APPROVAL SHEET

Title of Thesis: yInMem: A Parallel Distributed Indexed In-Memory Computation System for Big Data Analytics

Name of Candidate: Yin Huang
Ph.D. in Computer Science,
2017

Thesis and Abstract Approved: _____

Dr. Yelena Yesha
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of dissertation: yInMem: A Parallel Distributed Indexed In-Memory Computation System for Big Data Analytics

Yin Huang, Doctor of Philosophy, 2017

Dissertation directed by: Ph.D. Yelena Yesha, Professor
Ph.D. Shujia Zhou, Professor
Department of Computer Science and Electrical Engineering

Cluster computing is experiencing a surge of interest in in-memory computing system with the advances in hardware such as memory. However, the network media has the smallest bandwidth as compared to memory and disk in a typical setting of cluster computing environment. In addition, the sparse nature of graph applications, such as social network, imposes new challenges for in-memory computing system. Examples of such challenges are data locality, workload balance and memory management. As a result, fine control over data partitioning and data sharing plays a crucial role in improving the speed of large-scale data-parallel processing systems by reducing the cross-node communication. In order to maximize the performance, in-memory computing system should be offering optimized data throughput for parallel computation in large-scale data analytics.

This dissertation presents yInMem: a parallel, distributed, indexed, in-memory computing system for big data analytics. With the goal of building an in-memory computing system that enables optimal data partitioning and improves efficiency

of iterative machine learning and graph algorithms, yInMem bridges the gap between HPC and Hadoop by parallelizing the computation with MPI while obtaining the advantage of distributed data storage, such as NoSQL database built on top of Hadoop. The novelty of yInMem results from introducing indexes or associative arrays to the in-memory computing system. Such a design offers benefits of fine control over data distribution with parallel computation to maximize the computing resources usage in the cluster.

By analyzing the linear algebra characteristics of iterative machine learning and graph algorithms, such as spectral clustering and PageRank, we find that yInMem is capable of maximizing the usage of computing resources in the cluster. Leveraging the insights of Sparse Matrix-Vector Multiplication (SpMV), we also provide an optimal data partitioning algorithm on top of yInMem for load balance and data locality.

In order to evaluate yInMem, we investigate iterative machine learning and graph algorithms using both synthetic benchmarks and real user applications. yInMem matches or exceeds the performance of existing specialized systems.

yInMem: A Parallel Distributed Indexed In-Memory Computation System for Big Data Analytics

by

Yin Huang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the status of
Doctor of Philosophy
2017

Advisory Committee:
Professor Yelena Yesha, Chair/Advisor
Professor Shujia Zhou
Professor Milton Halem
Professor Yaacov Yesha
Dr. Walid Keyrouz

To my family

ACKNOWLEDGMENTS

I would like to thank Yelena Yesha, Milton Halem, Shujia Zhou, Yaacov Yesha, and Walid Keyrouz for their support and guidance through this incredible journey. I would also like to thank the Center for Hybrid Multicore Productivity Research and all CHMPR lab mates for their company and help throughout my work, especially Tim Blattner, Kimberly Blattner, Dorsa Ziaei, Navid Golpa, Asen Radov, Lawrence Sebald, Phuong Nguyen, Smriti Prathapan, Hadis Dashtestani, Shruti Sanjeev and Robert Ginsburg. I would also like to thank IBM for providing me fellowship. Thank you to my family, particularly my parents for their encouragement and also my niece Wan-ru.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The advent of big data requires more and more applications to scale out to distributed systems. The growing data sources produce large and valuable data. The examples are ranging from the scientific instruments, business operations to social media. More and more companies are either deploying their data center into cloud storage or building their own distributed storage system. As the hardware prices keep decreasing, it is common to see clusters of hundreds of machines. The performance of cluster computing, therefore, plays a key role in maximizing the value of analyzing the data sources.

Over the past few years, tremendous efforts have been made to improve the scalability, efficiency and fault-tolerance of large-scale data processing systems. Hadoop [1] has become the most popular open-source framework to handle Big Data Analytics, in which Hadoop Distributed File System(HDFS) [3], serves as the primary storage layer for Hadoop MapReduce [2] model. Current HDFS design can not leverage high-performance networks because of the application performance being bounded by the disk access. While MapReduce has been highly successful in implementing large-scale batch jobs, it is a poor fit for low-latency interactive and iterative computations, such as machine learning and graph algorithms.

To overcome the bottleneck mentioned above, cluster computing systems are experiencing a surge of interest in in-memory computation given the fast growing bandwidth gap among memory, disk and network. Typical examples of such in-memory computing models include Spark [29], MEM-HDFS [38], and Piccolo [46]. Examples of in-memory file systems include Alluxio (formerly known as Tachyon) [33] and Triple-H [40].

In-memory cluster computing has become a de-facto standard for big data workloads with the success of Apache Spark. Apache Spark extends MapReduce to Resilient Distributed Datasets(RDDs) [23] which can be cached into RAM to reduce their access latency. RDD is a fault-tolerant collection of objects distributed across a set of nodes that can be operated in parallel. However, Spark also inherits the limitations of MapReduce. Example includes wide dependencies operation, e.g. *GroupByKey*, which involves data shuffle across the network. In addition, Spark removes data replication for fault-tolerance by using lineage to rebuild lost or corrupted RDDs. Moreover, hash function based data distribution for sparse graphs is not able to balance the workload across the cluster. Meanwhile, MEM-HDFS maintains the default fault-tolerance mechanism from HDFS and caches the working sets across the cluster nodes. However, both Spark and MEM-HDFS use MapReduce as the computing model, which is not best fit for iterative computations, machine learning and graph processing algorithms for example. In contrast to Spark and MEM-HDFS, Piccolo offers a data-centric programming model for writing parallel in-memory applications by sharing a distributed mutable state key-value table. Such design facilitates data sharing for iterative machine learning and graph processing

algorithms. All of these computing models, however, do not offer any data partitioning mechanism to achieve data locality and workload balance. Data shuffling is considered as the bottleneck for Spark and MEM-HDFS. Moreover, computation gets skewed for iterative computations, which means under utilizing of cluster resources.

Moreover, practitioners and researchers have also built a wide array of programming frameworks for graph processing, examples are Graphlab [52], Dryad [5] and Pregel [45]. Graphlab targets asynchronous, dynamic, graph-parallel computation in the shared-memory setting. Since the cluster shares a global memory, it is easy to write asynchronous programs in Graphlab. Pregel offers a Bulk Synchronous Parallel model [6] abstraction and does not make assumption about the memory setting. The communication is done via message passing interface(MPI). Dryad provides a distributed computation engine for coarse-grain data-parallel applications and the programmer has to design the structure of the parallel computation. Dryad is a much lower-level programming model than both Graphlab and Pregel. Two problems of these frameworks are: (i) fine control over data distribution and (ii) workload balance for sparse graphs.

All computing frameworks and models typically relies on the underlying storage systems to maintain and manage data storage and distribution. Typical distributed data storage systems include Flat Datacetner Storage (FDS) [62], Google File System(GFS) [18], Megastore [63], Hyperdex [64], and RAMCloud [65].

FDS presents the distributed data storage system as a remote flat storage model, in which all compute nodes can access all storage with equal throughput.

Unlike FDS, HDFS extends GFS and uses a master and slave architecture for managing distributed data splits with fault-tolerance by replicating data. HDFS exploits locality by using local disks to co-locate the computation with data. However, many important computations, e.g. sort, distributed join, and matrix operations, fundamentally need to move data around. FDS spreads data over disks uniformly and offers a relatively fine grain for data placement. Megastore provides a scalable storage system for interactive services, emphasizing both strong consistency guarantees and high availability. HyperDex offers a distributed searchable key-value store which enables queries on secondary attributes. RAMCloud is a DRAM-based storage system, similar to Spark, both of which recover data after crashes rather than storing replicas in DRAM. Unlike Spark which is also a computing model, RAMCloud simply offers a high available storage system.

In-memory cluster computation system comes with several challenges for programmability. The first is parallelism. It requires rewriting applications in a parallel fashion, with programming models that can capture a wide range of computations. Spark captures the parallel computations by enabling operations on top of RDDs. Such design significantly improves parallel coding efficiency. However, the optimal performance is not guaranteed due to the lack of data partitioning for data locality. The second is fault-tolerance. Since data cached in RAM are typically volatile, how to recover from hardware failure is important. Spark uses lineage to track the logical construction of RDDs. MEMHDFS relies on data replication in memory level. And RAMCloud also reconstructs lost data in parallel. Recovering data is appealing because it significantly improves writing performance but it generally takes a long

time. By replicating the data, it will guarantee the availability of data but also impact the writing performance and data consistency. Finally, it is hard to estimate the memory usage to maximize the performance. This challenge requires an integration of computation model and storage system. That means the computation model should be able to control storage system to optimize the performance. For example, while caching can dramatically improve read performance, unfortunately, data shuffling is still the bottleneck for iterative machine learning and graph processing algorithms within these frameworks. This is due to inefficient support from underlying data storage system for applications with a chain of multiple jobs. Consider, iterative machine learning algorithms, for example. The output from previous iteration serves as the input for next iteration. Fine control over how to save and distribute these intermediate results determines the network traffic.

All existing cluster computing and data storage systems have been designed to handle specific workloads. This work focuses on iterative machine learning and graph processing algorithms for sparse graph because of the popularity of social media analysis. Problems with existing cluster computing system for this task are as follows:

1. **Data partitioning** determines the system performance for iterative machine learning and graph processing algorithms. Current in-memory data storage systems, RAMCloud, Spark, Triple-H, and Alluxio e.g., fail to offer fine control over data distribution to co-locate computation with data. Data shuffling becomes the bottleneck for such workloads. Spark extends MapReduce to

construct RDDs that can be cached into RAM to reduce expensive disk I/O. But it also inherits the limitation of all-to-all communication to exchange RDDs. As a result, it is imperative to introduce a system, which enables fine control of data distribution, to maximize the cluster computing efficiency for sparse graph mining.

2. **Workload balance** plays an equally crucial role as data partitioning for sparse graph applications. The reason is due to the sparse nature of such workloads, in general, lead to computation skew for iterative algorithms. This problem is partially related to the initial data distribution as mentioned above. This problem is also credited to the parallel computing model, MapReduce for example.

3. **Memory management** is generally not well supported within existing in-memory computing system. The primary reason is that the underlying parallel computation model, (e.g. MapReduce), hides the programmer away from how data is saved in the distributed file system.

There are some existing systems trying to tackle the first two problems at the same time. For example, Graphulo [49] utilizes *Iterator* framework in Accumulo database to co-locate graph data with computation. However, Graphulo optimizes the data partition in HDD level rather than DRAM level. Without statistical analysis of the sparse graph, Graphulo fails to balance the workload. Pronto [60] extends R to a distributed system for machine learning and graph processing with sparse matrices. Pronto deploys a dynamic data partitioning to mitigate load imbalance

by tracking the execution time of each partition and task, which might cause extra burdens on the system for tracking the performance of each partition and then moving around the data.

In this dissertation, we present yInMem, a novel in-memory cluster computation system which targets on iterative machine learning and graph processing algorithms for sparse graph applications. The novelty results from the introduction of indexes to the data storage system. Such design gives the computation model fine access to the data storage system. As a result, yInMem is capable of maximizing the performance by reducing cross-node communication and achieving workload balance. The key advantages of yInMem over existing systems are:

1. yInMem offers fine control over data distribution to co-locate computation with data by saving matrix representation of graph data in NoSQL database. The database serves as a global accessible distributed data storage to which all worker nodes have uniform access. Meanwhile, the database also offers fault tolerance on HDD level.

2. yInMem provides a data partitioning algorithm for sparse matrices with the goal of achieving data locality. Based on the matrix operation choice, the user can partition the matrix into sub-blocks and cache these blocks into the in-memory file system. Additionally, the in-memory file system supports caching through HDD for fault-tolerance. Moreover, it is much easier to manage the memory usage across the cluster because the size of sub-blocks can be estimated prior to computation.

3. yInMem also balances the workload across the cluster using both the data partitioning algorithm and message passing interface (MPI). First, the data partitioning algorithm ensures that all workers will bear more or less the same workloads. Second, MPI orchestrates the workloads equally among all workers, maximizing the computing resource and avoiding data skew in the cluster.

4. yInMem minimizes the data shuffling between iterations and offers the maximum theoretical performance for iterative machine learning and graph algorithms with sparse graphs.

1.1 Thesis Statement

We propose yInMem, a parallel, distributed, indexed, in-memory cluster computing system which maximizes the performance of Sparse Matrix-Vector Multiplication for iterative algorithms such as spectral clustering for sparse graphs. This goal is achieved by ensuring data locality and workload balance with NoSQL database support for in-memory computing.

1.2 Contributions

- **A novel model for distributed in-memory parallel computation.**

  yInMem is a novel distributed computation system that offers a data splitting scheduler for iterative distribute in-memory parallel computation. yInMem distributes the input data, sparse graphs more specifically, across the distributed in-memory file system, achieving data locality and workload bal-

ance. Moreover, yInMem supports efficient data sharing mechanism, such as point-to-point, one-to-all, and all-to-one communications in a cluster. In addition, yInMem manages the memory efficiently to not only maximize the usage of cluster resources, but also speed up the numerical analysis in contrast to state-of-the-art systems. Consequently, yInMem optimizes the iterative machine learning and graph algorithms for sparse graphs by maximizing the usage of cluster resources. Chapter 3 explains the overall architecture of yInMem in details.

- **A novel model with data partitioning for data locality.**

  Due to the high latency of network, it is imperative to keep data close to the computing resource when it comes to distribute systems. However, sparse graphs, in general, are hard to be divided across the distribute storage system to reduce the data shuffling cost. What's worse, most sparse graph algorithms are iterative. As a result, the cost of shuffling the data becomes the bottleneck for existing distribute systems.

  yInMem saves sparse graphs in a NoSQL database which collects the statistical information of the input data. Using this information, a data partitioning algorithm can be optimized for improving the data locality. Therefore, the data shuffling cost is minimized for iterative machine learning and graph algorithms. Data partitioning is supported by using a triple store data structure called associative array. The data partitioning algorithm is further illustrated in Chapter 5.

