdot p_{ij}N) \tag{3.11}$$

for each edge $e_{ij} \in E$. Here, ξ denotes the distance from the expected mean in the context of Chernoff bound.

In Eq. (3.11), dividing both sides of the inequality $|F_I(e_{ij}) - p_{ij}N| \geq \xi \, p_{ij}N$ in

the function $Pr[\cdot]$ by N and taking $\xi = \theta/p_{ij}$ yields

$$\begin{aligned} Pr[|\hat{p}_{ij} - p_{ij}| \geq \theta] &\leq 2\,exp(-\frac{\theta^2}{2p_{ij}+\theta} \cdot N) \\ &\leq 2\,exp(-\frac{\theta^2}{2+\theta} \cdot N) \end{aligned} \tag{3.12}$$

which denotes the upper bound for the probability that the accuracy θ is not achieved for a single edge e_{ij} (the last inequality in Eq. (3.12) follows, since $p_{ij} \leq 1$). Moreover, RHS of Eq. (3.12) is independent from the value of p_{ij} and its value is the same for all edges in E, which enables us to apply the same bound for all of them. However, our objective is to find the minimum value of N to achieve accuracy θ for all edges simultaneously with a probability at least $1 - \delta$. For that purpose, we need to find an upper bound for the probability that there exists at least one edge in E for which the accuracy θ is not achieved. We can compute this upper bound using Union bound as follows:

$$Pr[|\hat{p}_{ij} - p_{ij}| \geq \theta\,, \exists e_{ij} \in E] \leq 2|E|\,exp(-\frac{\theta^2}{2+\theta} \cdot N) \tag{3.13}$$

Here, we simply multiply RHS of Eq. (3.12) by $|E|$, since for each edge in E, the accuracy θ is not achieved with a probability at most $2\,exp(-\frac{\theta^2}{2+\theta} \cdot N)$. In order to achieve the Eq. (3.10), RHS of Eq. (3.13) need to be at most δ. That is,

$$2|E|\,exp(-\frac{\theta^2}{2+\theta} \cdot N) \leq \delta \tag{3.14}$$

Solving this equation for N yields

$$N \geq \frac{2+\theta}{\theta^2} \cdot \ln \frac{2|E|}{\delta} \tag{3.15}$$

which indicates the minimum value of N to achieve θ accuracy for all edges in E with a probability at least $1 - \delta$.

The accuracy θ determines how much error is made by the graph partitioning algorithm while it performs the optimization. As shown in Eq. (3.5), for a partition Π of G' obtained by the graph partitioning algorithm, the cut size $\chi(\Pi)$ is equal to the value of main objective function (3.2). However, the cost values associated with the edges of G' are estimations of their exact values and therefore, the partition cost $\chi(\Pi)$ might be different from the exact value of the objective

function. In this regard, the difference between the objective function and the partition cost can be expressed as follows:

$$\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(v)] - \chi(\Pi) \leq \theta \cdot |\mathcal{E}_{\text{cut}}^{\Pi}| \tag{3.16}$$

Here, the error is computed by multiplying the accuracy θ by the number of cut-edges of G under the partition Π, since for each edge in $\mathcal{E}_{\text{cut}}^{\Pi}$, at most θ error can be made with a probability at least $1-\delta$. Therefore, even if it were possible to solve the graph partitioning problem optimally, the solution returned by Algorithm 3.1 would be within $\theta \cdot |\mathcal{E}_{\text{cut}}^{\Pi}|$ of the optimal solution for the original problem with a probability at least $1-\delta$. Consequently, as the value of θ decreases, the partition obtained by Algorithm 3.1 will incur less error for the main objective function, which will enable the graph partitioning algorithm to perform a better optimization for the original problem.

Complexity analysis. The proposed algorithm consists of two main computational phases. In the first phase, for an accuracy θ with confidence δ, the set I is generated via performing at least $N = \frac{2+\theta}{\theta^2} \cdot \ln \frac{2|E|}{\delta}$ randomized BFS algorithms and each of these BFS executions takes $\Theta(V+E)$ time. The second phase of the algorithm performs the partitioning of the undirected graph G' which is constructed from the directed graph G by using $F_I(e_{ij})$ values computed in the first phase. The construction of the graph G' can be performed in $\Theta(V+E)$ time. The partitioning complexity of the graph G', however, depends on the partitioning tool used. In our implementation we preferred Metis which has a complexity of $\Theta(V+E+K\log K)$, where K being the number of partitions. Therefore, if θ and δ are assumed to be constants, the overall complexity of Algorithm 3.1 to obtain a K-way partition can be formulated as follows:

$$\begin{aligned}\Theta(\frac{2+\theta}{\theta^2}\ln\frac{2|E|}{\delta}(V+E)) + \Theta(V+E+K\log K)& \\ = \Theta((V+E)\log E + K\log K).&\end{aligned} \tag{3.17}$$

Eq. (3.17) denotes serial execution complexity of Algorithm 3.1. The proposed

algorithm's scalability can be improved even further via parallel processing, since the estimation technique is embarrassingly parallel. Given P parallel processors, N propagation trees in I can be computed without necessitating any communication or synchronization (i.e., each processor can generate N/P trees by separate BFS executions). The only synchronization point is needed in the reduction of $F_I(e_{ij})$ values computed by these processors. This reduction operation, however, can be efficiently performed in $\log P$ synchronization phases. Additionally, there exist parallel graph partitioning tools (e.g., ParMetis [63]) which can improve the scalability of the graph partitioning phase.

Extension to the LT model. Even though we have illustrated the problem and solution for the IC model, both our problem definition and proposed solution can be easily extended to other models such as the LT (linear threshold) model. It is worth to mention that the proposed solution does not depend on the IC model or the probability distribution defined over edges (i.e., w_{ij} probabilities). As long as the random propagation trees can be generated, the proposed solution does not require any modification for the use of any different cascade model or the probability distribution defined over edges.

We skip the description for the LT model and just provide the equivalent process of LT model proposed in [48]. In the equivalent process of the LT model, an unweighted directed graph g is generated from G by realizing only one incoming edge of each vertex in V. That is, for each vertex $v_i \in V$, each incoming edge e_{ji} of vertex v_i has probability w_{ji} of being selected and only the selected edge is realized in g. Given a directed graph g generated by this equivalent process, a propagation tree $I_g(v)$ rooted on vertex v again can be computed by performing a BFS starting from v on g. Different from the equivalent process of IC model, there can be only one propagation tree for each vertex, since all vertices have only one incoming edge to these vertices. However, a propagation tree $I_g(v)$ under LT model still encodes the same information as in IC model; that is, each edge $e_{ij} \in I_g(v)$ encodes the information that a content propagates from v_i to v_j.

In the problem definition part, we make use of the notion of propagation trees in such a way that edges in a propagation tree that are crossing different parts are assumed to necessitate communication operations between servers. This assumption also holds for the LT model, since propagation trees generated by the equivalent processes of IC and LT models encode the same information. Therefore, minimizing the expected number of communication operations during an LT propagation process starting from a randomly chosen user still corresponds to minimizing the expected number of cut edges in a random propagation tree. In this regard, we do not need any modification for the objective function (3.2) and we still want to compute a partition Π^* that minimizes the expected number of cut edges in a random propagation tree (the only difference is in the process of computing a random propagation tree under LT model).

In the solution part, we generate a certain number of random propagation trees in order to estimate a probability distribution defined over all edges in E. The estimated probability distribution associates each edge with a probability value denoting how likely an edge is included in a random propagation tree under the IC model. The associated probability values are also later used as costs in the graph partitioning phase. However, both the estimation method and the overall solution do not depend on anything specific to the IC model and only require a method for generating random propagation trees which is mentioned above. Moreover, concentration bounds attained for the estimation of the probability distribution still holds under the LT model and the number of random propagation trees forming the set I in Algorithm 3.1 should satisfy Eq. (3.15).

Processes starting from multiple users. The method proposed for the propagation processes starting from a single user can be generalized for propagation processes that start from multiple users as follows: Instead of the definition of random propagation trees, we define random propagation forest $I_g(S)$ for a randomly selected subset of users $S \subseteq V$. The only difference between the two definitions is that a random propagation forest consists of multiple propagation trees that are rooted on the vertices in S. However, these propagation trees must be edge-disjoint and if a vertex is reachable from two different vertices in S, this

vertex can be arbitrarily included in one of the propagation trees rooted on these vertices. As noted earlier, the IC model does not prescribe an order for activating inactive neighbors; therefore, a random propagation forest over the set S can be computed by first drawing a graph $g \sim G$, and then performing a multi-source BFS on g starting from the vertices in S. The order of execution of multi-source BFS determines the form of propagation trees in the propagation forest $I_g(S)$.

In a partition Π of propagation forest $I_g(S)$, each cut edge incurs one communication operation. So, the total number of communication operations induced by Π is defined to be the number of cut edges which we denote as $\lambda_g^{\Pi}(S)$. These new definitions do not require any major modification for the optimization problem introduced in Eq. (3.2) and we just replace the expectation function with $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(S)]$. That is, our objective becomes computing a partition that minimizes the expected number of cut edges in a random propagation forest.

To generalize the proposed solution, we redefine p_{ij} value of an edge e_{ij} as the probability of edge e_{ij} included in a random propagation forest instead of a random propagation tree. With this new definition of p_{ij} values, Eqs. (3.3) and (3.4) can still be satisfied; hence, a partition Π^* that minimizes the sum of p_{ij} values of edges crossing different parts also minimizes the expectation $\mathbb{E}_{v,g\sim G}[\lambda_g^{\Pi}(S)]$.

The new definition of p_{ij} values necessitates some modifications to the estimation method proposed earlier. Recall that, for the previous definition of p_{ij} values, we generate a set I of random propagation trees and compute the function $F_I(\cdot)$ for each edge e_{ij}. For the new definition of p_{ij} values, the estimations can be obtained with a similar approach; however, the set I must now consist of random propagation forests and $F_I(\cdot)$ must denote frequencies of edges to appear in these random propagation forests. Therefore, the only modification required for Algorithm 3.1 is to replace the step that the set I is generated by performing N randomized BFS algorithms. The new set I of random propagation forests can be obtained with a similar approach such that instead of performing randomized single-source BFS algorithms, we perform randomized multi-source BFS algorithms. These two BFS algorithms are essentially the same except that multi-source BFS starts execution with its queue containing a randomly selected

subset of vertices instead of a single vertex. The new definitions together with the modifications performed on the overall solution do not affect the concentration bounds obtained in Eq. (3.15).

3.4 Extensions and Limitations

Here, we show how the proposed cascade-aware graph partitioning algorithm (CAP) can be incorporated into other graph partitioning objectives.

Non-cascading queries. Queries such as "reading-friend's-posts" and "read-all-posts-from-friends" can be observed more frequently than cascading (i.e., re-share) operations in a typical OSN application. The number of communication operations for such non-cascading queries may require minimizing the number of cut edges if query workload is highly changing or not available, or minimizing the total traffic crossing different parts if it can be estimated. The cascade-aware graph partitioning aims at reducing the cut edges that have high probability of being involved in a random propagation process under a specific cascade model. Assigning unit weights to all edges (i.e., $c_{ij} = 1$ for each edge e_{ij}) makes the objective same as minimizing the number of cut edges. A combination of objectives can easily be achieved by assigning each edge cost $c_{ij} = 1 + \alpha(p_{ij} + p_{ji})$, where α determines the relative weight of traffic/cascade-awareness.

Intra-propagation balancing among servers. This paper considers the number of nodes/users as the only balancing criteria for the proposed cascade-aware partitioning. On the other hand, the proposed formulation can be enhanced to handle balance on multiple workload metrics via a multi-constraint graph partitioning. For example, a balanced distribution of the number of content propagation operations within servers can be attained via the following two-constraint formulation. We assign the following two weights to each vertex v_i:

$$w^1(v_i) = 1 \quad \text{and} \quad w^2(v_i) = \sum_{e_{ki} \in E} p_{ki}. \tag{3.18}$$

Here, the summation in the second weight represents the sum of p probabilities of the incoming edges of vertex v_i. Under this vertex weight definition, the two-constraint partitioning maintains balance on both the number of users assigned to servers as well as the number of intra-propagation operations to be performed within servers. The latter balancing holds because of the fact that the expected number of propagations within a part V_k is

$$\sum_{e_{ij} \in \mathcal{E}(V_k)} p_{ij} \tag{3.19}$$

where $\mathcal{E}(V_k)$ denotes the set of edges pointing towards the vertices in V_k.

Repartitioning. As graph databases are usually dynamic, i.e., new vertices and edges are added or removed etc., repartitioning is necessary [5–7,41]. Repartitioning methods aims to maintain the quality of an initial partition via reassigning vertices to parts as the graph structure changes. However, the costs of new edges should be computed for repartitioning. That is, if a new direct edge is established in G, then its p value need to be computed before repartitioning. The p_{ij} value of a new edge e_{ij} can be computed using p_{ki} value of each incoming edge e_{ki} of vertex v_i as follows:

$$p_{ij} = w_{ij} \times \left[1 - \prod_{e_{ki} \in E} (1 - p_{ki})\right] \tag{3.20}$$

That is, the content propagation probability w_{ij} is multiplied by the probability of there being at least one edge e_{ki} incoming to vertex v_i is activated during a random propagation process. It is important to note that establishing the new edge e_{ij} also affects p_{jk} value of each outgoing edge e_{jk} of vertex v_j. If these values also need to be updated during repartitioning, Eq.(3.20) can be applied for each edge e_{jk}, in succession for updating the value of p_{ij}. In short, while moving vertices between parts during repartitioning, the p_{ij} value of any edge e_{ij} can be updated via applying Eq. (3.20) in a correct order. By updating p_{ij} values on demand, the existing repartitioning approaches can be adapted for the cascade-aware graph partitioning problem.

Replication. Replication strategies need some modifications in order to be used for the cascade-aware graph partitioning. It should be noted that, even though the cut size of graph G' can be reduced via replication of some vertices among multiple parts, this approach also incurs additional communication operations. This is because, when a replicated vertex becomes active during a content propagation process, the content needs to be transferred to each server that the vertex is replicated.

3.5 Experimental Evaluation

In this section, we experimentally evaluate the performance of the proposed solution on social network datasets. We develop an alternative solution, which produces competitive results, as a baseline algorithm in our experiments. The baseline algorithm directly makes use of propagation probabilities between users in the partitioning phase (i.e., w_{ij} values). Additionally, we also test various algorithms previously studied in the literature [3, 4] and compared with the proposed solution.

3.5.1 Experimental Setup

Datasets. Table 3.2 displays the properties of the real-world social networks used in our experiments. Many of these datasets are used in the context of influence maximization research [54]. The first 13 datasets (`Facebook`–`LiveJournal`) are collected from Stanford Large Network Dataset Collection[1] [64] and they contain friendship, communication or citation relationships between users in various real-world social network applications. The remaining datasets: `Twitter (large)` is collected from [65], `uk-2002` and `webbase-2001` are collected from Laboratory

[1] `https://snap.stanford.edu/data`

for Web Algorithmics[2] [66] and `sinaweibo` is collected from Network Repository[3] [67]. Additionally, we also make use of a synthetic graph, named as `random-social-network`, which we generate by using graph500 [68] power law random graph generator. The graph500 tool is initialized with two parameters namely as *edge-factor* and *scale* in order to produce graphs with 2^{scale} vertices and *edge-factor*$\times 2^{scale}$ directed edges. We set both *scale* and *edge-factor* to 16 to produce `random-social-network` dataset.

All datasets are provided in the form of a graph, where users are represented by vertices and relationships by directed or undirected edges. To infer the direction of content propagation between users, we interpret these social networks as follows: For directed graphs, we assume that a propagation may occur only in the direction of a directed edge, whereas for undirected graphs, we assume that a propagation may occur in both directions along an undirected edge. Therefore, we did not apply any modifications to the directed graphs, whereas we modified the undirected graphs by replacing each undirected edge with two opposite directed edges.

[2] `http://law.di.unimi.it`
[3] `http://networkrepository.com`

Table 3.2: Dataset Properties

	number of		in degree		out degree		
	Vertices	**Edges**	**max**	**avg**	**max**	**avg**	**Descripton**
Facebook	4,039	176,468	1,045	44	1,045	44	Social network from Facebook
wiki-Vote	7,115	103,689	893	15	457	15	Wikipedia who-votes-on-whom network
HepPh	34,546	421,534	411	12	846	12	Arxiv High Energy Physics paper citation network
email-Enron	36,692	367,662	1,383	10	1,383	10	Email communication network
Epinions	75,879	508,837	1,801	7	3,035	7	Who-trusts-whom network of Epinions.com
Twitter (small)	81,306	1,768,135	1,205	22	3,383	22	Social network from Twitter
Slashdot	82,168	870,161	2,510	11	2,552	11	Slashdot social network from February 2009
email-EuAll	265,214	418,956	929	2	7,631	2	Email communication network
dblp	317,080	2,099,732	343	7	343	7	DBLP collaboration network
youtube	1,134,890	5,975,248	28,754	5	28,754	5	Youtube online social network
Pokec	1,632,803	30,622,564	8,763	19	13,733	19	Pokec online social network
wiki-Talk	2,394,385	5,021,410	100,022	2	3,311	2	Wikipedia talk (communication) network
LiveJournal	4,847,571	68,475,391	20,292	14	13,905	14	LiveJournal online social network
Twitter (large)	11,316,811	85,331,843	564,512	7	214,381	7	Social network from Twitter
uk-2002	18,484,123	292,243,668	194,942	16	2,449	16	Web graph crawled (2002) under `.uk` domain
sinaweibo	58,655,849	522,642,104	278,490	9	278,490	9	Sina Weibo online social network
webbase-2001	118,142,155	1,019,903,190	816,127	8	3,841	8	Web graph crawled (2001) by WebBase [69]
random-social-network	65,536	910,479	9,613	19	3,233	19	Generated by graph500 power law graph generator

In the datasets in Table 3.2, the information about the content propagation probabilities between users is not available. Therefore, for each dataset, we draw values uniformly at random from the interval $[0, 1]$ and associate these values with the edges connecting its pairs of users as the propagation probabilities. We repeat this process five times for each dataset and obtain five different versions of the same social network having different propagation probabilities associated with its edges. Using these versions of the social network, we performed the same set of experiments on each different version and reported the averages of the results obtained for that specific dataset.

Given an underlying social network with its associated propagation probabilities, our aim is to find a user partition that minimizes the expected number of communication operations during a random propagation process under a specific cascade model. There have been effective approaches in the literature to learn the propagation probabilities between users in a social network [45, 46]. Inferring these probability values using logs of user interactions is out of the scope of this paper. However, we also work on a real-world dataset, from which real propagation traces can be deduced, to test the proposed solution.

Baseline partitioning (BLP) algorithm. One can partition the input graph in such a way that the edges with high propagation probabilities are removed from the cut as much as possible. To achieve this, the sum of propagation probabilities of the cut edges can be considered as an objective function to be minimized in the graph partitioning problem. The baseline algorithm also builds an undirected graph from a given social network and makes use of a graph partitioning tool. Instead of computing a new probability distribution over all edges (i.e., the p_{ij} values), the baseline algorithm directly makes use of propagation probabilities associated with edges (i.e., the w_{ij} values). That is, the cost c_{ij} of an undirected edge e'_{ij} of G' is determined using w_{ij} and w_{ji} values instead of p_{ij} and p_{ji} values of edges e_{ij} and e_{ji}, respectively. By this way, the graph partitioner minimizes the sum of propagation probabilities associated with the edges crossing different parts. The difference between the baseline algorithm and the proposed solution is the cost values associated with the edges of the undirected graph provided to the graph partitioner.

Other tested algorithms. In our experiments, we also test three previously studied social network partitioning algorithms for comparison purposes. The first of these algorithms (CUT) is given in [3] and aims to minimize the number of links crossing different parts (i.e., basically minimizes the number of cut edges). The second algorithm (MO+) [3] makes use of a community detection algorithm and performs partitioning based on the community structures inherent to social networks.

As the third algorithm, we consider the social network partitioning algorithm provided in [4]. The social graph is partitioned in such a way that two-hop neighborhood of a user is kept in one partition, instead of the one-hop network. For that purpose, an activity prediction graph (APG) is built and its edges are associated with weights that are computed using the number of messages exchanged between users in a time period. Since the w_{ij} values can not be directly considered as the number of messages exchanged between users, we make use of $F_I(e_{ij})$ values computed by CAP algorithm. That is, we designate the number of messages exchanged in a time period between users as $F_I(e_{ij})$. Additionally, to compute edge weights, the algorithm uses two parameters which are the total number of past periods considered and a scaling constant (these parameters are referred to as K and C in [4]). We set these parameters to *one*, since we can not partition $F_I(e_{ij})$ values into time periods. Using these values, we construct the same APG graph and partition this graph. We refer to this algorithm as 2Hop in our experiments.

Content propagations. To evaluate the qualities of the partitions obtained by the tested algorithms, we performed a large number of experiments based on both real and IC based traces of propagation on real-world social networks. We generated the IC based propagation data as follows: First, we generate a randomly selected subset of users, then execute an IC model propagation process starting from the users in this set. The size of the set is randomly determined and chosen uniformly at random from the interval $[1, 50]$. During this propagation process,

we count the total number of propagation events that occurred between the users located on different parts. As mentioned earlier, such propagation events cause communication operations between servers according to our problem definition. For each of the datasets, we perform 10^5 such experiments and compute the average of the total number of communication operations performed under a given partition. This average corresponds to an estimation for the expected number of communication operations during a random propagation process.

Partitioning framework. The graphs generated by algorithms, except MO+, are partitioned using state-of-the-art multi-level graph partitioning tool Metis [14] using the following set of parameters: We specify partitioning method as multilevel k-way partitioning, type of objective as edge-cut minimization and the maximum allowed imbalance ratio as 0.10. All the other parameters are set to their default values. We implemented MO+ algorithm by using a community detection algorithm[4] provided in [70] with its default parameters.

In order to observe the variation of the relative performance of the algorithms, each graph instance is partitioned K-way for $K = 32, 64, 128, 256, 512$ and 1024, respectively. In order to observe the performance gain achieved by intelligent partitioning algorithms, all graph instances are also partitioned random-wise, which is referred to as random partitioning (RP) algorithm.

3.5.2 Experimental Results

Figure 3.2 compares the performance of the proposed CAP algorithm against the existing algorithms 2Hop, MO+, CUT as well as BLP. In the figure, we display the geometric means of the ratios of the communication operation counts incurred by the partitions obtained by CAP, BLP, CUT, MO+ and 2Hop to those by RP, for each different K value. We run CAP algorithm with accuracy $\theta = 0.01$ and $\delta = 0.05$. As seen in the figure, BLP performs much better than both 2Hop and

[4] `http://www.mapequation.org/code.html`

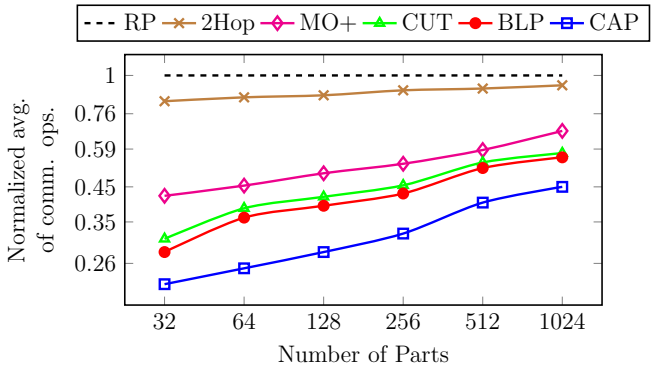


Figure 3.2: The geometric means of the communication operation counts incurred by the partitions obtained by BLP, CUT [3], CAP, MO+ [3] and 2Hop [4] normalized with respect to those by RP.

MO+, whereas it performs slightly better (6%–9% on average) than CUT. These experimental results justify the use of BLP as a baseline algorithm to test the validity of the proposed CAP algorithm. As also seen in the figure, the proposed CAP algorithm performs significantly better than all algorithm.

Table 3.3: Average number of communication operations during IC model propagations under partitions obtained by RP (Random partitioning), BLP (Baseline partitioning) and CAP (proposed cascade-aware graph partitioning) algorithms.

