4-19-2018

Distributed Iterative Graph Processing Using NoSQL with Data Locality

Ayam Pokhrel
Louisiana State University and Agricultural and Mechanical College, pokhrelayam82@gmail.com

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses
Part of the Other Computer Engineering Commons, and the Other Computer Sciences Commons

Recommended Citation

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
DISTRIBUTED ITERATIVE GRAPH PROCESSING USING NOSQL WITH DATA LOCALITY

A Thesis

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Master of Science

in

The Division of Computer Science and Engineering, School of EECS

by

Ayam Pokhrel
B.E., Kathmandu University, 2014
August 2018
Acknowledgments

This thesis has been possible due to the immense contribution of many people. First, my sincere gratitude to my thesis supervisor (and my Major Professor) Dr. Kisung Lee who has been constantly motivating, encouraging, and guiding me through the whole process. Being a student on one of his courses, Big Data Technologies, also motivated me towards this thesis. His patience, understanding, and expertise have been the foundation for my work. I would also like to express my sincere gratitude to Dr. Qingyang Wang and Dr. Seung-Jong Park who have been on my MS thesis committee and contributed their time.

I would like to acknowledge Mr. Sayan Goswami who has helped me throughout the technical part of my work. His participation and suggestions have been very valuable for the completion of my thesis. I would also like to thank Dr. Bijaya B. Karki and Dr. Frank Neubrander for all the support and guidance throughout my stay at LSU.

I am highly obliged to my roommates who have always made me feel like home here and have been very understanding. I would also like to acknowledge all my friends in LSU who gave me many memorable moments that I shall cherish for the lifetime.

I am very grateful to my parents Rajesh Pokhrel and Bhagawati Pokhrel for their ineffable love, support, and trust in me. I am also thankful to my sisters Ashma Pokhrel and Ayustha Pokhrel for being the support system in every moment of my life. Last but not the least, I would like to thank my girlfriend, Sudikshya Bhandari, for always trying to bring out the best in me.
# Table of Contents

ACKNOWLEDGMENTS ................................................................. ii
LIST OF TABLES ................................................................. iv
LIST OF FIGURES ................................................................. v
ABSTRACT ................................................................. vi

CHAPTER
1 INTRODUCTION ................................................................. 1
  1.1 Graph Computation Challenges ........................................ 1
  1.2 Existing Technologies .................................................. 2
  1.3 Objectives ................................................................. 4
  1.4 Thesis Organization .................................................. 4

2 BACKGROUND ................................................................. 5
  2.1 Graph ................................................................. 5
  2.2 Graph Problems .................................................. 7
  2.3 Programming Models .................................................. 7
  2.4 GraphMap ................................................................. 9

3 METHODOLOGY ................................................................. 11
  3.1 Graph Partitioning .................................................. 11
  3.2 Data Placement and Locality ........................................ 12
  3.3 System Architecture .................................................. 13

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS ..................... 16
  4.1 System Setup .................................................. 16
  4.2 Data Storage .................................................. 19
  4.3 Data Locality .................................................. 21
  4.4 Mincut-based Partitioning ........................................ 23
  4.5 Experimental Evaluation ........................................ 28

5 CONCLUSIONS ................................................................. 36

REFERENCES ................................................................. 38

APPENDIX
A PSEUDOCODES ................................................................. 42

VITA ................................................................. 43
## List of Tables

4.1 Region information file ................................................................. 22
4.2 Original dataset (top) and its Metis format (bottom) ......................... 26
4.3 The datasets used in the experiment ............................................... 29
4.4 Total time to store data ................................................................. 30
4.5 Execution time and memory used for partitioning .............................. 30
4.6 Total number of messages (Metis vs. Hash) .................................... 31
4.7 Total execution time (Metis vs. Hash) ........................................... 31
4.8 Total execution time (with and without locality) ............................... 35
List of Figures

2.1 A sample graph and its graph representations ......................... 6
3.1 System architecture .................................................................. 14
4.1 Experiment architecture ........................................................... 17
4.2 Graph representation in HBase .................................................. 20
4.3 A sample graph and its Metis representation ............................. 24
4.4 A sample graph and its partition file ........................................ 27
4.5 Edge distribution and iteration time per partition (Hollywood) ....... 33
4.6 Edge distribution and iteration time per partition (UK-2005) ........ 34
Abstract

A tremendous amount of data is generated every day from a wide range of sources such as social networks, sensors, and application logs. Among them, graph data is one type that represents valuable relationships between various entities. Analytics of large graphs has become an essential part of business processes and scientific studies because it leads to deep and meaningful insights into the related domain based on the connections between various entities. However, the optimal processing of large-scale iterative graph computations is very challenging due to the issues like fault tolerance, high memory requirement, parallelization, and scalability. Most of the contemporary systems focus either on keeping the entire graph data in memory and minimizing the disk access or on processing the graph data completely on a single node with a centralized disk system. GraphMap is one of the state-of-the-art scalable and efficient out-of-core disk-based iterative graph processing systems that focus on using the secondary storage and optimizing the I/O access. In this thesis, we investigate two new extensions to the existing out-of-core NoSQL-based distributed iterative graph processing system: 1) Intra-worker data locality and 2) Mincut-based partitioning. We design an additional suite of data locality that moves the computation towards the data rather than the other way around. A significant improvement in performance, up to 39%, is demonstrated by this locality implementation. Similarly, we use the mincut-based graph partitioning technique to distribute the graph data uniformly across the workers for parallelization so that the inter-worker communication volume is minimized. By extensive experiments, we also show that the mincut-based graph partitioning technique can lead to improper parallelization due to sub-optimal load-balancing.
Chapter 1
Introduction

A huge volume of data is generated everyday [1]. Processing data is essential as it may lead to valuable insights or optimal solutions to any business or scientific problem (e.g., [2]). Various distributed big data systems, such as Hadoop [3] and Spark [4], exist to process a tremendous amount of heterogeneous data unprecedentedly generated from multiple diverse sources such as social networks, smartphones, sensor devices, and Internet of Things (IoT). Graph is also a kind of data that is massively generated along with all other heterogeneous kinds. A graph may simply be visualized as a set of connected entities. The entities may be users if we consider social media such as Facebook or they may be web pages if we think of search engines such as Google. Graphs express the relationship between multiple real-world entities. Thus, they are ubiquitous (e.g., social networks, protein networks, transportation networks). Analyzing graph data brings out deeper insights into the explicit and implicit relationship between the complex entities. Because of these reasons, graph data is getting so much attention.

1.1 Graph Computation Challenges

Due to the invaluable properties of the graph data, large-scale graph analytics has become an essential part of the business process and scientific study. However, the processing of large graphs is highly challenging [5]. For instance, the widely used programming model, MapReduce [6], can be a good option to solve many large-scale computations (e.g., [7, 8]). It can also be used to perform large graphs computations [9, 10]. However, it may lead to sub-optimal solutions and usability issues [11].

For the efficient large-scale graph analytics, programmers have to deal with several technical challenges [5,11]. First, the size of the graph data itself is a major hurdle because as it grows in size, it quickly goes beyond the node’s capacity [5]. Real-world graphs are very huge, and they are continuously increasing at a very high rate. For example, there are more than one billion users on the Facebook social network [12]. If each user is represented
as a vertex, the friendship graph of Facebook will have more than one billion vertices and edges. Processing such amount of graph data requires tremendous storage and computing resources. Moreover, it generates intermediate data and communication messages [13]. The intermediate data generated during such graph computations is usually much larger than the original input graph data [14].