- **A novel model balances the workloads in the cluster.**

  Balancing the workloads in the cluster computing systems and/or cloud computing systems is critical to optimize the resource usage, speed up the performance and thus make timely decisions. However, it is a challenging task for existing cluster computing systems which lacks the fine control over data distribution. To be more specific, the parallel computation engine lacks direct control of accessing the input data. Consider MapReduce for example. MapReduce is independent from HDFS in terms of distributing the data. As a result, workloads tend to be skewed as the computation progresses. By analyzing the linear algebra characteristics of iterative machine learning and graph algorithms, we have identified the Sparse Matrix-Vector Multiplication (SpMV) as the key component for balancing the workloads. yInMem offers APIs for parallelizing such operations, which balances the workloads in the cluster.

- **Memory management for in-memory computation.** Due to the limited capacity of memory, it is critical to manage the memory efficiently for optimizing the system performance. Sparse graph applications make it hard to estimate the memory usage during runtime. Without careful considerations for memory usage, it is easy to lead to out-of-memory error for in-memory computation. The reason for this problem is the lack of prior knowledge about the sparse graphs. One major problem with MapReduce based system is the varying sizes of intermediate results. When caching these intermediate

results in memory, the system should be able to distribute the results to avoid the out-of-memory errors. However, it is generally hard to estimate the size of intermediate results prior to computation. It becomes the programmer's responsibility to avoid such errors.

yInMem frees the programmer of such responsibilities by first analyzing the input sparse graphs and distributing the data in a way to balance the workloads and second reducing the inter-node communications. More importantly, the programmer can easily obtain the memory usage information prior to implementing the algorithms by collecting statistical information about the sparse graphs.

1.3 Dissertation outline

The dissertation consists of the following chapters as an approach to the contributions stated above.

- Chapter 2: Background and related work in the state of art for big data analytics platform.
- Chapter 3: yInMem architecture description.
- Chapter 4: Workloads characterization and evaluation model.
- Chapter 5: Data partition algorithm.
- Chapter 6: Evaluation.

- Chapter 7: Conclusion and future work.

Chapter 2: Background on cluster computation

In this chapter, we present a background on cluster computation and its architecture in order to provide context for the rest of this work. We review the distributed data storage system and parallel computing frameworks. Examples include HDFS, high performance distributed NoSQL databases such as HBase and Accumulo, and Hadoop based computing engine for performing massive scientific and iterative graph algorithms. In particular, we focus on Apache Spark and Alluxio. In addition, we present machine learning and graph processing algorithms used to evaluate and assess yInMem.

2.1 Cluster computation and architecture

A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource [19]. Fig. 2.1 shows the typical architecture of a cluster.

In general, a computer node can be a single or multiprocessor system with memory, I/O facilities, and an operating system. It is also a common practice to assume that each node is prone to either hardware or software failure. Also, the

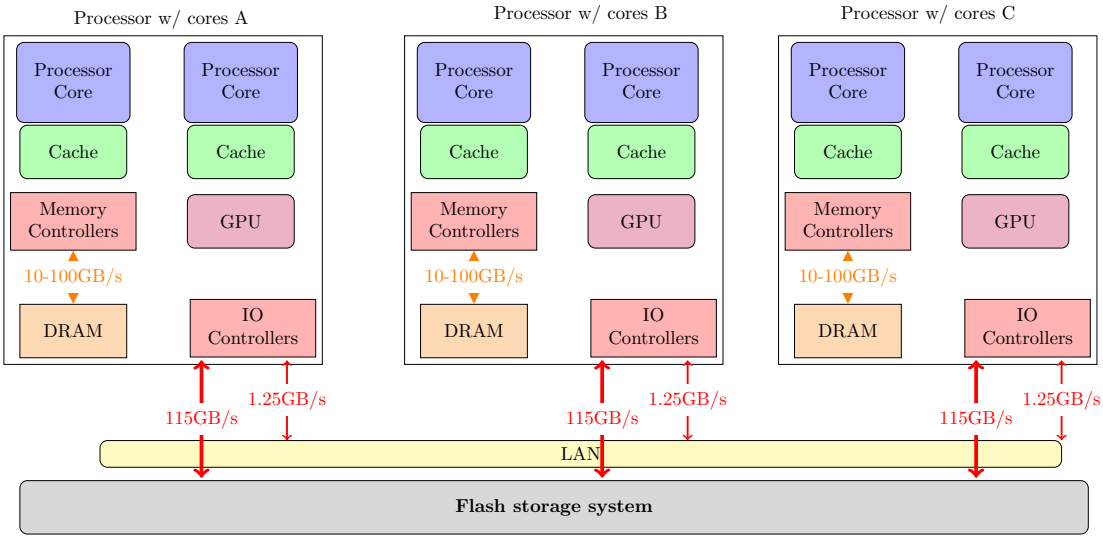


Figure 2.1: Cluster computer architecture

cluster is inter-connected via a LAN. Such a system offers a cost-effective way to gain features and benefits that have historically been found only on more expensive shared memory systems. As a result, the system performance is generally bounded by the bandwidth of hardware and network. Table 2.1 shods the bandwidth of HDD, SSD, network, and memory in a typical data center node.

Media	**Capacity**	**Bandwidth**
HDD (x12)	12-36 TB	0.2-2 GB/sec
SSD (x4)	1-4 TB	1-4 GB/sec
Network	N/A	1.25 GB/sec
Memory	128-512 GB	10-100 GB/sec

Table 2.1: Bandwidth comparison: HDD, SDD, Network and Memory

In the above table, memory bandwidth is one to three orders of magnitude higher than the aggregate disk bandwidth on a node. And this gap is becoming larger. The emergence of SSDs improves random access latency but still much

slower than memory. Since all nodes are connected by the network, reducing the data shuffling plays a major role in improving overall system performance. Moreover, with the hardware advancement of memory, in-memory computing system has emerged as the new standard for cluster computing.

Hadoop has become the most popular open-source framework to handle Big Data analytics, in which HDFS serves as the primary storage layer for Hadoop MapReduce model. However, current Hadoop design can not leverage high-performance networks because of the application performance being bounded by the disk access. Even though MapReduce has been highly successful in implementing large-scale batch jobs, it is a poor fit for low-latency iterative applications. The advantage of Hadoop is the underlying distributed file system which offers great scalability and fault-tolerance. Most existing cluster computing models are extensions of Hadoop, Spark for example.

2.2 HDFS

HDFS is structured similarly to a regular Unix file system except that data storage is distributed across several machines. HDFS is based on Google File System(GFS) [18]. It has in built mechanisms to handle machine outages, and is optimized for throughput rather than latency.

1. Datanode - where HDFS actually stores the data.

2. Namenode - the master machine which maintains the metadata.

3. Secondary Namenode , a backup service for namenode.

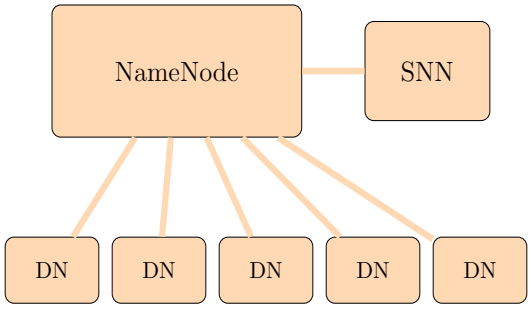


Figure 2.2: HDFS components: namenode, datanode and secondary namenode

Fig. 2.2 shows the HDFS architecture and components. HDFS follows a master-slave architecture in which the namenode is the master and datanode is the slave. When files are uploaded to the system, HDFS automatically partitions the file and saves the splits to the local disk of datanodes. Default partition mechanism is simply splitting the files sequentially to same sizes. Default data block size is 64MB to reduce the burden on the namenode.

Although HDFS significantly simplifies the data storage in a distributed manner, it is not the best mechanism to save sparse graphs. For example, when adjacent edges of one particular vertex are saved across the cluster, a single query of adjacent edges of this vertex will result in reading the whole file system.

This default data partitioning mechanism not only leads to unbalanced workloads but also incurs inter-node communication for sparse graphs.

2.3 MapReduce

In Hadoop, MapReduce is the parallel computing model. Many existing frameworks like Spark and MEM-HDFS are extensions of MapReduce. Map tasks perform

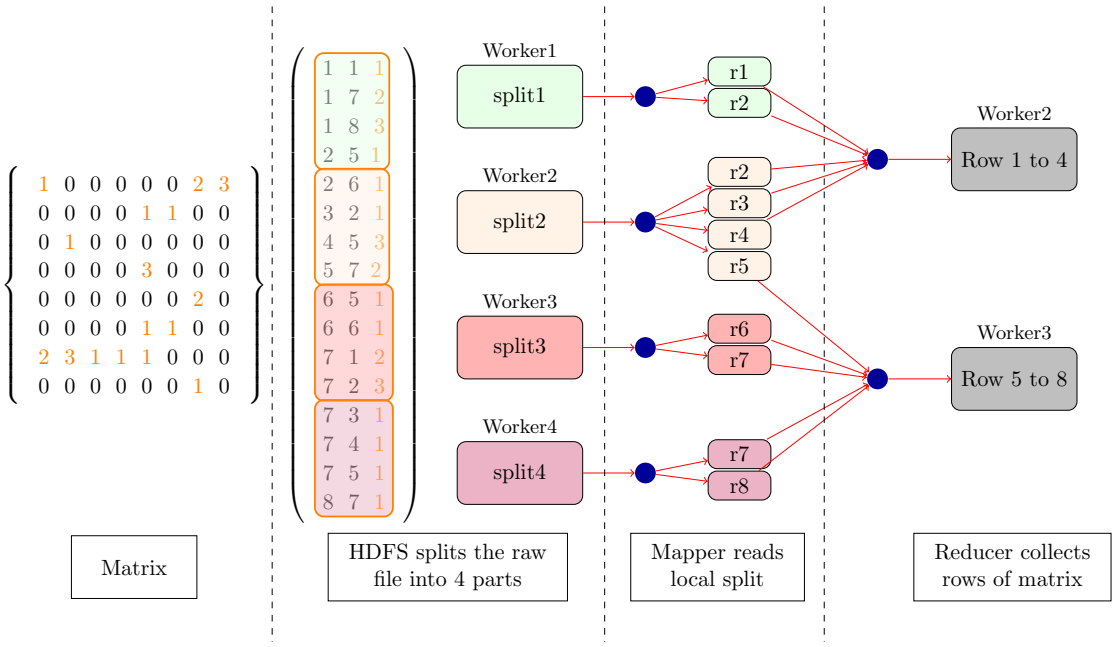


Figure 2.3: Work flow example of partitioning a sparse matrix in Hadoop using MapReduce

a transformation while Reduce tasks perform an aggregation. Between map and reduce there are 3 other stages: Partitioning, Sorting, and Grouping. The output to a map and reduce task is always a $(key, value)$ pair. The input to a reduce is $(key, ITERABLE[value])$. Reduce is called exactly once for each key output by the map phase. The $ITERABLE[value]$ is the set of all values output by the map phase for that key.

Advantages of MapReduce:

1. It is easy to write parallel program.

2. It is easy to scale the cluster thanks to the distributed file system.

3. Fault tolerance is supported due to data replication.

Fig. 2.3 shows a work flow example of how Hadoop partitions a sparse matrix using

MapReduce. For the sake of simplicity, we used a 8×8 sparse matrix. When the sparse matrix uploaded to HDFS, HDFS automatically splits the raw file into 4 parts, shown in different colors. Mappers will read their local split and aggregate rows into the same iterator. Afterwards, reducers collect these iterators.

Limitations of MapReduce:

1. Expensive disk I/O for intermediate results.

2. The shuffling of intermediate results is expensive.

3. It requires chains of MapReduce jobs for complex manipulations.

2.4 Distributed NoSQL databases

A NoSQL database environment is, simply put, a non-relational and largely distributed database system that enables rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types. There are four types of NoSQL databases, each with their own specific attributes: key-value store, column-store, document database, and graph database. Instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data. While this simple description sounds like the inverse of a standard database, wide-column stores offer very high performance and a highly scalable architecture. Examples include: HBase, Accumulo, BigTable and HyperTable.

2.4.1 Accumulo

Apache Accumulo is a scalable, distributed, NoSQL data store for Hadoop that provides accessible storage and fast read/write access for very large data sets. Apache Accumulo is based on Google BigTable [7], but it differs from other implementations by having cell-level security labels and a server-side programming mechanism that can greatly enhance the performance of read/write access and analytics.

2.4.1.1 Bigtable

Bigtable systems are a type of database that users row and column identifiers as general purpose keys for data lookup. They are sometimes referred as a data store rather than a database since they lack features commonly found in a traditional database. For example, they lack typed columns, secondary indexes, triggers, or query languages. Fig. 2.4 shows the key structure in Bigtable store which is similar

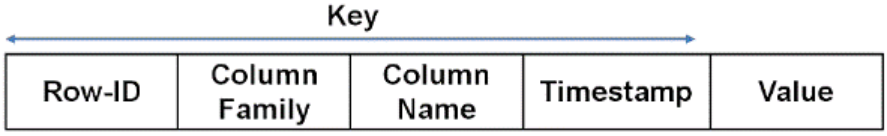


Figure 2.4: The key structure in Bigtable stores

to a spreadsheet with two additional attributes. In addition to the column names, a column family is used to group similar column names together. The addition of a timestamp in the key also allows each cell in a Bigtable store to store multiple versions of a value over time. To lookup an entry, users should provide a row id and

column name. Benefits of such store involve *higher scalability, higher availability, easy to add new data, saving space as empty cells are not store.*

2.4.2 D4M: Dynamic Distributed Dimensional Data Model

D4M is developed in MIT with the goal of bringing associative arrays into database engines. Associative arrays play a key role in offering two dimensional querying for sparse matrix. The limitations of most MapReduce based framework result from the missing part of indexing data elements. We argue that introducing associative arrays into cluster computation can significantly facilitate the data distribution for load balance and help achieve data locality for Sparse Matrix-Vector multiplication (SpMV).

Associative array saves data in a triple store *(Row, Column, Value)*. Users can query on both row and column dimension. Associative arrays also provide a mathematical interface, most importantly, supporting the linear algebra operations. yInMem extends D4M in a distributed environment and distributes data across the cluster prior to the computation with the goal of reducing data shuffling and achieving workload balance.