	K=32						K=64					
	RP	BLP		CAP			RP	BLP		CAP		
	comm	comm	cut	comm	cut	%imp	comm	comm	cut	comm	cut	%imp
Facebook	3,787	1,818	0.32	1,647	0.38	9.40	3,854	2,344	0.52	2,187	0.57	6.66
wiki-Vote	2,127	1,681	0.74	1,360	0.82	19.11	2,160	1,824	0.82	1,497	0.87	17.91
HepPh	12,048	4,092	0.29	3,155	0.44	22.91	12,580	5,095	0.36	3,858	0.52	24.28
email-Enron	25,153	6,539	0.39	5,083	0.45	22.27	25,514	8,073	0.45	6,193	0.52	23.29
Epinions	29,801	12,781	0.59	8,290	0.66	35.14	30,264	13,232	0.64	8,699	0.70	34.26
Twitter (small)	68,391	11,901	0.18	8,827	0.23	25.83	69,470	14,348	0.22	10,379	0.27	27.66
email-EuAll	27,543	3,709	0.21	2,592	0.31	30.11	28,050	24,421	0.75	2,934	0.35	87.99
Slashdot	57,426	29,724	0.71	21,497	0.77	27.68	58,313	29,416	0.74	22,433	0.79	23.74
dblp	240,487	36,939	0.17	33,853	0.20	8.36	244,170	40,818	0.19	37,270	0.22	8.69
youtube	648,399	147,215	0.33	122,535	0.40	16.76	659,107	163,117	0.38	134,613	0.46	17.47
Pokec	429,046	129,345	0.26	117,943	0.31	8.82	435,335	154,124	0.33	141,410	0.39	8.25
wiki-Talk	729,773	115,825	0.44	93,005	0.47	19.70	740,897	125,724	0.47	98,511	0.50	21.64
LiveJournal	1,152,319	279,846	0.24	234,804	0.30	16.10	1,168,167	317,001	0.29	267,697	0.36	15.55
Twitter (large)	4,183,322	1,950,300	0.57	1,194,675	0.75	38.74	4,250,880	1,942,093	0.68	1,268,234	0.79	34.70
uk-2002	6,933,807	107,593	0.01	60,539	0.03	43.73	7,046,263	114,284	0.01	64,713	0.03	43.38
sinaweibo	40,403,232	12,192,759	0.46	9,541,550	0.56	21.74	40,404,365	12,356,095	0.50	10,169,184	0.61	17.70
webbase-2001	29,301,906	614,458	0.02	329,646	0.03	46.35	29,774,985	651,124	0.02	358,722	0.04	44.91
norm avgs wrto RP	**1.00**	**0.21**		**0.16**		**25.16**	**1.00**	**0.26**		**0.17**		**31.82**

	K=128						K=256					
Facebook	3,884	2,774	0.70	2,629	0.73	5.24	3,893	3,107	0.82	2,980	0.85	4.09
wiki-Vote	2,174	1,931	0.88	1,837	0.96	4.89	2,184	2,125	0.96	2,101	0.97	1.13
HepPh	12,409	5,954	0.44	4,389	0.60	26.29	12,850	7,034	0.53	5,048	0.68	28.23
email-Enron	25,714	9,596	0.53	7,636	0.59	20.42	25,820	11,505	0.59	10,208	0.65	11.27
Epinions	30,506	13,285	0.68	9,094	0.72	31.55	30,653	13,635	0.71	9,574	0.74	29.78
Twitter (small)	69,993	17,111	0.27	12,336	0.34	27.90	70,169	20,687	0.34	14,709	0.41	28.89
email-EuAll	28,266	24,241	0.79	3,231	0.41	86.67	28,348	24,172	0.81	4,207	0.47	82.60
Slashdot	58,716	29,501	0.76	23,149	0.81	21.53	58,946	30,265	0.79	23,667	0.81	21.80
dblp	245,923	43,376	0.20	39,053	0.24	9.97	247,006	45,240	0.21	40,534	0.25	10.40
youtube	664,146	177,097	0.43	146,781	0.51	17.12	666,821	195,400	0.49	158,057	0.54	19.11
Pokec	438,930	182,158	0.40	168,153	0.49	7.69	440,504	210,461	0.48	201,080	0.61	4.46
wiki-Talk	747,415	130,856	0.49	111,842	0.51	14.53	747,916	142,779	0.50	138,546	0.52	2.97
LiveJournal	1,177,061	347,405	0.33	295,910	0.40	14.82	1,182,573	368,749	0.36	318,370	0.44	13.66
Twitter (large)	4,284,834	3,449,396	0.86	1,308,157	0.81	62.08	4,301,563	3,610,392	0.93	1,343,795	0.83	62.78
uk-2002	7,101,990	117,968	0.01	68,482	0.03	41.95	7,129,786	126,795	0.01	71,213	0.03	43.84
sinaweibo	40,726,570	12,836,470	0.53	10,679,002	0.63	16.81	40,886,216	13,232,762	0.57	11,027,580	0.65	16.66
webbase-2001	30,011,343	682,865	0.02	383,526	0.04	43.84	30,129,511	717,763	0.02	405,536	0.04	43.50
norm avgs wrto RP	**1.00**	**0.28**		**0.19**		**32.04**	**1.00**	**0.31**		**0.21**		**29.97**

	K=512						K=1024					
Facebook	3,912	3,625	0.95	3,555	0.95	1.91	3,914	3,721	0.97	3,687	0.97	0.91
wiki-Vote	2,186	2,135	0.97	2,131	0.97	0.21	2,190	2,138	0.96	2,115	0.97	1.11
HepPh	12,822	7,956	0.61	5,555	0.76	30.18	12,863	8,910	0.70	6,075	0.82	31.82
email-Enron	25,835	13,560	0.66	12,733	0.71	6.10	25,897	15,679	0.72	14,621	0.75	6.75
Epinions	30,631	14,396	0.74	10,320	0.75	28.32	30,674	15,302	0.77	11,424	0.77	25.34
Twitter (small)	70,309	24,979	0.42	18,020	0.50	27.86	70,430	33,555	0.55	23,861	0.64	28.89
email-EuAll	28,388	24,255	0.83	6,109	0.58	74.81	28,390	25,421	0.86	10,569	0.70	58.42
Slashdot	59,053	30,532	0.81	24,340	0.82	20.28	59,161	30,944	0.82	24,938	0.83	19.41
dblp	247,367	47,106	0.22	41,875	0.26	11.11	247,658	49,605	0.24	43,372	0.27	12.57
youtube	668,146	200,723	0.52	166,605	0.56	17.00	668,800	212,618	0.55	180,943	0.58	14.90
Pokec	441,892	232,194	0.55	214,309	0.67	7.70	442,150	249,415	0.60	223,356	0.72	10.45
wiki-Talk	752,402	635,183	0.90	622,024	0.92	2.07	752,048	672,420	0.93	663,852	0.92	1.27
LiveJournal	1,185,101	389,357	0.39	340,986	0.48	12.42	1,185,655	412,393	0.42	362,391	0.52	12.12
Twitter (large)	4,310,133	3,498,242	0.94	1,472,766	0.85	57.90	4,314,342	3,917,869	0.97	3,397,889	0.94	13.27
uk-2002	7,144,002	138,545	0.02	75,149	0.04	45.76	7,150,971	156,001	0.02	82,082	0.05	47.38
sinaweibo	40,966,473	13,630,831	0.60	11,516,372	0.67	15.51	40,983,365	14,209,656	0.63	11,904,051	0.69	16.23
webbase-2001	30,188,276	735,969	0.02	427,812	0.04	41.87	30,218,102	760,751	0.02	441,437	0.04	41.97
norm avgs wrto RP	**1.00**	**0.36**		**0.26**		**27.36**	**1.00**	**0.38**		**0.30**		**22.14**

"%imp": improvement of CAP over BLP, "cut": ratio of number of cut edges to total number of edges, K: number of parts/servers.

Table 3.3 compares the performance of the proposed CAP algorithm against BLP and RP on each graph for each K value, in terms of average number of communication operations during IC model propagation simulations. Here, partitioning of a graph for each different K value constitutes a partitioning instance. For each K value, the last column entitled "%imp" displays the percent improvement of CAP over BLP for each dataset in terms of the number of communication operations. For each K value, the last row entitled "norm avgs wrto RP" displays the geometric means of the ratios of the communication operation counts incurred by the partitions obtained by BLP and CAP to those by RP. The table also contains a "cut" column which displays the ratio of the number of cut edges to the total number of edges for each partitioning instance.

As seen in Table 3.3, BLP performs significantly better than RP in all partitioning instances. This is because, BLP successfully reduces the sum of propagation probabilities of cut edges and reduces the chances for propagation events to occur between different parts. On average, partitions obtained by BLP incurs 4.76x, 3.84x, 3.57x, 3.22x, 2.77x and 2.63x less communications than RP for $K = 32, 64, 128, 256, 512$ and 1024 servers, respectively. The decrease in the performance gap between BLP and RP with increasing K can be attributed to the performance degradation of the graph partitioning tool for high K values. Especially, whenever the average number of vertices assigned to parts (i.e., $|V|/K$) decreases below some certain threshold (e.g., for $K = 1024$ and $K = 512$ on `Facebook` and `wiki-Vote` datasets), improvements of Metis significantly degrades as can be seen from Table 3.3. However, for the case of web graphs (e.g., `uk-2002` and `webbase-2001`), Metis provides significantly better partitions, providing cut ratios below 0.1 (i.e., structures of graphs also have effect on quality of partitions produced by Metis). As a result, the number of inter-partition communication operations are significantly less in cases of these graphs as compared to other cases.

As seen in Table 3.3, CAP performs significantly better than BLP in all of the partitioning instances. If the cut ratio values are closely inspected, partitions obtained by CAP leaves more edges in the cut (i.e., higher cut ratios); but these partitions incur less communication operations. On average, CAP achieves

25.16%, 31.82%, 32.04%, 29.97%, 27.36% less communication operations than BLP for $K = 32, 64, 128, 256, 512$ and 1024 servers, respectively. Especially, the best improvement is obtained on `email-EuAll` social network for $K = 64$, where the partitions obtained by CAP achieve 88% less communication operations than those by BLP. Also in this partitioning instance, CAP achieves a cut ratio of 0.35 which is significantly less than 0.75 of BLP. However, as the value of K increases, the improvement of CAP over BLP decreases for some social networks; especially for `wiki-Talk` and `wiki-Vote`, where 19.11% and 19.70% improvements of CAP over BLP for $K = 32$, respectively, decrease to 1.11% and 1.27% for $K = 1024$. This can be attributed to Eq. (3.16), since as the value of K increases, the number of cut edges is also expected to increase. As shown in Eq. (3.16), the number of cut edges directly affects the error made by CAP algorithm: the upper bound of the error made by CAP algorithm is shown to be proportional to the number of cut edges. Indeed, the performance improvement of CAP over BLP is observed to be the lowest on the partitioning instances for which CAP incurs the highest cut ratio. For instance, on datasets `Facebook`, `wiki-Talk` and `wiki-Vote` for $K = 1024$, partitions generated by CAP have cut ratios of 0.97, 0.97 and 0.92 respectively.

The performance decrease of CAP can be alleviated by making more accurate estimations for the p_{ij} values and decreasing the value of θ. However, the cut ratio depends on the graph partitioning tool performance, dataset characteristics and imbalance ratios used during partitioning. In order to get better cut ratios, the imbalance constraint can be relaxed and increased to higher values (e.g., we used imbalance ratio of 0.1 in our experiments).

To observe how the improvement of CAP algorithm changes with respect to the cut ratio, we perform the same set of experiments also on `random-social-network`. As seen in Table 3.4, partitions obtained by CAP algorithm cause 43% less communication operations for $K = 32$ even though the fraction of edges are 6% more than that of BLP. As noted in previous experimental results, the improvement of CAP over BLP decreases as the value of K and the cut ratio increases: the percent improvement of CAP over BLP decreases from 43% to 30% as the fraction of cut edges increases from 0.91 to 0.95 on $K = 32$

Table 3.4: Results for IC model propagations on `random-social-network`. "cut" column denotes the fraction of edges remain in the cut after partitioning, "comm" column denotes the average number of communication operations and "%imp" column denotes the percent improvement of CAP over BLP.

	RP		BLP		CAP		
K	**cut**	**comm**	**cut**	**comm**	**cut**	**comm**	**%imp**
32	0.97	24,690	0.85	20,981	0.91	11,890	43.33
64	0.98	25,119	0.92	18,638	0.93	12,317	33.91
128	0.99	25,328	0.93	18,484	0.94	12,635	31.64
256	0.99	25,378	0.94	18,546	0.94	12,961	30.11
512	0.99	25,400	0.95	19,911	0.95	13,515	32.12
1024	0.99	25,511	0.95	20,279	0.95	14,229	29.83

and $K = 1024$, respectively.

Sensitivity analysis. We performed experiments to see how the accuracy parameter θ affects the performance of the CAP algorithm. For different values of θ and K, we compare the performance of CAP against RP on `random-social-network`. In Figure 3.3, we designate the size of set I as $|I| = 10, 10^2, 10^3, 10^4$ and 10^5 respectively. Experiments are performed under K-way partitions for $K = 32, 64, 128$ and 256. We plot the percent increase in the performance of CAP over RP on y-axis. The accuracy values are computed for confidence $\delta = 0.05$ and displayed on the right side of the figure. As seen in the figure, with increasing size of set I, the value of θ decreases exponentially and the improvement of CAP increases logarithmically. Additionally, as also observed earlier, the relative performance of CAP decreases with increasing K. The best performance improvement is obtained for $K = 32$ where CAP performs 2x better than RP. These results can be attributed to the results of Eq. 3.16, since for higher values of K, both the cut ratio and the error made by the CAP algorithm increases. However, as the accuracy increases, the error made by CAP decreases and the overall optimization quality improves.

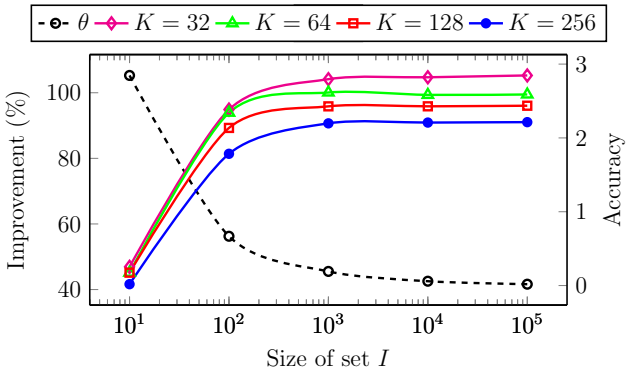


Figure 3.3: Variation in the improvement of CAP over RP with different sizes of set I. Dashed curve denotes the accuracy θ, whereas solid lines denotes variations in the improvements for `random-social-network` on $K = 32, 64, 128$ and 256 parts/servers.

Relationship with Minimizing Cut Edges. We also performed experiments to observe how much the cascade-based estimation of traffic is related to the performance measure of minimizing the number of cut edges. Previously, we asserted that different objectives can be encoded in the same cut size definition through assigning different weights to costs associated with edges by each objective (i.e. $c_{ij} = 1 + \alpha(p_{ij} + p_{ji})$). The parameter α controls how much the cascade-based estimation of the traffic is considered. Figure 3.4 displays the average number of communication operations and ratio of cut edges, by varying parameter α, obtained by CAP for `HepPh` dataset on $K = 32$ parts/servers (i.e., the α value is multiplied by $F_I(e_{ij})$ value of each edge). As seen in the figure, with increasing α, the average number of communication operations decreases, whereas the ratio of the number of cut edges increases. On the other hand, the increase in the cut size slows down after $\alpha = 10^{-2}$ and cut ratio becomes at most 0.44, since the cut size also has effect on the average number of communication operations. Note that for the smallest value of α, CAP becomes almost equivalent to CUT, providing equally well partitions in terms of number of cut edges (black-dashed curve denotes the cut value obtained by CUT algorithm). If the query workload is dominated by non-cascading queries and there is comparably small number of cascades, than α value can be set to smaller values, or vice versa.

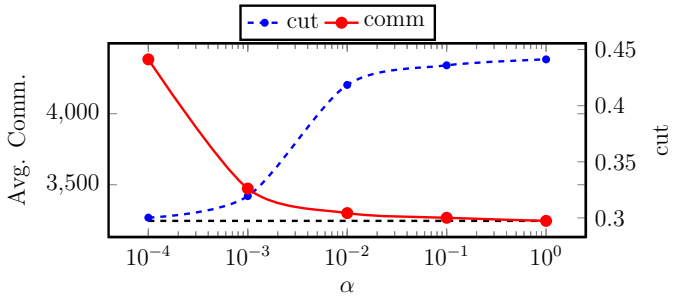


Figure 3.4: Variation of average number of communication operations and cut values obtained by CAP for different α values. Experiments are performed for `HepPh` dataset on $K = 32$ parts/servers. Solid curve denotes communication values, dashed curve denotes cut values (Black-dashed curve denotes the cut value obtained by CUT algorithm).

Running times. We performed our experiments on a server having 256 GB main memory and *two Intel(R) Xeon(R) CPU E7-8860 v4 @ 2.20GHz* processors, each of which consists of 18 cores and is able to run 36 threads concurrently. One of the largest social networks we used in our experiments, `sinaweibo`, consists of approximately 58M vertices and 522M edges. For this social network, the estimation of p_{ij} values took approximately 5.9 hours and required 59GB main memory. For the estimation phase, we employed shared memory parallelism and generated the set I via using 72 threads. The partitioning phase of undirected graph G' into $K = 1024$ parts via Metis took approximately $1,25$ hours and required 71GB main memory. It is worth to mention that partitioning time of `sinaweibo` via Pulp [71], which is a label propagation-based partitioning tool, took approximately 1.1 hour in case of BLP algorithm. Similarly, the biggest graph we used in our experiments, `webbase-2001`, has 118M vertices and 1B edges. For this dataset, the estimation phase took approximately 2.2 hours and required 130GB main memory. However, even though `webbase-2001` has more vertices and edges than `sinaweibo`, partitioning this graph into $K = 1024$ parts took approximately 0.5 hours and required 51GB main memory. Note that the estimation phase of CAP takes only 4.7x and 4.4x more time than the partitioning phase on the two large datasets `sinaweibo` and `webbase-2001`, respectively. These relative runtime comparisons suggest that cascade-aware graph partitioning, considered as an offline process that will be used in relatively long time intervals, runs in reasonable time limits for large-scale social networks.

3.5.3 Experiments on Digg Social Network with Real Propagation Traces

In this section, we use actual propagation traces collected from `Digg`[5] [39] social news portal. `Digg` is an OSN where users can share and vote for news stories, and designate others as friends to inform about their activities. Friendships can be designated as one-way relationships in such a way that if a user v_i declares another user v_j as a friend, then v_i is informed of activities of v_j but not vice versa. Users follow activities of their friends in their news feeds where the stories their friends shared or voted for are displayed. With these properties of `Digg` social network, a story can propagate through users once it is shared or voted for. Propagation of news stories can be considered as propagation of contents in our problem definition.

The `Digg` dataset contains a directed graph $G = (V, E)$ representing the underlying social network which consists of 71,367 users and 1,731,658 friendship links. As friendships are formed as one-way relationships, they are represented by directed edges. Each directed edge $e_{ij} \in E$ means that user v_j is following the activities of user v_i; therefore, the content propagation can occur in the direction of v_i to v_j. Additionally, the dataset contains log $\mathcal{L}$ of past activities of users over a set $\mathcal{N}$ of news stories. Each entry $(v_i, n_k, t_i) \in \mathcal{L}$ means that user $v_i \in V$ has voted for news story $n_k \in \mathcal{N}$ at time t_i. The dataset contains 3,018,197 votes made on 3,553 news stories (i.e., $|\mathcal{L}| = 3{,}018{,}197$ and $|\mathcal{N}| = 3{,}553$).

In order to deduce the content propagation traces from log $\mathcal{L}$, we follow the approach proposed in [59]. In this approach, if user v_i votes for the news story n_k, then it is assumed that v_i is probably influenced by one of its friends that have voted for the same story before. However, in order for v_i to be influenced by its friends, the difference between their voting times should be within a time window t_Δ. Let P_i^k denote the set of users that potentially influence user v_i in voting for news story n_k, we define P_i^k as

$$P_i^k = \{v_j \in V \mid (v_j, v_i) \in E \wedge t_i - t_j \leq t_\Delta\}. \tag{3.21}$$

[5] www.isi.edu/ lerman/downloads/digg2009.html

In our experiments, we set the time window t_Δ as *one* month following the approach in [59]. The set P_i^k induces a subgraph $g_k = (V, E_k)$ of G, where potential influencers of each user are denoted by the directed edges in E_k as follows:

$$E_k = \{(v_j, v_i) \in E \mid v_j \in P_i^k\}. \tag{3.22}$$

The subgraph g_k is reminiscent of a directed graph $g \sim G$, where each directed edge e_{ij} is associated with a propagation probability w_{ij} and g is generated by the equivalent process of the IC model as described in Section 3.2. Note that g is also a subgraph where each user may have multiple potential influencers and one of them can be arbitrarily selected to generate a propagation tree/forest. Therefore, we generate a propagation forest for the news story n_k on g_k as follows: Let $I_k(S)$ denote a propagation forest on g_k, where propagation trees are rooted on vertices in S and the set S is composed of vertices that are having no incoming edges (i.e., users in the set S do not have any potential influencers).

The propagation forest $I_k(S)$ can be computed by performing a multi-source BFS starting from vertices in S on g_k as if a random propagation tree is built from a $g \sim G$. It is important to note that multiple propagation forests can be generated depending on the execution of multi-source BFS on g_k. Edges in the propagation forest $I_k(S)$ still encode the information as to propagation traces through users. We generate propagation trees/forests for all news stories in Log $\mathcal{L}$, and use them instead of performing the IC model propagation simulations. Algorithm 3.2 presents computations we performed on log $\mathcal{L}$ to deduce the content propagation traces.

After generating the propagation trees/forests for all news stories available in log $\mathcal{L}$, we sample 90% of these trees/forests to use in Algorithm 3.1. That is, instead of randomly generating random propagation forests, we use real propagations in log $\mathcal{L}$ to compute the function $F_I(\cdot)$ and estimate p_{ij} values of edges in G. We use the remaining 10% of the propagation forests to test the qualities of the partitions returned by Algorithm 3.1. If an edge in a propagation forest crosses different parts, we count that edge as one communication operation.

We compare the qualities of the partitions produced by CAP algorithm against

Algorithm 3.2 Generating Propagation Trees/Forests from logs of past propagation traces

Input: $G = (V, E)$, $\mathcal{L}, t_\Delta$
Output: I

Partition log $\mathcal{L}$ based on news stories and obtain $\mathcal{L}_k$ for each story $n_k \in \mathcal{N}$
Initialize an empty set I of propagation forests
for each $\mathcal{L}_k$ **do**
    Sort entries $(v_i, n_k, t_i) \in \mathcal{L}_k$ according to their timestamps t_i in increasing order
    Initialize a directed graph $g_k = (V, E_k)$, where $E_k = \emptyset$
    for each entry $(v_i, n_k, t_i) \in \mathcal{L}_k$ **do**
        Mark v_i as activated
        for each $(v_j, v_i) \in E$ **do**
            if v_j is activated **and** $t_j - t_i \leq t_\Delta$ **then**
                $E_k = E_k \cup \{(v_j, v_i)\}$
    Initialize set $S = \{v_i \mid \text{in-degree of } v_i = 0 \text{ in } g_k\}$
    Perform multi-source BFS on g_k starting from the vertices in S and generate a propagation forest I_k
    $I = I \cup \{I_k\}$
return I

those of a slightly modified version of the BLP algorithm presented previously. In the modified version of BLP, we associate unit cost with each edge of the undirected graph that is produced from the input social graph. This modification causes BLP to regard only the friendship structure of `Digg` social network and produce partitions that minimize the number of friendship links crossing different parts. In this way, BLP and CUT algorithms become equivalent.

In Table 3.5, we present the results of the experiments on `Digg` social network. In addition to CAP and the modified version of BLP, we also include the results for partitions generated by RP. For each of these partitions, we compute the average number of communication operations induced on the propagation trees that are generated and sampled from 10 percent of log $\mathcal{L}$.

As seen in the Table 3.5, BLP performs much better than 2Hop, CUT, MO+ and RP algorithms. For $K = 32$, the partition generated by BLP incurs approximately 2x less communication operations than RP. The performance improvements of BLP is less for higher values of K. For example, BLP performs 2 times

Table 3.5: Experimental results on Digg social network. For each tested algorithm, average number of communication operations induced during propagation of news stories are displayed. "%imp" column denotes the percent improvement of CAP over BLP.

K	**RP**	**MO+**	**2Hop**	**BLP**	**CAP**	**%imp**
32	192	189	151	101	40	60.80
64	195	193	160	86	44	48.11
128	196	196	167	119	62	47.97
256	197	197	172	128	77	39.65
512	198	198	174	133	102	23.06
1024	198	198	177	152	131	13.53

better than RP for $K = 1024$. CAP algorithm, on the other hand, consistently performs better than BLP for all values of K. Especially, for $K = 32$, CAP algorithm incurs 60% less communication operations. However, as the value of K increases from 32 to 1024, the overall improvement of CAP over BLP decreases to 13%. This is because the accuracy obtained by 90% of the propagation trees/forest sampled from log $\mathcal{L}$ remains constant as we increase the value of K and therefore the error made by CAP algorithm increases as we have showed in Eq. 3.16. Additionally, the performance of the graph partitioning tool is expected to decrease for higher values of K where the average number of vertices per part reduces below 100 for $K = 1024$.

Results displayed in Table 3.5 illustrates the effectiveness of the CAP algorithm in a case where actual propagation traces are used instead of the IC model simulations.

3.6 Conclusion

We studied the problem of cascade-aware graph partitioning, where we seek to compute a user-to-server assignment that minimizes the communication between servers/parts considering content propagation processes.

We employed a sampling-based method to estimate a probability distribution by which each edge of a graph is associated with a probability of being involved

in a random propagation process. We use these estimates as part of the input of graph partitioning. The proposed solution works under various cascade models and requires that parameters of these models are given beforehand. Theoretic results that show how our solution achieves the stated objectives are also derived. To the best of our knowledge, this is the first work that incorporates the models of graph cascades into a systems performance goal.

We performed experiments under the widely used IC model and evaluated the effectiveness of the proposed solution in terms of partitioning objectives. We implemented the solution over real logs of propagation traces among users, in addition to using their social network structure. Experiments demonstrate the effectiveness of the proposed solution both in presence and absence of actual propagation traces.

Chapter 4

Scaling Sparse Matrix-Matrix Multiplication in the Accumulo Database

Relational databases have long been used as data persisting and processing layer for many applications. However, with the advent of big data, the need for storing and processing huge volumes of information made relational databases an unsuitable choice for many cases. Due to the limitations of relational databases, several NoSQL systems have emerged as an alternative solution. Today, big Internet companies use their own NoSQL database implementations especially designed for their own needs (e.g., Google Bigtable [72], Amazon Dynamo [73], Facebook Cassandra [74]). Especially, the design of Google's Bigtable has inspired the development of other NoSQL databases (e.g., Apache Accumulo [75], Apache HBase [76]). Among them, Accumulo has drawn much attention due to its high performance on ingest (i.e., writing data to database) and scan (i.e., reading data from database) operations, which make Accumulo a suitable choice for many big data applications [77].

Solutions for big data problems generally involve distributed computation and

see [29] for the original work

need to take the full advantage of data locality. Therefore, instead of using an external system, performing computations inside a database system is a preferable solution [26]. One approach to perform big data computations inside a database system is using NewSQL databases [78]. These type of databases seek solutions to provide scalability of NoSQL systems while retaining the SQL guarantees (ACID properties) of relational databases. However, even though using a NewSQL database can be a good alternative, some researchers take a different approach and seek solutions based on performing big data computations inside NoSQL databases [26]. To that extent, the Graphulo library [24] realizing the kernel operations of Graph Basic Linear Algebra Subprogram (GraphBLAS) [79] in Accumulo NoSQL database is recently developed. GraphBLAS is a community that specifies a set of computational kernels that can be used to recast a wide range of graph algorithms in terms of sparse linear algebraic operations. Therefore, realizing GraphBLAS kernels inside NoSQL databases enables performing big data computations inside these systems, since many big data problems involve graph computations [80].

One of the most important kernel operations in GraphBLAS specification is Sparse Generalized Matrix Multiplication (SpGEMM). SpGEMM forms a basis for many other GraphBLAS operations and used in a wide range of applications in big data domain such as subgraph detection and vertex nomination, graph traversal and exploration, community detection, vertex centrality and similarity computation [23–25]. An efficient implementation for SpGEMM in Accumulo NoSQL database is proposed in [27]. The authors actually discuss two multiplication algorithms which are referred to as inner-product and outer-product. Among the two algorithms, the outer-product is shown to be more efficient, and therefore is included in Graphulo Library [24]. The inner-product has the advantage of write-locality while ingesting the result matrix, but has the disadvantage of scanning one of the input matrices multiple times. On the other hand, the outer-product algorithm can not fully exploit write-locality, but requires scanning both of the input matrices only once.

In this work, we focus on improving the performance of SpGEMM in Accumulo for which we propose a new SpGEMM algorithm that overcomes the trade-offs

presented earlier. The proposed solution alleviates scanning of input matrices multiple times by making use of Accumulo's batch-scanning capability which is used for accessing multiple ranges of key-value pairs in parallel. Even though the use of the batch-scanning introduces some latency overheads, these overheads are alleviated by the proposed solution and by using node-level parallelism structures. Moreover, the proposed solution provides write-locality while ingesting the result matrix and does not require further computations when the result matrix need to be scanned, which was not the case for the previously proposed SpGEMM algorithm in [27].