Secondly, graphs represent complex relationships between various data entities. The complex relationships among vertices and edges are important for providing valuable insights for graph analytics, and they together guide the overall graph computation. Therefore, partitioning them for parallel computation can be difficult [5]. Lastly, real-world graphs usually have the power-law [15] edges distribution [16]. In other words, they have extremely skewed distribution in terms of the number of edges per vertex. For example, in the fan following graph of Twitter, a vertex representing a celebrity can have millions of edges while most of the vertices have only a small number of edges. Load balancing and proper parallelization in the computation of such kind of graph data is hard to achieve [16].

1.2 Existing Technologies

Many technologies have been developed in recent years to overcome the challenges faced during the iterative graph computations [11,14,16–28]. Existing graph systems can be categorized into three types based on their system architecture: 1) Centralized disk-based systems, 2) Distributed memory-based systems, and 3) NoSQL-based out-of-core systems.

GraphChi [17], X-Stream [18], FlashGraph [19], PathGraph [20], TurboGraph [21], GraphTwist [14], and Mosiac [22] are some of the centralized disk-based graph computation systems on a single machine. These systems mainly focus on designing graph representations or algorithms for the optimized HDD or SDD accesses and maximizing parallelism among the multiple cores. They demonstrate a significant improvement in performance for various iterative graph computations performed in a single machine. However, they are not capable of processing graphs that demand bigger computing and storage requirements.
than the resources available in a single machine. Thus, they have limited scalability.

The distributed memory-based iterative graph computation systems on a cluster of commodity computers, such as Pregel [11], Apache Giraph [24], Apache Hama [23], Giraph++ [25], GraphX [26], Pregelix [27], GraphLab [28], and PowerGraph [16], provide the scalable solutions. They can handle large graphs by adding more commodity resources to the cluster. However, they store the entire graph data, including all the intermediate results and the communication messages, on the distributed memory. The intermediate results generated during the computation may be several orders of magnitude greater than the original input graph dataset [14]. Although the input graph is partitioned and distributed across the multiple compute nodes, the systems may fail if the least powerful compute node among them cannot hold all the intermediate data and the messages along with its partitioned input graph data. Some of the graph processing systems among them like Giraph [24] and Pregelix [27] support the external memory computation for processing large graphs. However, their focus is mainly on reducing the memory requirement rather than utilizing the external memory effectively for improving the performance.

GraphMap [13] is the scalable NoSQL-based out-of-core distributed iterative graph processing system that addresses the issues discussed above. It not only reduces the memory requirement but also speeds up the computation by effectively storing and accessing a significant part of the graph data from the secondary storage [13]. GraphMap keeps only the mutable or the modifiable graph data in the memory. The immutable or the read-only data are stored on the disk. Additionally, the immutable graph data have optimized disk-representation so that they are read efficiently from the secondary storage when required [13]. Moreover, GraphMap balances the data load across multiple nodes in the cluster by using the efficient hash partitioning approach [13].
1.3 Objectives

The objectives of this research are as follows:

1. To extend the state-of-the-art NoSQL-based out-of-core iterative graph processing system, GraphMap [13], by adding a different suite of data locality that moves the computation towards data rather than the other way around for improving the performance.

2. To use the mincut-based graph partitioning technique for distributing the input graph uniformly across the workers to reduce the total volume of inter-worker communication during the iterative graph computation.

3. To study the impact of mincut-based graph partitioning technique on the overall performance of the system.

1.4 Thesis Organization

The organization of the rest of the chapters is as follows. In Chapter 2, we first introduce the preliminaries required to understand our system. We then briefly describe GraphMap [13] and its data representations. In Chapter 3, we explain our methodologies for carrying out our objectives. In Chapter 4, we first describe how we implement our methodologies. Later, we present our experimental results. Finally, we present our conclusions in Chapter 5.
Chapter 2
Background

In this section, we provide the background information on the concepts, computation models, and the frameworks used in our work. We start from the preliminary concepts like the graph and its representation. We then discuss the various approaches towards the iterative graph computation.

2.1 Graph

A graph, denoted by \( G \), is a set of pairs \( (V, E) \), where \( V \) is the set of vertices and \( E \) is either a set of directed edges (in case of a directed graph) or a set of undirected edges (in case of an undirected graph) \[29\]. If \( s \) is the source vertex, \( d \) is the destination vertex, and \( s, d \in V \), then a directed edge, \( e \), is an ordered pair \( (s, d) \in E \) (or \( V \times V \)). Similarly, an undirected edge, \( e \), is a pair or a set \( (s, d) \in E \) without having any orientation. That is, an undirected edge \( (s, d) \) is identical to the edge \( (d, s) \). The edge \( e = (s, d) \) is an out-edge of the vertex \( s \) and an in-edge of the vertex \( d \) \[13\]. The edge \( (s, s) \) is known as a loop. The cardinalities of \( V \) and \( E \), represented by \(|V|\) and \(|E|\) respectively, are the total number of vertices and the total number of edges in the graph.

The edges in a graph can be weighted. Weight is a numerical value assigned to represent additional information that exists on the edge between the connecting vertices. For example, if a map of a town is represented in a graph, a road (an edge) between two streets (vertices) can be weighted. Weight may represent the length of the road or the distance between the two streets. Besides, some of the graph representations may have vertex weights and vertex sizes for holding more information required for the graph processing (e.g., \[30\]).

There are two standard ways of representing graphs: as an adjacency matrix or as a collection of adjacency list \[29\]. However, we may modify the standard representations for our specific needs. Any vertex in a graph can be represented by an identifier. The adjacency list representation of a graph \( G \) is a collection of \(|V|\) adjacency lists, one list for each vertex \( v \) in \( G \) \[29\]. The adjacency list for each vertex \( v \) includes all the vertices \( u \)
that are directly connected to \( v \). In other words, the adjacency list of the vertex \( v \) contains the adjacent vertex \( u \) if the edge \((v, u) \in E\) exists [29]. Edge weight, if exists, can be kept alongside the vertex \( u \). The sum of the lengths of all the adjacency lists for any directed graph is \(|E|\), whereas the sum of the lengths of all the adjacency lists for any undirected graph is \(2|E|\) [29]. The adjacency matrix representation stores a graph \( G \) in a matrix \( A \) of order \(|V| \times |V|\). If \( A = (a_{ij}) \) then, \( (a_{ij}) \) can be defined as [29]:

\[
    a_{ij} = \begin{cases} 
        1, & \text{if } (i, j) \in E \\
        0, & \text{otherwise.} 
    \end{cases}
\]

If edge weights are provided, \( a_{ij} \) is the weight of the edge \((i, j) \in E\) instead of 1 [29]. Figure 2.1 shows a sample graph and its two graph representations: adjacency list and adjacency matrix. We base our data storage on the adjacency list representation.

![Sample Graph and its Graph Representations](image)

**Figure 2.1: A sample graph and its graph representations**
2.2 Graph Problems

Many real-world applications require solving big graph problems. For example, the World Wide Web itself has a graph structure that yields new insights for various problems like crawling, searching and community discovery [31]. Social networks, protein interaction networks, and the mobile call networks are the other domains that require big graph computations [32]. Some of the fundamental big-graph problems are Single Source Shortest Path, Connected Component, and PageRank [33].