Fig 2.5 demonstrates the difference between HDFS and D4M for saving data in a distributed way. We use a sparse matrix as the input file. When this sparse matrix uploaded to HDFS, it will be divided into multiple splits and saved to different worker nodes following *(Row, Column, Value)* format. Meanwhile, D4M saves the sparse matrix file in associative arrays on which the programmer can query. For

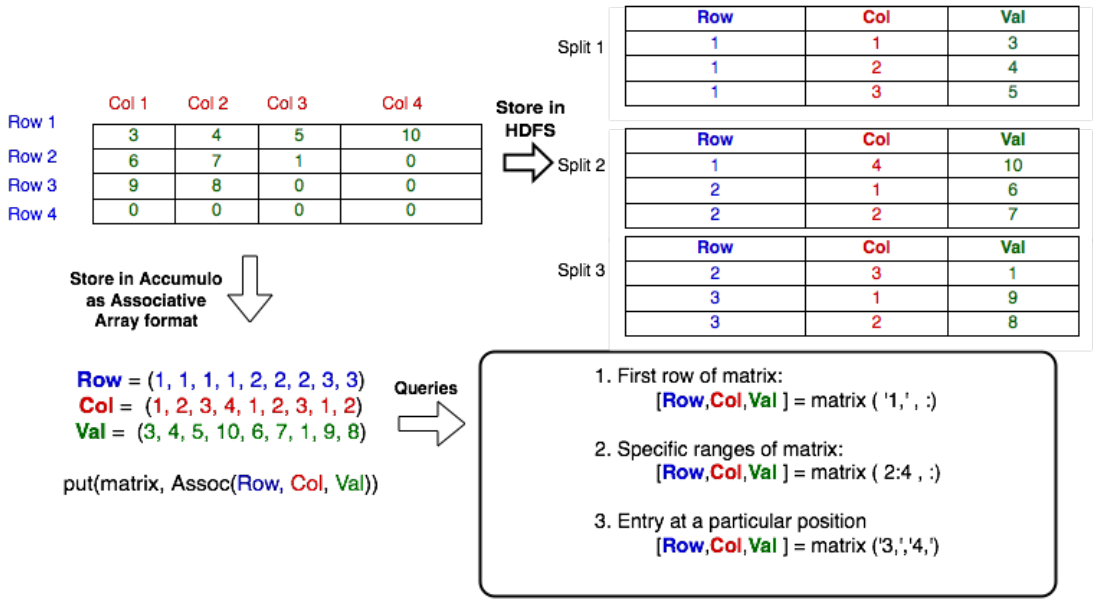


Figure 2.5: Data storage using HDFS and D4M

example, to get the matrix elements on row 2 to 3, the following query will suffice: **matrix(2:3,:)**. yInMem saves associative arrays in Accumulo database. MapReduce, on the other hand, will scan the whole data set and then return row 2 and row 3 elements.

2.5 Alluxio, Apache Spark and pMatlab

2.5.1 Alluxio

Alluxio, formerly known as Tachyon, is a distributed file system enabling reliable data sharing at memory speed across cluster computing frameworks. yInMem deploys Alluxio as the in-memory data storage system. Since D4M offers the facility to query required data elements, yInMem caches these results into the RAM of corresponding worker nodes. Alluxio maintains and manages this in-memory file

system. Using a separate in-memory file system like Alluxio simplifies the memory management for cluster computing systems.

Alluxio deploys a master-slave architecture (Fig. 2.6) similar to GFS. The master acts as the manager which maintains the metadata for the system. Slave nodes typically maintains their own metadata and can respond to client's read/write request.

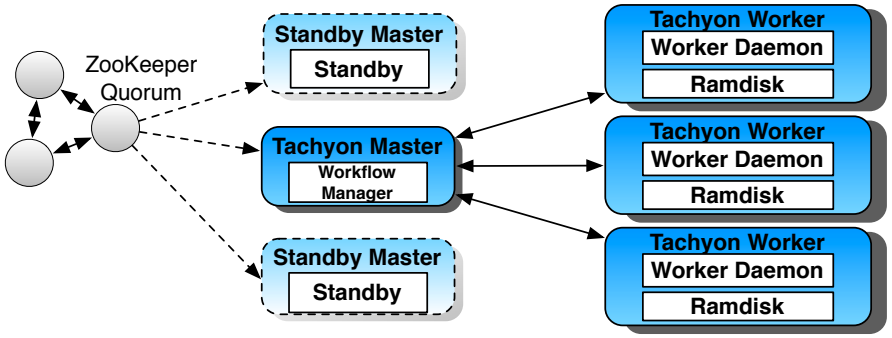


Figure 2.6: Tachyon Architecture

2.5.2 Apache Spark

Apache Spark is a distributed computation engine based on RDDs. RDDs are extensions of MapReduce but cached in the RAM of worker nodes. Thanks to the caching mechanism, Spark has significant performance improvement over Hadoop for certain applications.

However, it also inherits the limitations of MapReduce. For example, data shuffling is still considered as the bottleneck for iterative machine learning and graph algorithms such as *Spectral Clustering* and *PageRank*. New challenges with caching RDDs into the RAM of local workers are the memory usage. A typical

error is the **collect()** operation which aggregates results into the driver. This can potentially lead to the out of memory error without knowing the size of resulting RDD. In addition, point-to-point communication is not supported in Spark.

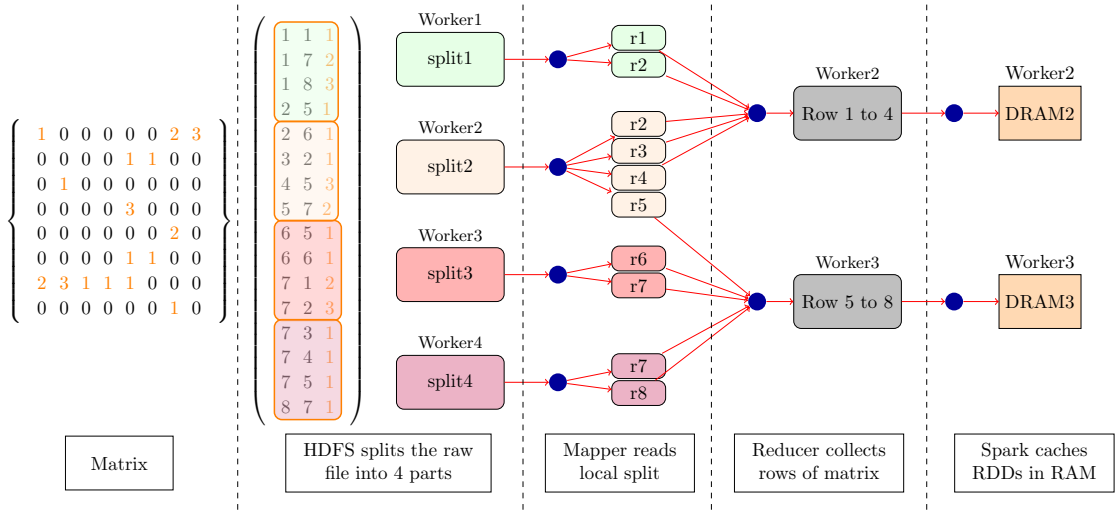


Figure 2.7: Work flow example of partitioning a sparse matrix in Spark extending MapReduce

Fig. 2.7 shows how Spark extends MapReduce to partition a sparse matrix into RDDs. Data skew typically happens when the reducer number is smaller than the mapper, 2 reducers v.s. 4 mappers in our example. Moreover, for iterative computations, the data exchange happens during the shuffling stage, which follows an all-to-all strategy. All-to-all data sharing can easily saturate the network bandwidth.

yInMem provides the memory usage estimation prior to computation by distributing the partitioned data across the cluster. Out-of-memory error is not encountered unless the partitioned input is too large to fit in the memory in the beginning. yInMem also enables point-to-point communication by using pMatlab as the parallel computation engine.

2.5.3 pMatlab

pMatlab is a parallel library of MPI for matlab based on MatlabMPI. yInMem utilizes pMatlab as the parallel computation engine. In cluster computing, pMatlab requires the specification of the total number of processes in the whole cluster with a fixed number of machines. pMatlab will sequentially assign equal number of proceses to each machine. pMatlab follows the Bulk Synchronous Parallel model in which the main process (PID=0) serves as the synchronization point. Each parallel operation will synchronize at the end by sending a signal to the leader.

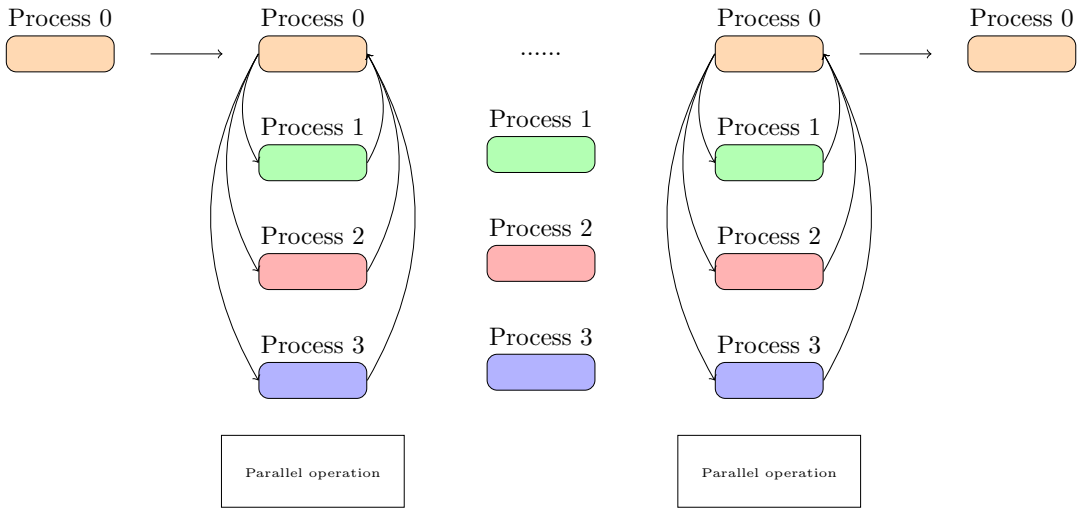


Figure 2.8: Parallel computation model in pMatlab

Fig. 2.8 illustrates the parallel computation model for pMatlab, which is an integration of OpenMP and MPI. Process 0 is the leader process which will spawn sub-processes on worker machines. Parallel operations are done and synchronized on all processes. Unlike MapReduce framework which has HDFS support, pMatlab is simply a parallel computation model without underlying distributed data storage

system. And the programmer should take care of data partitioning. To address the scheduling problem, we need get a global statistical information regarding input matrix and random access to the matrix. Accumulo degree table provides such information about the total number of entries for each column; and D4M enables us to access matrix elements by providing x and y coordinations. Most importantly, Accumulo tables serve as a shared file system to which working nodes in the cluster have access. A partitioning algorithm has been devised to ensure balanced workloads across the cluster. The partitioned data will be cached into Alluxio for iterative computations.

Advantages of pMatlab:

1. Programmers have fine control over which process resides in which machine. In our example, each process resides on different machines, say Process 1 is spawned on Machine 1. This significantly facilitates our sparse data partitioning since we have fine knowledge of which data partition resides on which process. This feature can help avoid data skew because all processes are participating in the iterative computation.

2. pMatlab enables point-to-point communication. Because each process is identified by one process ID, this enables the programmer to establish communication channels between any pairs.

3. Programmer can utilize existing linear algebra support from Matlab.

However, there are overheads when using pMatlab in a large cluster environment. The first is the time of spawning subprocesses at each worker. This is because the

leader takes charge of spawning and this often is determined first by the network speed and second by the number of subprocesses. The second is the synchronization cost, which is also proportional to the number of processes. But the synchronization is correlated to the algorithm choice, there is no need to synchronize if the algorithm has no dependency between iterations.

2.6 Iterative machine learning algorithms in this work

We strive to understand the characteristics of parallelizing iterative machine learning and graph processing algorithms. In this work, we briefly analyze three iterative machine learning algorithms to guide our investigation of how different applications perform on different cluster computation systems.

- **K-Means clustering**

  k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells, minimizing the within-cluster sum of squares (WCSS) (sum of distance functions of each point in the cluster to the K center). Data parallelism is a typical way to parallelize K-Means. All workers will broadcast local means to the driver to update the global means. K-means is a computation intensive algorithm with small number of data sharing.

- **PageRank**

  PageRank ranks the popularity of a vertex in a graph, and it was originally

used to sort web search results [37]. PageRank determines the popularity of a vertex v not only by the number of vertices that point to v, but also the popularity of the vertices that point to v. More formally, below formula shows the PageRank score (PR) for a vertex v (damping factor d (0.85)) :

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N-(v)} \frac{PR(u)}{|N+(u)|} \tag{2.1}$$

From linear algebra perspective, PageRank is equivalent to compute the dominant eigenvector of Google matrix using a power method in which the primary operation is Sparse Matrix-Vector multiplication(SpMV). Compare to K-means, PageRank has a larger number of data exchange for each iteration.

- **Spectral clustering**

  Spectral clustering makes use of the spectrum (eigenvalues) of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions. It first runs an eigenvalue decomposition on the similarity matrix and then applies k-means clustering to reduce the computation complexities for high dimension data. Spectral clustering has the advantage over K-means for noisy and high dimensional input. Efficient eigenvalue decomposition plays a critical role in determining the performance. Lanczos-SO algorithm extends power method to compute the top k eigenvalues and eigenvectors. Another advantage of using Lanczos-SO algorithm is because the primary operation only involves SpMV which can be parallelized easily.

Iterative algorithms normally works on large input datasets and will scan them every iteration until an object function converged.

Characteristics of iterative algorithms

1. Input data remains the same.
2. It takes several iterations to converge.
3. Data exchange should be minimized.

Since input data is not updated, it is beneficial to cache them to speed up. However, the partition of the input data is critical to the performance because intermediate results are shared. That's why algorithms involving Sparse Matrix-Vector multiplication(SpMV) should be prioritized over others. Because the output of SpMV is a vector which will minimize the data shuffling for each iteration. **PageRank** and **Spectral clustering** are two examples. **K-Means clustering** results in smaller data exchange compared to the other two.