We also propose a matrix partitioning scheme that improves the performance of the proposed SpGEMM algorithm via reducing the total communication volume and providing a balance on the workloads of the servers. Since matrices in Accumulo can only be partitioned according to sorted order of their rows and split points applied on rows, we propose a method that reorders input matrices in order to achieve the desired data distribution with respect to a precomputed partitioning. We cast the partitioning of matrices as a graph partitioning problem for which we make use of a previously proposed bipartite graph model in [30]. We propose a modification to this graph model in order to better comply with the proposed SpGEMM iterator algorithm and Accumulo's own architectural demands.

We conduct extensive experiments using 20 realistic matrices collected from various domains and synthetic matrices generated by graph500 random graph generator [68]. On all test instances, the proposed algorithm significantly performs better than the previously proposed solution without the use of any intelligent input matrix partitioning scheme. The performance of the proposed algorithm is improved even further with the use of the proposed matrix partitioning scheme.

4.1 Background

4.1.1 Accumulo

Accumulo is a highly scalable, distributed key-value store built on the design of Google's Bigtable. Accumulo runs on top of Hadoop Distributed File System (HDFS) [81] to store its data and uses Zookeeper [82] to keep coordination among its services. Data is stored in the form of key-value pairs where these key-value pairs are kept sorted at all times to allow fast look up and range-scan operations. Keys consist of five components namely as row, column family, column qualifier, visibility and timestamp. Values can be considered as byte arrays and there is no restriction for their format or size. These key-value pairs are kept sorted in ascending, lexicographical order with respect to the key fields.

Accumulo groups key-value pairs into tables and tables are partitioned into tablets. Tablets of a table are assigned to tablet servers by a master which stores all metadata information and keeps coordination among tablet servers. Tables are always split on row boundaries and all key-value pairs belonging to the same row are stored by the same tablet server. This allows modifications to be performed atomically on rows by the same server. All tables consist of one tablet when they are created for the first time, and as the number of key-value pairs in a tablet reaches to a certain threshold, the corresponding tablet is split into two tablets and one of these tablets is migrated to another server. It is also possible to manually add split points to a table to create tablets a priori and assign them to servers. This eliminates the need of waiting the tablets to split on their own and allows writing or reading data in parallel, which increases the performance of ingest and scan operations.

Reading data on the client side can be performed using sequential-scanner or batch-scanner capabilities of Accumulo. Sequential-scanning allows access to a range of key-value pairs in sorted order, whereas batch-scanning allows concurrent access to multiple ranges of key-value pairs in unsorted order. Similarly, writing data is performed through using batch-writer which provides mechanisms

to perform modifications to tablets in parallel.

It is also possible to perform distributed computations inside Accumulo by using its iterator framework. Iterators are configured on tables for specific scopes, and forms an iterator tree according to their priority. After configured on tables, iterators are applied in succession to the key-value pairs during scan or compaction times. Since a table may consist of multiple tablets and span to multiple tablet servers, each tablet server executes its own iterator stack concurrently, which provides a distributed execution. Users can implement customized iterators that can be plugged into available iterator tree on a table and obtain distributed parallelism by performing a batch-scan operation over a range of key-value pairs. An iterator applied during a batch-scan operation is executed on tablet servers that are spanned by the given range of key-value pairs.

4.1.2 Related Work

Parallelization of SpGEMM operation on shared-memory architectures have been extensively studied in many research works [83–85]. More recently, matrix partitioning schemes that utilize spatial and temporal locality in row-by-row parallel SpGEMM on many-core architectures are proposed in [28].

Several publicly available libraries exist to perform SpGEMM on distributed memory architectures [20, 86]. Buluc and Gilbert [1] studies sparse SUMMA algorithm, a message passing algorithm, which employs 2D block decomposition of matrices. Akbudak and Aykanat [32] propose hypergraph models for outer-product message passing SpGEMM algorithm to reduce communication volume and provide computational balance among processors. More recently, hypergraph and bipartite graph models are proposed in [30] for outer-product, inner-product and row-by-row-product formulations of message passing SpGEMM algorithms on distributed memory architectures.

Graphulo library[1] [24] also provides a distributed SpGEMM implementation

[1] https://github.com/Accla/graphulo

developed by Hutchison et al. [27], running on top of Accumulo's iterator framework. The method proposed in [27] utilizes the outer-product formulation of SpGEMM in the form of $C = AB$. In this approach each column of A is multiplied by its corresponding row of B (i.e., ith column of A is multiplied by ith row of B), and the resulting matrices of partial products by such multiplications are summed to get the final matrix C. To benefit from high performance attained by rowwise table accesses in Accumulo, A^T and B are stored in separate tables (i.e., scanning columns of A corresponds to scanning rows of A^T).

The SpGEMM algorithm proposed in [27] is executed by an iterator applied on a batch-scan performed on the table storing matrix A^T. Therefore, each tablet server iterates through local rows of A^T and scans the corresponding rows of B to perform the outer-product operations between them. The required rows of B can be stored locally as well as stored by other servers, since the A^T and B matrices are stored as separate tables and Accumulo may assign respective tablets to different servers even the same split points are applied on these tables. If we assume that scanning rows of B does not incur any communication costs (i.e., the respective tablets of A^T and B are co-located), writing the resulting C matrix back to the database still incurs significant amount of communication costs in this approach. This is because, the partial result matrices cannot be aggregated before being written back to the database and in the worst case, a partial result matrix can have nonzero entries that should be broadcast to almost all tablet servers available in the system, which may necessitate a tablet server to communicate with all the other tablet servers during this phase. Because of these reasons, writing phase of this outer-product SpGEMM algorithm becomes the main bottleneck. Therefore, Hutchison et al. [27] discuss also an alternative approach and propose an inner-product algorithm which requires scanning the whole B matrix for each local row of A. Although this inner-product approach has the advantage of write-locality, it is considered infeasible due to the necessity of scanning the whole B matrix for each row of A. Therefore, the outer-product implementation of the SpGEMM is included in the Graphulo library.

Our solution to perform SpGEMM in Accumulo differs from [27] in the way that it provides the advantage of write-locality, similar to the one provided by

the inner-product approach, and alleviates scanning all rows of matrix B for each row of A by a tablet server. To provide that, our solution makes use of Accumulo's batch-scanning capability which enables accessing to multiple rows of matrix B in parallel. However, our approach suffers from the latency overheads introduced by performing a batch-scan operation for each local row of A. As we discuss in the following sections, this latency overhead can be hugely resolved by batch-processing of multiple local rows of matrix A by tablet servers and using multi-threaded parallelism.

In our experimental framework, MPI-based distributed SpGEMM algorithms are omitted due to the significant differences between MPI and Accumolo iterators, since Accumulo's architectural properties violates fairness of performance comparisons between the proposed SpGEMM iterator algorithm and MPI-based algorithms. For instance, (1) Accumulo provides fault-tolerance mechanisms which incur additional computational overheads (e.g., disk accesses, data replication, cache updates etc.). (2) In MPI-based implementations, input matrices are present in memory of processors before performing communication and arithmetic operations, whereas in Accumulo, input matrices may be present in disk as well. (3) Communication operations are performed very differently in MPI and Accumulo: communication operations between servers are performed in a streaming manner in Accumulo, whereas in MPI, communication operations are performed in synchronized steps.

4.2 Row-by-Row Parallel SpGEMM Iterator Algorithm

Iterators are the most convenient way to achieve distributed parallelism in Accumulo, since they are concurrently executed by tablet servers on their locally stored data. The proposed iterator algorithm is based on row-by-row parallel matrix multiplication and data distribution.

4.2.1 Row-by-Row Parallel SpGEMM

We summarize the row-by-row SpGEMM which leads to row-by-row parallelization: Given matrices A, B and C, the product $C = AB$ is obtained by computing each row $C(i,:)$ as follows:

$$C(i,:) = \sum_{A(i,j) \in A(i,:)} A(i,j)B(j,:). \tag{4.1}$$

Here, $A(i,:)$, $B(j,:)$ and $C(i,:)$ denote the ith row, jth row and ith row of matrices A, B and C, respectively. That is, each nonzero $A(i,j)$ of row i of A is multiplied by all nonzeros of row j of B and each multiplication $A(i,j)B(j,k)$ for a nonzero $B(j,k)$ of row j of B produces a partial result for the entry $C(i,k)$ of row i of C.

Accumulo's data model presents a natural way of storing sparse matrices in such a way that each key-value pair stored in a table has row, column and value subfields which can together store all the necessary information to represent a nonzero entry. Therefore, tables can be seen as sparse matrices that are rowwise partitioned among tablet servers, since tables are always split on row boundaries among tablet servers and all key-value pairs belonging to the same row are always contained in the same tablet server. Due to this correspondence between key-value pairs and nonzero entries, and between tables and sparse matrices, we use these term pairs interchangeably.

Another important feature of Accumulo is that it builds row indexes on tables and allows efficient lookup and scan operations on rows. However, scanning key-value pairs in a column is impractical, because Accumulo does not keep a secondary index on columns and this operation necessitates scanning all rows of a table. If both columns and rows of a table need to be scanned, transpose of the table also need to be stored in a separate table in Accummulo. For instance, the outer-product parallel SpGEMM algorithm [27] needs to scan columns of matrix A for the multiplication $C = AB$; and therefore, keeps A^T instead of A.

4.2.2 Iterator Algorithm

Let matrices A, B and C are stored by K tablet servers where we denote the kth tablet server by T_k for $k = 1 \dots K$. For now, also assume that these matrices are stored in separate tables (e.g., matrix A is stored in table A).

Since these matrices are rowwise partitioned among tablet servers, also assume that each tablet server T_k stores kth row blocks A_k, B_k and C_k of matrices A, B and C, respectively. Note that a row block of a matrix consists of a number of consecutive rows of the respective matrix (e.g., A_k may consist of rows $A(i,:), A(i+1,:), \dots, A(j,:)$). Here, row blocks C_k and A_k are conformable, i.e., $C(i,:) \in C_k$ iff $A(i,:) \in A_k$. Before performing the computation $C = AB$, the matrix C can be thought of as an empty table. In this setting, the proposed iterator algorithm should be configured on a batch-scanner provided with a range covering all rows of matrix A (i.e., the entire range of table A). Performing this batch-scan operation ensures that each tablet server T_k concurrently executes the iterator algorithm on its local portion A_k of A.

After configured on the batch-scanner on matrix/table A, the proposed iterator algorithm proceeds as follows: Each tablet server T_k iterates through its locally stored rows of A_k and computes the corresponding rows $C(i,:)$ according to Eq. (4.1). It is important to note that Accumulo provides a programming framework that allows only iteration through key-value pairs (i.e., nonzero entries); and therefore, we can assume that each tablet server T_k is provided with a sorted stream of its local nonzero entries in A_k. These nonzero entries are lexicographically sorted with respect to their first row and then their column indices. Thus, during the scan of this stream, it is possible to scan all nonzeros of A_k in row basis by keeping the nonzero entries belonging to the same row in memory before proceeding to the next row. In this way, iterations can be considered as proceeding rowwise in A_k.

During the iteration of each row $A(i,:) \in A_k$, the computation of row $C(i,:)$ necessitates scanning row $B(j,:)$ for each nonzero $A(i,j) \in A(i,:)$ to perform multiplication $A(i,j)B(j,:)$. Some of these required rows of B are stored locally,

Algorithm 4.1 Iterator Algorithm

Input: Matrices A, B and C are distributed among K tablet servers.
Input: Local matrices A_k, B_k and C_k stored by server T_k.
Output: Arithmetic results are written back to C_k

```
1: Initialize a thread cache
2: Initialize sets Φ, B(Φ)
3: while source iterator has key-value pairs (i.e., ∃A(i,j) ∈ A_k) do
4:    Iterate through all nonzero entries of A(i,:)
5:    for all A(i,j) ∈ A(i,:) do
6:       if j ∉ B(Φ) then
7:          B(Φ) = B(Φ) ∪ {j}
8:    Φ = Φ ∪ {A(i,:)}
9:    if |B(Φ)| > threshold then
10:      Execute MULTIPLY(Φ, B(Φ)) by a worker thread in the thread cache
11:      Initialize new sets Φ, B(Φ)
12: Join with all threads in the thread cache
13: return
```

whereas the remaining rows are stored by other tablet servers. Therefore, to retrieve these B-matrix rows, T_k performs a batch-scan operation provided with multiple ranges covering these required rows over matrix B. As mentioned earlier, Accumulo allows simultaneous access to multiple ranges of key-value pairs via its batch-scanning capability. This operation provides an unsorted stream of nonzero entries belonging to the required rows of B, because these nonzero entries can be retrieved from remote servers in arbitrary order and batch-scan operation does not guarantee any order on them. As these nonzero entries are retrieved, for each retrieved nonzero $B(j,k)$ of a required row $B(j,:)$, the computation $C(i,k) = C(i,k) + A(i,j)B(j,k)$ is performed. The final row $C(i,:)$ is obtained after the stream of nonzero entries belonging to the required rows of B are all processed. The pseudocode implementation of the proposed iterator algorithm is presented in Algorithm 4.1.

4.2.3 Communication and Latency Overheads

For each row of A_k, the tablet server T_k performs a batch-scan operation on multiple ranges covering the required rows of B (i.e., ranges need to cover each row

$B(j,:)$ of B for each nonzero $A(i,j) \in A(i,:)$ of A_k). This operation necessitates T_k to perform one lookup on the row index of B, which is stored by the master tablet server, in order to determine the locations of the required B-matrix rows. After this lookup operation and the servers storing these B-matrix rows are determined, data retrieval is performed via communicating with multiple tablet servers in parallel.

This operation incurs significant latency overheads due to the lookups performed on the master tablet server for each row $A(i,:) \in A_k$ and establishing connections with tablet servers storing required rows of B. Additionally, this approach may necessitate redundant communication operations and increase the total communication volume among tablet servers, since the same row of B may be retrieved multiple times by the same tablet server for computing different rows of C. In other words, a tablet server T_k needs to retrieve row j of B for each row of A_k that has a nonzero at column j. For instance, if A_k contains two nonzero entries $A(i,k)$ and $A(j,k)$, then two different batch-scan operations performed by T_k to compute rows $C(i,:)$ and $C(j,:)$ necessitates retrieval of the same row $B(k,:)$ twice.

In the proposed algorithm, the aforementioned shortcomings are alleviated by processing multiple rows of A simultaneously. That is, a tablet server T_k iterates through multiple rows in A_k and creates batches of rows to be processed together before performing any computation or a batch-scan operation. Let $\Phi \subseteq A_k$ denote such a batch that is generated by iterating through multiple rows and contains rows $A(i,:), A(i+1,:), \ldots, A(j,:)$. Further, let $B(\Phi)$ denote an index set indicating the required B-matrix rows to compute the corresponding rows $C(i,:), C(i+1,:) \ldots C(j,:)$ for the batch Φ. The row indices in the set $B(\Phi)$ are determined as the union of indices of columns on which rows in Φ have at least one nonzero. That is,

$$B(\Phi) = \{j \mid \exists A(i,j) \in A(i,:) \wedge A(i,:) \in \Phi\}. \tag{4.2}$$

By computing the set $B(\Phi)$, a single batch-scan operation can be performed for multiple rows in Φ to retrieve all of the required rows of B at once instead of

performing separate batch-scans for each $A(i,:) \in \Phi$. After performing a single batch-scan over rows in $B(\Phi)$, the tablet server T_k is again provided with an unsorted stream of nonzero entries belonging to the rows corresponding to row indices in $B(\Phi)$. Each retrieved nonzero entry $B(j,k)$ of a required row $B(j,:)$ is then multiplied with each nonzero entry in the jth column segment of Φ. Then, each partial result obtained by multiplication $A(i,j)B(j,k)$ is added to the entry $C(i,k) \in C(i,:)$. That is, each nonzero $B(j,k)$ is multiplied by column j of A_k to contribute to the column j of C_k. So, the local matrix multiplication algorithm performed by T_k is a variant of column-by-column paralel SpGEMM although the proposed iterator algorithm utilizes the row-by-row parallelization to gather the B-matrix rows needed for processing nonzeros in A_k.

Processing rows in A_k in batches significantly reduces the latency overheads and communication volume among tablet servers, because both the number of lookup operations and the redundant retrieval of rows of B are reduced by this approach. Here, the size of a batch becomes an important parameter, since it directly affects the number of lookup operations and the communication volume among tablet servers. As the size of a batch increases, both the number of lookup operations and the total communication volume among tablet servers decreases, since as more rows of A_k are added to a single batch, it is more likely that nonzero entries belonging to different A-matrix rows share the same column. However, the size of a batch is bounded by the hardware specifications of tablet servers (i.e., total memory). In our experiments, we determined the best suitable batch size by testing various values in our experimentation environment.

4.2.4 Thread Level Parallelism

Even though the batch processing of multiple rows of A_k by the tablet server T_k significantly reduces the latency overheads, further enhancements for the proposed iterator algorithm can be achieved via using node-level parallelism structures, such as threads.

In a single-threaded execution, the main execution thread of T_k stays idle and

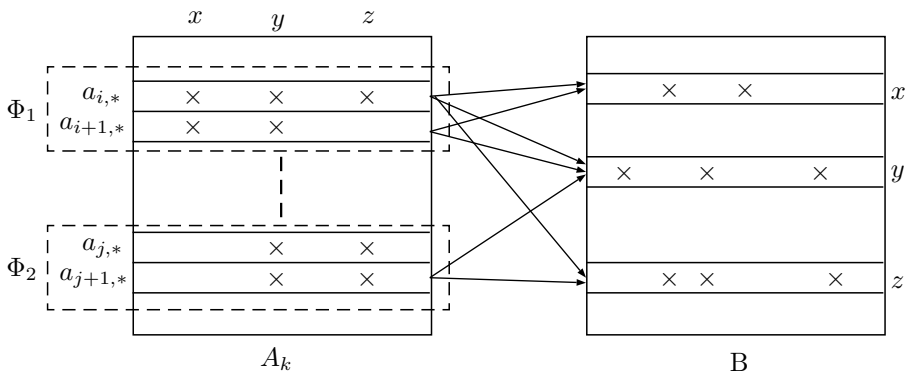


Figure 4.1: Sample execution of the proposed iterator algorithm by a tablet server T_k. Batches Φ_1 and Φ_2 are processed by *two* separate threads. Arrows indicate the required rows of matrix B by each worker thread.

performs no computation by the time between initiating the batch-scan and the stream of nonzero entries become ready to be processed during the processing of a batch Φ. In order to avoid this idle time, we utilize a multi-threaded approach in which the main thread assigns the task of processing the current batch Φ of A_k to a worker thread and continues iterating through the remaining rows to prepare the next batch. Whenever the main thread prepares the next batch, it assigns the task of processing this batch to a new worker different than the previous worker(s). This multi-threaded execution enables processing multiple batches of A_k concurrently by worker threads and thus achieving node-level parallelism in a streaming manner. After a batch Φ is fully processed by a worker thread, the resulting C-matrix rows are written back to database by the same worker thread. This write operation will not incur communication if row blocks A_k and C_k are stored by the same server.

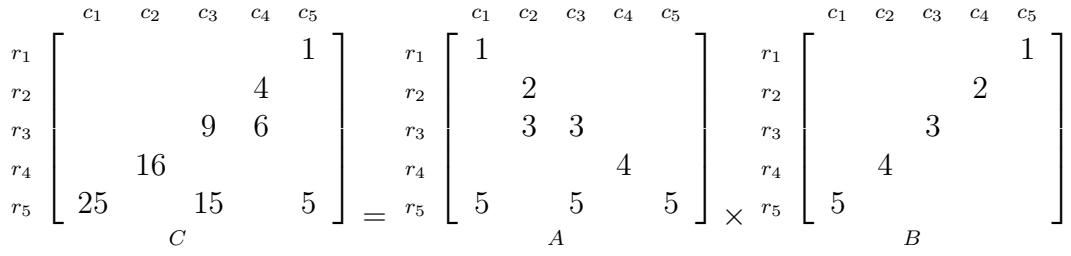
In the proposed implementation, creating a new thread for each batch Φ also incurs additional computational cost and latency. In this regard, we make use of thread caches which create new threads only if there is no available thread to process the current batch (Java standard library also provides various thread caches/pools having different implementation schemes). These thread caches create new threads only if necessary and reuses them in order to alleviate the cost of creating new threads.

Figure 4.1 displays a sample execution of the proposed iterator algorithm by a tablet server T_k. The main execution thread creates batches Φ_1 and Φ_2 and assigns them to worker threads which then perform batch-scan operations to retrieve required rows of B and compute $\{A(i,:)B, A(i+1,:)B\}$ and $\{A(j,:)B, A(j+1,:)B\}$, respectively. For each batch, the required rows of B are denoted by arrows pointing to the respective rows. For instance, the worker thread processing batch Φ_1 needs to receive rows x, y and z of matrix B, since rows $A(i,:)$ and $A(i+1,:)$ have nonzeros only in these three columns. Similarly, the worker thread processing batch Φ_2 concurrently performs a batch-scan to retrieve rows y and z of matrix B. However, rows y and z need to be retrieved by tablet server T_k twice, since each worker thread scans these rows separately. The redundant retrieval of rows y and z increases the total communication volume, but can be avoided by increasing the batch size. For instance, instead of processing Φ_1 and Φ_2 by separate threads, these batches can be merged into a single batch and processed by the same thread. By this way, a single batch-scan can be performed to retrieve rows $B(y,:)$ and $B(z,:)$, and the redundant retrieval of rows y and z can be avoided.

4.2.5 Write-locality

To obtain write-locality during ingestion of rows in row block C_k, the responsibility of storing C_k must be given to the same tablet server storing row block A_k, since rows of C_k are locally computed on that server. However, if matrices A and C are stored as two different tables, Accumulo can not guarantee that row blocks A_k and C_k are stored by the same tablet server, since Accumulo's load balancer may assign corresponding tablets to different servers according to the partition of key space. Therefore, creating different tables for each matrix may necessitate redundant communication operations during ingestion of the resulting matrix C.

To achieve write-locality discussed as above, we use a single table M instead of three different tables for matrices A, B and C. This approach ensures that rows $A(i,:)$, $B(i,:)$ and $C(i,:)$ are stored by the same tablet server and these



(a)

row	family	qualifier	value
1	A	1	1
1	B	5	1
1	C	5	1
2	A	2	2
2	B	4	2
2	C	4	4
3	A	2	3
3	A	3	3
3	B	3	3
3	C	3	9
3	C	4	6

Tablet 1

row	family	qualifier	value
4	A	4	4
4	B	2	4
4	C	2	16
5	A	1	5
5	A	3	5
5	A	5	5
5	B	1	5
5	C	1	25
5	C	3	15
5	C	5	5

Tablet 2

(b)

Figure 4.2: A sample SpGEMM instance in which matrices A, B and C are stored in single a table M and partitioned among *two* tablet servers.

rows together belong to ith row of table M. Here, it is worth to note that storing row $B(i,:)$ along with rows $A(i,:)$ and $C(i,:)$ is not necessary for the proposed iterator algorithm and matrix B can be stored in a separate table. On the other hand, we preferred to store all three matrices in a single table M due to the requirements of the input matrix partitioning scheme discussed later in Section 4.3. To distinguish the nonzero entries of these matrices in table M, we use the column family subfield of a key. However, scanning nonzero entries of a specific row of a matrix necessitates scanning nonzero entries of other matrices as well, since ith row of M contains nonzero entries of rows $A(i,:)$, $B(i,:)$ and $C(i,:)$. This inefficiency can be resolved with the use of locality groups which directs Accumulo to separately store the key-value pairs belonging to different column families. That is, even rows $A(i,:)$, $B(i,:)$ and $C(i,:)$ belong to the same row of M and stored by the same tablet server, these rows are separately stored

on disk. This allows scanning a range of key-value pairs belonging to the same column family without accessing key-value pairs belonging to the other column families.

Keeping matrices A, B and C under different column families and defining locality groups for these matrices in table M creates an of illusion three different tables being stored in a single table. It is still possible to perform batch-scan over rows of A and B separately without accessing nonzero entries of each other. For instance, the proposed SpGEMM iterator algorithm can be configured on a batch-scanner, given the range covering column family of A, on table M. Each tablet server is still provided with a sorted stream of nonzero entries in A_k and nothing need to be modified in the proposed iterator algorithm. The only difference is the range provided for the batch-scanner on table M.

In Figure 4.2, the contents of table M are displayed for a sample SpGEMM instance, in which two tablet servers are used and a split point 3 is configured on table M. The nonzeros of the first three rows are stored in Tablet 1, whereas the others are stored in Tablet 2. Although the figure shows nonzero entries belonging to different matrices are placed one after each other and kept in lexicographically sorted order,these nonzero entries are stored separately on disk and it is possible to scan any row of a matrix in table M without redundantly accessing nonzero entries of other matrices. However, if the table M is scanned without specifying any column family, all nonzero entries are retrieved in this order.

4.2.6 Implementation

In Algorithms 4.1 and 4.2, we present the pseudocode of our implementation for the proposed solution. Algorithm 4.1 is the main iterator algorithm executed by the main thread running on each tablet server T_k. The main thread iterates through rows in A_k and prepares batches of rows of matrix A and assigns these batches to worker threads to perform multiplication and communication operations. Algorithm 4.2 is executed by worker threads to process a given batch Φ.

Accumulo iterators are Java classes that implement SortedKeyValueIterator (SKVI) interface which tablet servers load and execute during scanning or compaction phases on tablets of a table. In order to have custom logic inside Accumulo iterators, a Java class that implements SKVI interface should be written. These iterators are added to iterator trees of tables according to their priority, and the output of an iterator is used as the input of the next iterator whose priority is less than the previous. Therefore, the source of an iterator can be Accumulo's own data sources as well as another iterator having higher priority in the iterator tree. We implemented our algorithm in the *seek()* method of SKVI, which is the first method executed by the iterator after initialized by the tablet server (i.e., Algorithm 4.1 is executed in the *seek()* method).

In Algorithm 4.1, the main iterator thread starts with initializing a thread cache and two sets Φ and $B(\Phi)$. For the thread cache, we use Java's own cached thread pool implementation. The set Φ is represented with a two dimensional hash-based table which is available in Google Guava Library [87]. The table data structure supports efficient access to its cells since it is backed with a two dimensional hash-map (i.e., accessing to any cell in the table is performed in constant time). The set $B(\Phi)$ is a typical list data structure and used to keep distinct column indices of nonzero entries in the current batch Φ.

After the initialization step, the main thread starts iterating through the rows in local portion A_k of matrix A. As each row $A(i,:) \in A_k$ is retrieved, it is included into the current batch Φ until the batch-size threshold is reached. The set $B(\Phi)$ is used to keep distinct column indices of nonzero entries encountered so far in the current batch. These column indices correspond to the rows of B that are required to perform all the multiplications for the batch Φ. These operations are carried out between lines 4 to 12 in Algorithm 4.1.