If $G$ is a graph as defined in section 2.1, then the Single Source Shortest Path (SSSP) is the list of the shortest paths from a single source $s$ to all other vertices in $G$ [29]. Finding the Connected Components (CC) is a fundamental graph problem which involves finding all the set of vertices in the graph wherein there is a sequence of paths between any two vertices [29]. Similarly, PageRank [33] is the measure that measures the relative importance of the web pages in the web graph. It has applications in areas like search and browsing.

2.3 Programming Models

The vertex-centric programming model is the most common and well-established programming model developed for addressing the challenges faced in the large-scale graph computations [34]. It was first inspired by the Bulk Synchronous Parallel (BSP) [35] computing model [11, 34]. In what follows, we briefly describe the BSP [35] and the vertex-centric computing model.

2.3.1 Bulk Synchronous Parallel Computing

Bulk Synchronous Parallel (BSP) [35] computation model performs any parallel task by a series of synchronous supersteps of computation and communication. Each superstep consists of the following tasks: 1) Local computation, 2) Communication with other workers, and 3) Barrier-style Synchronization. The parallel workers first perform the computation locally on their share of the input data. Then, there is a communication phase where message transmissions and arrivals happen among multiple workers. Messages carry useful computation information such as the updates or the results from the local computa-
tions. In the end, there is a barrier-like synchronization which checks whether the superstep has been completed by all the workers or not. If yes, the computation proceeds to the next superstep [35].

2.3.2 Vertex-Centric Computing

Many contemporary graph-processing systems like Pregel [11], Giraph [24], Power-Graph [16], GraphLab [28], and Hama [23] are based on the vertex-centric programming model (or the Think Like A Vertex or TLAV programming model). We describe this model of computation based on GraphMap [13]. In the vertex-centric programming model, a sequence of iterations (or supersteps) carry out the graph computation. To define each iteration, users have to think from the viewpoint of a vertex and write a common vertex-centric function, which gets executed by all the vertices of the input graph in parallel [13]. The vertex-centric model performs the following actions in every iteration:

1. Each vertex reads the inbound messages (or updates) sent by its connecting vertices along with its current value.

2. The vertex then executes the provided algorithm to update its value based on the values from step 1.

3. Next, the vertex sends the result (or its updated value) to its connected neighboring vertices.

4. Finally, the vertex executes sync. That is, it waits for all other vertices to complete the above three steps. Bareer-like synchronization is marked by the execution of sync by all the vertices.

Additionally, all the vertices are equipped with a transition flag that indicates whether the vertex is active or not [13]. Only the active vertices carry out the computation. Any vertex can choose to deactivate itself in a iteration if it has no role in the remaining computation. However, the deactivated vertexreactivates if it receives any further message. The
computation terminates if all the vertices are deactivated or if the user-defined convergence point is reached (e.g., the number of iteration reaches some threshold) [13]. In the next section, we briefly describe GraphMap.

2.4 GraphMap

As stated earlier, GraphMap [13] is the state-of-the-art scalable NoSQL-based out-of-core iterative graph computation framework. It ensures a competitive performance against other graph computation frameworks like Hama [13]. Unlike the existing distributed memory graph processing systems like Pregel [11], Hama [23], and Giraph [24], it does not store the entire input graph along with the intermediate results in the memory. It separates the mutable part of the graph data from the immutable part, and stores only the mutable part in the memory [13]. For a weighted graph, each edge can be the attribute of the source vertices. Thus, edges can be considered immutable data of the mutable vertices. For example, in the SSSP algorithm, edges represent the distance between any two vertices and are immutable [13]. The read-only or the immutable part of the graph data is stored on the disk-based NoSQL system in a read-optimized way. Therefore, GraphMap reduces the memory requirement for the iterative graph computation [13]. Additionally, GraphMap achieves data-level load balance across the machines by using hash-based partitioning technique to divide the graph data uniformly across them [13]. Furthermore, GraphMap provides dynamic access pattern with which any worker can dynamically select either the sequential disk access pattern or the random disk access pattern based on its algorithm type and the ongoing performance to read the out-of-core data [13]. In the subsections below, we briefly describe the graph representation and the algorithms in GraphMap.

2.4.1 Data Representation

Graph data is represented in the form of a vertex block (VB) [13]. If $G$ is a graph as defined in 2.1, then for each vertex $v$ in the graph $G$, the corresponding VB representation is the vertex $v$ itself (called anchor vertex) along with all of its directly connected vertices $u$ such that $(u, v) \in E$ is an edge in case of in-edge VB or $(v, u) \in E$ is an edge in case
of out-edge VB or both \((u, v), (v, u) \in E\) are the edges in case of bi-edge VB [13]. A vertex \(v\) is represented with a unique vertex identifier. Additionally, it may have set of attributes for representing its state or properties [13]. Hash-based graph partitions (VB-Partitions) in GraphMap are the sets or the groups of VBs whose anchor vertices hash to the same value when using the global hash function. The VB representation places the edges together in the disk. Thus, they can be accessed together. Additionally, the VBs are stored contiguously so that they can be accessed sequentially from the disk [13].

2.4.2 Programming Approach

Along with the vertex-centric programming approach, GraphMap also supports the VB partition-centric approach [13]. The VB partition-centric approach defines what each VB partition does, instead of just a vertex, for each iteration [13]. Every partition executes the same VB partition-centric function in each iteration. The VB partition-centric function receives all the messages sent to it as a formal parameter in every iteration. Moreover, it stores all the mutable data (or the anchor vertices) of a partition in the memory. In other words, it keeps a map for storing the values of every vertex in its partition. The pseudo-code for the CC algorithm in terms of the GraphMap’s VB partition-centric model is provided in Appendix A.
### Chapter 3
#### Methodology

In this chapter, we present our approach to distributive iterative graph computations using a NoSQL database (HBase [36]). The overall architectural design of our system, the data placement strategy, the data locality implementation techniques, and the support for graph partitioning are briefly discussed.

#### 3.1 Graph Partitioning

We store the graph data as a table of vertex blocks (VBs) [13] in the NoSQL database where the anchor vertex ID [13] is the row-key. To support proper parallelization of the graph computation, the graph data needs to be uniformly distributed across multiple workers in the cluster for minimizing the total solution time. One way of doing this is by using the lightweight global hash-based partitioning technique as in GraphMap [13]. Another approach involves implementing the mincut-based partitioning technique for dividing the graph dataset. The objective of mincut-based partitioning is to minimize the total communication volume between the partitions during the graph computation (e.g., [37]). In other words, the mincut-based partitioning creates almost same-sized partitions by minimizing the total number of edges that straddle different partitions [38, 39]. To reduce the communication volume between the balanced partitions, we use the mincut-based partitioning technique for distributing the graph data across multiple workers in the cluster. The mincut-based partitioning technique divides the graph logically into $n$ chunks (or $n$ partitions) for $n$ workers in our cluster. The number $n$ is chosen such that the total size of the mutable data in any partition fits in the memory allocated for the worker processing that partition. The partitions are balanced in terms of per-iteration cost, and we design each worker to pick up just one partition of the graph data for the graph computation. Thus, the inter-worker data-level load balance is ensured.

We partition the graph data based on the anchor vertex IDs because they are the primary identifiers for the VBs. Mincut-based partitioning technique logically divides the
graph into multiple disjoint sets of VBs, each set belonging to one partition. If the graph is divided into n logical partitions, all the anchor vertices along with their VBs belonging to the partition $i$, where $0 \leq i \leq n - 1$, have the single and the same partition ID $i$. We need a new data structure or a new data representation to hold the partition assignment for each vertex. We refer to this new structure whenever the partition assignment of any vertex is required. Partition assignment of the vertex is required in the steps like storing the graph data in the underlying NoSQL system and distributing the updated messages across the cluster.