2.7 Related work

Related works focus on both caching systems and cluster programming models. **Caching systems:** MEM-HDFS [38] performs intelligent caching and replication of HDFS data blocks in Memcached [39]. Memcached allows easy indexing of data-packets against corresponding block identifiers. However, unlike associative arrays which directly manipulate matrix elements, Memcached operates on data blocks, which provides no insights of data distribution according to parallel computation.

Due to the data replication, the write performance of MEM-HDFS is slower than Spark/yInMem. Triple-H [40] is another example of in-memory file system, which takes advantage of different storage devices (e.g. RAMDisk, SSD, HDD, Lustre, etc). Unlike Tachyon which eliminates data replication, HHH-M actually caches replication on memory level. CIEL [41] and FlumeJava [42] can likewise cache task results but do not provide in-memory caching or explicit control over which data is cached. In general, yInMem differs from these caching systems in that cached intermediate results are indexed for easy data sharing.

Cluster Programming models: MapReduce based models include: Twister [43] and HaLoop [44], which are iterative MapRedue runtimes. Pregrel [45] provides iterative graph applications. These models inherit the limitations of MapReduce for expensive data shuffling. Other in-memory computation system include Piccolo [46] which deploys a distributed hash table for read and update operations. Other similar models are distributed shared memory (DSM) [47] systems and key-value stores like RAMCloud [65]. These models lack a higher-level programming interface for in-memory data manipulation.

2.8 Summary

We have presented a background on cluster computation and its architecture, reviewed distributed data storage system and parallel computing frameworks to offer a context for the rest of the work. We have also discussed the characteristics of iterative machine learning algorithms to understand the challenges of parallelizing

them in current systems.

yInMem is so far the first computing system that introduces index into the distributed storage system and integrates closely with parallel computing engine to achieve data locality and load balance. The next chapter will provide a thorough description of yInMem architecture.

Chapter 3: yInMem architecture

3.1 Overview

This chapter describes the components of yInMem.

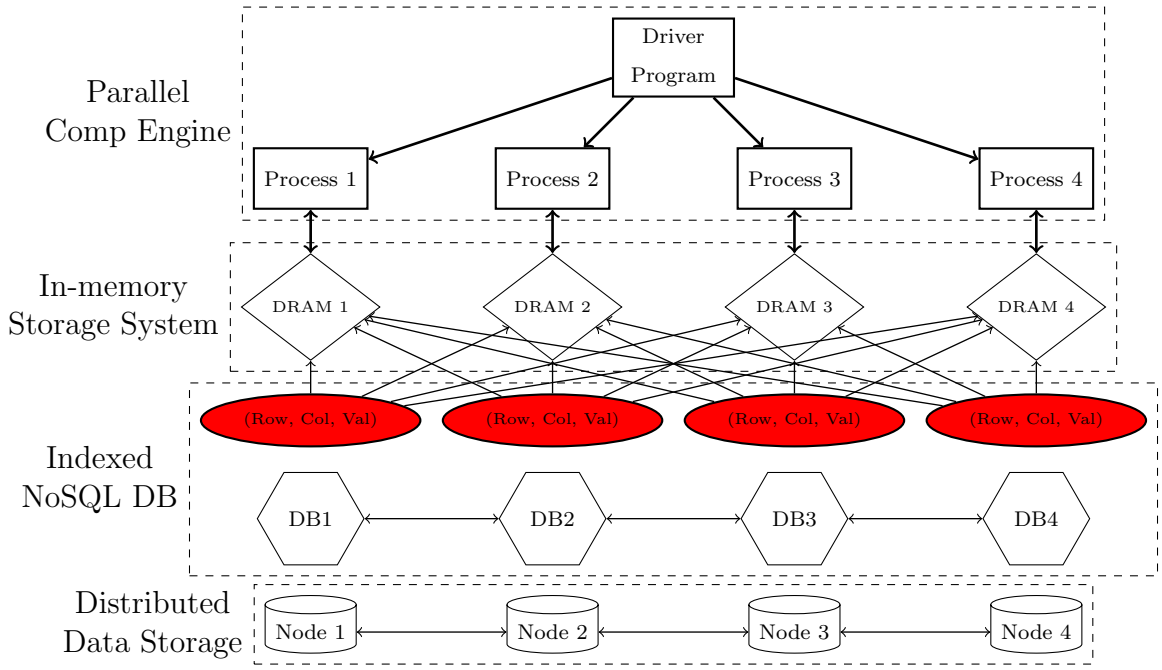


Figure 3.1: yInMem system architecture: the first layer segments and stores data in a distributed manner, the second layer indexes the data and answers queries from the user in a (RowIndex, ColumnIndex, Value) format, the third layer is the in-memory storage system for data accessing and data sharing, the fourth layer is parallel computation engine where the driver program spawns and manages the processes across the cluster.

Fig 3.1 describes yInMem architecture which consists of the following 4 layers:

(1) Distributed data storage system saves data in a distributed fashion (2) NoSQL

database serves as a shared file system for fast store and query data of interest (3) In-memory file system is used to cache partitioned data and intermediate results (4) Parallel computation engine offers parallel programming environment.

The integration of associative arrays enables yInMem to re-arrange the data saved in the first layer to achieve data locality and load balance. MapReduce typically reads file splits from local HDD. And how the input file is divided across the cluster is managed by HDFS. Programmer has no knowledge of which worker saves which segment of the input file. We define the computation data locality as the data partition is saved in the same location as the computing resource. MapReduce fails the computation data locality because it can't guarantee the data partition saved locally will be the input for the hosting worker. As a result, data shuffling becomes the bottleneck for operations that require moving data around. Iterative computation makes it worse. Although, there are some hash function based data partitioning approaches, in general, it is impossible to achieve workload balance for sparse graphs because the programmer has no statistical information about the sparse elements distribution in the graph.

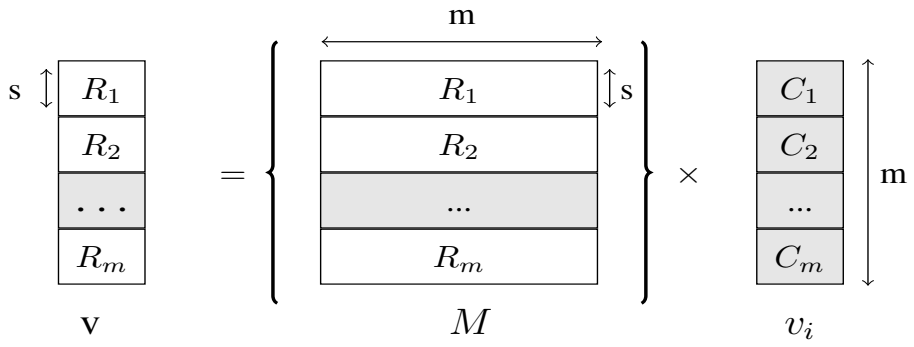


Figure 3.2: Sparse matrix-vector multiplication. The input of the multiplication are the whole row of matrix M and the whole vector v_i.

Consider, Sparse Matrix-Vector multiplication(SpMV) (Fig. 3.2), for example. The input of the multiplication happens between the whole row of matrix M and the whole vector v_i (the gray area). Row-based matrix decomposition is a common practice for parallelizing the computation. The reasons are first we are considering sparse matrix, which means worker node is likely to save multiple rows and the vector in memory, second it is easy to implement, and finally the output is a small vector, reducing the data shuffling. HDFS splits the input file based on the size. As a result, it is most unlikely to distribute the splits based on the matrix rows. In order to do a simple SpMV operation, multiple MapReduce tasks are assigned to reduce the elements from the same row in the same worker (computation data locality).

Spark constructs data from HDFS into RDDs which can be cached into DRAM to reduce disk I/O latency. Spark also provides a user-defined partition function *partitionBy* to RDDs so that data from the same row will be aggregated on the same worker. This type of consistent partitioning feature is one of the main optimizations in specialized frameworks like Pregel. However, there is one problem with this optimization, it is hard to estimate the work load balance using *partitionBy*. For example, one worker might aggregate most rows of matrix that would exceed the memory capacity. Even if the aggregated result can fit the memory, it is hard to ensure all workers have the same amount of work. The imbalanced workloads among the workers can significantly slow down the performance. yInMem provides a data partitioning algorithm to achieve data locality and workload balance to maximize the performance (Chapter 4).

Advantages of yInMem:

- Programmer can easily achieve computation data locality.
- Workload balance improves the computation time.
- yInMem offers point-to-point communication.

3.2 Cluster computer in Bluewave

This section describes the cluster computer configurations in Bluewave at our CHMPR lab. All experiments shown in following chapters are conducted in this cluster. Our experiments use a 32-nodes cluster connected with 10Gbps switch with the following configuration: each node has 8 processors each with Quad-Core AMD Opteron(tm) processor 2376, 25GB of RAM, L1 cache 512KB, L2 cache 2MB. Hadoop 2.2.0. Accumulo 1.5.2. Zookeeper 3.4.6. D4M, pMatlab, and Matlab 2010bSP2. CentOS 6.5 64-bit. Apache Spark 1.6.1.

One node is served as the namenode for HDFS while other thirty two nodes are datanodes. The same applies to Accumulo. The namenode also hosts the Accumulo master node while the rest serve as slave nodes.

3.3 NoSQL database

Non-traditional, relaxed consistency, triple store databases provide high performance on commodity computing hardware to I/O intensive applications. yInMem deploys Accumulo as the indexed NoSQL database using the Dynamic Distributed

Dimensional Data (D4M). The reasons are as follows: (1) Accumulo supports D4M (2) Accumulo is designed to handle unstructured data (3) Only non-empty entries in a table/matrix are stored.

The primary purpose of Accumulo is first to save and index input data in a triple store and second distribute the partitioned data to the corresponding worker node and then cached into Alluxio.

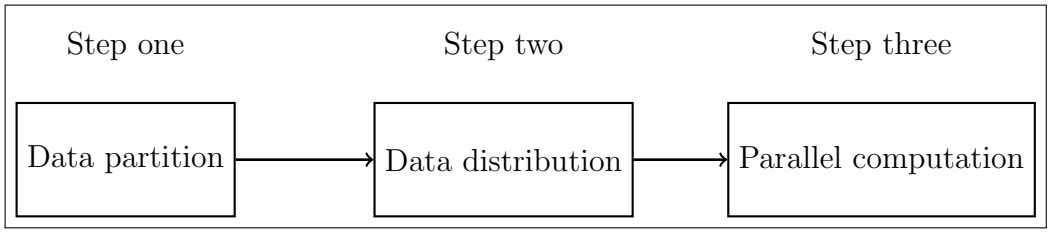


Figure 3.3: A typical data processing flow in yInMem

Fig. 3.3 shows a typical data processing flow in yInMem. The first step is to run a data partition algorithm (Chapter 4) which will partition the data across the cluster to achieve data locality and minimize the data exchange. Step two is loading data to the RAM of each worker according to the data partitioning algorithm. And this operation will only be called once for each experiment. Since most algorithms are iterative, the input data will be cached in Alluxio and so are intermediate results. In this section, we briefly discuss the write and read performance of Accumulo in our cluster computer environment. To benchmark the ingest rate of Accumulo in Bluewave, we use the following steps:

1. Start Accumulo on N_{server} servers.

2. Create the table and table splits.

3. Launch N_{ingest} processes on each node using pMatlab.

4. Generate Graph500 graph with 2^{scale} nodes

5. All N_{ingest} processes will insert above data to Accumulo table.

3.3.1 Graph500 Benchmark

The Graph500 benchmark has been designed to measure graph performance following the power-law. The number of vertices and edges in a graph are configured via a positive integer called the $SCALE$ parameter. Typically the number of vertices, N, and the number of edges M can be computed as follows:

$$N = 2^{SCALE} \qquad M = 8N \tag{3.1}$$

The Graph500 will generate N number of vertices in the graph, and M number of edges. This graph can represent a large $N * N$ sparse matrix M, where $M(i, j) = 1$ indicates an edge from vertex i to vertex j. In our experiment, we use $SCALE = 24$ to generate a graph with 16 million vertices. And the total number of entries will be $N_{server}N_{ingest}N$, which is around 1.9 billion.

Fig. 3.4 shows the accumulo ingest rate for the benchmark of generating 16 million vertices graph. It takes around 15 mins to generate 1.9 billion entries at an average rate of 7.8 million entries/s.

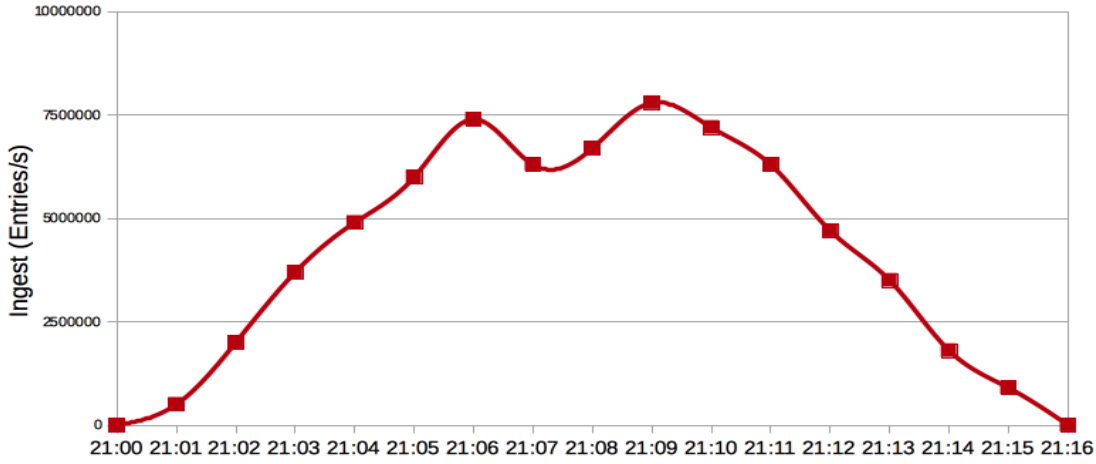


Figure 3.4: Accumulo ingest performance vs time. The benchmark runs around 15 mins with the peak ingest rate at 7.8 million entries/s using 16 servers and 4 processes per node.