In the proposed algorithm, we define the batch size threshold in terms of the number of distinct column indices of the nonzero entries in Φ (i.e., the size of the batch is $|B(\Phi)|$). In this way, we try to increase the number of nonzeros that share the same column indices in Φ and therefore reduce the redundant retrieval of B-matrix rows as much as possible. For instance, if two distinct rows $A(i,:)$

Algorithm 4.2 Multiplication Algorithm

Input: Φ,$B(\Phi)$

1: Perform batch-scan on matrix B over rows $B(j,:)$ for each $j \in B(\Phi)$
2: Initialize an empty $C(i,:)$ for each $A(i,:) \in \Phi$
3: **for all** retrieved nonzero $B(j,k) \in B(j,:)$ for a $j \in B(\Phi)$ **do**
4: **for all** entry $A(i,j) \in \Phi$ **do**
5: $C(i,k) = C(i,k) + A(i,j) * B(j,k)$
6: **for all** computed $C(i,:)$ **do**
7: Add $C(i,:)$ to batch-writer queue
8: **return**

and $A(j,:)$ have nonzero entries in common column indices and these rows are processed in different batches, then the same B-matrix rows corresponding to these column indices need to be redundantly retrieved for each of these batches by the same tablet server. In a different perspective, if a row of A_k introduced to Φ does not increase the batch size, then all of its nonzeros must share the same column indices with the rows added to Φ earlier. Hence, by increasing the batch size threshold, the likelihood of reducing redundant retrieval of B-matrix rows is possible. For example, in one extreme, if the whole A_k is processed in a single batch, then there will be no redundant communication, since each required B-matrix row need to be retrieved only once. Here, if the batch size is set to a large number, the level of concurrency may degrade in a tablet server, since fewer batches and threads will be generated and all CPU cores may not be efficiently utilized. On the other hand, if the batch size is set to a small number, the number of lookups and the total communication volume will increase. The size of the current batch is controlled in line 10 of Algorithm 4.1.

After the current batch Φ is prepared, Algorithm 4.2 provided with the parameters Φ and $B(\Phi)$, is executed on a worker thread chosen from the thread cache. Communication and multiplication operations for the batch Φ are handled by this worker thread. In Algorithm 4.2, the worker thread first initializes a batch-scanner on matrix B and provides this scanner with a range covering all rows $B(j,:)$ for each $j \in B(\Phi)$.

After performing the batch-scan operation, the worker thread initializes data structures representing C matrix rows to be computed. To represent an empty

$C(i,:)$ row, we again make use of the two dimensional hash-based table data structure, which was previously used to represent the batch Φ. As mentioned earlier, this data structure enables efficient access to its entries (i.e., each nonzero $c_{i,j}$) and allows us to efficiently combine partial results contributing to the same entries of matrix C, as performed in line 6 of Algorithm 4.2.

In line 7, a batch-writer is initialized after computing row $C(i,:)$ for each row $A(i,:) \in \Phi$. In the for-loop between lines 8 and 9, each computed row $C(i,:)$ is added to batch-writer queue buffer via a mutation object and written back to the database (i.e., to the ith row of table M under the respective column family for matrix C). As mentioned earlier, these operations do not necessitate any communication operations and can be locally performed due to the usage of a single table M and the write-locality achieved through this approach.

In Algorithm 4.1, the execution of the main thread continues until each row $A(i,:) \in A_k$ is processed. In line 13, the main thread waits for the worker threads in the thread cache to join. Upon joining with all threads, the iterator algorithm finishes and the resulting matrix C is available in table M. It is important to note that to scan the final matrix C, there is no need to apply a summing-combiner or another iterator, as was not the case in the algorithm previously proposed in [27].

The computational complexity of Algorithm 4.1 depends on the number of nonzero arithmetic operations $flops(A \cdot B)$ required to perform the multiplication $C = AB$ where $flops(A \cdot B) = \sum_i \sum_{a_{ij} \in A(i,:)} nnz(B(j,:))$. Insert and lookup operations performed on data structures Φ and $B(\Phi)$ can be performed in constant time, since these data structures are 2-dimensional hash-map-based table and hash-map-based set data structures, respectively. Elements in $B(\Phi)$ can be traversed in time linear time to the number of elements in this data structure. Assuming that the perfect workload load balance is achieved, each server approximately performs $\frac{flops(A \cdot B)}{K}$ nonzero arithmetic operations. Therefore, if the multi-threaded execution is not enabled, computation time of Algorithm 4.1 can be given as $\Theta(\frac{flops(A \cdot B)}{K})$, since the term $\frac{flops(A \cdot B)}{K}$ dominates other hidden factors associated with the number of local nonzero entries A_k, B_k and C_k on each server

T_k. For the communication complexity, each server, in the worst case, may communicate with all other tablet servers and receive $\mathcal{O}(\frac{flops(A \cdot B)}{K})$ nonzero entries of matrix B, since the number of nonzero entries retrieved can not be higher than the number nonzero arithmetic operations performed by a tablet server (i.e., at most, one B-matrix entry can be retrieved for each nonzero arithmetic operation). The communication cost of Algorithm 4.1 is $\mathcal{O}(t_s(K-1) + t_w \frac{flops(A \cdot B)}{K})$ where t_s and t_w denotes per-message latency and per-word bandwidth costs, respectively. Therefore, the parallel execution time of Algorithm 4.1 can be given as $\mathcal{O}(t_s K + (1 + t_w)\frac{flops(A \cdot B)}{K})$.

4.3 Partitioning Matrices

Here, we adapt the bipartite graph model recently proposed in [30] for row-by-row parallelization SpGEMM on distributed memory architectures. In this model, the SpGEMM instance $C = AB$ is represented by the undirected bipartite graph $G = (V_A \cup V_B, E)$. The vertex sets V_A and V_B represent the rows of A and B matrices, respectively. That is, V_A contains vertex u_i for each row i of A and V_B contains vertex v_j for each row j of B.

A K-way vertex partition of G

$$\Pi(V) = \left\{V^1 = V_A^1 \cup V_B^1, V^2 = V_A^2 \cup V_B^2, \cdots, V^K = V_A^K \cup V_B^K\right\}$$

is decoded as a mapping of rows of input matrices to tablet servers as follows: $u_i \in V_A^k$ and $v_j \in V_B^\ell$ correspond to assigning row i of A and row j of B to tablet servers T_k and T_ℓ, respectively. Here, a vertex u_i also represents row $C(i,:)$, since the tablet server that owns row $A(i,:)$ is given the responsibility of computing and storing row $C(i,:)$. Therefore, a partition obtained over rows of A also determines the partition of rows of C.

Through our experimentation and analysis over the proposed iterator algorithm, we observed that the numbers of nonzero entries stored for each of the A,

B and C matrices by a server are better measures than the number of floating-point operations performed for representing the associated computational load. That is, nonzero entries of each of the matrices A, B and C should be evenly distributed in order to achieve a workload balance among servers. Therefore, we assume that row i of A and row j of B respectively incur the computational loads of $nnz(A(i,:))$ and $nnz(B(j,:))$ to the servers they are assigned to. Here, $nnz(\cdot)$ denotes the number of nonzeros in a row. Note that storing row i of C also incurs the computational load of $nnz(C(i,:))$, because writing nonzero entries of output matrix C may require more computation time as compared to the scanning entries of input matrices.

Instead of estimating the relative computational loads associated with individual nonzeros of input and output matrices, we propose a three constraint formulation in which we associate three weights with each vertex as follows:

$$\begin{aligned} &w^1(u_i) = nnz(A(i,:)), \; w^2(u_i) = 0, && w^3(u_i) = nnz(C(i,:)), \; \forall u_i \in V_A \\ &w^1(v_j) = 0, && w^2(v_j) = nnz(B(j,:)), \; w^3(v_j) = 0, && \forall v_j \in V_B \end{aligned}$$

This 3-constraint partitioning captures maintaining a balanced distribution of nonzero entries of all matrices A, B and C among tablet servers. We should note here that the bipartite graph model given here differs from the model given in [30] because of this multi-constraint formulation.

In order to compute $w^3(v_i)$ of vertex v_i, we need to know the total number of nonzero entries in row $C(i,:)$ before partitioning, which necessitates performing a symbolic multiplication. This symbolic multiplication can be efficiently performed for just one time, using the proposed SpGEMM algorithm without adopting any partitioning scheme.

In graph G, there exists an undirected edge $e_{ij} \in E$ that connects vertices $u_i \in V_A$ and $v_j \in V_B$ for each nonzero $A(i,j) \in A$. We associate each edge e_{ij} with a cost equal to the number of nonzeros in the respective row j of B, i.e.,

$$cost(e_{ij}) = nnz(B(j,:))$$

This edge-cost definition refers to the amount of communication volume to incur if row i of A and row j of B are assigned to two different tablet servers.

In a given partition Π, uncut edges do not incur any communication. Cut edge e_{ij}, where $u_i \in V_A^k$ and $v_j \in V_B^\ell$, refers to the fact that tablet server T_ℓ stores row j of B, whereas tablet server T_k stores row i of A and is responsible of computing row i of C. Hence, this cut edge will incur the transfer of B matrix row $B(j,:)$ from tablet server T_ℓ. Thus, the partitioning objective of minimizing the cutsize according to Eq. (2.3) relates to minimizing the total communication volume that will be incurred due to the transfer of B-matrix rows. However, the cutsize overestimates the total communication volume in some cases: Consider two cut-edges e_{ij} and e_{hj}, where $u_i \in V_A^k$, $v_j \in V_B^\ell$ and $u_h \in V_A^k$. These two cut-edges show the need of tablet server T_k to retrieve row j of B for computing rows h and i of C. The cutsize incurred by these cut-edges according to Eq. (2.3) will be equal to $cost(e_{ij}) + cost(e_{hj}) = 2nnz(B(j,:))$. However, tablet server T_k may process rows h and i of matrix A in a single-batch, which causes T_k to retrieve row j of B only once, thus necessitating a communication volume of only $nnz(B(j,:))$ rather than $2nnz(B(j,:))$.

In general, consider a B-matrix row vertex that has d neighbors (A-matrix row vertices) in V_k. The cutsize definition encodes this situation as incurring a communication volume of $d \times nnz(B(j,:))$; however, the actual communication volume will vary between $nnz(B(j,:))$ and $d \times nnz(B(j,:))$, depending on the number distinct batches of T_k that require row j of B. For example in Figure 4.1, the degree of B-matrix row vertex v_y is 3 and its weight is $nnz(B(y,:))=3$. The cutsize encodes the total communication volume as 9. However, row y of B will be retrieved from the respective server to T_k only once for each of the two batches Φ_1 and Φ_2, thus the total communication volume will be 6 instead of 9.

Accumulo partitions tables into tablets via split points defined over row keys. For instance, if the split points a, b, c are applied on table M, rows of matrices A, B and C will be distributed to intervals $[0,a]$, $(a,b]$, $(b,c]$, $(c,\infty]$. For instance, if a row index $i \in [0,a]$, than rows $A(i,:)$, $B(i,:)$ and $C(i,:)$ will be stored by the tablet server responsible for storing interval $[0,a]$.

For a given partition Π of G, the desired data distribution of matrices A, B and C can only be achieved by reordering these matrices in table M and applying

a set of proper split points. That is, the sorted order of the new row indices of matrices A, B and C, together with the set of split points, ensure Accumulo to automatically achieve the desired data distribution. So, a given partition Π is decoded as inducing a partial reordering on the rows of the matrices as follows: C-/A-matrix rows corresponding to the vertices in V_A^{k+1} are reordered after the rows corresponding to the vertices in V_A^k and B-matrix rows corresponding to the vertices in V_B^{k+1} are reordered after the rows corresponding to the vertices in V_B^k. The row ordering obtained by this method is referred to as a partial ordering; because rows corresponding to the vertices in a part are reordered arbitrarily. Then the split points are easily determined on the part boundaries of row blocks according to Π.

The size of the proposed bipartite graph model is linear in the number of rows, columns and nonzero entries of matrix $A \in \mathbb{R}^{m \times n}$, since there exist a vertex v_i for each row $A(i,:)$, a vertex v_j for each row $B(j,:)$ and an edge for each nonzero entry $A(i,j)$ in matrix A. So the topology of the bipartite graph can be built in $\Theta(m+n+nnz(A))$ time. The first and second weights ($w^1(v_i)$ and $w^2(v_i)$) of vertices can be determined from input matrices in $\Theta(nnz(A)+nnz(B))$ time. Computing the third weight ($w^3(v_i)$) of vertices necessitates the symbolic multiplication of input matrices A and B (since $w^3(v_i) = nnz(C(i,:))$). Hence, the complexity of building the proposed graph model is $\Theta(flops(A \cdot B) + m + n + nnz(A) + nnz(B))$. On the other hand, complexity of the partitioning phase depends on the partitioning tool. In [88], complexity of Metis is reported as $\Theta(V + E + K \log K)$ where V $(= m + n)$ is number of vertices, E $(= nnz(A))$ is number of edges in a graph and K is the number of parts. The running time of the partitioning phase becomes $\Theta(m+n+nnz(A)+K \log K)$ thus leading to overall running time complexity of $\Theta(flops(A \cdot B)+m+n+nnz(A)+nnz(B)+K \log K) = \Theta(flops(A \cdot B)+K \log K)$.

4.4 Experimental Evaluation

We compare the performance of the proposed SpGEMM iterator algorithm (RRp) against the baseline algorithm (BL) [27], which is currently available in the Graphulo Library, on a fully distributed Accumulo cluster. We also evaluate the performance of the graph partitioning-based RRp (gRRp), where the input and output matrices are reordered using the partitioning scheme proposed in Section 4.3. We use the state-of-the-art graph partitioning tool Metis to partition the graph model in gRRp. We set the maximum load imbalance $\epsilon = 0.005$ and performed edge cut minimization. We used both realistic and synthetically generated sparse matrices as SpGEMM instances in our experiments.

4.4.1 Datasets

Table 4.1 displays properties of matrices used in the experiments. These matrices are included in our dataset since they arise in various real-world applications and also used in recent research works [28, 85, 89, 90].

We performed our experiments in two different categories: In the first category, a sparse matrix is multiplied with itself (i.e., $C = AA$), and in the second category, two different conformable sparse matrices are multiplied (i.e., $C = AB$). The first category $C = AA$ arises in graph applications such as finding all-pairs-shortest paths [91], self similarity joins and summarization of sparse dataset [92, 93]. The second category $C = AB$ is a more general case and especially arise in applications such as collaborative filtering [94] and similarity joins of two different sparse datasets [93].

The $C = AA$ category contains 13 sparse matrices all selected from UFL sparse matrix collection [95]. As seen in Table 4.1 all of these matrices contain more than 100K rows.

Table 4.1: Dataset Properties

Matrix	Number of			Number of nonzeros in a row		in a column	
	Rows	Columns	Nonzeros	Avg	Max	Avg	Max
$C = AA$							
2cubes_sphere	101,492	101,492	1,647,264	16	31	16	31
filter3D	106,437	106,437	2,707,179	25	112	25	112
598a	110,971	110,971	1,483,868	13	26	13	26
torso2	115,967	115,967	1,033,473	9	10	9	10
cage12	130,228	130,228	2,032,536	16	33	16	33
144	144,649	144,649	2,148,786	15	26	15	26
wave	156,317	156,317	2,118,662	14	44	14	44
majorbasis	160,000	160,000	1,750,416	11	11	11	18
scircuit	170,998	170,998	958,936	6	353	6	353
mac_econ_fwd500	206,500	206,500	1,273,389	6	44	6	47
offshore	259,789	259,789	4,242,673	16	31	16	31
mario002	389,874	389,874	2,101,242	5	7	5	7
tmt_sym	726,713	726,713	5,080,961	7	9	7	9
$C = AB$							
cfd2 (A)	123,440	123,440	3,087,898	25	30	25	30
cfd2.P (B)	123,440	4,825	528,769	4	10	110	181
boneS01 (A)	127,224	127,224	6,715,152	53	81	53	81
boneS01.P (B)	127,224	2,394	470,235	4	10	196	513
shipsec5 (A)	179,860	179,860	10,113,096	56	126	56	126
shipsec5.P (B)	179,860	2,959	541,099	3	13	183	456
thermomech_dK (A)	204,316	204,316	2,846,228	14	20	14	20
thermomech_dM (B)	204,316	204,316	1,423,116	7	10	7	10
offshore (A)	259,789	259,789	4,242,673	16	31	16	31
offshore.P (B)	259,789	9,893	1,159,999	4	13	117	221
amazon0302 (A)	262,111	262,111	1,234,877	5	5	5	420
amazon0302-user (B)	262,111	50,000	576,413	2	302	12	27
amazon0312 (A)	400,727	400,727	3,200,440	8	10	8	2,747
amazon0312-user (B)	400,727	50,000	882,813	2	1,675	18	38
$C = AB$ (graph500)							
scale=15 (A)	32,768	32,768	441,173	13	5,942	13	2,067
scale=15 (B)	32,768	32,768	441,755	13	5,976	13	2,041
scale=16 (A)	65,536	65,536	909,301	13	9,719	13	3,273
scale=16 (B)	65,536	65,536	909,854	13	9,670	13	3,407
scale=17 (A)	131,072	131,072	1,864,398	14	15,643	14	5,227
scale=17 (B)	131,072	131,072	1,864,338	14	15,743	14	5,301
scale=18 (A)	262,144	262,144	3,806,212	14	25,332	14	8,303
scale=18 (B)	262,144	262,144	3,804,831	14	25,324	14	8,277

The $C = AB$ category contains seven SpGEMM instances. The SpGEMM instances *amazon0302* and *amazon0312* are used for collaborative filtering in recommendation systems [94]. In these instances, A matrices represent similarities of items and B matrices represent preferences of users and synthetically generated following the approach in [28], where the item preferences of users follow Zipf distribution. The SpGEMM instances *boneS01*, *cfd2*, *offshore* and *shipsec5* are used during the setup phase of Algebraic multigrid methods (AMG) [96]. In these instances, A matrices are selected from UFL and their corresponding interpolation operators are generated as the B matrices by using a tool[2] in [96] (matrices with suffix ".P" in Table 4.1). The last SpGEMM instance contains two conformable matrices *thermomech_dK* and *thermomech_dM*.

The $C = AB$ category also contains four SpGEMM instances whose A and B matrices are synthetically generated by using the Graph500 power law graph generator [68]. These synthetic matrices are previously used in [27] to evaluate the performance of the BL algorithm. We also use this tool with the same set of parameters. The tool takes two parameters, referred to as *scale* and *edge-factor*, and produces square matrices with 2^{scale} rows and *edge-factor* $\times\, 2^{scale}$ nonzero entries. We fix *edge-factor* to 16 as in [27] and set *scale* $= 15, 16, 17, 18$ to generate *four* different sized SpGEMM instances (e.g., for *scale* $= 15$ we generate two different square matrices A and B with 2^{15} rows and 16×2^{15} nonzero entries).

4.4.2 Accumulo Cluster

The cluster we used in our experiments consists of 12 nodes and these nodes are connected via *DGS-3120-24TC* ethernet switch. Each node has *two Intel-Xeon-E5-2690-v4* processors each of which consists of 14 cores and is able to run 28 threads concurrently. Additionally, each node has 256 GB main memory in addition to its 16 TB local storage. We designate *two* of these nodes as control nodes on which we run ZooKeeper, HDFS NameNode, Accumulo master, garbage collector and monitor processes. The remaining nodes are used as worker nodes and

[2]https://github.com/pyamg/pyamg

run only HDFS DataNode and Accumulo tablet server processes. In order to conduct strong scalability analysis, we run BL, RRp and gRRp algorithms for all SpGEMM instances on $K = 2, 4, 6, 8$ and 10 tablet servers.

4.4.3 Evaluation Framework

To measure the running time of BL, we first ingest input matrices A and B as separate tables, then we define split points that distribute rows of matrices to tablet servers evenly. By this partitioning, the first $K - 1$ tablet servers are assigned $\lfloor n/K \rfloor$ rows (n being the number of rows of a matrix) and the last tablet server is assigned all the remaining rows. The Graphulo library does not support any other load balancing scheme other than defining split points on rows as performed in our implementation, and this approach is also followed in [27]. After ingesting the A and B matrices, we call the SpGEMM routine provided by Graphulo library from a client process running on one of the control nodes. We measure the running time of BL as the total time the corresponding routine takes within the client process.

To measure the running time of RRp, we ingest matrices A and B in a single table M and we define split points on table M to evenly distribute rows of M among tablet servers, as done in BL. After this operation, we call the proposed SpGEMM routine within a client process running on one of the control nodes and measure the total running time of the multiplication operation. In this routine, the proposed iterator is applied on a batch-scanner which is provided with a range covering all rows of A under the column family of table M. The running time of gRRp is measured similarly.

The running times of all algorithms are obtained by averaging 5 successive runs. These times cover the entire process of performing multiplications of matrices A and B and writing the resulting matrix C back to database. However, in BL, outer-products results that contribute to the same nonzero entries of C are not combined/summed before being written back to table C. Summations of these partial results are performed by applying a scanning-time summing-combiner on

table C. Therefore, to obtain the final matrix C in parallel by all tablet servers, a batch-scan operation covering all rows of matrix C need to be performed. On the other hand, the C matrices computed by RRp and gRRp do not require applying a summing-combiner or performing any further computations, since all partial results contributing to the same nonzero entries of C are already combined/summed before being written back to the database. This difference violates the fairness of the comparisons made among the algorithms, since BL performs less computation than both RRp and gRRp during SpGEMM operations. In this regard, we also compare the time required to scan the C matrices produced by all algorithms and include in our experimental results. In order to get scanning times, we performed a batch-scan operation covering all rows of C, after performing the SpGEMM operation on matrices A and B, and measure the time required to receive all the nonzero entries of C by the client process running on one of the control nodes.

4.4.4 Experimental Results

Table 4.2 displays the measured running times of BL, RRp and gRRp to perform SpGEMM instances on $K = 2, 4, 6, 8$ and 10 tablet servers. The rows entitled as "norm avgs wrto BL" display the geometric means of the ratios of the running times of RRp and gRRp to those of BL in the respective categories.

As seen in Table 4.2, in both $C = AA$ and $C = AB$ categories, RRp consistently performs much better than BL on all test instances except for *amazon0312* on $K = 2$ servers. Moreover, the performance improvement of RRp over BL increases with increasing K. On average, in the $C = AA$ category, RRp runs 1.75x, 2.56x, 2.85x, 3.33x and 3.44x faster than BL on $K = 2, 4, 6, 8$ and 10 tablet servers, respectively. Similarly, these values become 2.50x, 3.70x, 3.80x, 4.34x and 4.34x in the $C = AB$ category. Additionally, we also observe that BL can not scale on some instances, especially on *scircuit* and *tmt_sym* instances, where the running time of BL increases as K increases from 8 to 10. However, RRp displays better scalability than BL, since in all test cases the running time of RRp decreases with the increasing value of K.

Table 4.2: Multiplication Times (ms)

	$K=2$			$K=4$			$K=6$			$K=8$			$K=10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AA$															
2cubes_sphere	16,571	6,735	**6,057**	13,145	3,741	**3,135**	10,045	2,803	**2,182**	11,332	2,264	**1,731**	7,373	1,826	**1,500**
filter3D	47,867	**19,199**	19,893	29,667	10,175	**9,116**	22,837	6,898	**5,999**	19,025	5,663	**4,659**	12,916	5,055	**3,711**
598a	15,009	10,745	**7,812**	14,126	6,070	**3,920**	13,145	5,266	**2,669**	13,991	4,368	**2,195**	9,579	3,491	**1,654**
torso2	7,434	3,277	**3,039**	7,091	2,095	**1,775**	5,949	1,495	**1,250**	5,851	1,162	**1,033**	4,810	938	**863**
cage12	25,809	16,024	**12,069**	23,050	9,076	**6,067**	15,319	6,367	**4,301**	11,264	4,855	**3,113**	13,282	4,272	**2,632**
144	21,852	15,422	**10,875**	14,429	9,070	**5,725**	13,038	6,436	**3,696**	12,838	5,955	**3,061**	11,974	5,093	**2,352**
wave	19,179	**8,478**	8,686	16,237	4,553	**4,193**	11,275	3,389	**2,757**	13,612	2,915	**2,330**	11,856	2,440	**1,873**
majorbasis	12,518	7,030	**6,109**	10,648	3,623	**3,170**	9,990	2,787	**2,274**	9,499	2,123	**1,776**	8,074	1,951	**1,524**
scircuit	7,203	4,635	**3,780**	7,298	2,949	**2,089**	5,770	2,169	**1,592**	5,093	1,809	**1,225**	6,363	1,500	**1,020**
mac_econ_fwd500	8,454	5,238	**4,858**	6,743	2,899	**2,681**	5,210	2,329	**2,067**	5,582	1,795	**1,580**	4,630	1,670	**1,310**
offshore	40,348	23,047	**21,880**	28,652	12,457	**9,985**	22,906	8,163	**6,276**	21,400	7,274	**5,157**	18,868	5,602	**3,892**
mario002	10,042	8,053	**6,866**	10,164	4,702	**4,146**	8,039	3,160	**2,438**	8,945	3,058	**2,023**	9,074	2,914	**1,697**
tmt_sym	23,497	13,831	**14,059**	18,026	7,341	**7,271**	13,184	5,111	**5,107**	12,031	4,084	**4,059**	13,554	**3,350**	3,340
norm avgs wrto BL	1.00	0.57	0.50	1.00	0.39	0.31	1.00	0.35	0.27	1.00	0.30	0.22	1.00	0.29	0.20
$C = AB$															
cfd2	11,505	4,958	**4,345**	12,265	2,471	**2,254**	9,891	1,746	**1,620**	8,993	1,336	**1,322**	7,765	**1,125**	1,220
boneS01	21,955	7,693	**7,169**	21,759	3,994	**3,914**	16,436	2,717	**2,665**	13,695	**2,123**	2,175	13,159	**1,810**	1,907
shipsec5	27,973	10,443	**10,336**	25,502	5,659	**5,463**	21,102	3,829	**3,780**	18,731	**2,983**	3,115	17,891	**2,430**	2,599
thermomech_dK	13,074	8,471	**6,836**	13,364	4,575	**3,738**	10,799	3,396	**2,616**	10,552	2,914	**2,205**	11,056	2,626	**1,758**
offshore	15,680	7,854	**6,977**	13,580	4,437	**3,826**	12,951	3,270	**2,761**	12,436	2,754	**2,231**	11,439	2,330	**1,844**
amazon0302	4,573	3,913	**3,429**	5,283	2,353	**2,062**	4,821	1,716	**1,514**	4,941	1,592	**1,551**	4,410	1,595	**1,439**
amazon0312	9,862	10,394	**8,498**	9,304	6,333	**4,616**	8,735	4,625	**3,606**	8,458	4,223	**3,123**	8,408	3,870	**2,828**
$C = AB$ (Graph500)															
scale=15	19,452	3,776	**3,186**	15,058	**2,228**	2,739	10,273	1,611	**1,476**	7,794	**1,273**	1,346	6,130	1,285	**1,049**
scale=16	38,634	7,037	**6,750**	25,109	4,064	**3,351**	12,283	3,231	**2,641**	12,202	2,617	**1,888**	11,679	2,171	**1,627**
scale=17	69,797	16,079	**15,161**	38,901	8,959	**7,652**	23,787	6,280	**5,472**	21,996	5,139	**4,313**	12,760	4,518	**3,422**
scale=18	126,538	34,802	**32,704**	59,441	18,703	**16,498**	37,432	13,187	**11,333**	26,735	10,890	**9,366**	21,816	8,738	**7,560**
norm avgs wrto BL	1.00	0.40	0.35	1.00	0.27	0.24	1.00	0.26	0.22	1.00	0.23	0.21	1.00	0.23	0.20