The workers exclusively process all the VBs belonging to one partition. To reduce the non-sequential disk access, all the VBs in each partition are kept contiguous. Therefore, the workers can process their respective logical partitions sequentially for faster access. In the next section, we describe how we store the logical partitions in the underlying NoSQL system.

3.2 Data Placement and Locality

Usually, database tables are physically divided into multiple segments for load balancing. In a NoSQL database system like HBase, regions are the basis for any table distribution [36]. A table can be divided into many regions distributed across multiple machines in the cluster [36]. A region stores only a fraction or a chunk of the original table data. Once we logically partition our dataset into multiple partitions as described in section 3.1, we store each partition in a separate table region (or a segment). In some special cases where the sizes of logical partitions are relatively small, multiple graph partitions can be stored in a single region. In brief, partitions are stored in multiple regions in the cluster in such a way that the total data across the multiple worker machines are well balanced. The data in each region (or the VBs) is kept contiguous to facilitate the sequential disk access to the workers. Moreover, they are stored in the sorted order of their anchor vertex IDs, which are indexed. Creating an index over anchor vertex IDs protects us from performing a full table scan while searching any particular VB. As a result, our system facilitates an
efficient random access of any VB.

Our design goal is to move the workers (or tasks) towards the data instead of moving the data towards the workers during the graph computation. This goal is achieved if a worker in any machine only processes the partition of the graph data that is physically stored on the same machine. To enforce this, we create a globally accessible data structure that stores all the partitions and their respective physical locations. When the workers are dispatched by the master machine, they query the data structure for listing all the partitions that are local to them. Each of the workers then exclusively selects one unique local partition to proceed the computation. In the special case where a region contains multiple partitions, a separate data structure is required to indicate that multiple partitions belong to the same region. A worker selecting one of such partitions has to select all the partitions belonging to the same region. To manage the inter-worker communication, an additional data structure is created which links partition IDs to the worker IDs. In every iteration of the graph computation, a worker processes the same local region. Thus, the intra-worker (or the worker-partition) data locality is achieved.

GraphMap [13] not only reduces the memory requirement but also offers a read-optimized data representation to our system. Only the mutable data is stored in the memory. The immutable or the invariant data are accessed from the NoSQL database [13]. In other words, only the anchor vertices are stored in the memory while their corresponding VBs are stored in the disk. Due to the intra-worker data locality, all the mutable data or the anchor vertices of a region are always accessed by the same worker during the entire computation. Thus, the vertex access locality [13] is achieved. Similarly, all the immutable data or the edges for any anchor vertex are placed together in the form of a VB, and they are accessed together. So, the edge access locality [13] is enforced.

3.3 System Architecture

Our system is an extension to GraphMap [13], which is based on the BSP [35] and the message-passing model. Figure 3.1 shows the overall architecture of our system. The
worker machines have rooms for running multiple workers (or tasks). Master machine does not run any worker. However, it takes the iterative graph computation task form the user and dispatches the workers in the worker machines to run the given computation in parallel based on the BSP and the message passing modules. Additionally, the master controls and coordinates the communication among the workers including the barrier synchronization. Data storage, inter-worker load balancing, and intra-worker data locality are performed by the workers themselves.

Figure 3.1: System architecture. Dotted lines without arrow zoom in the components, solid lines with arrows represent the inter-worker communication, and dashed lines with arrows represent the communication with the master.
Graph data is partitioned into multiple partitions. To store the graph partitions, a NoSQL table is split into multiple parts across the cluster. A graph partition is stored entirely in a single region. In figure 3.1, the NoSQL table for a dataset gets split into multiple regions: A, B, C, and so on. Similarly, the input graph dataset is divided into multiple parts: P0, P1, P2, and so on. Each graph data partition is stored in one region. The workers in the worker machines pick up one of the local partitions (or regions) of the graph data to proceed the computation. They carry out the computation by interacting with each other through message passing.

During the graph computation, workers store only the local anchor vertices, their updated values, and the partition assignment in their working memory. They read the edges information (or the respective VBs of the anchor vertices) form the underlying NoSQL regions. To update the anchor vertices, the workers carry out the computation by using the inbound messages and the respective edges information. Upon update, they communicate the result to the desired workers.
Chapter 4
Implementation and Experimental Results

We measure the performance of our system by adding a suite of data locality in the underlying NoSQL database for the iterative graph algorithms. In addition to this, we also analyze the effects of replacing the hash-based partitioning technique for the data-level load balancing across the workers with the mincut-based partitioning technique. The language chosen for the implementation and the experiments is Java. The major tools and frameworks that are used are Apache HBase [36], Apache Hama [23], Hazelcast [40], Metis [37], and Apache Hadoop [3]. In this chapter, the high-level system architecture, datasets, data storage, partitioning, data locality implementation, and the experimental results are briefly discussed.

4.1 System Setup

The implementation and the experiments are carried out on Emulab [41]. We create a cluster of 21 machines of which 20 are workers and the one is the master. All the nodes are equipped with 64 GB memory and Intel Xeon E5-2630v3 processor with 8 hardware cores. Further, they are configured with CentOS 7.3 operating system with Kernel version 3.10. The 1-GigE network connects all the nodes. Besides, all the machines have 162 GB local filesystem mounted on /mnt, which we use during our experiment. In addition, we use one GB filesystem that is shared across all the nodes for storing global configurations. We run three parallel JVM tasks per node in the worker machines (i.e. there are three worker slots per machine). The maximum configured JVM heap size per task is 16.5 GB. Data is uniformly distributed across the multiple workers. Each worker processes a unique section of the data, called the data partition. The master does not run any task. It schedules and controls the worker machines. We use the term worker or peer interchangeably for the workers, which run in parallel on each worker machine. Figure 4.1 shows the high-level system setup.
Figure 4.1: Experiment architecture. Dashed lines zoom in the slave machine and solid lines represent the network connection.

We create the workers using Apache Hama (version 0.6.3). Similarly, we use Apache HBase (version 0.96), a NoSQL database system, to store our graph data. We use Hadoop (version 1.0.4) for Hadoop Distributed File System (HDFS) and data preprocessing. Additionally, Metis (version 5.1.0) and Hazelcast (version 3.8.2) are used in our system for the mincut-based partitioning and the distributed in-memory data structure, respectively. The tools and software that we use are briefly discussed in the subsections below.

4.1.1 Hadoop

Hadoop [3] is an open-source framework for storing and processing big data. It stores data in the Hadoop Distributed File System (HDFS), which runs on top of the local filesystem. HDFS divides any data into blocks and distributes it across the cluster. The default replication factor of three for each block provides fault tolerance to our system. Hadoop processes data using MapReduce [6] programming model. MapReduce program solves any problem by using a combination of mappers and reducers. The mapper reads the chunks
of each data block, called the input splits, then produces key-value pairs as intermediate results. The reducer further processes the intermediate results to provide the final solution. Along with the mapper and the reducer, we also use the RecordReader and the Partitioner class of Hadoop. The mapper internally uses the RecordReader class, which defines a way to read an input split from HDFS. Partitioner is used to distribute the intermediate data across multiple reducers [3].

4.1.2 HBase

Apache HBase [36] is an open source wide column store database that runs on top of HDFS. It is fault tolerant as it utilizes the block replication provided by the underlying HDFS. We use HBase to store our dataset in the form of VB. We also use the range-scan or the prefix-scan feature provided by the HBase to read the VBs. All the rows having a common prefix can be retrieved using the range-scan feature. Additionally, we use region pre-splitting feature of HBase to split our tables into multiple parts. Table creation and pre-splitting are performed using HBaseAdmin, which is the class for managing all sorts of table metadata [36].