3.4 Alluxio: in-memory file system

Alluxio provides an in-memory data sharing system where different frameworks can share intermediate results. The construction of this in-memory file system is very similar to GFS and HDFS. The difference is Alluxio is RAM based. Each worker maintains a local RAMdisk where data are mapped from local storage system(e.g. HDD, SSD). Alluxio uses two different storage types: Alluxio managed storage and under storage. Alluxio managed storage is the memory, SSD, and/or HDD allocated to Alluxio workers. Under storage is the storage resource managed by the underlying storage system, such as S3, Swift, HDFS or even HDD. yInMem deploys Alluxio in a local HDD mode which means the underlying storage system is the local HDD. The reason is our application can be easily embarrassing parallelized. In addition, yInMem achieves computation data locality by specifying which data partitioning

will be cached into which worker node. The first time to write to Alluxio happens in step two in Fig. 3.3.

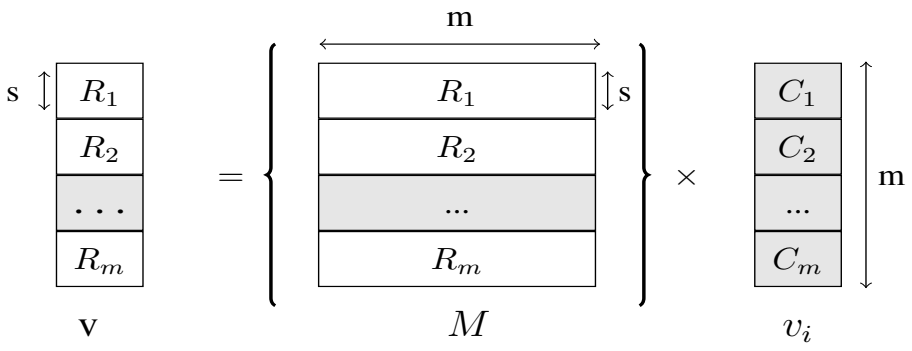


Figure 3.5: Sparse matrix-vector multiplication. The input of the multiplication are the whole row of matrix M and the whole vector v_i.

For example, consider SpMV operation where the whole row of the matrix should be multiplied with the whole vector. yInMem queries rows of matrix from Accumulo using D4M and caches these rows to the RAM of each worker. D4M returns three vectors: Row, Col, and Val, corresponding to the non-zero entries in the sparse matrix. After caching the input to each worker, the Matlab process will be reading from Alluxio and writing intermediate results to Alluxio.

3.4.1 Data sharing

Data sharing happens in iterative algorithms after each iteration.

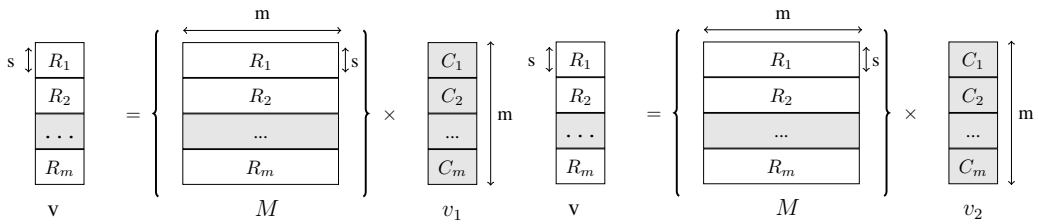
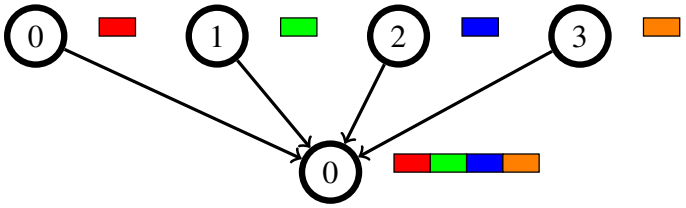
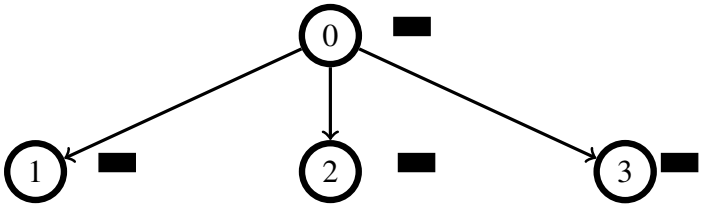


Figure 3.6: Two consecutive iterations of Sparse Matrix-Vector multiplication.

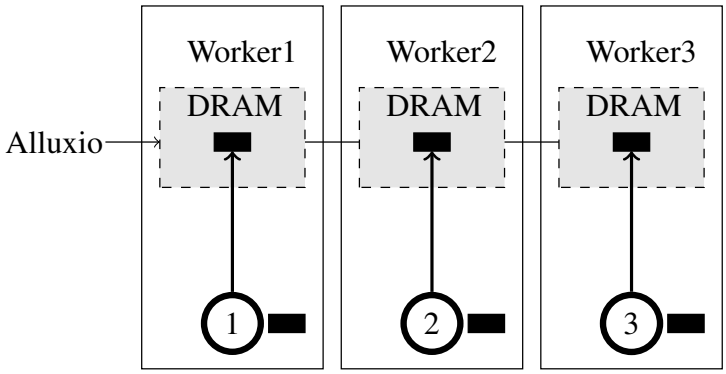
Fig. 3.6 shows two consecutive iterations of SpMV multiplication. The output of the left multiplication v serves as the input v_2 for the right multiplication. Notice that each worker will generate a partial result to produce the output v. As a result, the output from each worker should be first aggregated and then broadcast to all workers. yInMem offers an easy point-to-point communication and also enables aggregation and broadcast operations. Fig. 3.7 shows data aggregation and data broadcasting in yInMem. Fig. 3.7(a) explains how intermediate results from workers



(a) The leader process or Process 0 collects partial results from other processes



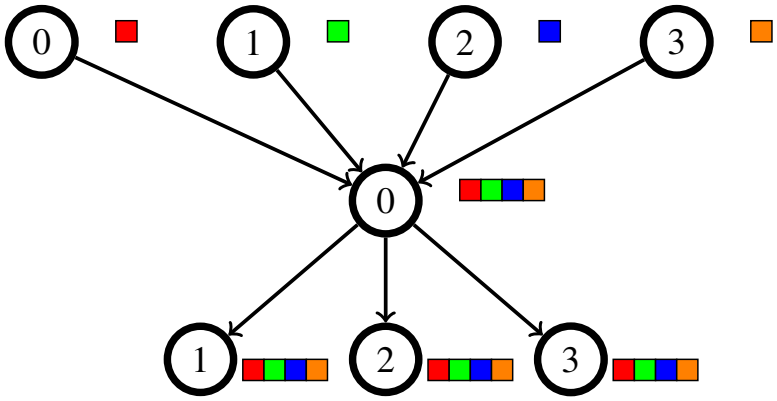
(b) The leader process or Process 0 broadcasts the collected results to the HDD/SSD of other workers via linux secure copy (scp)



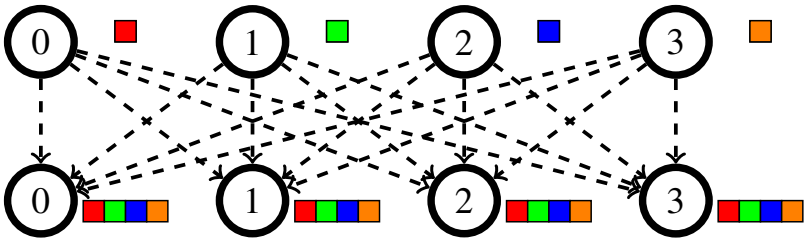
(c) Each worker process uploads the data from HDD/SSD to the DRAM. Alluxio can be viewed as a collection of DRAMs from workers

Figure 3.7: yInMem data aggregation and data broadcasting with Alluxio for SpMV.

are collected (T1) and then (b) broadcast to all workers (T2). Intermediate results are differentiated by process ID. The driver program collates these files and write a collated version to Alluxio located at the main memory of the driver node. In our experiment, we find out that Alluxio does not support parallel in-memory copy of the collated result, we end up using (b) linux scp command to copy to HDD and then (c) load to the DRAM (T3). Two potential improvement over Alluxio could be file appending and parallel reading.



(a) YinMem: data sharing by all-to-one and then one-to-all



(b) MapReduce: data sharing by all-to-all

Figure 3.8: yInMem VS MapReduce for data sharing

Comparison: Fig. 3.8 compares the data sharing mechanism between yInMem and MapReduce: (a) yInMem adapts an all-to-one and then one-to-all strategy; (b) MapReduce communicates via all-to-all. Advantages of yInMem over

MapReduce are as follows: 1. yInMem reduces the number of communication. The total number of communication channels is n (all-to-one) in the former while MapReduce n^2 (all-to-all). 2. yInMem re-assembles the result once. The cost is typically proportional to the size of the output. Spark, however, offers *aggregate* and *broadcast* function to share the results. The difference is yInMem can easily estimate the size of data to be shared because we distribute the data beforehand. We can manage the memory more efficiently.

3.5 pMatlab

pMatlab, an open source software that runs Matlab in parallel, serves as the parallel computation engine in yInMem. pMatlab gives the application/programmer precise control of its computations and communications. pMatlab also follows the master-slave architecture. The driver program or the master, can launch N_p instances of Matlab processes for the life of the program. Each process has a unique identifier P_{ID}, and can directly communicate with all the other instances. The communication is handled by message passing. This is the essential difference from MapReduce based parallel framework.

One potential limitation for pMatlab is the overhead of spawning sub-processes. This is particularly true for large clusters. And iterative algorithms make it even worse. Because pMatlab follows a bulk synchronous parallel (BSP) model, which means each parallel operation will synchronize at the end. More precisely, all sub-processes will terminate and send a complete signal to the master. For the next

iteration, new processes are launched again on the same worker. This can be easily optimized by maintaining the life time of a Matlab instance for synchronization.

The benefits of using pMatlab come from the precise control of its computations and communications. And an evaluation model (Chapter 4) can be devised thanks to the fine control over data distribution, data computation, and data sharing.

3.6 APIs

The core APIs of yInMem for implementing iterative machine learning algorithms can be categorized into the following three:

1. Database connector.

2. In-memory file system connector (Alluxio connector).

3. Core APIs.

Since yInMem uses pMatlab as the parallel computing engine, we have inherited existing libraries, database connector for example. During the course of our work, pMatlab has now included support for federated databases besides Accumulo [61]. Database connector is used to establish a connection from MATLAB to Accumulo database. And we are using the existing connector. In-memory file system connector mainly deals with reading/writing data into Alluxio. For iterative algorithms, after running data partitioning algorithm (Chapter 5), each process will read data partition from Accumulo using database connector. And these data partitions are

cached in Alluxio using in-memory file system connector. Core APIs are designed to support simple linear algebra operations, namely Sparse Matrix-Vector Multiplication(SpMV), Vector-Vector Multiplication(VV), Scalar-Vector Multiplication(SV) and normalization. Parallel computation is implemented by running core APIs, in which the input is read from Alluxio and output is also written to Alluxio.

3.6.1 Database connector

In order to establish a connection to Accumulo database, *DBserver* can be called to return a DB object that contains information about the specific database being connected to [61].

DB = DBserver(host, type, instanceName, user, pass)

Inputs:

host = database host name

type = type of database (Accumulo)

instanceName = database instance name

username = username in database

password = password associated with username

Outputs:

DB = database object with a binding to a specific DB

After obtaining the DB object, one can create a binding to a specific table in the database. Binding to the table offers query and insert functionalities.

```
A = T(rows, cols)

Inputs:

        T = database table

        rows = row keys to select

        cols = column keys to select

Outputs:        A = associative array of all non-empty row/columns
```

The following example describes how one can connect to Accumulo to obtain the total number of machines from table *NumOfMachines* in the instance named *myaccumulo*.

```
DB = DBserver('host','Accumulo','myaccumulo','root',passwd);

machines_t = DB('NumOfMachines');

NumOfMachines = str2num(Val(machines_t(:,:)));
```

3.6.2 Alluxio connector

Alluxio connector APIs provides the interface to read and write files in Alluxio file system. It mainly consists of the two following functions: *AlluxioWriteRead* and *javaMethod*.

AlluxioWriteRead establishes the connection to Alluxio The constructor of *AlluxioWriteRead* consists of the following 5 parameters:

1. mMasterLocation: the URI for Allxuio master.

2. mFilePath: the file name to be operated on.

3. mReadOptions: option to open the file, default CACHE.

4. CreateFileOptions: option to write the file, default CACHE_THROUGH.

5. Input: string to be written.

We have listed the constructor of AlluxioWriteRead.java in the following box.

```
public class AlluxioWriteRead {
        private final AlluxioURI mMasterLocation;
        private final AlluxioURI mFilePath;
        private final OpenFileOptions mReadOptions
        private final CreateFileOptions mWriteOptions
        private final String input;
   public AlluxioWriteRead(AlluxioURI masterLocation, AlluxioURI filePath,
ReadType readType, WriteType writeType, String input) {
mMasterLocation = masterLocation;
mFilePath = filePath;
mReadOptions = OpenFileOptions.defaults().setReadType(readType);
mWriteOptions = CreateFileOptions.defaults().setWriteType(writeType);
contentToBeWritten = input;} }
```

Example of calling *AlluxioWriteRead* in Matlab:

```
myConn = AlluxioWriteRead(['alluxio://n117:19998'||'inputFilePath']);
```

javaMethod is used to call Java *write()* or *read()* defined in *AlluxioWriteRead* for Matlab.

write() takes FileSystem and input string *s* as input. And it uses ByteBuffer to stream the input to the file system.

read() uses ByteBuffer to read FileInStream which is reading *mFilePath* with reading option (CACHE default).

```
public void write(FileSystem fs, String s) throws IOException, AlluxioExcep-
tion, FileAlreadyExistsException, InvalidPathException{
// Using bytebuffer to hold the string
    ByteBuffer buf = ByteBuffer.allocation(s.length());
    buf.order(ByteOrder.nativeOrder()); // Set the byteorder
// add input s to the buffer
    buf.put(s.getBytes(StandardCharsets.UTF_8));
// define the file name and write option in os
    FileOutStream os = fs.createFile(mFilePath, mWriteOptions);
    os.write(buf.array()); os.close(); }
```

```
public void read(FileSystem fs) throws IOException, AlluxioException, File-
DoesNotExistException{
    FileInStream is = fs.openFile(mFilePath, mReadOptions);
    ByteBuffer buf = ByteBuffer.allocate((int) is.remaining());
    is.read(buf.array());
    buf.order(ByteOrder.nativeOrder());
    String myString = new String(buf.array(),StandardCharsets.UTF_8);
    is.close(); }
```

Example of using javaMethod to read/write to Alluxio is listed as below.