BL : SpGEMM implementation provided in the Graphulo library

RRp : The proposed row-row parallel SpGEMM algorithm

gRRp : RRp algorithm enhanced via the proposed graph partitioning scheme

Table 4.3: Scanning Times (ms)

	$K=2$			$K=4$			$K=6$			$K=8$			$K=10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AA$															
2cubes_sphere	52,724	**28,486**	29,493	31,742	**14,585**	14,674	35,208	**10,045**	10,429	16,774	8,007	**7,780**	20,923	6,377	**6,309**
filter3D	143,178	63,810	**54,195**	71,325	33,302	**29,067**	91,690	23,924	**20,777**	43,151	18,290	**15,669**	60,406	15,005	12,848
598a	44,983	24,419	**22,803**	38,324	12,456	**11,541**	29,689	8,943	**8,173**	19,749	6,939	**6,187**	17,522	5,901	**5,040**
torso2	15,661	8,171	**7,052**	12,447	4,482	**3,907**	8,923	3,132	**2,692**	5,429	2,438	**2,096**	6,408	1,968	**1,782**
cage12	81,726	60,256	**49,012**	74,981	34,524	**25,288**	53,328	25,208	**17,928**	32,859	18,130	**13,724**	31,844	15,248	**10,953**
144	61,819	34,998	**33,596**	39,872	18,325	**17,037**	41,106	13,134	**11,940**	22,026	9,814	**9,240**	26,003	7,829	**7,312**
wave	57,264	34,081	**29,535**	37,259	18,428	**15,747**	39,820	12,480	**10,996**	23,362	9,700	**8,391**	24,837	7,897	**6,898**
majorbasis	35,260	26,675	**23,223**	32,860	13,289	**11,997**	22,657	9,313	**7,940**	15,874	7,178	**6,223**	15,125	5,774	**5,083**
scircuit	19,146	19,087	**13,420**	19,463	11,357	**6,972**	16,226	7,783	**5,068**	12,304	6,151	**3,899**	9,678	4,739	**3,114**
mac_econ_fwd500	22,171	20,928	**19,090**	24,874	**9,806**	10,086	15,467	6,917	**6,417**	12,342	5,323	**4,878**	9,316	4,570	**3,941**
offshore	126,598	72,887	**62,443**	69,840	38,687	**33,695**	85,213	26,481	**23,554**	46,630	20,672	**18,455**	55,269	16,722	**14,625**
mario002	30,367	22,647	**17,219**	24,750	11,726	**10,092**	19,768	8,002	**6,398**	14,689	6,351	**4,855**	12,770	5,171	**4,001**
tmt_sym	63,949	43,907	**41,354**	50,607	23,569	**22,175**	47,486	16,106	**14,788**	24,992	12,495	**11,182**	28,101	9,471	**9,017**
norm avgs wrto BL	1.00	0.65	0.56	1.00	0.45	0.39	1.00	0.35	0.30	1.00	0.45	0.38	1.00	0.35	0.30
$C = AB$															
cfd2	14,216	**3,325**	3,429	7,159	**1,871**	1,935	8,791	**1,360**	1,367	4,971	**1,092**	1,140	5,963	**988**	1,017
boneS01	28,430	3,043	**2,989**	13,640	**1,576**	1,708	15,648	**1,179**	1,258	6,992	**1,016**	1,045	10,483	**905**	918
shipsec5	26,568	3,305	**3,236**	14,656	**1,800**	1,872	19,807	**1,286**	1,346	7,750	1,132	**1,127**	12,417	**953**	1,000
thermomech_dK	34,299	22,103	**21,557**	34,025	**11,605**	11,799	24,330	8,292	**7,475**	15,009	6,475	**6,359**	14,238	**5,095**	5,266
offshore	24,088	**9,488**	9,670	15,157	5,361	**5,013**	15,392	3,786	**3,541**	9,649	3,061	**2,889**	9,346	2,717	**2,476**
amazon0302	8,886	**7,411**	7,474	5,022	3,954	**3,920**	6,998	2,826	**2,793**	4,930	2,282	**2,260**	4,318	**1,891**	1,978
amazon0312	22,306	19,897	**19,057**	15,425	10,800	**10,051**	17,656	7,871	**7,431**	13,814	6,147	**5,599**	11,674	**5,047**	5,090
$C = AB$ (Graph500)															
scale=15	14,498	**12,362**	12,814	7,738	6,541	**6,537**	9,342	4,663	**4,501**	4,039	3,702	**3,587**	5,381	3,159	**2,920**
scale=16	29,010	**25,494**	26,976	14,348	**13,200**	13,704	16,136	9,293	**9,172**	7,530	**7,162**	7,236	11,131	6,079	**5,856**
scale=17	62,098	60,138	**59,154**	34,251	**30,955**	31,535	35,721	**21,170**	21,173	16,650	**16,006**	16,410	24,192	**13,063**	13,249
scale=18	137,408	**122,971**	125,845	68,557	64,576	**64,350**	74,625	**43,305**	44,737	36,117	34,335	**34,205**	53,888	**27,252**	27,574
norm avgs wrto BL	1.00	0.49	0.49	1.00	0.45	0.46	1.00	0.29	0.28	1.00	0.43	0.43	1.00	0.30	0.30

BL : SpGEMM implementation provided in the Graphulo library

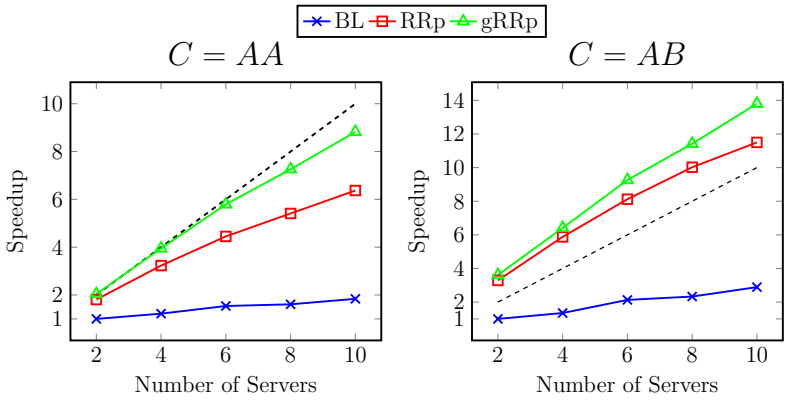
RRp : The proposed row-row parallel SpGEMM algorithm

gRRp : RRp algorithm enhanced via the proposed graph partitioning scheme

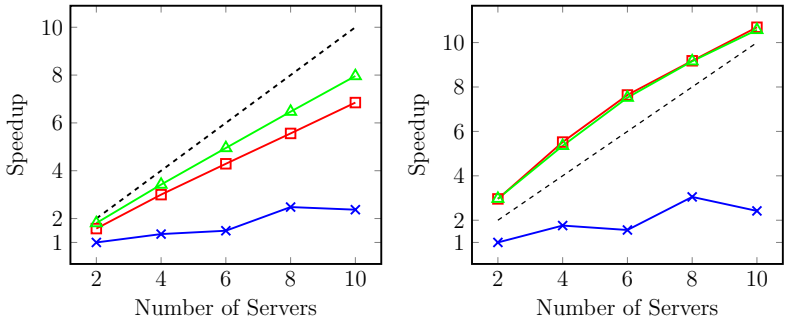
The performance of RRp is further improved by gRRp on almost all SpGEMM instances. On average, gRRp provides an improvement of 12% over RRp on $K = 2$ servers and this improvement significantly increases to 32% on $K = 10$ servers in the $C = AA$ category. In the $C = AB$ category, gRRp performs 12% better than RRp on $K = 2$ servers and this improvement slightly increases to 13% on $K = 10$ servers. These results indicate that the graph partitioning approach increases the scalability of RRp, since the decrease in the communication volume among tablet servers increases the efficiency of parallel algorithm.

Figure 4.3(a) displays the average speedup curves for the multiplication phase over each SpGEMM category. Speedup values on an SpGEMM instance are attained with respect to the running time of BL for the same instance on $K = 2$ tablet servers. In both categories, both RRp and gRRp scale linearly and display significantly better scalability than BL. For example, on $K = 10$ servers, RRp and gRRp achieve speedup values of 6.37 and 8.82 respectively, whereas BL achieves only 1.84 in the $C = AA$ category. Similarly, these speedup values become 11.50, 13.80 and 2.89 respectively in the $C = AB$ category. Additionally, as can be inferred from the speedup curves, the efficiency of RRp is further increased by gRRp and higher speedup values are obtained via intelligent partitioning of input matrices. As also seen in Figure 4.3(a), the performance improvement of gRRp over RRp on the scalability is much more pronounced in the $C = AA$ category than in the $C = AB$ category.

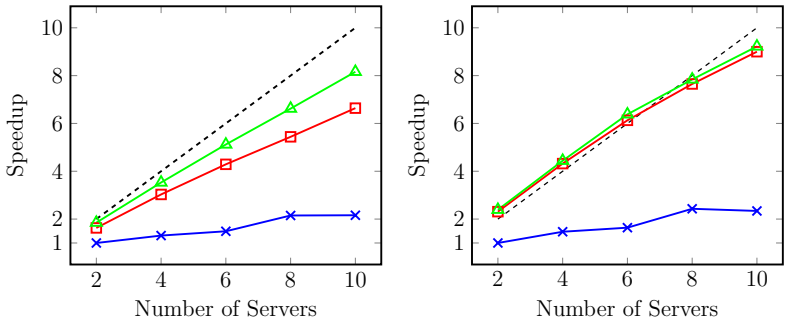
Table 4.3 displays the times required to scan C matrices produced by all algorithms after they perform respective SpGEMM instances. As seen in the table, scanning phase of BL requires significantly more time than those of RRp and gRRp due to the summing-combiner applied on table C by Graphulo library. In the $C = AA$ category, on average, scanning phase of RRp and gRRp algorithms runs 1.55x and 1.78x faster than that of BL on $K = 2$ servers and this performance improvement increases to 2.85x and 3.33x on $K = 10$ servers, respectively. Similarly in the $C = AB$ category, scanning phase of RRp and gRRp runs 2.04x faster than that of BL on $K = 2$ servers and this improvement increases to 3.33x on $K = 10$ servers, respectively.



(a) Speedup curves for multiplication phase.



(b) Speedup curves for scanning phase.



(c) Speedup curves for the overall execution time.

Figure 4.3: Average speedup curves of BL, RRp and gRRp with respect to the running time of BL on $K = 2$ tablet servers. a) Multiplication phase. b) Scanning phase. c) Overall execution time.

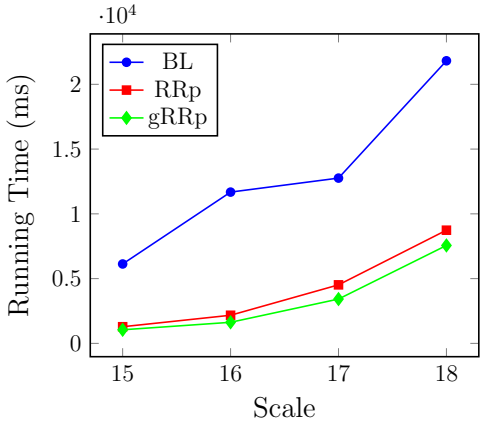


Figure 4.4: Running times of BL, RRp and gRRp to perform SpGEMM instances, which are generated by graph500 tool, on $K = 10$ tablet servers.

Figure 4.3(b) displays the average speedup curves for the scanning phase over each SpGEMM category. As seen in Figure 4.3(b), both RRp and gRRp displays significantly better scalability than BL. Comparison of Figures 4.3(a) and 4.3(b) show that the performance improvement of gRRp over RRp is considerably less on the scanning phase than the multiplication phase, as discussed earlier. This is because the scanning phase involves communication only between the control server and tablet servers (i.e., no communication among tablet servers) and the graph partitioning only encapsulates the amount of communication volume during the multiplication phase. The considerable performance improvement of gRRp over RRP in $C = AA$ category can be attributed to the fact that gRRp provides better nonzero distribution of C matrix among tablet servers, due to the third constraint placed on vertices (i.e., the w_i^3 weight), as compared to RRp, and therefore, achieves better load balance during the scan of C-matrix.

It is worth to mention that speedup values attained by BL in the scanning phase decrease as the number of tablet servers increases in some cases (e.g., when the number of servers increases from 8 to 10 in Figure 3 (b)). This performance decrease can be mainly attributed to the partitioning scheme used in BL where input matrices are partitioned rowwise among servers without considering the workload associated with these rows (i.e., each server is assigned $\lfloor n/K \rfloor$ rows). However, the computational load associated with each row may drastically deviate, since

these rows, especially dense rows, may necessitate many summing-combining operations due to the partial results contributing to the same nonzero entries. Thus BL suffers from load imbalance due to uneven nonzero entry distribution among tablet servers, leading to lower speedup values.

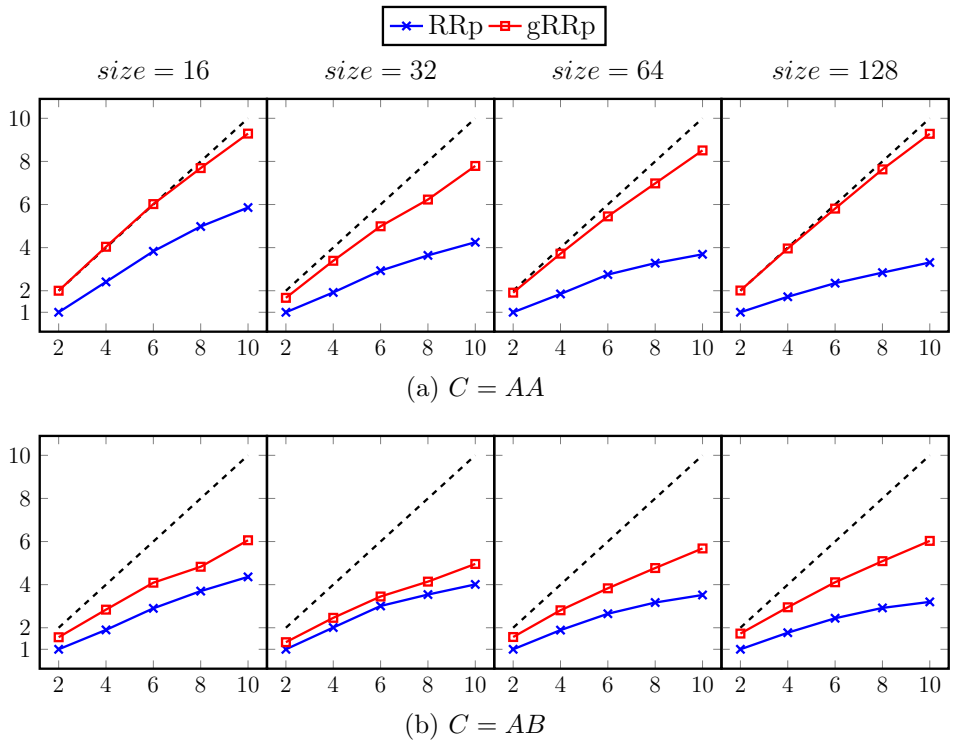
Figure 4.3(c) displays the average speedup curves for the overall execution time (i.e. the total time spent on multiplication plus scanning phases) of the SpGEMM operations. As seen in Figure 4.3(c), both RRp and gRRp scales significantly better than BL in the total execution time.

Figure 4.4 displays the variation of the running times of the multiplication phase of all algorithms on $K = 10$ servers with increasing scale factor of SpGEMM instances that are produced by the graph500 tool. This figure is included here in accordance with the experimental results reported in [27]. As seen in the figure, both RRP and gRRp perform significantly better than BL, whereas gRRp performs slightly better than RRp.

4.4.5 Varying Key-Value size

In this section, we report the results of the experiments conducted to show the variation of the performance improvement of gRRp over RRp with increasing key-value pair sizes. For the previous experiments, we designate the sizes of row, column and value fields of key-value pairs as 8 bytes each (i.e., the size of each nonzero is 24 bytes). Hence, in order to obtain different sized key-value pairs for an SpGEMM instance, we keep the sizes of row and column fields fixed and set the size of the value field to 16, 32, 64 and 128-bytes. The scaler multiplication of two value fields is performed via Java's BigDecimal Class.

In Figure 4.5, we plot the average speedup curves of RRp and gRRp, with different key-value sizes, over each SpGEMM category. For each SpGEMM instance and value-field size, speedup values are computed with respect to the running time of RRp on $K = 2$ tablet servers. As seen in the figure, the performance improvement of gRRp over RRp increases significantly for both categories with



(a) $C = AA$

(b) $C = AB$

Figure 4.5: Average speedup curves of RRp and gRRp with varying key-value sizes. Speedup values are computed with respect to the running times of RRp on $K = 2$ tablet servers.

increasing key-value pair size. For example, for the $C = AA$ category on $K = 10$ servers, gRRp runs 1.58x, 1.83, 2.30 and 2.80x faster than RRp for value-field sizes of 16, 32, 64 and 128 bytes, respectively. Similarly, for the $C = AB$ category on $K = 10$ servers, gRRp runs 1.39x, 1.24x, 1.61x and 1.88x faster than RRp for value-field sizes of 16, 32, 64 and 128 bytes, respectively. This increase in the performance improvement of gRRp over RRp is because of the fact that the communication volume increases with increasing value-field sizes and hence the improvements to be attained by graph partitioning, which has the objective of minimizing total communication volume, become more pronounced on the overall performance. So the preprocessing overhead is expected to amortize in such applications, since the partitioning overhead is independent of the sizes of key-value pairs.

4.5 Conclusion

We proposed an iterator algorithm to perform distributed SpGEMM in Accumulo database. The proposed algorithm utilizes row-by-row parallel SpGEMM which achieves write-locality during ingestion of the output matrix. However, this approach necessitates performing multiple batch-scan operations, which introduces significant latency overheads. These overheads are alleviated by performing local SpGEMM operations via multiple batches and utilizing multi-threaded parallelism. Extensive experiments performed on a wide range of realistic and synthetic SpGEMM instances showed that the proposed algorithm outperforms the outer-product implementation provided in the Graphulo library, by a large margin.

We also proposed a matrix partitioning scheme that reduces the total communication volume while maintaining workload balance among servers. The experiments also showed that the proposed matrix partitioning scheme provides significant improvements. The preprocessing overhead due to graph partitioning is expected to amortize in applications that require repeated SpGEMMs involving input matrices having the same sparsity patterns as well as applications that

have large key-value sizes. The latter is because of the fact that the partitioning overhead is independent of the sizes of key-value pairs, whereas communication-volume overhead increases with increasing sizes of key-value pairs thus increasing the performance improvement attained by using graph partitioning.

The proposed SpGEMM algorithm adopts 1D matrix partitioning scheme in which matrices are rowwise partitioned among tablet servers. However, 1D algorithms face communication bottlenecks when the number of tablet servers increases, since the total communication volume and the number of messages need to be handled by a single server may drastically increase. In order to address this issue, as a future work, we will also investigate 2D matrix partitioning schemes in which matrices are partitioned both rowwise and column-wise among servers.

Iterative methods to solve problems such as sparse non-negative matrix factorization [97–100] heavily depend on repeated sparse matrix and low-rank matrix multiplication. We observed significant computational and latency overheads during repeated invocation of iterator algorithms in Accumulo. Therefore, we are planning to investigate potential performance enhancements to the proposed SpGEMM iterator algorithm for efficient use in such cases.

Chapter 5

Cartesian Partitioning Models for 2D and 3D Parallel SpGEMM Algorithms.

Sparse general matrix multiplication (SpGEMM) is a kernel operation in many scientific computing applications such as finite element simulations [101], molecular dynamics [102,103], linear programming (LP) [104,105] and linear solvers [17, 18]. Additionally, SpGEMM is also utilized in high-performance graph computations such as graph contraction [19], betweenness centrality computation [20], Markov clustering [21], triangle counting [22] and graph traversal [23].

Extensive research is made for parallelizing SpGEMM on various parallel computing platforms [1, 28, 29]. Considering the sparsity structure of the problem, much research is devoted to graph/hypergraph partitioning models to offer efficient data and task partitioning among processors [28, 30, 32]. The proposed graph/hypergraph partitioning models incur preprocessing overhead. Hence, applications, that involve repeated SpGEMM in which the sparsity patterns of input matrices remain the same in all iterations, benefit more from these models. For instance, as mentioned in [30, 32], similarity join [106] and collaborative filtering [94] algorithms utilize SpGEMM of the forms $C = AWA$ or $C = AWB$ where

W-matrix is used for adjusting feature ranking and importance of items in the filtering. Additionally, LP problems are solved via interior point methods which require the solution of linear system $(AD^2A^T)x = b$. In each iteration, the linear system is solved via forming the coefficient matrix $C = AB$ where the sparsity patterns of both A and $B = D^2A^T$ matrices remain unchanged.

Iteration space of SpGEMM operation can be visualized as a sparse 3D cube (workcube) and parallel SpGEMM algorithms are categorized according to the partitioning of this workcube [2]. In this categorization, 1D, 2D and 3D algorithms are defined according to the number of dimensions by which the workcube is partitioned. Efficient implementations of 1D, 2D and 3D parallel SpGEMM algorithms are given in [1, 2, 30]. An important drawback of 1D parallel SpGEMM algorithms is that these algorithms face communication bottlenecks, since the volume/number of messages handled by processors may drastically increase with increased number of processors. However, by utilizing additional dimensions in processor grids and partitioning the iteration space in multiple dimensions, these overheads can be significantly reduced.

In this paper, we propose hypergrah partitioning models for 2D and 3D parallel SpGEMM algorithms given in [1, 2]. In other words, we offer intelligent matrix partitioning schemes to improve the scalability and efficiency of these algorithms. The graph/hypergraph models given in [28, 30, 32] partition the iteration space of SpGEMM in a single dimension and can be considered as 1D parallel SpGEMM algorithms. The fine-grained hypergraph model proposed in [31] achieves a multi-dimensional partitioning on the SpGEMM's workload, but the task distribution is performed on processors that are logically arranged in a single dimension, thus becoming 1D algorithm as well.

We conduct extensive experiments to evaluate our partitioning models on SpGEMM instances arising from real-world applications as well as on synthetically generated instances. Experimental results demonstrate that our models provide significant improvements for the algorithms given in [1, 2] and improve these algorithms' scalability and efficiency on real-world datasets. On synthetically generated instances, improvements of our models are slightly less due to the

limited availability of optimization on these datasets.

The rest of the paper is organized as follows. Section 5.1 present related work. Section 5.2 summarizes the existing 2D and 3D SpGEMM algorithms in the literature. Sections 5.3.1 and 5.3.2 describe our proposed Hypergraph partitioning models for 2D and 3D SpGEMM algorithms. Section 5.4 presents experimental results. Finally, Section 5.5 concludes the paper.

5.1 Related Work

In the literature, there are extensive research work considering parallelization of SpGEMM algorithms for various shared-memory parallel architectures. An efficient SpGEMM implementation is provided by Intel's math kernel library (MKL) [107]. Various partitioning and cache optimization techniques, which significantly improve the performance of MKL, are studied by Patwary et al. [84]. Hypergraph and bipartite graph models that exploit spatial and temporal locality of row-by-row parallel SpGEMM on Intel Xeon Phi many core processor architecture are considerd by Akbudak and Aykanat [28]. Studies considering GPU architectures also exist [85, 89, 96, 108]; and libraries such as CUSPARSE [109] and CUSP [110] provide efficient GPU-based SpGEMM implementations.

SpGEMM algorithms are also extensively studied for distributed memory systems. There are publicly available libraries such as Trilinos [86] and Combinatorial BLAS (CombBLAS) [20] which offer efficient distributed memory SpGEMM implementations. CombBLAS library provides an implementation of sparse SUMMA algorithm [1] which operates on a two-dimensional (2D) virtual processor grid and utilizes 2D block partitioning of input and output matrices. To perform local computations, this algorithm uses a sequential SpGEMM kernel based on heap and double-compressed-sparse-column data structures. By considering an extra third dimension in the virtual processor grid, Azad et al. [2] propose an extension for the sparse SUMMA algorithm. Their algorithm also extends the sequential SpGEMM kernel in sparse SUMMA algorithm by considering multithreaded execution. Akbudak and Aykanat [32] consider a distributed SpGEMM

outer-product formulation and propose hypergraph-partitioning-based models for effective task and data distribution. Hypergraph and bipartite graph partitioning models are studied in [30] for outer-product, inner-product and row-by-row-product formulations for distributed-memory SpGEMM. Ballard et al. [31] propose a fine-grain hypergraph model that allows multi-dimensional partitioning of the work required by SpGEMM for different class of algorithms (e.g., 1D/2D/3D SpGEMM algorithms).

In [111, 112], theoretical lower bounds on communication costs of parallel SpGEMM algorithms are studied. A survey of parallel SpGEMM algorithms is given in [113], along with their theoretical assessment of the expected communication costs on random matrices. Sparsity-dependent communication lower bounds for parallel SpGEMM algorithms are also discussed through the use of the fine-grain hypergraph model in [31].

In the fine-grain model, each nonzero scalar multiplication is represented by a vertex and each nonzero entry of the input and output matrices is also represented by a different vertex. Therefore, it is able to determine task and data distribution simultaneously. It can model different classes of SpGEMM algorithms (i.e., 1D/2D/3D algorithms) by using hypergraph coarsening methods. The fine-grain hypergraph model, however, is described as a theoretical approach [31] and is found to be impractical [2, 30] due to the hypergraph's considerably large size. The fine-grain model achieves multi-dimensional partitioning in a single partitioning phase without taking into account the processor arrangement (i.e., without considering dimensionality in processor grid).

Our proposed hypergraph partitioning models significantly differ from the fine-grain model given in [31]. In order to match the sizes of the virtual processor grid and to capture the nice upper-bounds provided by multi-dimensional grid on the number of messages handled by processors, we suggest a multi-phase partitioning framework. Within the multi-phase partitioning framework, we suggest multi-constraint partitioning formulations to encode computational load-balance among processors.