4.1.3 Hama

Apache Hama [23] is a distributed computing framework based on the BSP computing technique. It includes a powerful graph package as well, making it an open source alternative to Google’s Pregel [11], which is a distributed system for large-scale graph processing. We use Hama for its BSP and the message passing modules. Hama has three major components: the BSPMaster, the GroomServer, and the Zookeeper. GroomServers are basically our workers, which carry out BSP tasks. The BSPMaster schedules and controls the BSP tasks, the supersteps, and the barrier synchronization. All the configurations required for the barrier synchronization are managed by the ZooKeeper [23].

4.1.4 Hazelcast

Hazelcast [40] is an open source in-memory data grid. It is based on Java. We use Hazelcast for our intra-worker data locality. Intra-worker data locality or the worker-partition
data locality is implemented using the two distributed data structures: IAtomicLong and IMap. IMap is the concurrent distributed map which can be accessed by all the workers. Similarly, IAtomicLong is a distributed long object which changes atomically [40]. We use IAtomicLong as a distributed sequence.

4.1.5 Metis

Metis [37] is a state-of-the-art software package for partitioning graphs. It is faster than other common graph partitioning algorithms by one to two orders of magnitude [37]. Metis is written in ANSI C, and it provides various programs and options for validating graphs and partitioning them. We use two Metis programs in our experiment which are graphchk [30] and gpmetis [30]. The input to these programs is a file having a special graph representation, which we will discuss in the later sections. The graphchk tool validates the input format whereas the gpmetis tool divides the graph into multiple partitions using the mincut-based multilevel k-way partitioning algorithm [37] by default.

4.2 Data Storage

We create a separate HBase table for storing each graph dataset. To distribute the graph dataset uniformly across \( n \) workers, we first divide the table for each dataset into \( n \) segments (or regions) using HBase pre-splitting [36]. Pre-splitting means splitting the table before inserting any data on it. We use the prefix-based splitting on the row-keys to distribute the graph dataset across multiple regions. The BSPMaster assigns the workers with the identifiers 0 through \( n - 1 \). Each of which exclusively processes a unique region. Thus, we create \( n \) splitting points in the table having splitting prefixes 0 to \( n - 1 \) with a hyphen (-) attached at the end. HBase usually distributes the regions uniformly across the slave machines. Therefore, each of the 20 workers in our cluster gets 3 regions per table. A simple Java application is used for creating and splitting the HBase table. The application uses HBaseAdmin class provided by the HBase. Once the table is created and split, we use a MapReduce application to store the graph dataset uniformly across the multiple regions.
Data is stored using the Vertex Block (VB) [13] representation. Since a separate data structure or data representation is required for holding the partition assignment for each vertex for the mincut-based partitioning, we define the Extended Vertex Block (EVB) representation. In both the VB and the EVB representations, the data for any vertex is stored in the form of a block where the anchor vertex is followed by its adjacency list. In case of a VB, the adjacency list for any vertex is a whitespace delimited list of its directly connected vertices. Whereas, in case of an EVB, it is the whitespace delimited list of directly connected vertices along with their partition IDs separated by a hyphen (-). We refer to the partition file (our global map) for concatenating each connected vertex in the connection list with its partition assignment. Partition file is described in detail in the
coming sections. The adjacency list for any vertex in our experiment consists only of the
out-edges.

The MapReduce application creates the VB or the EVB representation of the data
before storing it into the HBase table. Partitioning of the graph data into regions is done
based on the prefix of the row-key. Thus, we prepend each anchor vertex in the VB with
its corresponding partition ID. For the hash-based partitioning, division hash function is
used to find the prefix partition ID. For instance, consider a scenario where the number
of workers is 20. To divide the data among these 20 workers, we create 20 regions in the
HBase table. These regions have prefixes ranging from 0- to 19-. A worker $n$ gets to
process the region $n$. If the anchor vertex in any VB is $m$, then the entire VB is stored in
the partition $m\%20$ and the anchor vertex is stored as $m\%20-m$ where $m\%20$ is the prefix.
For the mincut-based partitioning, the prefix partition ID is queried from the partition file.

Figure 4.2 shows the storage of a sample graph data in the HBase. The top part
of the figure shows the VB representation. Similarly, the bottom part shows the EVB
representation, which contains the partition assignment of each of its connected vertices as
well. All the anchor vertices belonging to the partition 0 have the prefix 0−. Similarly,
all the anchor vertices belonging to the partition 1 have the prefix 1−. The partitions are
stored in different regions as they have different prefixes.

4.3 Data Locality

A distinct row having a distinct row-key belongs to only one region in our HBase setup.
Similarly, a HBase region is kept inside only one worker machine. Consequently, all the VBs
or the EVBs in the same partition are stored inside the same machine. Thus, we achieve
the vertex access locality [13] in our experiment. Additionally, HBase internally stores
the row-keys in the lexicographical order in the flat-file. Consequently, the non-sequential
access of the VBs is reduced by the range scan of HBase. In addition, the full table scan is
also avoided when searching a particular VB. Therefore, the performance of random access
of the data is improved. We store the entire adjacency list or the VB of any anchor vertex
in the same column. HBase stores same column data together in the disk. Hence, we can access the edges of any anchor vertex together. This satisfies the edge access locality [13] approach.

As discussed earlier, a separate worker is created for processing each partition of the graph dataset. Each worker is identified by a unique ID given by the BSP module. Worker IDs have the same nomenclature as the partition IDs. However, a worker may not get a chance to process its local data that is on its local disk. Usually, workers need to fetch the data from other worker nodes which are on different machines. In figure 4.1, the three workers \( m, n, \) and \( o \) in slave1 uniquely process the three partitions of the HBase table. However, the three partitions may not lie in the same slave machine if the intra-worker data locality is not implemented. In other words, the workers \( m, n, \) and \( o \) may not be processing the partitions \( i, j, \) and \( k \) respectively. Our aim is to implement the intra-worker data locality which makes the workers process only the local partition. This can be achieved if the workers \( m, n, \) and \( o \) get to process partitions from the pool of \( i, j, \) and \( k \).

Table 4.1: Region information file

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
</tr>
</tbody>
</table>

To enforce the intra-worker (or the worker-partition) data locality, we first extract all the region information of a table from HBase. The region information for any region includes its partition ID, region server, and the host machine on with it exists. To every worker machine in the cluster, we then send the list of all the local regions that are inside it. This information is locally kept in a region information file inside each machine. The region information file is shown in table 4.1. First column shows the line numbers. The numbers 25, 30, and 19 are the region IDs local to the machine to which the region information file belongs.

The region information file is read by the workers in the order governed by the dis-
tributed sequence, IAtomicLong, of HazelCast. A unique sequence number is generated for any worker when it accesses the IAtomicLong object. We create a separate IAtomicLong object for every slave machine. A worker belonging to a machine only accesses the IAtomicLong object created for that machine. There are 3 workers in a single machine, and they get the value 1, 2, and 3 respectively from the distributed sequence created for that machine. They read the same line number from the region information file as their distributed sequence number. That is, the worker machine getting the sequence value 2 reads the second line of the file. Therefore, it gets to process the region which is mentioned in the second line. In this way, every worker gets to process the region that is local to it. Once all the workers get their respective region IDs, we store them in a distributed in-memory map provided by Hazelcast for future use. In the vertex-centric approach, vertices need to receive the updated information after each round of computation. In order to send the updated message for any vertex, a worker needs to find the whereabouts of that vertex. Distributed in-memory map of Hazelcast is used to serve this purpose.