Read: $myVal = javaMethod('read', myconn)$;

Write: $javaMethod('write', myconn, string)$;

3.6.3 Core APIs

Core APIs implemented in this work include: Sparse Matrix-Vector Multiplication(SpMV), Vector-Vector Multiplication(VV), Scalar-Vector Multiplication(SV) and normalization. We focus on SpMV operation for its complexity.

[v] = SpMV('Matrix', 'Vector')

Inputs:

Matrix: the matrix file name in Alluxio

Vector: the vector file name in Alluxio

Outputs:

v vector produced from SpMV.

myVector = *readVecor*(Vector); %% read vector

myMatrix = *readMatrix*(Matrix); %% read matrix

v = myMatrix * myVector; %% multiplication

SpMV: SpMV operation is composed of the following three parts: 1. Read data partitions (rows of matrices) from RAM in *(Row, Col, Val)* format (*Row* is a vector showing the row coordinates, *Col* is a vector of column coordinates and *Val* is the value vectors). 2. Read vector from RAM. 3. Multiplication. Recall that all input data have been saved in Alluxio, we use Alluxio connector to read input

and also write intermediate results to Alluxio.

There are 2 parameters for **SpMV**. *Matrix*: the matrix file name in Alluxio. Since Alluxio is a file system, each file is uniquely identified by a file name. *Matrix*: the vector file name in Alluxio. Two APIs calls used for SpMV are: *readVector* and *readMatrix*.

a. readVector: takes the input vector file name as input and it will return a sparse vector. Since we read row vector and value vector from Alluxio using Java, we re-construct the vector using Matlab *sparse()* function.

```
[vector] = readVector('Vector')
vRow = AlluxioWriteRead(['host' Vector '_r']);
vVal = AlluxioWriteRead(['host' Vector '_v']);
myRow = javaMethod('readFile',vRow);
myVal = javaMethod('readFile',vVal);
vr = char(myRow); vv = char(myVal); %% convert to char
vr = sscanf(vr, '%d'); vv = sscanf(vv,'%f'); %% convert to integer and float
vector = sparse(vr, 1, vv, NumOfNodes, 1); %% construct sparse vector
```

b. readMatrix: inputs the matrix file and it will return the matrix. Notice that we are partitioning the matrix based on row, when reconstructing the sparse matrix, we need reduce the row index from the beginning row number of the corresponding process (based on partition table (Chapter 4)). The last two parameters in *sparse* define the dimension of our sub-matrix, in which the x dimension is the row range based on the partition table and the y dimension equals to the size of the

matrix.

```
[matrix] = readMatrix('Matrix')

vRow = AlluxioWriteRead(['host' Matrix '_r']);

vCol = AlluxioWriteRead(['host' Matrix '_c']);

vVal = AlluxioWriteRead(['host' Vector '_v']);

myRow = javaMethod('readFile',vRow);

myCol = javaMethod('readFile',vCol);

myVal = javaMethod('readFile',vVal);

vr = char(myRow); vc = char(myCol); vv = char(myVal);

vr = sscanf(vr, '%d'); vc = sccanf(vc, '%d'); vv = sscanf(vv,'%f');

vector = sparse(vr-startR+1, vc, vv, endR -startR +1 ,NumOfNodes);
```

3.7 Related work

Most related works involve graph processing frameworks such as GraphX [50], Pregel [45], Gaffer [51], and Graphlab [52] optimized for graph operations on data in databases. We focus on Graphulo [49] given the similarity to yInMem.

Graphulo is a processing framework that enables GraphBLAS kernels in the Apache Accumulo database. Graphulo utilizes *Iterator* framework in Accumulo database to co-locate storage and computation. More precisely, Graphulo exploits the Key-Value Data Model to save edge information. That is, row (source vertex), column qualifier (destination vertex) and the Value. This is actually equivalent to the triple store in associative arrays. yInMem differs from Graphulo in that we cache

the graph data in Alluxio to achieve faster performance than Graphulo. However it is ideal to integrate graph load balance and pre-split the graph to Accumulo when the graph is ingested.

3.8 Summary

This section describes all the components of yInMem: Accumulo, the NoSQL database for underlying data storage, Alluxio, the in-memory file system for caching graph data, and pMatlab, the parallel computation engine which gives the programmer precise control over data computation and communication. We also list the advantages of yInMem over other computing frameworks, as well as the data sharing mechanism in yInMem.

We also present the three iterative machine learning and graph processing algorithms for sparse graph, namely K-Means clustering, PageRank, and Spectral Clustering. By identifying the characteristics of these algorithms, we understand the key points to optimize their parallelization in in-memory cluster computing systems.

In addition, the hardware and software environment has been presented for assessing yInMem. Moreover, we have also listed the core APIs of yInMem: 1. database connector. 2. Alluxio connector. 3. Core APIs. Programmers can easily write parallel iterative algorithms with the APIs.

Chapter 4: Workload characterization and evaluation model

In this chapter, we first characterize the workload, namely the sparse graph constructed from real life applications. Secondly, we illustrate the common ground of iterative machine learning and graph processing algorithms to optimize the performance in a cluster computing setting. Finally, we present the evaluation model to assess yInMem theoretically.

4.1 Workload characterization

To best understand iterative machine learning and graph algorithms in cluster computer, we analyze the performance of three algorithms: (1) K-Means clustering (2) PageRank and (3) Spectral clustering. The graph we choose to experiment on include synthetic graph generated by Graph500 benchmark and real-world graph, both of which are low-diameter and high-diameter graphs, and both mesh and social network topologies. We conquer the following two conventional challenges of parallelizing these algorithms efficiently:

1. Load imbalance

2. Synchronization overheads

To draw a solid conclusion, we not only consider a range of input graphs in any analysis but also different graph sizes and topologies. yInMem gives consistent best results on given experimental environment.

Graph	Description	Vertices(m)	Edges(m)	Degree	Directed
Kron	Synthetic	4k-16	0.2-24000	Varies	N
Friendster	social network	65	1800	27.7	N
Twitter	social network	17	476	28	Y
Road	USA road network	1.9	2	1.05	N
Orkut	social network	3	117	39	Y

Table 4.1: Graphs used for evaluation. All graphs are real-world data [53] except Kron which is generated by Graph500 benchmark.

In this work, we use the diverse set of graphs listed in Table 4.1 to guide our investigations. It is important to have a diverse set of graphs, since the topology of a graph can impact the characteristics of the workload. For example, the average number of edges for each vertex varies for all the datasets. Our primary focus is on social network since they are more challenging than meshes.

All the real-world data we used in this work can be found at [53]. We are grateful for these public available data because real-world social network data is often difficult to obtain due to anonymity concerns. Social networks typical indicate online communities by constructing the links between users, examples include Friendster, Twitter and Orkut. We also include the USA road network as it contrasts with the social networks, since Road has a high diameter, low average degree and low maximum degree.

Among all the graphs, Kron is the only synthetic one using Graph500 Benchmark. We generate the kron graph from the Kronecker generator [54]. Table 4.2

shows the detailed synthetic datasets used in this work. We generate large degree synthetic dataset to fill the memory capacity of our cluster.

Matrix size	Edges	Data file size	Degree
4,096	0.2 million	4MB	48.8
8,192	0.7 million	14MB	85.5
16,384	2 million	40MB	122
65,536	43 million	860MB	6,561
262,144	0.6 billion	1.2GB	22,888
524,288	1.4 billion	2.8GB	23,121
1,048,576	5 billion	10GB	26,122
16,777,216	24 billion	48GB	14,305

Table 4.2: Synthetic graph generated using Kronnecker generator

4.1.1 Matrix representation of graph

The graph abstraction is a way to model the connections between objects and the nature of these connections can have many properties. For example, a graph $G(V, E)$ consist of a set of vertices V and a set of edges E. Two vertices u and v are connected via an edge (u, v). A directed edge (u, v) represents a connection from u to v, while an undirected edge represents a bidirectional connection. If a graph is composed only by undirected edges, it is an undirected graph. Otherwise, it is called a directed graph. The degree of a vertex is the number of edges connected to it. The degree of a graph is the average degree of all of its vertices ($|E|/|V|$). To capture the diversity of vertex degrees within a graph, the degree distribution is the distribution of degrees over the vertices within a graph.

From linear algebra perspective, it is conventional to represent graph abstraction as an adjacency matrix. Using the linear algebra abstraction not only provides

great notational conciseness and expressibility, it can often allow for reusing optimized linear algebra libraries. This adjacency matrix is a symmetric one. All vertices are ordered both on the rows and columns. And the matrix entry value indicates the weight of their edge. In his work, we present all these graphs into matrix format and saved in Accumulo tables.

4.1.2 Graph topologies

The structure of the graphs is another important feature that should be taken into serious consideration for parallel computation. Why? The sparsity of a graph has a big impact on the load balance. A graphs sparsity is determined by its average degree. There is no formal mathematical way to distinguish a dense graph from a sparse graph. But in general, for a graph of n vertices, a dense graph has $O(n^2)$ edges while a sparse graph has $O(n)$ edges. We focus on the sparse graphs because first they appear much in real-world applications and second processing sparse graphs incurs greater communication inefficiencies.

Social networks typically have a low diameter, or small communities and a power-law degree distribution. More specifically, a large number of vertices in a graph are not neighbors of one another; but most vertices can be reached from every other within small hops. Since we are not analyzing graph traversal related algorithms, this only affects the load balance. Fig. 4.1 shows a degree distribution of a typical social network. Normally a certain number of vertices have more connections than other vertices. For example, the celebrities, the politicians, and

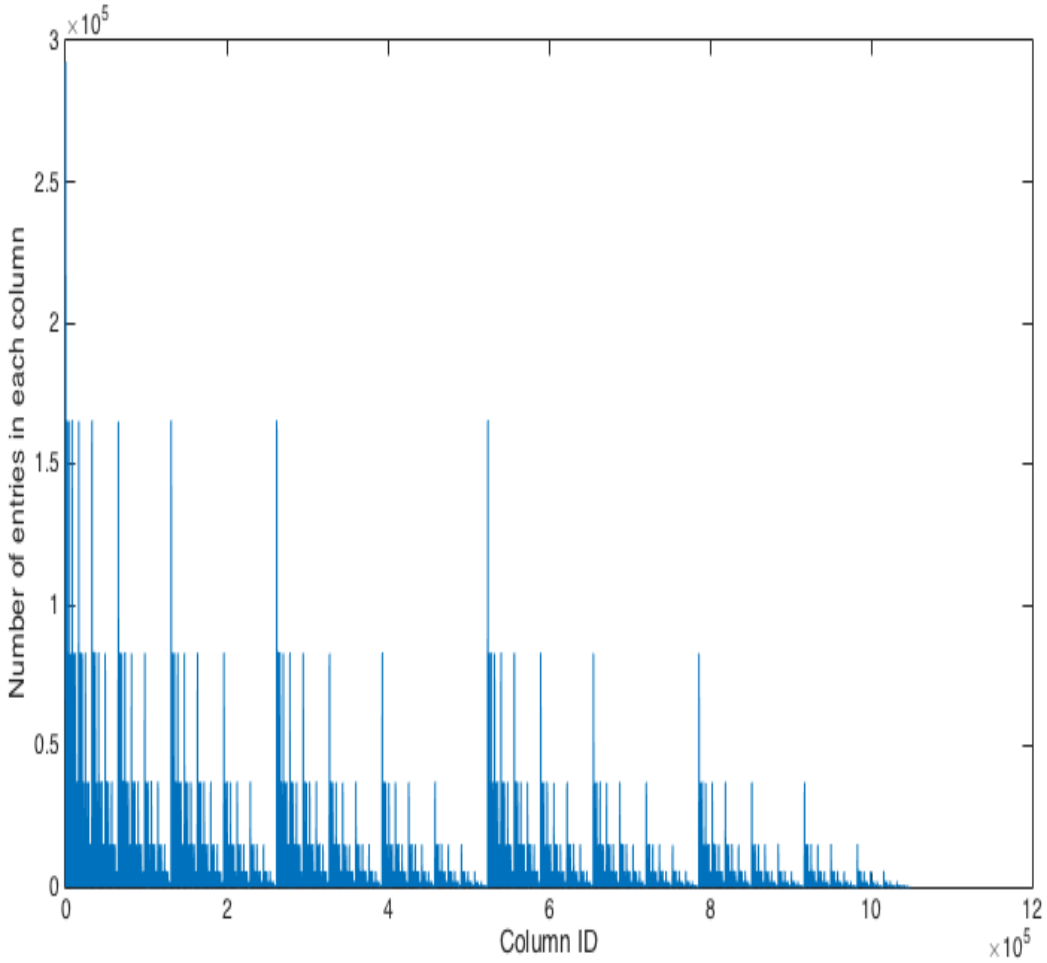


Figure 4.1: Example of the degree distribution of a typical social network.

news channels. Community discover has become a renewed interest for social networks [55]. The imbalanced graph input itself makes it hard for conventional graph processing frameworks to parallelize the computation. Moreover, the synchronization overheads are also directly connected with the data distribution.

4.2 Iterative algorithm characteristics

In this section, we strive to understand the characteristics of iterative algorithms and then try to optimize the performance in cluster computer. The reason we focus on iterative algorithms comes from the fact that it is conventionally hard to optimize the performance using existing parallel processing frameworks. The sparsity nature of the objects to be analyzed results in imbalanced input. And this is becoming worse as the data size grows bigger and bigger. Nowadays, the in-memory computation system significantly reduces the latency of disk I/O, but data shuffling is still considered the bottleneck for most of existing frameworks.

We have identified the roots for expensive data shuffling incurred in iterative algorithms are the following respects:

1. Initial data distribution. As shown in Fig. 4.1, the data originated from social network is not evenly distributed. Most distributed storage systems only provide a storing mechanism to save the data. In addition, distributed storage system tends to replicate data to handle fault or error from hardware. The co-location of computation and data is mostly lost. One solution to this problem is to design a data management system that can co-locate the computation and data.