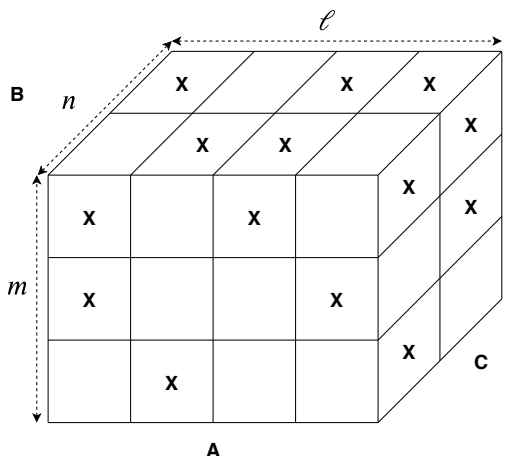


Figure 5.1: Workcube W of an SpGEMM instance $C = AB$ where $A \in \mathbb{R}^{3\times4}$, $B \in \mathbb{R}^{4\times2}$ and $C \in \mathbb{R}^{3\times2}$. "x" denotes a nonzero entry in the respective matrix. Intersections of projections of nonzero entries produce voxels in W.

5.2 SpGEMM Algorithms

5.2.1 Workcube Representation

Given matrices $A \in \mathbb{R}^{m\times\ell}$, $B \in \mathbb{R}^{\ell\times n}$ and $C \in \mathbb{R}^{m\times n}$, a sparse 3D cube (workcube) W of size $m\times\ell\times n$ can be utilized to represent the iteration space of the SpGEMM operation $C=AB$ (See Figure 5.1) [2, 31]. In W, each voxel $W(i,j,k)$, whose projections onto A- and B-faces contain nonzero entries, represents a nontrivial scalar multiplication $A(i,k)B(k,j)$ (i.e., both $A(i,k)\neq0$ and $B(k,j)\neq0$). The nonzero pattern of matrix C is determined by projections of these voxels onto C-face. Subcubes of W, which are respectively called "layers" and "fibers", can be obtained by fixing one and two indices. So, $W(i,:,:), W(:,j,:)$ and $W(:,:,k)$ denote the ith horizontal, jth frontal and kth lateral layers, respectively. Fibers, which are denoted by $W(i,:,j)$, $W(i,k,:)$ and $W(:,j,k)$, are obtained through intersecting layers along different dimensions. For instance, the intersection of ith horizontal and jth frontal layers is fiber $W(i,:,j)$.

Horizontal layer $W(i,:,:)$ represent all computations using the nonzeros of row $A(i,:)$ and the task of computing row $C(i,:)$. Frontal layer $W(:,j,:)$ represent all computations using column $B(:,j)$ and the task of computing column $C(:,j)$. Lateral layer $W(:,:,k)$ represent computation of the outer-product of column $A(:,k)$ with row $B(k,:)$.

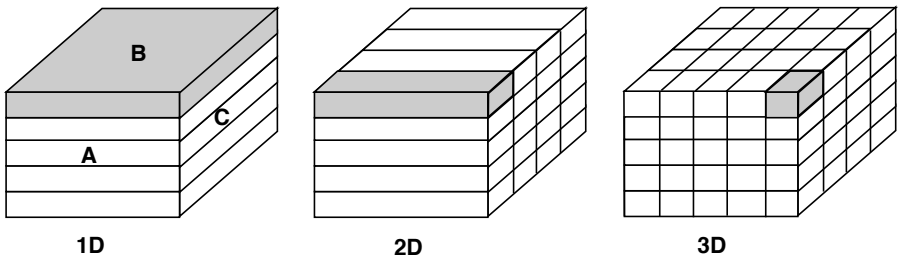


Figure 5.2: Workcube partitioning for 1D, 2D and 3D SpGEMM algorithms. 1D algorithm partitions horizontal layers, 2D algorithm partitions both horizontal and frontal layers, 3D algorithm partitions horizontal, frontal and lateral layers among processors. Gray shaded areas show horizontal blocks, fiber blocks and cuboids assigned to a processor.

In [2], distributed-memory SpGEMM algorithms are categorized depending on the number of dimensions by which the workcube is partitioned. In this categorization, 1D, 2D and 3D algorithms are defined (see Figure 5.2). 1D, 2D and 3D algorithms necessitates communication on one, two and all three of the matrices, respectively. Communication on an input matrix relates to the expand-type communications of the corresponding input matrix nonzero entries. Communication on the output matrix relates to the fold-type communications on the partial results for the same output matrix entries generated by distinct processors. In the following two subsections, we concisely describe 2D- and 3D-parallel SpGEMM algorithms for which we propose intelligent partitioning schemes.

5.2.2 2D: Sparse SUMMA algorithm [1]

In sparse SUMMA algorithm, multiplication $C = AB$ is performed on a 2D virtual $p_x \times p_y = p$ processor grid. The xth processor-row and the yth processor-column of the 2D grid are respectively denoted by $P_{x,:}$ and $P_{:,y}$. So the processor in the xth row and the yth column of the grid is denoted by $P_{x,y}$.

A, B and C matrices are partitioned into 2D blocks in such a way that each matrix is partitioned rowwise among p_x processor-rows and columnwise among p_y processor-columns (see Figure 5.3). Rows of C are partitioned conformably with rows of A, whereas columns of C are partitioned conformably with columns of

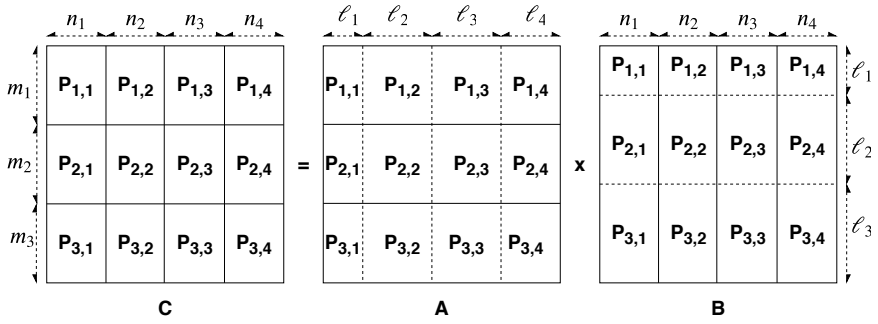


Figure 5.3: 2D block partitioning of matrices on 3×4 processor grid. Solid lines show that A-matrix rows and B-matrix columns are partitioned conformably with the workcube/task partition. Dotted lines show that B-matrix rows and A-matrix columns are partitioned independent from the task partitioning.

B. Submatrices $A_{x,y}$ and $B_{x,y}$ are stored by $P_{x,y}$ and submatrix $C_{x,y}$ is computed by the same processor. Let m_x and ℓ_x be the number of A-/C-matrix rows and B-matrix rows assigned to processor-row $P(x,:)$ and let n_y and ℓ_y be the number of B-/C-matrix columns and A-matrix columns assigned to processor-column $P(:,y)$. Then, submatrices $A_{x,y}$, $B_{x,y}$ and $C_{x,y}$ are of sizes $m_x \times \ell_y$, $\ell_x \times n_y$ and $m_x \times n_y$, respectively.

Under this data distribution among processors, all submatrices along the xth row block of A are needed by each processor $P_{x,y}$ in processor-row $P_{x,:}$. Similarly, all submatrices along the yth column block of B are needed by each processor $P_{x,y}$ in processor-column $P_{:,y}$. Hence, each processor $P_{x',y'}$ broadcasts its local submatrix $A_{x',y'}$ along the processor-row $P_{x',:}$ as well as local submatrix $B_{x',y'}$ along processor-column $P_{:,y'}$. This 2D partitioning strategy provides an upper bound on the volume and the number of messages exchanged between processors during these broadcast operations, since the collective communication operations are restricted to the rows and columns of the processor grid.

The parallel sparse SUMMA algorithm [1] performs the collective communication operations in stages in order to reduce the local memory requirements of the processors. Although this approach reduces the processors' local memory requirement, it increases the latency overhead due to the significantly increased number of collective operations.

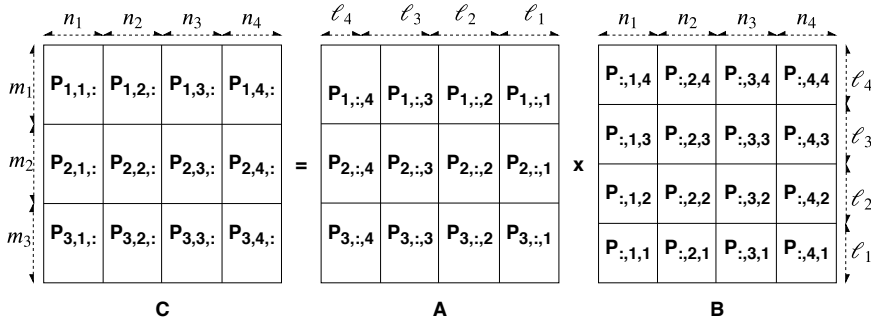


Figure 5.4: 3D partitioning of the workcube on a $3 \times 4 \times 4$ grid. Solid lines show that rows and columns of all matrices are partitioned conformably with the workcube/task partition.

In this algorithm, the workcube is partitioned into $p_x \times p_y$ fiber blocks. Each fiber block $W_{x,y,:}$ is of size $m_x \times \ell \times n_y$ and the tasks/voxels in the fiber block $W_{x,y,:}$ is computed by processor $P_{x,y}$. In other words, $p_x \times p_y$ 2D block partitioning is obtained on the C-face of the workcube and each submatrix $C_{x,y}$ is locally computed by $P_{x,y}$. Figure 5.5 displays a sample workcube partitioning on a 2D grid of size $3 \times 4 = 12$.

5.2.3 3D: Split-3D-SpGEMM algorithm [2]

Split-3D-SpGEMM algorithm runs on 3D virtual $p_x \times p_y \times p_z = p$ processor grid and extends 2D algorithm [1] by considering an additional third dimension. Let $P_{x,y,z}$ be the processor in the xth horizontal, the yth frontal and the zth lateral layer of the grid and let the xth horizontal, the yth frontal and the zth lateral layers of the 3D grid are respectively denoted by $P_{x,:,:}$, $P_{:,y,:}$ and $P_{:,:,z}$. Moreover, let $P_{x,y,:}$, $P_{x,:,z}$ and $P_{:,y,z}$ denote the processor-fibers obtained by intersecting the respective processor-layers.

This algorithm splits the 2D blocks of 2D-partitioned A, B and C matrices into distinct third-dimensional disjoint subblocks. That is, A, B and C matrices are partitioned into 2D blocks among $p_x \times p_z$, $p_y \times p_z$ and $p_x \times p_y$ processors, respectively (see Figure 5.4). Then, 2D blocks of A, B and C matrices are split into subblocks among p_y, p_x and p_z processors, respectively. For instance, p_y subblocks

$A_{x,1,z}, A_{x,2,z}, \ldots, A_{x,y,z}$ are obtained by splitting each 2D block $A_{x,z}$ of matrix A and these subblocks are respectively assigned to processors $P_{x,1,z}, P_{x,2,z}, \ldots, P_{x,y,z}$ of processor-fiber $P_{x,:,z}$.

The broadcast of $A_{x,z}$ from a single processor $P_{x,z}$ in 2D algorithm is split and concurrently performed between p_y processors in this novel scheme. That is, processors $P_{x,1,z}, P_{x,2,z}, \ldots, P_{x,y,z}$ concurrently broadcast A-matrix subblocks $A_{x,1,z}, A_{x,2,z}, \ldots, A_{x,y,z}$, respectively. Similarly, the broadcast of $B_{y,z}$ is split among p_x processors. Because of C-matrix blocks are also split along the z dimension, local partial C-subblock results are combined through reduce (fold) type of operations along the z dimension to obtain the final C-block results. The 3D algorithm is more scalable compared to the 2D algorithm and offers better upper bounds on the communication overheads [2] at the cost of reduce-type operations along the z dimension.

In this algorithm, the workcube is partitioned into $p_x \times p_y \times p_z$ cuboids. Each cuboid $W_{x,y,z}$ is of size $m_x \times \ell_z \times n_y$ and the tasks/voxels in the cuboid $W_{x,y,z}$ is computed by processor $P_{x,y,z}$. In other words, $p_x \times p_y$ 2D block block partitioning is applied on the C-face of the workcube and each subblock by $C_{x,y}$ is collectively computed among the processors of the processor-fiber $P_{x,y,:}$. Figure 5.6 displays a sample workcube partitioning among $3 \times 4 \times 4 = 48$ processors.

5.3 Partitioning Models

In this section, we propose hypergraph models, that achieve 2D and 3D cartesian partitioning of the workcube, to improve the performance of 2D- and 3D-parallel SpGEMM algorithms given in Sections 5.2.2 and 5.2.3, respectively.

5.3.1 2D Cartesian Partitioning of Workcube

2D cartesian partitioning of the workcube is performed through two partitioning phases ϕ_1 and ϕ_2. In ϕ_1, a p_x-way partitioning on the horizontal layers of the workcube is obtained and each part is assigned to a distinct processor-row. In phase ϕ_2, a p_y-way partitioning on the frontal layers of the workcube is obtained and each part is assigned to a distinct processor-column.

For phase ϕ_1, we define a hypergraph $H(\phi_1) = \{V^H, N^L\}$ with m vertices, ℓ nets and $nnz(A)$ pins. Here, $nnz(\cdot)$ corresponds to the number nonzeros in the respective matrix. $H(\phi_1)$ contains a vertex $v_i^H \in V^H$ for each horizontal layer $W(i,:,:)$ and each vertex v_i^H represents the computation of $C(i,:)$. $H(\phi_1)$ contains a net $n_k^L \in N^L$ for each lateral layer $W(:,:,k)$ and each net n_k^L represents row $B(k,:)$. A net $n_k^L \in N^L$ connects each vertex v_i^H for which the intersection of horizontal layer $W(i,:,:)$ and fiber $W(i,:,k)$ is nonempty. Formally,

$$pins(n_k^L) = \{v_i^H \mid \exists W(i,j,k) \in W(i,:,:) \cap W(:,:,k)\}$$

Alternatively, in matrix theoretic view,

$$pins(n_k^L) = \{v_i^H \mid \exists k \in cols(A(i,:))\},$$

where $cols(A(i,:))$ is the set of column indices of the nonzeros in row $A(i,:)$.

We associate each vertex v_i^H with a weight $w(v_i^H)$ that is equal to the number of voxels in horizontal layer $W(i,:,:)$. That is,

$$w(v_i^H) = \sum_{k \in cols(A(i,:))} nnz(B(k,:)) = |W(i,:,:)|.$$

We associate each net with a cost equal to the number of nonzeros in the respective B-matrix row, i.e.,

$$cost(n_k^L) = nnz(B(k,:))$$

A p_x-way partition $\Pi_{p_x}(\phi_1) = \{V_1^H, V_2^H, \ldots, V_{p_x}^H\}$ of $H(\phi_1)$ induces the following task partitioning: All tasks corresponding to vertices in $V_x^H \in \Pi_{p_x}(\phi_1)$ are assigned to processor-row $P_{x,:}$. That is, the task of computing an individual row

of matrix C is performed by the processors of the same row of the grid. Additionally, this partitioning induces a partial reordering on the horizontal layers so that the horizontal layers belonging to the same part are reordered consecutively (in any order) to form a horizontal block. This partially reordered workcube will be referred to as $\overline{W}$. Then, the tasks corresponding to the xth horizontal block $\overline{W}_{x,:,:}$ of $\overline{W}$ is assigned to processor-row $P_{x,:}$.

The weight of a part V_x^H is equal to the number of voxels in the horizontal block $\overline{W}_{x,:,:}$. Hence, maintaining balance on part weights encodes the balance on the p_x horizontal blocks' voxel counts, thus encoding computational balance among p_x processor-rows.

For a cut-net n_k^L with connectivity set $\Lambda(n_k^L)$, each part $V_x^H \in \Lambda(n_k^L)$ corresponds to processor-row $P_{x,:}$ which is assigned the horizontal layers whose intersection with lateral layer $W(:,:,k)$ is nonempty. In other words, for each part $V_x^H \in \Lambda(n_k^L)$, all tasks assigned to processor-row $P_{x,:}$ in $\Lambda(n_k^L)$ require row $B(k,:)$.

Here, a B-matrix row distribution is consistent with task partition $\Pi(\phi_1)$ if each row $B(k,:)$ is stored by one of the processor-rows in $\Lambda(n_k^L)$. The distribution of nonzeros of a B-matrix row along the respective processor-row is determined by the partition obtained in phase ϕ_2. Under this data distribution, assume that row $B(k,:)$ is stored by processor-row $P_{x,:}$ in $\Lambda(n_k^L)$. Processor-row $P_{x,:}$ expands row $B(k,:)$ to all processor-rows in $\Lambda(n_k^L) - \{P_{x,:}\}$, thus cut-net n_k^L incurs the communication of

$$nnz(B(k,:)) \times \big(|\Lambda(n_k^L)| - 1\big)$$

words. The total communication volume between processor-rows can be given as

$$\text{ExpVol}(B) = \sum_{n_k^L \in \mathcal{N}_e} nnz(B(k:)) \times \big(|\Lambda(n_k^L)| - 1\big)\,.$$

Therefore, minimizing the cutsize according to Eq. (2.3) corresponds to minimizing the total communication volume on B-matrix rows.

For phase ϕ_2, we define a hypergraph $H(\phi_2) = \{V^F, N^L\}$ with n vertices, ℓ nets and $nnz(B)$ pins. $H(\phi_2)$ contains a vertex $v_j^F \in V^F$ for each frontal layer $W(:,j,:)$

and each vertex v_j^F represents the computation of column $C(:,j)$. $H(\phi_2)$ contains a net $n_k^L \in N^L$ for each lateral layer $W(:,:,k)$ and each net n_k^L represents column $A(:,k)$. A net n_k^L connects each vertex v_j^F for which the intersection of frontal layer $W(:,j,:)$ and lateral layer $W(:,:,k)$ is nonempty. Formally,

$$pins(n_k^L) = \{v_j^F \mid \exists W(i,j,k) \in W(:,j,:) \cap W(:,:,k)\}$$

Alternatively, in matrix view,

$$pins(n_k^L) = \{v_j^F \mid \exists k \in rows(B(:,j))\},$$

where $rows(B(:,j))$ denotes the set of row indices of the nonzeros in row $B(j,:)$. We associate each net n_k^L with a cost equal to the number of nonzeros in the respective A-matrix column, i.e.,

$$cost(n_k^L) = nnz(A(:,k)).$$

A p_y-way partition $\Pi_{p_y}(\phi_2) = \{V_1^F, V_2^F, \ldots, V_{p_y}^F\}$ of hypergraph $H(\phi_2)$ induces the following task partitioning: All tasks corresponding to vertices in $V_y^F \in \Pi_{p_y}(\phi_2)$ are computed by processor-column $P_{:,y}$. That is, the task of computing an individual column of matrix C is restricted to the processors of the same column of the grid. This partitioning also induces a partial reordering on the frontal layers so that the frontal layers belonging to the same part are reordered consecutively (in any order) to form a frontal block. Then, frontal block $\overline{W}_{:,y,:}$ of the reordered workcube is computed by processor-column $P_{:,y}$.

The p_x-way horizontal partition $\Pi_{p_x}(\phi_1)$ together with p_y-way partition $\Pi_{p_y}(\phi_2)$ form fiber blocks $\overline{W}_{x,y,:}$ including voxels in the intersection of horizontal block $\overline{W}_{x,:,:}$ and frontal block $\overline{W}_{:,y,:}$ of the reordered workcube. So, the partition $(\Pi(\phi_1), \Pi(\phi_2))$ induces assigning fiber block $\overline{W}_{x,y,:}$ to processor $P_{x,y}$.

In phase ϕ_2, we employ a multi-constraint partitioning formulation to achieve a balanced voxel distribution on fiber blocks. For this purpose, we associate each vertex v_j^F of V^F with p_x weights $w^c(v_j^F)$ for $c = 1, 2, \ldots, p_x$. Here, $w^c(v_j^F)$ is set equal to the number of voxels in the intersection of frontal layer $W(:,j,:)$ and horizontal block $\overline{W}_{c,:,:}$. That is,

$$w^c(v_j^F) = |W(:,j,:) \cap \overline{W}_{c,:,:}|.$$

Here, $|\cdot|$ denotes the number of voxels in the respective subcube of W. Alternatively, in matrix view,

$$w^c(v_j^F) = \sum_{v_i^H \in V_c^H} |\{k \mid k \in cols(A(i,:)) \wedge k \in rows(B(:,j))|$$

For a given partition $\Pi_{p_y}(\phi_2) = \{V_1^F, V_2^F, \ldots, V_{p_y}^F\}$ of $H(\phi_2)$, the cth weight of part $V_y^F \in \Pi_{p_y}(\phi_2)$ is equal to the number of voxels in the fiber block $\overline{W}_{c,y,:}$. That is,

$$\begin{aligned} W^c(V_y^F) &= \sum_{v_j^F \in V_y^F} w^c(v_j^F) \\ &= \sum_{v_j^F \in V_y^F} |W(:,j,:) \cap \overline{W}_{c,:,:}| \\ &= |\overline{W}_{:,y,:} \cap \overline{W}_{c,:,:}| \end{aligned}$$

So, maintaining balance on the cth part weights corresponds to maintaining balance on the voxel counts of the fiber blocks in the horizontal block $W_{c,:,:}$. The horizontal partition $\Pi_{p_x}(\phi_1)$ already produces horizontal blocks having roughly equal number of voxels. Therefore, the single partitioning constraint in ϕ_1 together with the multiple (i.e., px constraints) partitioning constraints in ϕ_2 encodes maintaining balance on the voxel counts in the individual fiber blocks. Since each fiber block is assigned to a separate processor, the proposed formulation encodes the computational load-balance between processors in the 2D grid.

For a cut-net n_k^L with the connectivity set $\Lambda(n_k^L)$, each part $V_y^F \in \Lambda(n_k^L)$ corresponds to the processor-column $P_{:,y}$ which is assigned the frontal layers whose intersection with lateral layer $W(:,:,k)$ is nonempty. In other words, for each part $V_y^F \in \Lambda(n_k^L)$, the tasks assigned to the processor-column $P_{:,y}$ in $\Lambda(n_k^L)$ require column $A(:,k)$.

An A-matrix column distribution is consistent with task partition $\Pi(\phi_2)$ if each column $A(:,k)$ is stored by one of the processor-columns in $\Lambda(n_k^L)$. Under this distribution, assume that $A(:,k)$ is stored by processor-column $P_{:,y}$ in $\Lambda(n_k^L)$. Processor-column $P_{:,y}$ expands $A(:,k)$ to all processor-columns in $\Lambda(n_k^L) - \{P_{:,y}\}$

so that cut-net n_k^L incurs communication of

$$nnz(A(:,k)) \times \big(|\Lambda(n_k^L)| - 1\big)$$

words. The total communication volume between processor-columns is

$$\text{ExpVol}(A) = \sum_{n_k^L \in \mathcal{N}_e} nnz(A(:,k)) \times \big(|\Lambda(n_k^L)| - 1\big)\,.$$

Therefore, minimizing the cut size according to Eq. (2.3) corresponds to minimizing the total communication volume on A-matrix columns.

As discussed earlier, task partition $\Pi(\phi_1)$ induces a distribution on B-matrix rows among processor-rows. The distribution of nonzeros of a B-matrix along a processor-row is determined by $\Pi(\phi_2)$ as follows: Assume that a processor-row $P_{x,:}$ in $\Lambda(n_k^L)$ stores row $B(k,:)$ by utilizing $\Pi(\phi_1)$. If $v_j^F \in V_y^F$ in $\Pi(\phi_2)$, each nonzero $B(k,j)$ of row $B(k,:)$ is stored by processor $P_{x,y}$. That is, the nonzero distribution of row $B(k,:)$ among processors of $P_{x,:}$ follows the partitioning obtained on the workcube's front layers. Expanding a B-matrix row from a processor-row is performed in such a way that each processor in that processor-row expands its local nonzero row segment along its processor-column. While the non-zero distribution of a B-matrix row may change the number of messages, it does not change the total volume of communication.

As also discussed earlier, task partition $\Pi(\phi_2)$ induces a consistent A-matrix column distribution among processor-columns. $\Pi(\phi_1)$ determines the distribution of nonzeros of an A-matrix column in a respective processor-column as follows: Assume that a processor-column $P_{:,y}$ in $\Lambda(n_k^L)$ stores column $A(:,k)$ by utilizing $\Pi(\phi_2)$. If $v_i^H \in V_x^H$ in $\Pi(\phi_1)$, each nonzero $A(i,k)$ of column $A(:,k)$ is stored by processor $P_{x,y}$. That is, the nonzero distribution of column $A(:,k)$ in processor-column $P_{:,y}$ follows the partitioning obtained on the workcube's horizontal layers. Expanding an A-matrix column from a processor-column is performed in such a way that each processor in that processor-column expands its local nonzero column segment along its processor-row. While the nonzero distribution of an A-matrix column may change the number of messages, it does not change the total volume of communication.

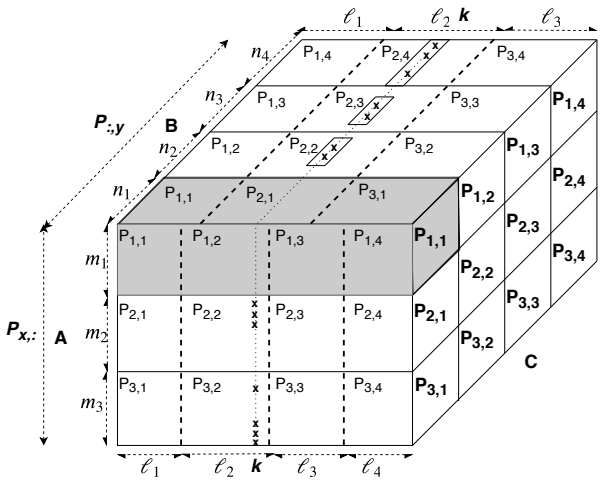


Figure 5.5: Sparse Summa (2D) algorithm. Partitioning of the workcube on a 3×4 2D grid. "x" denotes a nonzero entry in the respective matrix.

The discussion given in the above two paragraphs imply the following: The partitioning of A-matrix rows determines the total communication volume on B-matrix rows, whereas the partitioning of B-matrix columns does not have an effect on this volume. The total communication volume on A-matrix columns is determined by the partitioning of B-matrix columns whereas it is independent from the partitioning of A-matrix rows.

Figure 5.5 displays the proposed partitioning model. In phase ϕ_1, a cut-net n_k^L representing lateral layer $W(:,:,k)$ and row $B(k,:)$ has connectivity set $\Lambda(n_k^L) = \{P_{2,:}, P_{3,:}\}$, since horizontal-layer blocks $\overline{W}_{2,:,:}$ and $\overline{W}_{3,:,:}$ contain voxels in the intersection with the $W(:,:,k)$. Hence, nonzero row segments of row $B(k,:)$ are stored by processors $P_{2,2}$, $P_{2,3}$ and $P_{2,4}$ in processor-row $P_{2,:} \in \Lambda(n_k^L)$ and these processors expand the *three* row segments (drawn in *three* parallelograms) along their processor-columns. For example, processor $P_{2,2}$ expands its nonzero row segment to processor $P_{3,2}$ since fiber blocks $\overline{W}_{2,2,:}$ and $\overline{W}_{3,2,:}$ have voxels and require this row segment.