4.4 Mincut-based Partitioning

GraphMap uses global hash partitioning to distribute the graph uniformly across the workers [13]. VBs hashing to the same value fall under the same partition. We analyze the performance of mincut-based partitioning technique as an alternative to hash-based partitioning technique in our system. To perform the mincut-based graph partitioning in our dataset, we use Metis [37]. The entire steps involved in partitioning the graph data are discussed briefly in the subsections below.

4.4.1 Input Format

The input graph for Metis needs to be undirected [30]. Let us consider an undirected graph $G$ where $n$ is the total number of unique vertices and $k$ is the total number of unique undirected edges. Metis represents the graph $G$ as a flat text file having $(n + 1)$ lines where the first line is the header line [30]. Since vertex weights, edge weights, and vertex sizes in our experiment are all 1, we ignore the weights parameter. Therefore, the header only
contains the values $n$ and $k$ delimited by a whitespace. The remaining $n$ lines have the information for $n$ vertices, and they define the actual structure of the graph. Each vertex is represented in a separate line. The comments, which start with the percentage sign ($\%$), are not counted as lines. The vertices are numbered sequentially, strictly starting from 1. Each line $i$, where $2 \leq i \leq n + 1$, contains the whitespace delimited list of vertices which are adjacent to the vertex $i - 1$ [30].

A sample graph and its Metis representation

A sample graph and its Metis format graph file is shown in figure 4.3. The graph has 6 vertices and 8 edges. Thus, the first line in the graph file contains 6 and 8, and the total length of the file is 7. The second line in the file contains a list of vertices that are adjacent to the vertex 1. They are 2, 4, and 3. Similarly, the third line includes the list of all the vertices that are connected to the vertex 2. Any unique edge, such as the edge between vertex 1 and vertex 2, is counted only once. However, it is represented two times in the graph file: one in the adjacency list of the first vertex and the other in the adjacency list of the second vertex.
4.4.2 Graph Conversion

The input graph format for Metis is different from that of our system. Therefore, we convert our graph into the Metis input format. However, we do not stick to this format throughout our experiment. We use this format only to partition the graph. Once the graph is partitioned, we store the partition assignment generated by Metis in the HBase table along with the pristine dataset.

In the first step of the data conversion process, we number the vertices sequentially starting from 1. All the unique vertices in our graph are sorted numerically based on their IDs. The first vertex in the list gets the number 1. Similarly, the second vertex gets the number 2. In the same way, the last vertex gets the number \( n \) where \( n \) is the total number of vertices in the graph. We store the all the vertices and their corresponding sequence IDs in a map. We then refer to this map to convert the input file to another file having the same format, but the different naming of the vertices. The next step is to convert the transformed file into Metis input format.

A RecordReader [3] in Hadoop reads the input split from HDFS and presents every line in it as a key-value pair. We create a custom RecordReader to read the transformed graph file. The custom RecordReader, called the GraphRecordReader, ignores the comment in the graph file and emits both keys and values as long integers for the data line. A MapReduce program is created which utilizes the GraphRecordReader and converts the transformed file into the Metis input format. The program first converts the graph into an undirected graph by adding extra edges if needed. A directed edge \((u, v)\), where \( u \) and \( v \) are the two vertices, can be made undirected by adding another edge \((v, u)\). The mapper of our MapReduce program emits all the vertices as the keys and their individual connections or edges as the values. The reducer method groups all the adjacent vertices together for each unique vertex, and writes them to the output file in the sorted order of their respective vertices. In addition, the reducer also removes the loops in the graph. We use two counters to count the total number of vertices and edges. The count of unique vertices and unique
edges are written on the first line of the output file.

To speed up the conversion process, we distribute the intermediate data across multiple reducers. Some of the datasets are highly skewed in terms of the number of edges per vertex. Thus, we create a custom partitioner to distribute the data uniformly across multiple reducers. The partitioner also preserves the natural ordering of the vertices while dividing the data. Natural ordering is needed because Metis represents its graph data in the sorted order of the vertices [30].

Table 4.2: Original dataset (top) and its Metis format (bottom)

<table>
<thead>
<tr>
<th># Original Graph Dataset</th>
<th>Edges: 16518948</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes: 3774768</td>
<td></td>
</tr>
<tr>
<td># FromNodeId</td>
<td>ToNodeId</td>
</tr>
<tr>
<td>3858241</td>
<td>956203</td>
</tr>
<tr>
<td>3858241</td>
<td>1324234</td>
</tr>
<tr>
<td>3858241</td>
<td>3398406</td>
</tr>
<tr>
<td>3858241</td>
<td>3557384</td>
</tr>
<tr>
<td>3858241</td>
<td>3634889</td>
</tr>
<tr>
<td>3858242</td>
<td>1515701</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>% Equivalent Metis Format</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>% First line is the header</td>
<td></td>
</tr>
<tr>
<td>3774768</td>
<td>16518947</td>
</tr>
<tr>
<td>1760391</td>
<td>2439226</td>
</tr>
<tr>
<td>1913824</td>
<td>265037</td>
</tr>
<tr>
<td>1924676</td>
<td></td>
</tr>
<tr>
<td>1768138</td>
<td></td>
</tr>
<tr>
<td>3284975</td>
<td>3039282</td>
</tr>
<tr>
<td>3074384</td>
<td>3090013</td>
</tr>
<tr>
<td></td>
<td>3139274</td>
</tr>
</tbody>
</table>

Table 4.2 shows the parts of the original cit-Patents dataset at the top and its Metis-file at the bottom. Comments are not considered any lines. The first line in the Metis file has two columns. The first column is the number of vertices, and the second column is the number of unique undirected edges. Total number of vertices in both the formats are same. However, the number of edges are different because of the presence of loops. Loops are removed for the Metis input format.
4.4.3 Partitioning

Once the graph is converted into the Metis input format, it is partitioned using the gpmetis tool [30]. We use the multilevel k-way partitioning scheme [30], which is the default scheme for partitioning the graph. The gpmetis tool is the standalone tool which runs in a single machine [30]. It takes two mandatory arguments: the input graph file and an integer nparts, which is the number of segments into which the graph is partitioned [30]. The partitions are numbered 0 to nparts – 1. We partition the graph into 60 parts to match with the number of workers in our cluster. Small graph datasets are partitioned using one of the worker machines in our cluster. However, due to the high memory requirements, we use QB2, the Louisiana Optical Network Initiative (LONI) [42] machine having 1.5 TB memory and four 12-core 2.6 GHz E7-4860v2 Xeon processors for the large graph datasets.

![Figure 4.4: A sample graph and its partition file](image)

The gpmetis tool produces a partition file as an output. If a graph with n vertices is divided into nparts parts, then the partition file contains n partition IDs in a single column. The partition IDs have the value between 0 to nparts – 1. The vertex i, 1 <= i <= n, belongs to the partition defined in line i of the partition file [37]. Figure 4.4 shows the two partitions and the partition file for the sample graph shown in figure 4.3. Each of the first,
the second, and the fourth lines in the partition file contains the value 1. Therefore, the vertices 1, 2, and 4 belong to the partition 1. Similarly, the vertices 3, 5, and 6 belong to the partition 0. We add a new column to the partition file to store the corresponding vertex IDs. We then store the partition file in the shared global filesystem. The partition file is used in our experiment during the data storage phase. Partition information for any VB or EVB is mainly used for the inter-worker load balancing. It is also used during the message passing to find the right partition for any vertex.