2. Data sharing required by the algorithm. Most algorithms take an iterative way to update variables in order to achieve a goal for the goal function. And normally there is a dependency between each iterations. This dependency

should be minimized to reduce the communication. This is particularly true for applications with large input. Most parallel computing model follows a Bulk Synchronous Parallel pattern, which is a natural fit for iterative algorithms. The synchronous overheads are directly connected to the dependency between iterations.

K-Means clustering: K-Means clustering works to partition n objects into k clusters in which each object belongs to the cluster with shortest distance. However, there is no prior knowledge regarding the initialization of k clusters and should be computed from the data. The objective function with K-Means is to minimize the squared error function:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} ||{x_i}^j - c_j||^2 \tag{4.1}$$

J is the objective function, k is the number of clusters, n is the number of cases, ${x_i}^j$ is the node i, c_j is the centroid for cluster j. Algorithm 1 shows the serial K-Means

Algorithm 1 Serial K-Means clustering algorithm

Input:
D: training examples , k clusters and ϵ convergence rate, t=0
Randomly initialize k centroids: ${\mu_1}^t, {\mu_2}^t, ..., {\mu_k}^t$
1: **Repeat**
2: $t \leftarrow t + 1$
3: $C_j \leftarrow \emptyset$ for all $j = 1, ..., k$
4: **for** $x_j \in$ **D do**
5: $j^* \leftarrow argmin_i ||x_j - {\mu_i}^t||^2$ // assign x_j to closet centroid
6: $C_{j*} \leftarrow C_{j*} \cup \{x_j\}$
7: //Centroid update step
8: **for** $i = 1$ to k **do**
9: ${\mu_i}^t \leftarrow \frac{1}{||C_i||} \sum_{x_j \in C_i} X_j$
10: **until** $\sum_{i=1}^{k} ||{\mu_i}^t - {\mu_i}^{t-1^2}|| \leq \epsilon$

clustering algorithm. **Data parallelism** fits the parallelization of K-Means on the

assumption that data points are independent of each other. Our implementation of K-Means are as follows:

1. Partition N/P data points to each node.
2. Leader node randomly choose K points and assigns them as the cluster means and broadcast.
3. Each node finds membership for their local data point using the cluster mean.
4. Each node updates local means for each cluster.
5. Leader node collects these local means and broadcast the global mean.

Observations of our implementation:

1. Independent data points simplify the data distribution. We just split the data points equally among the cluster. This is much easier than matrix decomposition. Moreover, the data points remain unaltered.
2. Computation complexity is $O(n^2)$ for each worker node, computation intensive comparing to PageRank and Spectral clustering.
3. Communication happens at the end of each iteration for collection local means and then broadcasting the global mean.

PageRank: PageRank computes the score for each page (vertex) and then ranks all the pages according to this score. The core of PageRank algorithm involves computing the principal eigenvector of a Markov matrix representing the structured

graph. The simplest way to compute the left eigenvector is to apply the power method [56]. More formally, the definition of PageRank $\pi^T G = \pi^T$. Algorithm

Algorithm 2 Power method for PageRank

Input:
Matrix **G**, $k = -1$, pick $x^{(0)} > 0$, $||x^{(0)}||_1 = 1$
1: **Repeat**
2: $k = k + 1$
3: $[x^{(k+1)}]^T = [x^{(k)}]^T G$
4: **until** $||x^{(k+1)} - x^{(k)}|| \leq \epsilon$

2 shows the power method for computing the left eigenvector of matrix G. The difference of successive iterates in the stopping criterion is just the residual ϵ, which often lies between 10^{-8} and 10^{-4}. And the core operation in Algorithm 2 is SpMV as shown below. SpMV also serves as the essential part for Lanczos-SO algorithm in spectral clustering.

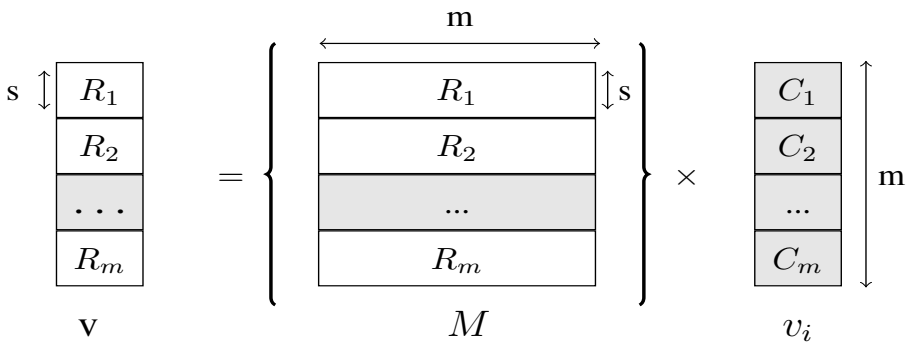


Figure 4.2: Sparse matrix-vector multiplication. The input of the multiplication are the whole row of matrix M and the whole vector v_i.

The simplest way to parallelize SpMV is to use row-based matrix to multiple the whole vector v_i. Since this is a sparse matrix, we assume that the RAM of a worker is able to save at least two vectors of size n (n is the size of the matrix).

Observations of our implementation:

1. Data distribution requires co-locating the computation with data since the

smallest unit is a vector. That means the data storage system should be able to provide a data partitioning interface to move vectors around based on the computation.

2. Above problem also leads to load balance. The goal of load balance is to ensure all workers will finish on the same time so synchronization won't de delayed by the most loaded worker.

3. Computation complexity is $O(n)$ for each iteration.

4. Communication happens at the end of each iteration to collect partial results and then broadcast to all workers.

Spectral clustering: Spectral clustering is an unsupervised clustering approach which not only tolerates noisy data but also produces better accuracy than typical clustering algorithms such as k-means. Spectral clustering performs a dimensionality reduction before running normal clustering in fewer dimensions, K-Means for example. And eigenvalue decomposition is the dimensionality reduction technique. Fig. 4.3 illustrates an example of using spectral clustering to partition a graph into 2 clusters. By representing the graph G into a spare matrix A, we can investigate the eigenvector space to identify clusters for the original graph. The key part of spectral clustering is to compute the top k eigenvalues and eigenvectors of adjacency matrix A.

We focus on Lanczos-SO algorithm for computing the top k eigenvalues and eigenvectors. The reasons why we choose Lanczos-SO algorithm is as follows:

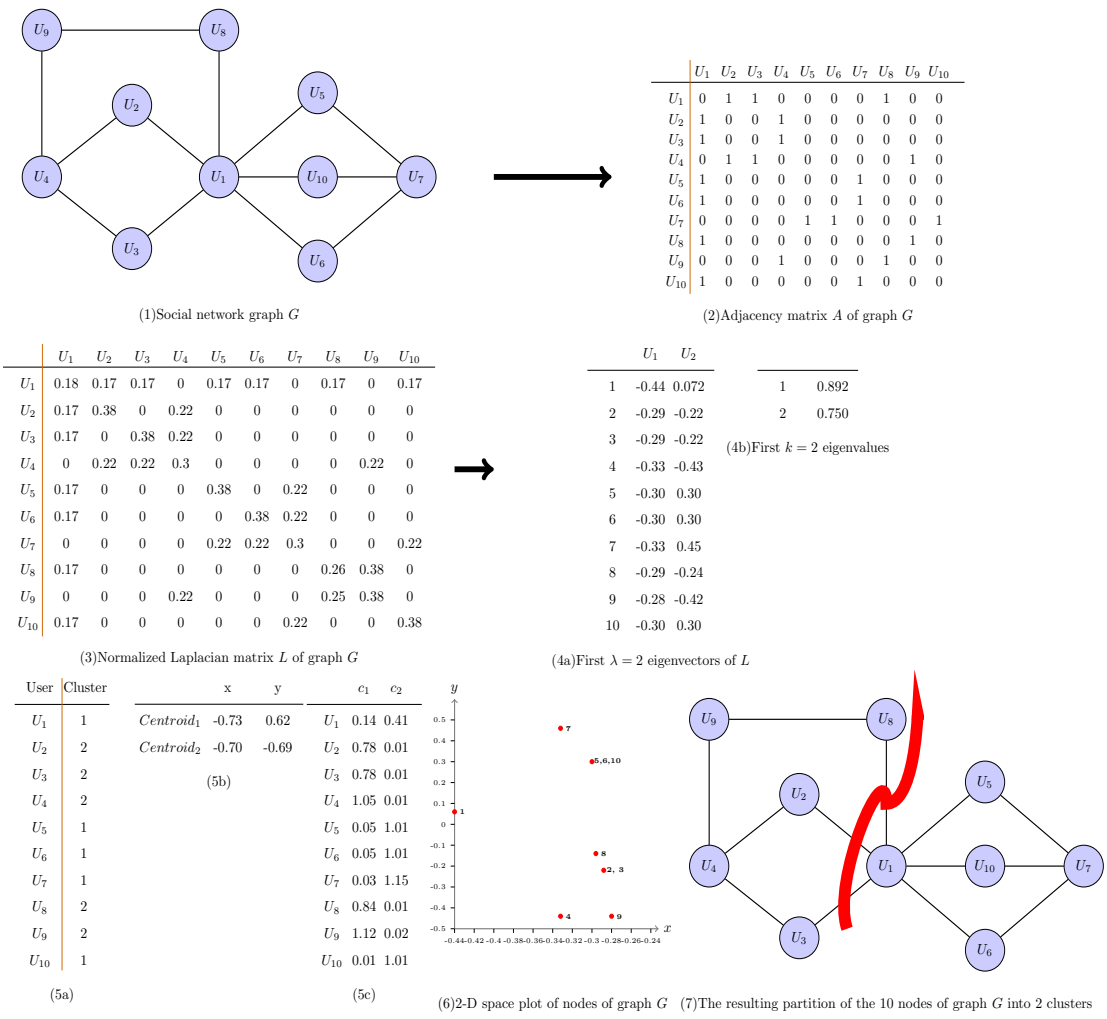


Figure 4.3: Example of using spectral clustering to partition a graph into 2 clusters.

1. Lanczos method generally calculates top k largest eigenvalues as compared to other algorithms such as power method.

2. Lanczos-SO filters spurious eigenvalues given the selective re-orthogonalizations. Algorithm like Lanczos-NO suffers from precision problems.

3. The most expensive operation, a sparse matrix-vector multiplication, is more cost-effective than matrix-matrix multiplication. This is particularly useful for sparse large graphs.

Algorithm 1. Lanczos -SO(Selective Orthogonalization)

Input: Matrix $A^{n\times n}$, random n-vector b, maximum number of steps m, error threshold ϵ
Output: Top k eigenvalues $\lambda[1..k]$, eigenvectors $Y^{n\times k}$

1: $\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$;
2: **for** $i = 1..m$ **do**
3: $v \leftarrow Av_i$; // Find a new basis vector
4: $\alpha_i \leftarrow v_i^T v$;
5: $v \leftarrow v - \beta_{i-1}v_{i-1} - \alpha_i v_i$; // Orthogonalize against two previous basis vectors
6: $\beta_i \leftarrow ||v||$;
7: $T_i \leftarrow$ (build tri-diagonal matrix from α and β);
8: $QDQ^T \leftarrow EIG(T_i)$; // Eigen decomposition of T_i
9: **for** $j = 1..i$ **do**
10: **if** $\beta_i|Q[i,j]| \leq \sqrt{\epsilon}||T_i||$ **then**
11: $r \leftarrow V_i Q[:,j]$;
12: $v \leftarrow v - (r^T v)r$; // Selectively orthogonalize
13: **end if**
14: **end for**
15: **if** (v was selectively orthogonalized) **then**
16: $\beta_i \leftarrow ||v||$; // Recompute normalization constant β_i
17: **end if**
18: **if** $\beta_i = 0$ **then**
19: break for loop;
20: **end if**
21: $v_{i+1} \leftarrow v/\beta_i$;
22: **end for**
23: $T \leftarrow$ (build tri-diagonal matrix from α and β);
24: $QDQ^T \leftarrow EIG(T)$; // Eigen decomposition of T
25: $\lambda[1..k] \leftarrow$ top k diagonal elements of D; // Compute eigenvalues
26: $Y \leftarrow V_m Q_k$; // Compute eigenvectors. Q_k is the columns of Q corresponding to λ

Figure 4.4: Lanczos-SO(selective orthogonalization) algorithm

Fig. 4.4 shows the Lanczso-SO algorithm. Essentially the most important steps are SpMV (line 3) and vector update (line 21). This is very similar to Algorithm 2, power method for PageRank. The remaining steps in Fig. 4.4 involve light operations like, vector-vector multiplication and scalar-vector multiplication, all of which can be easily parallelized. The only overhead of parallelizing these steps is the synchronization. pMatlab synchronizes all processes at the end of each parallel operation. And this algorithm actually help to evaluate the pMatlab overhead for iterative algorithms. Observations of our implementation:

1. Data distribution requires co-locating the computation with data partition. Moreover, following operations can re-use the same data partition.

2. Load balance is also very important to maximize the performance.

3. Computation complexity is $O(n)$ for each iteration.

4. Communication happens at the end of each iteration to collect partial results and then broadcast to all workers.

4.3 Evaluation model

To understand the behaviors of iterative algorithms in parallel computation system, we describe our evaluation model to assess the performance in a theoretical way. Major steps of doing numerical analysis involve:

1. Data collection and data preprocessing . This complexity of data collection is typically related with the applications. Data preprocessing aims to remove the noisy data to generate clean input data. We simplify this step by assuming data already collected and cleaned in the system.

2. Data distribution. Like discussed above, the initial data distribution plays a key role in improving the performance. Unlike yInMem, most existing parallel computation frameworks do not consider distribute data or partition data to co-locate the computation with data. For example, MapReduce utilizes the HDFS for data management, and HDFS lacks fine control over sparse matrix entries. As a result, most MapReduce based frameworks will generate a lot of

data traffic during computation stage. This is one key bottleneck within such frameworks. In Chapter 5, we will describe data partitioning algorithm with yInMem to achieve load balance and data locality.

3. Data shuffling. On one hand, data distribution has a big impact on the amount of exchanging data. On the other hand, the data collection and broadcast in iterative algorithms also lead to burdens on the network which has a limited bandwidth in the first place. Reducing both the data to be shuffled and also the communication channels are important factors to improve performance.