In phase ϕ_2, a net n_k^L representing lateral layer $W(:,:,k)$ and column $A(:,k)$ has connectivity set $\Lambda(n_k^L) = \{P_{:,2}, P_{:,3}, P_{:,4}\}$, since frontal-layer blocks $\overline{W}_{:,:,2}$, $\overline{W}_{:,:,3}$

and $\overline{W}_{:,:,4}$ contain voxels in the intersection with $W(:,:,k)$. Hence, nonzero column segments of column $A(:,k)$ are stored by processors $P_{2,2}$ and $P_{3,2}$ in processor-column $P_{:,2} \in \Lambda(n_k^L)$ and these processors expand these column segments along their processor-rows. For example, processor $P_{3,2}$ expands its nonzero column segment to processors $P_{3,3}$ and $P_{3,4}$ since fiber blocks $\overline{W}_{3,2,:}$, $\overline{W}_{3,3,:}$ and $\overline{W}_{3,4,:}$ have voxels and require this column segment.

5.3.2 3D Cartesian Partitioning of Workcube

The proposed model consists of three partitioning phases ϕ_1, ϕ_2 and ϕ_3. In the first phase ϕ_1, a p_x-way partitioning on the horizontal layers of the workcube is obtained and each part is assigned to a distinct horizontal layer of the processor grid. In the second phase ϕ_2, a p_y-way partitioning on the frontal layers of the workcube is obtained and each part is assigned to a distinct frontal layer of the processor grid. In the last phase ϕ_3, a p_z-way partitioning on the lateral layers of the workcube is obtained and each part is assigned to a distinct lateral layer of the processor grid. The hypergraph models $H(\phi_1)$ and $H(\phi_2)$ are the same with those proposed for the 2D partitioning model in Section 5.3.1.

For phase ϕ_3, we define a hypergraph $H(\phi_3) = \{V^L, N^Z\}$ with ℓ vertices, $nnz(C)$ nets and $|W|$ pins. $H(\phi_3)$ contains a vertex $v_k^L \in V^L$ for each lateral layer $W(:,:,k)$ and each vertex v_k^L represents the outer-product of column $A(:,k)$ with row $B(k,:)$. $H(\phi_3)$ contains a net $n_{i,j}^Z \in N^Z$ for each fiber $W(i,j,:)$ that has a voxel (partial product) contributing to nonzero entry $C(i,j)$. A net $n_{i,j}^Z$ connects each vertex v_k^L for which the intersection of lateral layer $W(:,:,k)$ and fiber $W(i,j,:)$ is nonempty. Formally,

$$pins(n_{i,j}^Z) = \{v_k^L \mid \exists W(i,j,k) \in W(i,j,:) \cap W(:,:,k)\}$$

Alternatively, in matrix view,

$$pins(n_{i,j}^Z) = \{v_k^L \mid \exists k \in cols(A(i,:)) \wedge \exists k \in rows(B(:,j))\}$$

We associate each net $n_{i,j}^Z$ with $cost(n_{i,j}^Z) = 1$.

A p_z-way partition $\Pi_{p_z}(\phi_3) = \{V_1^L, V_2^L, \ldots, V_{p_z}^L\}$ of hypergraph $H(\phi_3)$ induces the following task partitioning: All tasks corresponding to vertices in $V_z^L \in \Pi_{p_z}(\phi_3)$ are assigned to lateral layer $P_{:,:,z}$ of the processor grid. In other words, the task of computing an individual outer product of an A-matrix column with the corresponding B-matrix row is restricted to a lateral layer of the processor grid. Moreover, this partitioning induces a partial reordering on the lateral layers in such a way that the lateral layers belonging to the same part are reordered consecutively (in any order) to form a lateral block. Then, the responsibility of computing the zth lateral block $\overline{W}_{:,:,z}$ is given to the lateral processor layer $P_{:,:,z}$. The $p_x \times p_y$ fiber block partition induced by $(\Pi(\phi_1), \Pi(\phi_2))$ together with the p_z-way partition Π_{p_z} induces a $p_x \times p_y \times p_z$ cuboid partition such that a cuboid $\overline{W}_{x,y,z}$ contains voxels in the intersection of fiber block $\overline{W}_{x,y,:}$ and lateral block $\overline{W}_{:,:,z}$. Hence, the partition $(\Pi(\phi_1), \Pi(\phi_2), \Pi(\phi_3))$ is decoded as assigning cuboid $\overline{W}_{x,y,z}$ to processor $P_{x,y,z}$.

To maintain balance on the voxel counts of the cuboids, we associate each vertex $v_k^L \in V^L$ with $p_x \times p_y$ weights $w^{c,d}(v_k^L)$ for $c = 1, 2, \ldots, p_x$ and $d = 1, 2, \ldots, p_y$. For the sake of clarity of presentation, we denote constraints by two-dimensional array format (c, d), whereas they are actually stored as 1D vectors to be comformable with the input format requirements of the multi-constraint partitioners. Here, we set $w^{c,d}(v_k^L)$ equal to the number of voxels in the intersection of lateral layer $W(:, :, k)$ with fiber block $\overline{W}_{c,d,:}$ induced by the vertex parts V_c^H of $\Pi_{p_x}(\phi_1)$ and V_d^F of $\Pi_{p_y}(\phi_2)$. That is,

$$w^{c,d}(v_k^L) = |W(:, :, k) \cap \overline{W}_{c,d,:}|$$

Alternatively, in matrix view,

$$\begin{aligned} w^{c,d}(v_k^L) \;\; = \;\; & |\{C(i,j) \mid v_i^H \in V_c^H \wedge v_j^F \in V_d^F \\ & \wedge k \in cols(A(i,:)) \wedge k \in rows(B(:,j))\}| \end{aligned}$$

For a partition $\Pi_{p_z}(\phi_3) = \{V_1^L, V_2^L, \ldots, V_{p_z}^L\}$ of $H(\phi_3)$, the weight of part

$V_z^L \in \Pi_{p_z}(\phi_3)$ is equal to the number of voxels in the cuboid $\overline{W}_{c,d,z}$ That is,

$$\begin{aligned} W^{c,d}(V_z^F) &= \sum_{v_k^L \in V_z^L} w^{c,d}(v_k^L) \\ &= \sum_{v_k^L \in V_z^L} |W(:,:,k) \cap \overline{W}_{c,d,:}| \\ &= |\overline{W}_{:,:,z} \cap \overline{W}_{c,d,:}| = |\overline{W}_{c,d,z}|. \end{aligned}$$

So, maintaining balance on the (c,d)th weights of the parts corresponds to maintaining balance on the voxel counts of the cuboids in the horizontal fiber block $\overline{W}_{c,d,:}$. Recall that $(\Pi(\phi_1), \Pi(\phi_2))$ obtained in the first two phases already produces fiber blocks having roughly equal number of voxels. Hence, the single partitioning constraint in ϕ_1 and p_x partitioning constraints in ϕ_2 together with the $p_x \times p_y$ partitioning constraints in ϕ_3 ensures a balanced distribution on the voxel counts of the cuboids. The proposed multi-constraint partitioning model encodes the load balance between processors as each cuboid is computed by a distinct processor.

For a cut-net $n_{i,j}^Z$ with connectivity set $\Lambda(n_{i,j}^Z)$, each part $V_z^L \in \Lambda(n_{i,j}^Z)$ corresponds to a lateral processor-layer $P_{:,:,z}$ and $P_{:,:,z}$ that is assigned lateral layers of W whose intersection with fiber $W(i,j,:)$ is nonempty. Hence, each processor-layer $P_{:,:,z}$ corresponding to a part $V_z^L \in \Lambda(n_{i,j}^Z)$ produces partial results for $C(i,j)$. A C-matrix nonzero distribution is consistent with task partition $\Pi(\phi_3)$, if one of the processor-layers in $\Lambda(n_{i,j}^Z)$ stores and accumulates all partial results for $C(i,j)$.

Assume that the responsibility of storing the final value of $C(i,j)$ is given to processor-layer $P_{:,:,z}$ in $\Lambda(n_{i,j}^Z)$. Moreover, horizontal and frontal layers $W(i,:,:)$ and $W(:,j,:)$ are respectively assigned to horizontal and frontal processor-layers $P_{x,:,:}$ and $P_{:,y,:}$ by partition $(\Pi(\phi_1), \Pi(\phi_2))$. These assumptions induce the assignment of horizontal fiber $W(i,j,:)$ to processor-fiber $P_{x,y,:}$. Then, processor $P_{x,y,z}$ in processor-fiber $P_{x,y,:}$ will receive partial results for $C(i,j)$ from processors in $P_{x,y,:} \cap \{\Lambda(n_{i,j}^Z) - \{P_{:,:,z}\}\}$. Note that if a processor has multiple partial products for $C(i,j)$ assigned to another processor, a single partial result will be calculated by summation and sent to that processor. Hence, that cut-net $n_{i,j}^Z$ will incur

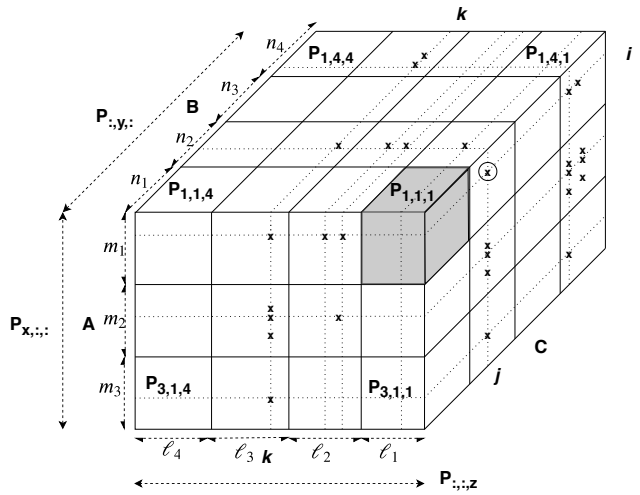


Figure 5.6: Split-3D-SpGEMM algorithm (3D) algorithm. Partitioning of the workcube on a $3\times4\times4$ 3D grid.

$\big(|\Lambda(n^Z_{i,j})|-1\big)$ words. So, the total communication volume associated with C-matrix nonzeros is

$$\text{FoldVol}(C) = \sum_{n^Z_{i,j}\in\mathcal{N}_e} \big(|\Lambda(n^Z_{i,j})|-1\big)$$

Therefore, minimizing the cut size according to Eq. (2.3) corresponds to minimizing the total communication volume during the fold type communications on C-matrix nonzeros.

Consistency of the distribution of nonzeros of C-matrix is defined earlier. Consistency of A-matrix columns and B-matrix rows with the overall task partitioning $(\Pi(\phi1),\Pi(\phi2),\Pi(\phi3))$ is an extension to the 2D discussion as follows: If a vertex v^L_k is assigned to $V^L_z \in \Pi(\phi_3)$, processor-layer $P_{:,:,z}$ stores column $A(:,k)$ and row $B(k,:)$. By utilizing $\Pi(\phi_1)$, assume that row $B(k,:)$ is assigned to a vertex part $V^H_x \in \Lambda(n^L_k)$ and correspondingly to processor-layer $P_{x,:,:}$. So, if $v^F_j \in V^F_y$ in $\Pi(\phi_2)$, processor $P_{x,y,z}$ stores nonzero $B(k,j)$ of row $B(k,:)$. That is, row $B(k,:)$ is stored by processors in processor-fiber $P_{x,:,z}$ and expanding B-matrix nonzero row segment(s) is performed along processor-fibers $P_{:,y',z}$. Similarly, assume that column $A(:,k)$ is assigned to a vertex part $V^F_y \in \Lambda(n^L_k)$ and correspondingly processor-layer $P_{:,y,:}$ by utilizing $\Pi(\phi_2)$. So, if $v^H_i \in V^H_x$ in $\Pi(\phi_1)$, processor $P_{x,y,z}$

stores nonzero $A(i,k)$ of column $A(:,k)$. That is, processor-fiber $P_{:,y,z}$ stores column $A(:,k)$ and expanding A-matrix nonzero column segment(s) is performed along processor-fibers $P_{x',:,z}$ of the grid.

The proposed partitioning model is depicted in Figure 5.6. In phase ϕ_1, cut-net n_k^L has $\Lambda(n_k^L)=\{P_{1,:,:}, P_{2,:,:}, P_{3,:,:}\}$. Hence, nonzero row segments of $B(k,:)$ can be stored by processors in one of the processor-layers in $\Lambda(n_k^L)$ and processors expand these row segments along their respective processor-fibers. For instance, since processor-layer $P_{:,:,3}$ is assigned lateral layer $W(:,:,k)$, processors $P_{1,2,3}$ and $P_{1,4,3}$ in processor-layer $P_{1,:,:} \in \Lambda(n_k^L)$ may store nonzero row segments of row $B(k,:)$ and these row segments are expanded along processor-fibers $P_{:,2,3}$ and $P_{:,4,3}$, respectively. In phase ϕ_2, cut-net n_k^L has $\Lambda(n_k^L) = \{P_{:,:,2}, P_{:,:,4}\}$. Hence, nonzero column segments of $A(:,k)$ can be stored by processors in one of the processor-layers in $\Lambda(n_k^L)$ and processors expand these row segments along their respective processor-fibers. For instance, processors $P_{1,2,3}$, $P_{2,2,3}$ and $P_{3,2,3}$ in processor-layer $P_{:,2,:} \in \Lambda(n_k^L)$ may store nonzero column segments of column $A(:,k)$ and expand these column segments along processor-fibers $P_{1,:,3}$, $P_{2,:,3}$ and $P_{3,:,3}$, respectively. In phase ϕ_3, the cut-net $n_{i,j}^Z$ representing fiber $W(i,j,:)$ and nonzero $C(i,j)$ has the connectivity set $\Lambda(n_k^L) = \{P_{:,:,2}, P_{:,:,3}\}$, since intersections of fiber $W(i,j,:)$ with lateral blocks $\overline{W}_{:,:,2}$ and $\overline{W}_{:,:,3}$ are nonempty. The task of accumulating and storing nonzero $C(i,j)$ can be given to a processor in one of the processor-layers in $\Lambda(n_k^L)$. For instance, the responsibility of storing the final $C(i,j)$ can be given to processor $P_{1,2,3}$. Hence, processor $P_{1,2,2}$ locally sums its *two* partial products and send a single partial result to processor $P_{1,2,3}$.

The proposed hypergraph $H(\phi_3)$ consists of $nnz(C)$ nets and $|W|$ pins Hence, for some SpGEMM instances, it may considerably increase the preprocessing overhead of the partitioning. To alleviate this problem, we represent a row $C(i,:)$ by a single net n_i^Z instead of introducing a net $n_{i,j}^Z$ for each nonzero entry $C(i,j)$. Then, we add v_k^L as a pin to net n_i^Z if $k \in cols(A(i,:))$. This modifications lead to a hypergraph with m nets and $nnz(A)$ pins. In this way, C-matrix entries are accumulated in row-basis rather than nonzero-basis since the accumulation of entries in the same row is performed by the processors in the same processor-layer without considering individual consistency conditions of nonzero entries. That is,

a nonzero $C(i,j)$ can be assigned to a processor-layer $P_{:,:,z} \notin \Lambda(n^Z_{i,j})$ even though $P_{:,:,z} \in \Lambda(n^Z_i)$. This approach drastically reduces the size of the hypergraph; however, the total volume of communication in the fold phase is overestimated the hypergraph model. This is because, even though a processor $P_{x,y,z}$ does not have a partial product for $C(i,j)$, this processor can accumulate partial results for $C(i,j)$ due to the assignment of row $C(i,:)$ to processor-layer $P_{:,:,z}$. In our experimental evaluation, we used this modified version of 3D partitioning scheme.

Table 5.1: Dataset Properties

	Number of		Max degree of	
Matrix	Rows/Cols	Nonzeros	Row	Col
$C = AA$				
crankseg_2	63,838	14,148,858	3,423	3,423
net4-1	88,343	2,441,727	4,791	4,791
Ge99H100	112,985	8,451,395	469	469
Ge87H76	112,985	7,892,195	469	469
Ga10As10H30	113,081	6,115,633	698	698
torso1	116,158	8,516,500	3,263	1,224
Ga19As19H42	133,123	8,884,839	697	697
bmwcra_1	148,770	10,644,002	351	351
para-10	155,924	5,416,358	6,931	6,931
mono_500Hz	169,410	5,036,288	719	719
ohne2	181,343	11,063,545	3,441	3,441
Si41Ge41H72	185,639	15,011,265	662	662
Si87H76	240,369	10,661,631	361	361
Ga41As41H72	268,096	18,488,476	702	702
coPapersCiteseer	434,102	32,073,440	1,188	1,188
coPapersDBLP	540,486	30,491,458	3,299	3,299
pre2	659,033	5,959,282	628	745
3Dspectralwave	680,943	33,650,589	117	117
Stanford_Berkeley	683,446	7,583,376	83,448	249
StocF-1465	1,465,137	21,005,389	189	189
$C = AB$				
rmat ($scale$=20)	1,048,576	8,259,994	1,181	1,158
rmat ($scale$=21)	2,097,152	16,570,170	1,576	1,555

5.4 Experiments

5.4.1 Experimental Setup

We test the performance of 2D- and 3D-parallel SpGEMM algorithms against the 1D row-by-row parallel algorithm [30], because it is reported to be the best performing 1D algorithm in general.

We implemented all parallel SpGEMM algorithms in *C++* and handled inter-process communication operations by *OpenMPI version 3.0.1*. For a fair comparison of partitioning algorithms, we iplemented local SpGEMM computations by utilizing row-by-row product formulation [23] for all parallel SpGEMM implementations. The sequential SpGEMM implementation, which is used to obtain the speedups of the parallel algorithms, also uses row-by-row product formulation. Instead of using the sequential implementation of Comblas library [20], we used our own sequential SpGEMM implementation, since our 2D- and 3D-parallel SpGEMM algorithms run faster on the SpGEMM instances utilized in the paper. We utilized random partitioning both for our parallel SpGEMM implementations and the one provided in Comblas library for experimental comparison.

We use PaToH [15], which supports multi-constraint hypergraph partitioning, to partition the proposed the hypergraph models. We set the allowed imbalance ratio to $\epsilon = 0.01$ in each phase of the proposed partitioning models. Since PaToH contains randomized algorithms, the averages of three partitioning runs, each randomly seeded, are reported.

We perfoemd our experiments on UHEMS's Sariyer system [114]. In this system, each node contains an *Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz* processor, consisting of 28 cores, and 128GB main memory. Each MPI job is submitted to the system by allocating the number of cores as required by each job, since the tested algorithms does not utilize shared memory parallelism.

We evaluate partitioning algorithms for parallel SpGEMM algorithms on $p = 25, 100, 225, 400, 625$ and 900 processors. We select these processor counts in order to configure 2D virtual processor grids as perfect squares 5×5, 10×10, 15×15, 20×20, 25×25 and 30×30, respectively. We select the 3D virtual grid sizes in such a way that lateral layers of processor grids are perfect squares, where we select the size of the third (i.e., z) dimension accordingly. So, 3D virtual girds of sizes of $5 \times 5 \times 4$, $5 \times 5 \times 9$, $10 \times 10 \times 4$ and $10 \times 10 \times 9$, are used for $p = 100, 225, 400$ and 900, whereas the numbers of processors in the remaining two 3D virtual grids $3 \times 3 \times 3 = 27$ and $9 \times 9 \times 8 = 648$ are slightly larger than the p values 25 and 625, respectively.

5.4.2 Datasets

We select SpGEMM instances under two categories, $C = AA$ and $C = AB$, where we choose the input matrices as squares for the sake of fairness of the tested algorithms. For $C = AA$, we collected matrices from SuiteSparse Matrix Collection (UFL) [95]. The properties of 20 matrices used under this category are displayed in Table 5.1. For $C = AB$, we use the recursive matrix generator *R-MAT* [115] to generate two SSCA matrices (HPCS Scalable Synthetic Compact Applications graph analysis benchmark [116]) as input matrices A and B. Matrices are generated for parameters $scale = 20$ and $scale = 21$, where for each value of $scale$, the tool produces a matrix of size $2^{scale} \times 2^{scale}$. Additionally, we choose parameters $a = 0.55$, $b = 0.1$, $c = 0.1$ and $d = 0.25$, which are the default settings in the tool.

5.4.3 Experimental Results

We use the following abbreviations for the algorithms: We use 1D, 2D and 3D to denote the row-by-row parallel [30], sparse SUMMA (Sections 5.2.2) and split-3D SPGEMM (Sections 5.2.3) algorithms. For 1D, we use the prefix "H" to

Table 5.2: Performance comparisons of H2D over R2D and H3D over R3D

		Volume		Messages				Volume		Messages		
p		Avg	Max	Avg	Max	S		Avg	Max	Avg	Max	S
25	R2D	3128	3180	8	8	11	R3D	7231	7909	6	6	9
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.11	0.16	0.81	0.96	1.27	H3D	0.24	0.38	0.84	0.97	1.39
100	R2D	1654	1714	18	18	31	R3D	3060	3468	11	11	26
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.12	0.19	0.76	0.93	1.48	H3D	0.29	0.55	0.78	0.95	1.40
225	R2D	1072	1136	28	28	54	R3D	1572	1843	16	16	44
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.13	0.22	0.71	0.90	1.53	H3D	0.34	0.66	0.76	0.95	1.45
400	R2D	769	837	38	38	76	R3D	1154	1456	21	21	72
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.13	0.25	0.65	0.86	1.52	H3D	0.34	0.72	0.72	0.92	1.32
625	R2D	586	645	48	48	85	R3D	731	919	23	23	86
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.14	0.27	0.63	0.83	1.43	H3D	0.37	0.88	0.73	0.93	1.34
900	R2D	466	603	58	58	63	R3D	564	726	26	26	92
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.15	0.31	0.58	0.81	1.63	H3D	0.38	0.95	0.69	0.92	1.31

For each p, the first row displays the actual values, whereas the second and third rows display normalized values for the resective algorithm.

indicate that we use the hypergraph partitioning model described in [30] for row-by-row parallel SpGEMM algorithm. For 2D and 3D, we use the prefix "H" to indicate that we use the hypergraph partitioning models proposed in Section 5.3 for the respective parallel SpGEMM algorithms. We use the prefix "R" to indicate that we use random partitioning.

Comparison of the relative performance of parallel SpGEMM algorithms in terms of multiple communication cost metrics as well as speedup values attained on the parallel system are provided in Tables 5.2 and 5.3. We categorized communication cost metrics under communication volume and message counts metrics. These metrics respectively relate to bandwidth and latency overheads of the parallel SpGEMM implementations. For both metrics, average and maximum volume/number of messages sent by a processor are displayed, since for a fixed p,

average message volume/count values also refer to total message volume/count values. Additionally, average message volume/count values are preferred to be displayed instead of total volume/count values to better see the deviation of maximum values from the average values. In Tables 5.2 and 5.3, for each p, results are displayed as averages (geometric means) over 20 $C = AA$ instances.

Comparisons of H2D against R2D as well as H3D against R3D are given in Table 5.2 to show the improvements of the proposed hypergraph partitioning models instead of random partitioning. As seen in the table, H2D and H3D respectively achieve 85%–89% and 62%–76% less average volume than R2D and R3D over all p values. H2D and H3D respectively achieve 69%–84% and 5%–62% less maximum message volume than R2D and R3D. H2D and H3D respectively achieve 19%–42% and 16%–31% smaller message counts than R2D and R3D. H2D and H3D respectively achieve 4%–19% and 3%–8% smaller maximum message counts than R2D and R3D. The improvement gap between hypergraph and random partitioning models decreases in both maximum message volume and maximum message count values which can be attributed to better message volume and count balancing achieved by the random partitioning. H2D and H3D respectively achieve 27%–63% and 31%–40% larger speedup values than R2D and R3D.

We compare the performance of hypergraph partitioning models on 1D-, 2D- and 3D-parallel SpGEMM algorithms in Table 5.3. In the table, we display the average imbalance ratios in the third column. The average imbalance ratios computed as the ratio of the computational load (voxel count) of the maximally loaded processor to the average processor load ($|W|/p$). In terms of computational load balance, H1D and H2D display comparable performances, whereas H3D displays considerably worse. The relative performance of H3D against H1D/H2D degrades with increasing number of processors. This is because, the third partitioning phase of H3D necessitates increased number of constraints, which adversely affects the load-balancing quality of PaToH [117], as the number of processors increases.

Table 5.3: Average performance comparison of H1D, H2D and H3D algorithms

			Message (Actual)					Message (Norm.)				
			Volume		Count			Volume		Count		
p		imb	Avg	Max	Avg	Max	S	Avg	Max	Avg	Max	S
	H1D	1.01	330	601	11	18	14	1.00	1.00	1.00	1.00	1.00
25	H2D	1.01	357	519	7	8	13	1.08	0.86	0.60	0.43	0.97
	H3D	1.05	1719	2987	5	6	12	5.21	4.97	0.46	0.33	0.90
	H1D	1.01	183	463	24	49	45	1.00	1.00	1.00	1.00	1.00
100	H2D	1.01	193	331	14	17	46	1.06	0.71	0.57	0.34	1.02
	H3D	1.12	889	1897	9	10	36	4.87	4.09	0.36	0.21	0.80
	H1D	1.04	131	430	35	84	65	1.00	1.00	1.00	1.00	1.00
225	H2D	1.03	134	249	20	25	83	1.03	0.58	0.56	0.30	1.29
	H3D	1.24	529	1212	12	15	64	4.03	2.82	0.34	0.18	0.98
	H1D	1.06	102	417	45	123	70	1.00	1.00	1.00	1.00	1.00
400	H2D	1.04	100	210	25	33	115	0.98	0.50	0.56	0.27	1.65
	H3D	1.37	388	1052	15	19	96	3.81	2.52	0.34	0.16	1.38
	H1D	1.10	86	394	52	161	62	1.00	1.00	1.00	1.00	1.00
625	H2D	1.10	81	171	30	40	122	0.94	0.43	0.59	0.25	1.96
	H3D	1.54	272	807	17	21	116	3.16	2.05	0.33	0.13	1.86
	H1D	1.13	75	415	55	196	56	1.00	1.00	1.00	1.00	1.00
900	H2D	1.10	68	190	34	47	103	0.91	0.46	0.61	0.24	1.85
	H3D	1.61	214	688	18	24	121	2.84	1.66	0.33	0.12	2.17

As seen in Table 5.3, H3D performs worse than both H1D and H2D in terms of both communication volume cost metrics. Two factors that explain this experimental findings are: First, many partial products are communicated among processors in the fold phase. Second, the cut-size quality of PaToH [117] is adversely affected by the increased number of constraints. On the other hand, we observe a decrease in the performance difference between H1D and H3D as the number of processors increases, since H3D incurs 5.21x more volume than H1D on $p=25$ processors, whereas it incurs 2.81x more volume on $p=900$ processors.

In terms of average message volume, performance of H2D is worse than H1D on small processor counts ($p = 25, 100$ and 225). However, on larger processor counts ($p = 400, 625$ and 900) performance of H2D becomes better than H1D.