Partition file for the hash-based partitioning can also be created by applying a global hash function to the sorted list of all the vertices for any graph. The vertices along with their hash partitions are stored in the two columns. We use the hash-based partition file for storing the hash-distributed graph data in EVB representation.

4.5 Experimental Evaluation

We carry out our experiment for three iterative graph computation algorithms: CC, SSSP, and PageRank [33]. The system setup for the experiments is same as described in the implementation section. GraphMap [13], its dynamic access pattern, and its programming API are used to perform all the graph computations. The vertex having the maximum reachability from other vertices is chosen as the source vertex for the SSSP algorithm. We stop the PageRank algorithm in 10 iterations. The number of active vertices change based on the type of computation done [13]. For the PageRank algorithm, all the vertices are active during the entire computation. For the CC algorithm, all the vertices are active to begin the computation. The number of active vertices then begins decreasing as the iterations continue. Similarly, for the SSSP algorithm, only the single source is active during the first iteration. Active vertices then increase over the next few iterations, and again decrease [13]. Each algorithm is run three times for each dataset using the same configuration. The fastest execution time among the three runs is recorded.
4.5.1 Datasets

Real-world datasets from different domains are used in the experiment. Table 1.1 shows the overview of the datasets that are used in our experiment. The datasets are of different sizes and possess different characteristics. They are stored in the text file in two columns. The first column is the source vertex, and the other column is the destination vertex. Comments, which start with the hash sign (#), are ignored. The Vertex weights, edge weights, and vertex sizes are not provided in our dataset. Thus, they all are assumed to be 1 if needed. Some of the graphs are undirected while the others are directed. We use the graphs in their own form in the experiment without any modification in their structure.

Table 4.3: The datasets used in the experiment

<table>
<thead>
<tr>
<th>Dataset</th>
<th>number of vertices (in millions)</th>
<th>number of edges (in millions)</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>cit-Patents [43]</td>
<td>3.8</td>
<td>16.5</td>
<td>0.26 GB</td>
</tr>
<tr>
<td>soc-LiveJournal1 [44]</td>
<td>4.8</td>
<td>69</td>
<td>1.1 GB</td>
</tr>
<tr>
<td>orkut [45]</td>
<td>3.1</td>
<td>224</td>
<td>1.7 GB</td>
</tr>
<tr>
<td>hollywood-2011 [46]</td>
<td>2.2</td>
<td>229</td>
<td>3.3 GB</td>
</tr>
<tr>
<td>uk-2005 [46]</td>
<td>39</td>
<td>936</td>
<td>16 GB</td>
</tr>
<tr>
<td>twitter [47]</td>
<td>42</td>
<td>1500</td>
<td>25 GB</td>
</tr>
</tbody>
</table>

The cit-Patents [43] dataset is a directed, temporal, and labeled graph of the citation network among US patents. The soc-LiveJournal1 [44] dataset represents the directed connections of LiveJournal online social network. Similarly, the orkut dataset [45] and the twitter [47] dataset are the datasets of Orkut friendship social network and the Twitter follower-following social graph. The hollywood-2011 [46] dataset contains the undirected connections of Hollywood actors based on their common or shared screen. Finally, the uk-2005 [46] dataset has the links and connections of websites with the .uk domain. Table 4.4 shows the time required to load the dataset first into HDFS and then into HBase.
Table 4.4: Total time to store data

<table>
<thead>
<tr>
<th>Dataset</th>
<th>total time to store</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HDFS</td>
<td>HBase</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>4 secs</td>
<td>3.3 mins</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>16 secs</td>
<td>4.15 mins</td>
</tr>
<tr>
<td>orkut</td>
<td>28 secs</td>
<td>3.2 mins</td>
</tr>
<tr>
<td>hollywood-2011</td>
<td>33 secs</td>
<td>3 mins</td>
</tr>
<tr>
<td>uk-2005</td>
<td>2.5 mins</td>
<td>31 mins</td>
</tr>
<tr>
<td>twitter</td>
<td>4 mins</td>
<td>42 mins</td>
</tr>
</tbody>
</table>

4.5.2 Mincut-Based partitioning

Table 4.5 provides the total execution time and the total memory used by the gpmetis tool for partitioning the datasets into 60 parts. It is fair to say that a significant amount of time and memory is required to perform the mincut-based partitioning although it is a one-time job.

Table 4.5: Execution time and memory used for partitioning

<table>
<thead>
<tr>
<th>Dataset</th>
<th>gpmetis tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total time</td>
</tr>
<tr>
<td></td>
<td>HDFS</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>38.215 secs</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>1.58 mins</td>
</tr>
<tr>
<td>orkut</td>
<td>2.753 mins</td>
</tr>
<tr>
<td>hollywood-2011</td>
<td>1.3 mins</td>
</tr>
<tr>
<td>uk-2005</td>
<td>7.547 mins</td>
</tr>
<tr>
<td>twitter</td>
<td>153.34 mins</td>
</tr>
</tbody>
</table>

We run the same experiments on two sets of partitions for the same dataset: hash-based partitions and the mincut-based partitions. For all the algorithms, messages originating from the same source worker (or partition) and going to the same destination worker (or partition) are combined in every iteration. Table 4.6 shows the total number of messages sent by all the workers across the cluster when running the algorithms on the hash-based partitions and the mincut-based partitions for the same graph dataset. We can clearly see that the number of messages is significantly reduced by the use of mincut-based partitioning technique (up to an order of magnitude). The edge-cuts between the partitions
are minimized in mincut-based partitioning. Thus, the inter-partition communication is reduced.

Table 4.6: Total number of messages (Metis vs. Hash)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SSSP</th>
<th>CC</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total Number of messages (millions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mincut</td>
<td>hash</td>
<td>mincut</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>0.178</td>
<td>0.211</td>
<td>5.063</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>46.45</td>
<td>51.26</td>
<td>215.50</td>
</tr>
<tr>
<td>orkut</td>
<td>27.97</td>
<td>77.78</td>
<td>114.86</td>
</tr>
<tr>
<td>hollywood-2011</td>
<td>15.40</td>
<td>80.28</td>
<td>60.944</td>
</tr>
<tr>
<td>uk-2005</td>
<td>74.075</td>
<td>444.913</td>
<td>661.821</td>
</tr>
<tr>
<td>twitter</td>
<td>283.451</td>
<td>585.185</td>
<td>651.428</td>
</tr>
</tbody>
</table>

Table 4.7 shows the total execution time for running the algorithms when the datasets are partitioned using the mincut-based partitioning and the hash-based partitioning approach. The algorithms on the mincut-based graph partitions are consistently slower than those on hash-based partitions.

Table 4.7: Total execution time (Metis vs. Hash)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SSSP</th>
<th>CC</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total execution time (sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mincut</td>
<td>hash</td>
<td>mincut</td>
</tr>
<tr>
<td>hollywood-2011</td>
<td>24.419</td>
<td>12.409</td>
<td>63.485</td>
</tr>
<tr>
<td>uk-2005</td>
<td>204.561</td>
<td>123.547</td>
<td>517.255</td>
</tr>
<tr>
<td>twitter</td>
<td>354.681</td>
<td>114.5</td>
<td>694.493</td>
</tr>
</tbody>
</table>

Surprisingly, the mincut-based partitioning is slower although the number of messages is significantly less (up to an order of magnitude). On drilling down the causes, we discover that the total number of edges in the mincut-based partitions are extremely skewed although the number of vertices are more or less balanced. Some of the partitions have a very high number of edges than the others. The number of edges corresponds to the number of immutable data. Active vertices in any partition need to process the immutable
edges data to update their values. They usually have to scan the NoSQL database to read all of their edges. Therefore, the partitions having a large number of edges need to pull and process more data from the disk. If one of the workers is slow in the BSP model, every other worker needs to wait for its completion to go to the next superstep. Thus, the partition having the largest number of edges is the bottleneck for the overall performance. Hence, the partitions formed by the mincut-based approach may not be suitable for proper parallelization and load balancing.

Figure 4.5 shows the distribution of the edges of the hollywood-2011 dataset at the top row. The graphs for the hash-based partitions are shown on the first column. The corresponding counterpart results for the mincut-based partitions are shown on the second column. Hash-based partitions are well balanced in terms of the number of edges per partition. However, the mincut-based partitions are skewed in terms of the number of edges per partition. The remaining two rows in the figure show the executions of the PageRank and the CC algorithms on the hollywood dataset. The execution time for both the algorithms corresponds to the edge-distribution pattern. As the number of edges increases, the computation time for the partition also increases given the vertices are active. In a BSP job, barrier synchronization marks the end of any iteration. Therefore, the worker which finishes the fastest must wait for the longest time (wait for all other workers). For the majority of the iterations in the figure, the partition having the highest number of edges is takes the longest time. The average time taken by the slowest partition to complete its iteration is much higher than that of any partitions in the hash-based partitioning. Thus, mincut-based partitioning is slower.

The same edge-distribution vs. iteration-time relationship is true for the Figure 4.6 as well. The iteration time to execute the algorithm for the different partitions corresponds to their respective edge-distribution for the UK dataset. In some cases, the partition having the highest number of edges takes less time to compute than others. This happens if the vertices in that partition are inactive.
Figure 4.5: Edge distribution and iteration times per partition for Hollywood-2011 dataset using hash-based vs. mincut-based partitioning
Figure 4.6: Edge distribution and iteration times per partition for UK-2005 dataset using hash-based vs. mincut-based partitioning
4.5.3 Data Locality

Implementation of the worker-partition data locality or the intra-worker data locality makes the workers process only the local partitions. The performance of this new locality based system in comparison with the original GraphMap system is shown in table 4.8. For small datasets, the overhead of using Hazelcast diminishes the advantage of using data locality. However, for the large datasets like twitter and uk-2005, the performance of worker-partition data locality is consistently better over the multiple rounds of experiments. The execution time increases up to 17% for the SSSP, 39% for the CC, and 8% for the PageRank algorithm, where the outbound messages to the same target partition are not combined. Since the workers do not fetch the data from remote nodes to start the computation, the performance improves. Workers read the immutable data locally. Therefore, network overhead during the iterative graph computation is reduced.

Table 4.8: Total execution time (with and without locality)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>total execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSSP</td>
</tr>
<tr>
<td></td>
<td>Locality Impl.</td>
</tr>
<tr>
<td>orkut</td>
<td>15.3</td>
</tr>
<tr>
<td>uk-2005</td>
<td>75.453</td>
</tr>
<tr>
<td>twitter</td>
<td>78.454</td>
</tr>
</tbody>
</table>
Chapter 5  
Conclusions

In this thesis, we have investigated two new extensions to GraphMap [13], the state-of-the-art distributed iterative graph computation framework in an out-of-core setting based on NoSQL database. The first extension is an additional layer of data locality, which moves the computation towards the data rather than moving the data towards the computation. With this new extension, the system can perform up to 39% faster. Summing up, we have shown an effective way of using NoSQL database for the processing of distributed iterative graph algorithms. NoSQL database is a feasible option to store the graph data for the efficient and scalable iterative graph computations, especially when the memory resources are limited.

The second extension is the support for using the mincut-based partitioning technique for distributing the graph data across multiple workers in the cluster. Mincut-based partitioning technique reduces the number of inter-partition messages. Thus, the total communication volume between the workers during the graph computation is reduced. However, the performance of the system may decline due to sub-optimal load balancing and improper parallelization. In case of real-world graphs where the vertices have an extremely varying degree of edges, the mincut-based partitioning technique may not divide the graph uniformly across the workers in terms of the number of edges. Some of the graph partitions may have a large number of edges compared to others. Although the number of inter-partition messages is reduced, some of the workers may still require high computation and I/O access time because of having an extremely large number of edges. In an iteration of a Bulk Synchronous Parallel model, if a worker is slow then every other worker is slow. Moreover, a worker with a large number of edges needs to spend a long time for the disk read. Thus, there is improper load balancing and sub-optimal parallelization across the workers.
Our system can be further studied by implementing the vertex-cut partitioning [48] technique instead of the mincut-based partitioning technique. Very little work has been done in the area of vertex-cut partitioning. Vertex-cuts are claimed to be significantly effective than the edge-cuts [48]. Similarly, edge-centric graph computing [18] model can be implemented on top of NoSQL database with data locality. Moreover, our system can be modified to use its own optimized version of NoSQL database to enforce the intra-worker data locality by default. Additionally, the performance of our system may improve if we can reduce the overhead of using extra data structures for achieving data locality.
References


Appendix A
Pseudocodes

The pseudocode for the Connected Component algorithm based on the VB partition-centric [13] model is shown below:

```plaintext
bspCompute(messages) // This method runs for each partition; messages are received as formal parameter
setPartition(getPartition()) // Sets the working-partition of the current worker
if getCurrentSuperstep() == 0 then // For the first superstep
    for Vertex v: readAllAnchorVertices() do // readAllAnchorVertices() reads the anchor vertices from HBase for the current local partition
        setMutableDataMap(v,v); // Mutable data is kept in memory. The initial value of the vertex is the vertex itself
        activate(v); // Sets the vertex v as active
    end for;
end if;
while (true) do // Beginning of iterations
    for Message msg: incomingMessages do // Reads all the inbound messages to the current worker (or partition)
        if msg.value < getMutableDataMap(msg.vertex) then // getMutableDataMap(v) reads the current value of the mutable data (or the vertex)
            setMutableDataMap(msg.vertex, msg.value);
            activate(msg.target);
        else
            deActivate(msg.target); // De-activates the vertex
        end if;
    end for;
    if convergence() == true then // Check if the convergence has reached
        break;
    end if;
    for Vertex v: readAllActiveVertices() do // Reads all the active vertices in the current iteration
        for Edge e: v.getEdges() do // getEdges() for any anchor vertex reads the corresponding VB from the NoSQL store
            msg = createMessage(e.endVertex, getMutableDataMap(v)); // Creates the message for the connected vertices
            sendMessage(getWorker(e.endVertex),msg); // Sends message to the right worker
        end for;
    end for;
end while;
```
Vita

Ayam Pokhrel was born on January 12, 1993. He graduated with a Bachelor in Computer Engineering from Kathmandu University, Dhulikhel, Nepal in 2014. Right after his graduation, he worked as software engineer in Verisk Information Technologies (rebranded as Verscend Technologies) for about 2 years. Thereafter, he joined Louisiana State University (LSU) to pursue Masters in Computer Science. He has been a teaching assistant since he joined LSU. Currently, he is teaching at McKinley Senior High School, Baton Rouge, Louisiana. He holds greater interests to work in Big Data Technologies, Distributed Systems, and Data Mining.