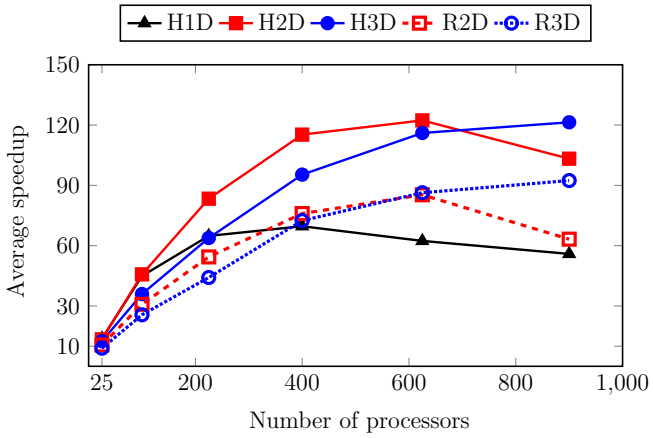


Figure 5.7: Average speedup curves for 20 $C = AA$ instances

In terms of maximum message volume, performance of H2D is significantly better than H1D on all processor counts and the performance difference increases in favor of H2D with increasing p in general. Moreover, the performance improvement of H2D over H1D is much higher in maximum message volume than its improvement in average message volume. For example, for $p = 900$, on average, H2D incurs 54% less maximum message volume than H1D, whereas H2D incurs only 9% less average message volume than H1D. This is because, dense rows/columns of input matrices incur large communication volume for processors storing these rows/columns. However, H2D largely resolves this issue, since the dense rows/columns of matrices are partitioned among multiple processors.

H3D performs the best in terms of both message count metrics, whereas H2D is the second best. With increasing number of processors, the performance difference of H2D over H1D and H3D over both H2D and H1D increase. For instance, on $p = 900$ processors, H2D provides improvements of 39% and 76% over H1D in average and maximum message count metrics, respectively. For $p = 900$, on average, H3D performs approximately 2x better than H2D in both average and maximum message count metrics. These experimental results are due to the upper bounds established by 2D and 3D algorithms where the number of messages handled by a processor is $O(\sqrt[2]{p})$ and $O(\sqrt[3]{p})$ in 2D and 3D algorithms, respectively. This upper bound is $O(p)$ in the 1D algorithm.

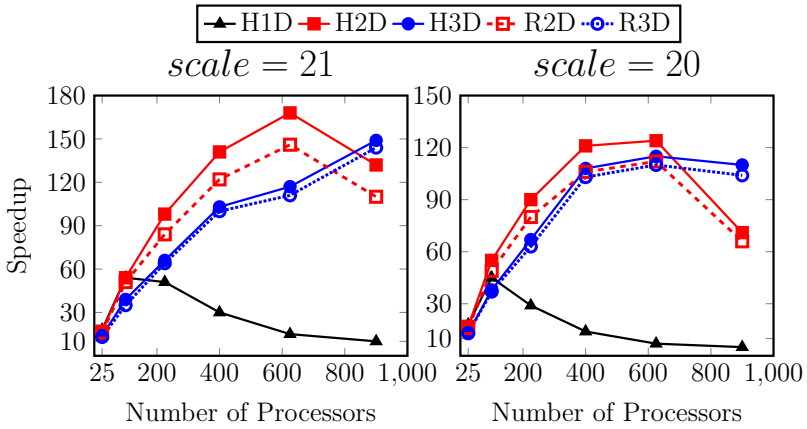


Figure 5.8: Speedup curves for *R-MAT* $C = AB$ instances

Table 5.3 shows that comparable average speedup values are obtained by H1D and H2D on small processor counts $p = 25$ and 100, whereas H3D performs worse. On the larger processors counts $p = 225, 400$ and 625, H2D performs the best and with increasing p, the performance difference between H2D and H1D increases in favor of H2D, whereas the performance gap between H2D and H3D closes. H3D performs the best on the largest processor count $p = 900$. These experimental findings on speedup values are consistent with the relative performance variation of H1D, H2D and H3D in various cost metrics of communication as mentioned above. On smaller number of processors, parallel SpGEMM algorithms are bandwidth bound, whereas they become latency bound on larger number of processor. Hence, H3D becomes the clear winner on $p = 900$ processor even tough incurring more communication volume than both H1D and H2D.

The average speedup curves (averaged over all 20 $C = AA$ instances) are displayed in Figure 5.7 for the tested algorithms. As seen in the figure, the best speedup performance is attained by H2D until $p = 625$ whereas it scales down on $p = 900$. On the other hand, H3D consistently scale until $p = 900$ and finally achieves higher speedup than H2D on $p = 900$. Observe that better speedup performances are attained by both R2D and R3D than H1D for larger processor counts $p \geq 400$. The speedup curves for two *R-MAT* matrices with $scale = 20$ and 21 are shown in Figure 5.8. The discussion given for the average speedup curves also apply for those of the two *R-MAT* matrices. Finally, speedup

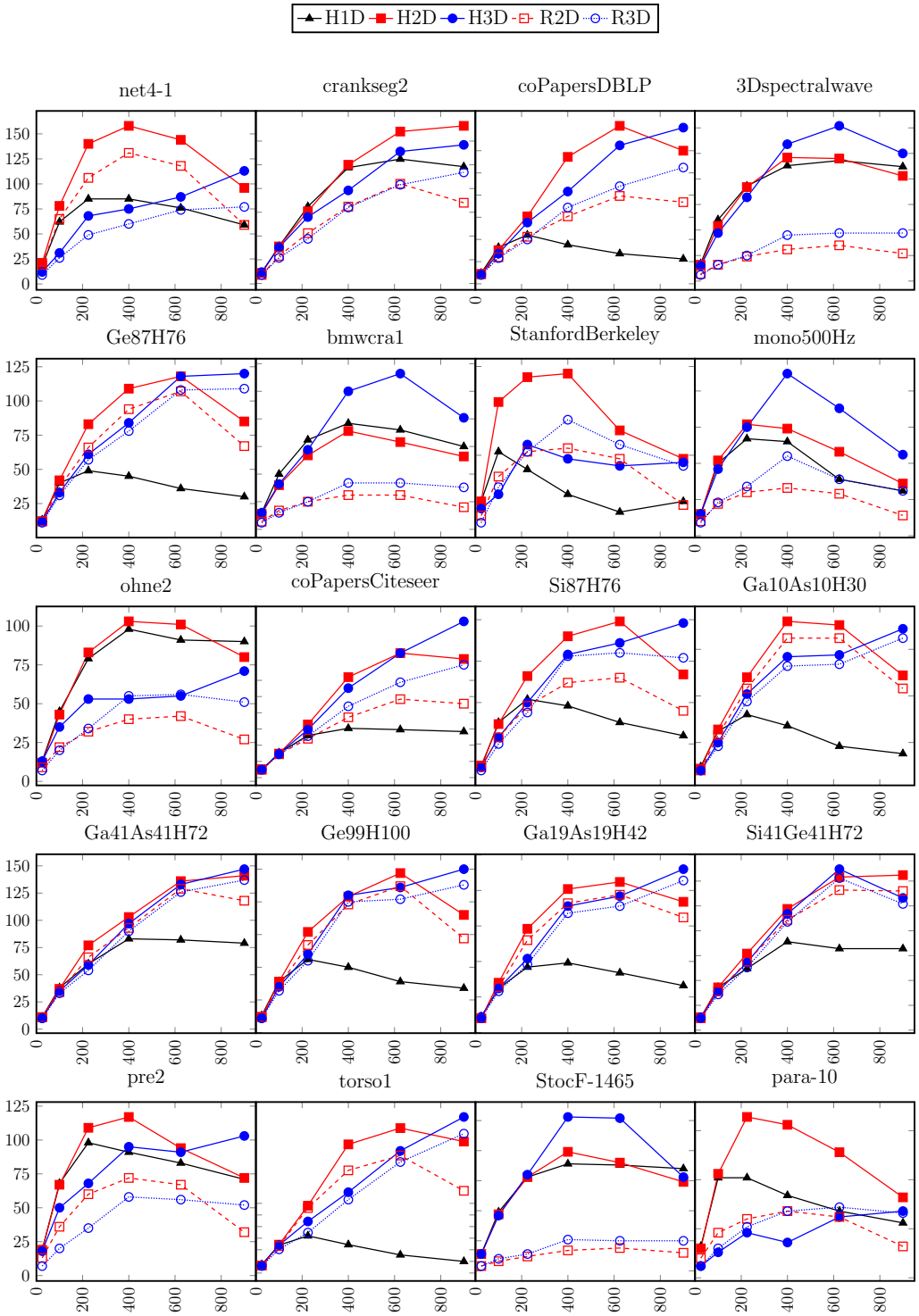


Figure 5.9: Speedup curves attained by each algorithm on all 20 $C = AA$ SpGEMM instances.

performances of the tested algorithms on each individual SPGEMM instance in the category $C = AA$ are displayed in Figure 5.9

For a more comprehensive comparison, the performance profiles [118] for parallel SpGEMM times of all tested algorithms are displayed in Figure 5.10. the running time obtained by an algorithm for a matrix on a given number of processors corresponds to a test instance. A point (x, y) for an algorithm in a profile denotes that performance of the algorithm is within x factor of the best performance attained in y fraction of the test instances. Performances of algorithms are compared under three different processor count groups: small ($p \in \{25, 100\}$), medium ($p \in \{225, 400\}$) and large ($p \in \{625, 900\}$). If the profile of an algorithm is closer to the y-axis, its performance is considered to be better.

As shown in Figure 5.10, H1D achieves the best results on approximately 70% of the instances for $p \in \{25, 100\}$ and its performance is within a factor of 1.6 of the best performances achieved in this category. On the other hand, H2D performs the best in 54% of the instances and its performance is within a factor of 1.2 of the best results in all instances. For $p \in \{225, 400\}$, H2D performs the best in approximately 75% of the instances and its performance is within a factor of 1.2 of the best results in all instances. The second best performance is achieved by H3D and H1D performs significantly worse in this processor group. Performances of H3D and H2D are respectively the best in 54% and 44% of the instances for $p \in \{625, 900\}$ and performance of both algorithms are within a factor of 1.5 of the best results in all instances.

As mentioned earlier, communication operations are performed in multiple stages by both 2D [1] and 3D [2] parallel SpGEMM algorithms through utilizing blocking factors to reduce processors' local memory requirements. The partitioning objective corresponding to minimizing the total communication volume also corresponds to minimizing the total sizes of the local communication buffers used for send and receive operations in the proposed hypergraph models. In other words, the proposed hypergrah models already address minimizing the increase in the processors' local memory requirements due to communication buffers. For example, for H2D on $p = 400$, a single-stage communication scheme incurs an

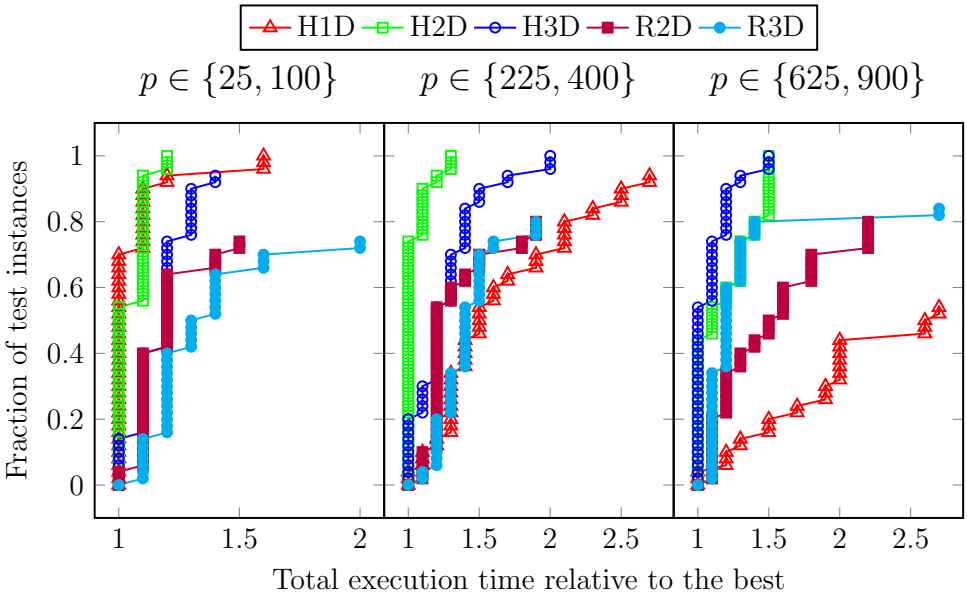


Figure 5.10: Performance profile

average increase of 84% (between 25%–200%) in processors' local memory requirements. This justifies the use of single-stage communications in H2D and H3D implementations as multi-stage communication significantly increases latency overheads.

5.5 Conclusion

We proposed two new Hypergraph models that address both bandwidth and latency costs of the communication requirements of 2D and 3D SpGEMM algorithms. Different from the previously proposed 1D partitioning models, our methods regard the multidimensional arrangement of processors and therefore; can significantly reduce the number of messages exchanged among processors. In this way, our partitioning models provide much better optimizations in the latency costs. However, the optimization achieved on communication volume costs by the proposed models are less than those by 1D counterparts. This is due to the multi-constraint hypergraph partitioning employed in our models; but the relative performance difference in the communication volumes decrease as the number of processors increases.

Our experimental analysis demonstrates that the proposed partitioning schemes provide significant improvements for scalability of 2D and 3D SpGEMM algorithms. We observed that as the number of processors increases, the latency overheads become more pronounced than the communication volume costs in the overall cost of communication. Therefore; our proposed models performs significantly better than 1D partitioning schemes on higher number of processors even though our models necessitate higher communication volume costs. Additionally, improvements of the proposed models significantly higher especially when the SpGEMM instances comprises of input matrices that contain dense rows. This is due to the fact that dense rows induce higher communication costs and these rows are handled by multiple processors in 2D and 3D algorithms and thus reducing the upper-bound on the communication cost induced on processors.

Lastly, sizes of the proposed hypergraph models are significantly smaller than the fine-grain model [31], which makes the partitioning of our proposed models more practical in many cases. Additionally, hypergraph partitioning costs are expected to amortize in applications that require repeated SpGEMMs which involve input matrices having the same sparsity patterns.

Chapter 6

Conclusion

In this thesis, we proposed graph/hypergraph partitioning models to improve the scalability of parallel graph computations on distributed-memory systems. The proposed models aim at reducing the total number of messages in case of computations that involve cascading operations. Moreover, our models also aim at reducing bandwidth- and latency-related overheads of the parallel graph computations through parallelizing SpGEMM algorithms efficiently on distributed-memory systems.

We studied the cascade-aware graph partitioning problem and developed a sampling-based method to partition users across servers. The user-to-server mapping obtained through the proposed partitioning scheme considerably minimizes the communication traffic among servers during cascade processes. The partitioning approach integrates the graph structure and propagation processes by estimating a probability distribution that associates each edge with a probability of being involved in a random propagation process. We derived theoretical results showing how the proposed solution achieves the stated goals. Under the widely used IC model, we conducted experiments and evaluated the effectiveness of the proposed solution in terms of partitioning objectives. We also tested the solution over a social network where the real logs of propagation traces among users are available. Experiments show the effectiveness of the proposed solution both in the existence and lack of real propagation traces.

We proposed an SpGEMM algorithm for the Accumulo database that utilizes row-by-row parallel SpGEMM to achieve write-locality during ingestion of the output matrix. The proposed algorithm requires multiple batch-scanning operations which incur latency overheads. By performing local SpGEMM operations via multiple batches and using multi-threaded parallelism, these overheads are alleviated. We also proposed a matrix partitioning scheme that reduces the total communication volume and provides a workload-balance among servers. We conducted extensive experiments on both realistic and synthetic SpGEMM instances. Experimental results showed that the proposed algorithm, without using the proposed matrix partitioning scheme, outperforms the Graphulo library's outer-product algorithm by a large margin. The results also showed that the proposed matrix partitioning scheme provides further improvements.

We proposed hypergraph graph partitioning models that attain multidimensional partitioning on SpGEMM's workload. The proposed models encode the communication and computational requirements of processors that are logically arranged as multidimensional grids and utilize the natural upper bound provided by 2D and 3D SpGEMM algorithms on the communication requirements of processors. Our experimental analysis showed that the proposed partitioning models provide significant improvements for scalability of 2D and 3D SpGEMM algorithms. The number of messages handled by processors significantly decreases and the latency-related costs of the overall communication overheads are reduced. We observed that the latency costs become more pronounced in the overall cost of communication as the number of processors increases. Therefore, the proposed partitioning models provide better scalability even though incurring higher total communication volumes.

Bibliography

[1] 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.

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

[3] Josep M Pujol, Vijay Erramilli, and Pablo Rodriguez. Divide and conquer: Partitioning online social networks. *arXiv preprint arXiv:0905.4918*, 2009.

[4] Mindi Yuan, David Stein, Berenice Carrasco, Joana MF Trindade, and Yi Lu. Partitioning social networks for fast retrieval of time-dependent queries. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 205–212. IEEE, 2012.

[5] Jiewen Huang and Daniel J Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment*, 9(7):540–551, 2016.

[6] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 145–156. ACM, 2012.

[7] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM*

SIGMOD International Conference on Management of Data, pages 517–528. ACM, 2012.

[8] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[9] Abdurrahman Yaşar, Buğra Gedik, and Hakan Ferhatosmanoğlu. Distributed block formation and layout for disk-based management of large-scale graphs. *Distributed and Parallel Databases*, 35(1):23–53, 2017.

[10] Josep M Pujol, Georgos Siganos, Vijay Erramilli, and Pablo Rodriguez. Scaling online social networks without pains. In *Proc of NETDB*, 2009.

[11] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine (s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 41(4):375–386, 2011.

[12] Berenice Carrasco, Yi Lu, and Joana MF da Trindade. Partitioning social networks for time-dependent queries. In *Proceedings of the 4th Workshop on Social Network Systems*, page 2. ACM, 2011.

[13] Ata Turk, R Oguz Selvitopi, Hakan Ferhatosmanoglu, and Cevdet Aykanat. Temporal workload-aware replicated partitioning for social networks. *Knowledge and Data Engineering, IEEE Transactions on*, 26(11):2832–2845, 2014.

[14] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[15] Umit V Catalyürek and Cevdet Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[16] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[17] Marian Brezina and Panayot S Vassilevski. Smoothed aggregation spectral element agglomeration amg: Sa-ρamge. In *International Conference on Large-Scale Scientific Computing*, pages 3–15. Springer, 2011.

[18] Ichitaro Yamazaki and Xiaoye S Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer, 2010.

[19] John R Gilbert, Steve Reinhardt, and Viral B Shah. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering*, 10(2):20–25, 2008.

[20] 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.

[21] S Van Dongen. Graph clustering by flow simulation [phd thesis].[utrecht (the netherlands)]: University of utrecht. 2000.

[22] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015.

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

[24] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Ben Miller, and Jeremy Kepner. Graphulo: Linear algebra graph kernels for nosql databases. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 822–830. IEEE, 2015.

[25] Aydın Buluç and John R Gilbert. Highly parallel sparse matrix-matrix multiplication. *arXiv preprint arXiv:1006.2183*, 2010.

[26] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Bill Howe. From nosql accumulo to newsql graphulo: Design and utility of graph algorithms

inside a bigtable database. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–9. IEEE, 2016.

[27] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Adam Fuchs. Graphulo implementation of server-side sparse matrix multiply in the accumulo database. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–7. IEEE, 2015.

[28] Kadir Akbudak and Cevdet Aykanat. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2258–2271, 2017.

[29] Gunduz Vehbi Demirci and Cevdet Aykanat. Scaling sparse matrix-matrix multiplication in the accumulo database. *Distributed and Parallel Databases*, pages 1–32, 2019. Reprinted from Distributed and Parallel Databases.

[30] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. Partitioning models for scaling parallel sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 4(3):13, 2018.

[31] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 3(3):18, 2016.

[32] Kadir Akbudak and Cevdet Aykanat. Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 36(5):C568–C590, 2014.

[33] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[34] Gunduz Vehbi Demirci, Hakan Ferhatosmanoglu, and Cevdet Aykanat. Cascade-aware partitioning of large graph databases. *The VLDB Journal*, pages 1–22, 2018. Reprinted from The VLDB Journal.

[35] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[36] Daniel Gruhl, Ramanathan Guha, David Liben-Nowell, and Andrew Tomkins. Information diffusion through blogspace. In *Proceedings of the 13th international conference on World Wide Web*, pages 491–501. ACM, 2004.

[37] Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada Adamic. The role of social networks in information diffusion. In *Proceedings of the 21st international conference on World Wide Web*, pages 519–528. ACM, 2012.

[38] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1029–1038. ACM, 2010.

[39] Digg social news portal. `http://digg.com/`, 2017.

[40] 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.

[41] Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.

[42] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.

[43] Jacob Goldenberg, Barak Libai, and Eitan Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing letters*, 12(3):211–223, 2001.

[44] Mark Granovetter. Threshold models of collective behavior. *American journal of sociology*, pages 1420–1443, 1978.

[45] Amit Goyal, Francesco Bonchi, and Laks VS Lakshmanan. Learning influence probabilities in social networks. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 241–250. ACM, 2010.

[46] Kazumi Saito, Ryohei Nakano, and Masahiro Kimura. Prediction of information diffusion probabilities for independent cascade model. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 67–75. Springer, 2008.

[47] Pedro Domingos and Matt Richardson. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 57–66. ACM, 2001.

[48] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146. ACM, 2003.

[49] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208. ACM, 2009.

[50] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429. ACM, 2007.

[51] Yu Wang, Gao Cong, Guojie Song, and Kunqing Xie. Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1039–1048. ACM, 2010.

[52] Hui Li, Sourav S Bhowmick, Aixin Sun, and Jiangtao Cui. Conformity-aware influence maximization in online social networks. *The VLDB Journal*, 24(1):117–141, 2015.

[53] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 946–957. Society for Industrial and Applied Mathematics, 2014.

[54] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 75–86. ACM, 2014.

[55] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 629–638. ACM, 2014.

[56] Yang Zhou and Ling Liu. Social influence based clustering of heterogeneous information networks. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 338–346. ACM, 2013.

[57] Zaixin Lu, Yuqing Zhu, Wei Li, Weili Wu, and Xiuzhen Cheng. Influence-based community partition for social networks. *Computational Social Networks*, 1(1):1, 2014.

[58] Rumi Ghosh and Kristina Lerman. Community detection using a measure of global influence. In *Advances in Social Network Mining and Analysis*, pages 20–35. Springer, 2010.

[59] Nicola Barbieri, Francesco Bonchi, and Giuseppe Manco. Cascade-based community detection. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 33–42. ACM, 2013.

[60] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331, 2008.

[61] Charles J Colbourn and CJ Colbourn. *The combinatorics of network reliability*, volume 200. Oxford University Press New York, 1987.

[62] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[63] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[64] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[65] Reza Zafarani and Huan Liu. Social computing data repository at ASU. `http://socialcomputing.asu.edu`, 2009.

[66] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[67] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[68] D Bader, Kamesh Madduri, J Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2:1–10, 2006.

[69] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[70] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009.

[71] George M Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 481–490. IEEE, 2014.

[72] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[73] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[74] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[75] Adam Fuchs. Accumulo–extensions to google's bigtable design. *National Security Agency, Tech. Rep*, 2012.

[76] Apache hbase. `https://hbase.apache.org/`. Accessed: 2018-04-15.

[77] Ranjan Sen, Andrew Farris, and Peter Guerra. Benchmarking apache accumulo bigdata distributed table store using its continuous test suite. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 334–341. IEEE, 2013.

[78] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[79] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[80] Timothy Weale, Vijay Gadepally, Dylan Hutchison, and Jeremy Kepner. Benchmarking the graphulo processing framework. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–5. IEEE, 2016.

[81] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[82] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.

[83] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[84] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*, pages 48–57. Springer, 2015.

[85] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[86] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[87] Bill Bejeck. *Getting Started with Google Guava*. Packt Publishing Ltd, 2013.

[88] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[89] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

[90] Michael McCourt, Barry Smith, and Hong Zhang. Sparse matrix-matrix products executed through coloring. *SIAM Journal on Matrix Analysis and Applications*, 36(1):90–109, 2015.

[91] Paolo D'Alberto and Alexandru Nicolau. R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.

[92] Carlos Ordonez. Optimization of linear recursive queries in sql. *IEEE Transactions on knowledge and Data Engineering*, 22(2):264–277, 2010.

[93] Carlos Ordonez, Yiqun Zhang, and Wellington Cabrera. The gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1905–1918, 2016.

[94] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[95] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[96] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[97] Hao Li, Kenli Li, Jiwu Peng, Junyan Hu, and Keqin Li. An efficient parallelization approach for large-scale sparse non-negative matrix factorization using kullback-leibler divergence on multi-gpu. In *Ubiquitous Computing*

and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on, pages 511–518. IEEE, 2017.

[98] Hao Li, Kenli Li, Jiwu Peng, and Keqin Li. Cusnmf: A sparse non-negative matrix factorization approach for large-scale collaborative filtering recommender systems on multi-gpu. In *Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on*, pages 1144–1151. IEEE, 2017.

[99] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. Mpi-faun: an mpi-based framework for alternating-updating nonnegative matrix factorization. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):544–558, 2018.

[100] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[101] Václav Hapla, David Horák, and Michal Merta. Use of direct solvers in tfeti massively parallel implementation. In *International Workshop on Applied Parallel Computing*, pages 192–205. Springer, 2012.

[102] H Bernhard Schlegel, John M Millam, Srinivasan S Iyengar, Gregory A Voth, Andrew D Daniels, Gustavo E Scuseria, and Michael J Frisch. Ab initio molecular dynamics: Propagating the density matrix with gaussian orbitals. *The Journal of Chemical Physics*, 114(22):9758–9763, 2001.

[103] Andrew D Daniels, John M Millam, and Gustavo E Scuseria. Semiempirical methods with conjugate gradient density matrix search to replace diagonalization for molecular systems containing thousands of atoms. *The Journal of chemical physics*, 107(2):425–431, 1997.

[104] Rob H Bisseling, TM Doup, and L Daniel JC Loyens. A parallel interior point algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 43(2):49–86, 1993.

[105] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 204–213. IEEE, 1994.

[106] Carlos Ordonez. Optimization of linear recursive queries in sql. *IEEE Transactions on Knowledge and Data Engineering*, 22(2):264–277, 2009.

[107] Intel MKL. Math kernel library (mkl). `https://software.intel.com/en-us/mkl`. Accessed: 2019.

[108] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25, 2015.

[109] CUDA Nvidia. Cusparse library. *NVIDIA Corporation, Santa Clara, California*, 2014.

[110] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Version 0.3. 0*, 35, 2012.

[111] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[112] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 77–79. ACM, 2012.

[113] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 222–231. ACM, 2013.

[114] UHEM Website. `http://www.uhem.itu.edu.tr/`. Accessed: 2019.

[115] 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.

[116] SSCA Benchmark. `http://www.graphanalysis.org/benchmark/`. Accessed: 2019.

[117] Cevdet Aykanat, B Barla Cambazoglu, and Bora Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.

[118] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.