To simplify the evaluation, we assume that all data can be cached in the cluster memory. With the increasing bandwidth and decreasing price of RAM, this is a legit assumption. Moreover, the communication cost becomes dominant in in-memory computation since the expensive disk I/O no longer slows the computation. As a result, the evaluation model will focus much on the communication cost.

Framework	Data distribution	Load balance	P2P	In-memory
Hadoop/Spark	Coarse control	N	N	Non-shared
Graphulo	Fine control	N	Y	N/A
yInMem	Fine control	Y	Y	Both

Table 4.3: Comparison of different frameworks for support of data distribution, load balance, point-to-point(P2P) communication and in-memory

Table 4.3 compares different frameworks for support of data distribution, load balance, point-to-point(P2P) communication and in-memory. yInMem strives to bridge the gap between HPC and Hadoop community by utilizing the distributed in-memory data management system from Hadoop and pMatlab for parallel com-

putation engine. Therefore, yInMem has support for all listed attributes.

Hadoop/Spark: The coarse control of data distribution in Hadoop/Spark results from HDFS, which automatically divides the file into splits and save them across the work nodes. This mechanism is only coarse level because it, in general, fails to co-locate the computation and the data. Moreover, load balance is not supported because of this. Point-to-point communication is also missing in MapReduce based approach. Programmer can claim that writing complicated *mapper* and *reducer* can channel direct communication between different *mapper* and *reducer*. However, the programmer has no identification of each *mapper* or *reducer*. As a result, the network is saturated with *(Key, Value)* pairs during shuffling stage. Spark extends MapReduce to RDDs which can be cached in memory. However, it also inherits the limitation of MapReduce. Even though, programmer can aggregate matrix elements from the same row to the worker nodes. A global load balance scheduler is missing. Consider, the sparse graph which follows power-law distribution, each row has various non-zero elements. It is hard to guarantee equal size of RDDs across the cluster.

Graphulo: Graphulo is a specified graph processing framework with linear algebra support and is based on Accumulo database. Graphulo utilizes the $Key - Value$ Data Model to model sparse graph in the database. One advantage of this approach is natural integration with the data storage format with HDFS. As a result, Graphulo offers fine control of data splitting. yInMem exploits the usage of associative array on top of Accumulo. The idea is similar, but the benefits of using associative array is first it is easy to program. It takes only a query to

get desired matrix elements from the database. Second, pre-splitting in Graphulo normally involves careful considerations and human intervene. yInMem offers a data partitioning algorithm to automate this process. Both Graphulo and yInMem use pMatlab as the parallel computing engine, so P2P is supported. In addition, yInMem provides in-memory computation, which can be deployed as shared or non-shared mode. Non-shared mode in general outperforms shared mode because there is no communication cost. However, non-shared mode requires the applications can de decomposed into embarrassing parallel tasks. The three algorithms under investigation fall into this category.

Evaluation model: We focus on the running time of a typical data analytical process (Fig. 4.5), which is composed of first data collection, data preprocessing, data partitioning, and computation. The first two steps are applications dependent,

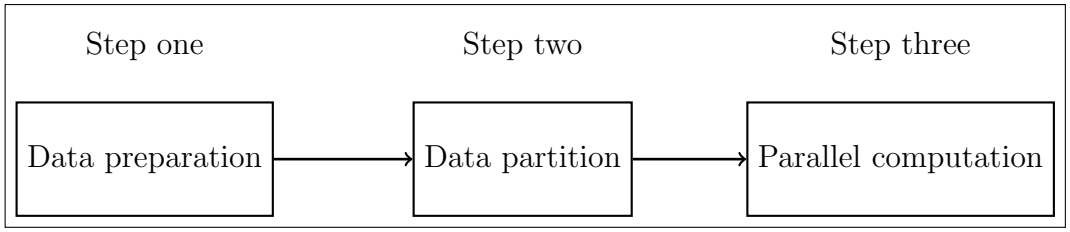


Figure 4.5: A typical data processing flow in yInMem

we simply use a constant value $T_{preparation}$ to represent. Data partitioning ($T_{partition}$) plays a key role in yInMem to achieve data locality and load balance. In iterative algorithms, it takes multiple iterations to converge. We use $T_{computation}$ to represent the time for computing. The following formula indicates the total running time of an iterative algorithm:

$$T = T_{preparation} + T_{partition} + T_{computation} \tag{4.2}$$

$$T_{computation} = \sum_{i=1}^{n} (t_i + t^i_{agg} + t^i_{broadcast} + t^i_{syncrhonization}) \tag{4.3}$$

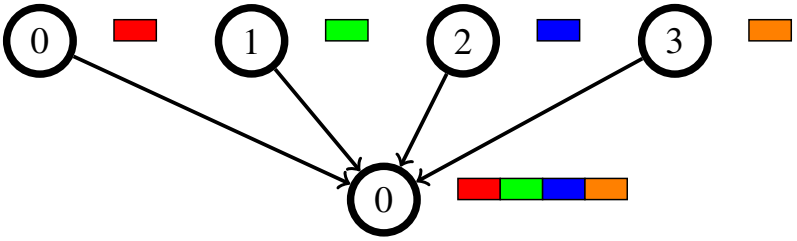
MapReduce model mixes $T_{partition}$ with $T_{computation}$. As a result, it is hard to accurately measure the time for each step. yInMem offers the partitioning algorithm and runs a parallel operation for each step, we argue that this can potentially achieve the theoretical maximum performance in cluster computers. Equation 4.3 shows the time for computation, in which n is the number of iterations, t_i the numerical operation time, t_{agg} the time to aggregate partial results, $t_{broadcast}$ the time to broadcast the variable, and $t_{syncrhonization}$ the time to synchronize all worker processes.

Data partition is important because t_i is proportional to the input size. Moreover, the $t_{syncrhonization}$ will always wait for the slowest process to finish and then proceed. Had the work loads not been evenly distributed, $t_{syncrhonization}$ will be equal to the sequential running time on the whole dataset (worst case). t_{agg} is the time for the leader process to aggregate partial results from workers. This time is proportional to the output size ($S_{v_{partial}}$) from this process and divide by the network speed ($S_{network}$). In SpMV, it will be a partial vector v. More formally:

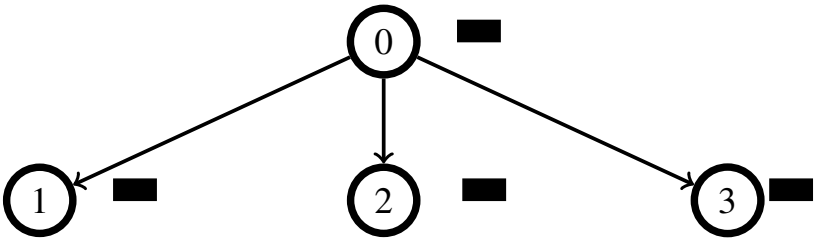
$$t_{agg} = S_{v_{partial}} / S_{network} \tag{4.4}$$

$t_{broadcast}$ is the time for the leader process to broadcast aggregated result to all worker processes. This time is also proportional to the size of the aggregated result ($S_{v_{agg}}$) and divide by the network speed (($S_{network}$)).

$$t_{broadcast} = S_{v_{agg}}/S_{network} \tag{4.5}$$



(a) The leader process or Process 0 collects partial results from other processes



(b) The leader process or Process 0 broadcasts the collected results to the HDD/SSD of other workers via linux secure copy (scp)

Figure 4.6: yInMem data aggregation and data broadcasting with Alluxio for SpMV.

Fig. 4.6 shows the operation of t_{agg} and $t_{broadcast}$. Typically, a good data partition should generate equal size of input at each process (colorful bars). The output or aggregated result is equal to the size of a vector. When compared with Hadoop/Spark, the advantage of yInMem is a fine control over data computation and data sharing. The limitation of this approach, however, is when the intermediate

results grow significantly large. The leader process might be over loaded. This case should be avoided by designing a good parallel algorithm, for example, pick algorithms with SpMV over matrix-matrix multiplication.

4.4 Summary

This chapter provides a complete analysis of workload characterization. More specifically, the characteristics of graph data have been discussed. In this work, we cover not only synthetic graph generated by Graph500 benchmark, but also real-world graph. The sparsity nature of these graphs have led to one of the biggest challenges for cluster computer, namely load imbalance. Exiting frameworks are not taking serious considerations to optimize the general performance, examples include Spark/Hadoop. yInMem is designed so that data balance can be achieved across the cluster to speed up the iterative algorithms.

This chapter also summarizes the characteristics of iterative machine learning or graph processing algorithms, including K-Means clustering, PageRank, and Spectral clustering. While picking the right algorithm is very important to gain speedup, a careful co-location of computation and data can also greatly impact the system performance. In addition, we present a theoretical analysis model for yInMem. In the next chapter, we will mainly discuss the data partition algorithm in yInMem for data locality and load balance.

Chapter 5: Data partitioning

Performance of cluster computing (e.g. MapReduce, DryadLINQ) heavily depends on how data is partitioned. Reducing imbalance is especially important for iterative algorithms as the overall execution time can be significantly high due to the skew among workers. As seen in Fig. 5.1, data partitioning reduces the completion time by around 22200 seconds (6.17 hours) when PageRank algorithm is run for 20 iterations with 16-million scale matrix.

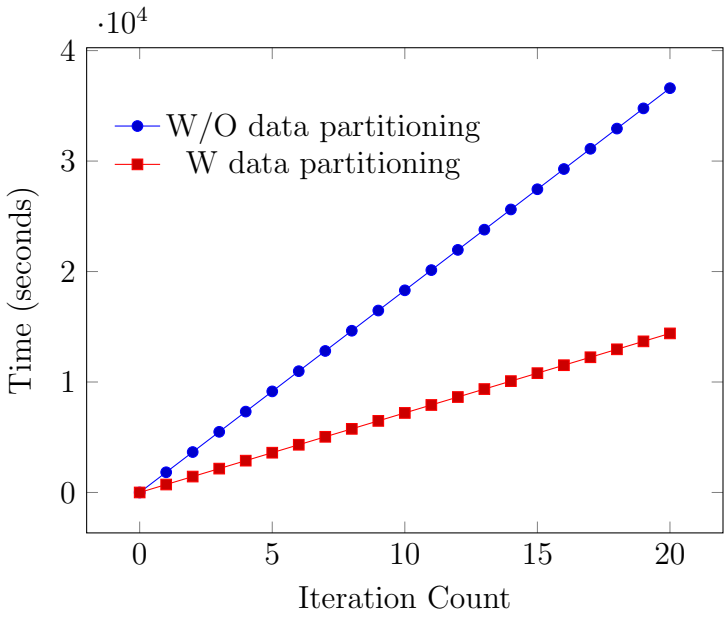


Figure 5.1: Comparison of overall execution time with and without data partitioning with PageRank. The lower the better.

The current state of the art systems have not yet to give an optimal solution

to partition the data to maximize the performance. By performance, we generally refer to various cost metrics including the number of processes required, CPU time, memory utilization, disk and network I/O.

Techniques proposed by the database community have shed some light on the optimal data partitions. yInMem is one example of utilizing the data management facilities from NoSQL database to find the optimal data partition for iterative machine learning algorithms.

5.1 Challenges with existing systems

Recent efforts in distributed computing frameworks have significantly simplified the development of distributed large-scale applications. Examples are MapReduce, Hadoop, and Dryad. In such systems, the parallelism is directly controlled by the data partition. The simplicity of MapReduce helps scaling the cluster horizontally. However, the performance of these systems are not competitive because the partitioning techniques are very primitive.

The simple hash and range partitioning are two most widely used methods to partition the datasets in these systems. There are many potential questions need addressing. For example, what partition function and how many partitions? Problems with existing systems are listed as follows:

1. It leads to unbalanced partitions in terms of data or computation using a hash function or a set of equally spaced range keys.

2. The number of partitions is hard to estimate for optimal performance. There is

a trade-off between the amount of computation per partition and the network traffic as discussed in Chapter 4.

3. For jobs with a chain of tasks, the data or computation skew is likely to occur in later tasks. One example is the number of reducers normally is smaller than the mappers. As a result, data distribution is skewed to these reducers.

4. Some real-time applications normally generate dynamic datasets (e.g., data streaming). As a result, old partitioning schemes might no longer be the best strategy.

yInMem tackles above challenges with the support from NoSQL database, which gives a statistical information about input data. Consider a real example from our previous work [55, 57] that discovers communities in Twitter during Hurricane Sandy. We first save tweets in associative arrays in Accumulo. Consider, for example, an associative array $Assoc('Tweet1','Status \mid 200')$ holding information about user's tweet Status (Table 5.1). Table 5.2 shows how to save tweets in D4M schema. Remember, non-zero cells are not saved and 1 means this cell exists.

TweetID	User	Status
Tweet1	Joe	200
Tweet2	Adam	200
Tweet3	Jane	301

Table 5.1: A simple example with information about three tweets

TedgeDeg in Table 5.3 is the degree table which sums up the total number of entries for each column. This information is extremely useful for partitioning the

TweetID	User| Joe	User| Adam	User| Jane	Status| 200	Status| 301
Tweet1	1			1	
Tweet2		1		1	
Tweet3			1		1

Table 5.2: Tweets expanded in D4M schema

data across the cluster, the reason being we obtain the global information about the sparse graph. In SpMV, $TedgeDeg$ represents how many non-zero entries in each row. We can rely on this table to understand the distribution of data and then balance the load by copying the load to each work node. Our scheduler relies on this table to obtain the sparseness information.

	User| Joe	User| Adam	User| Jane	Status| 200	Status| 301
Degree	1	1	1	2	1

Table 5.3: TedgeDeg: a degree table containing the total number of entries of each column

5.2 Data partition in yInMem

yInMem provides data partitioning by collecting statistical information of large sparse matrix from Accumulo. Below algorithm 3 is targeted to achieve load balance according to the computation resources of the cluster.

The input for algorithm 3 include: (1)N_p the total number of processes (2)n the matrix size (3)$startCol$, start column number, which begins with 1 (4)$TotalEn$ is the total non-zero entries in the matrix (5)$Load : TotalEn/(Np-1)$, load is the ideal average load per process (6) $myStep$ is a step range to speed up the computation and (7)$avgCol : floor(n/(Np-1))$, average columns per process. $avgCol$ serves

Algorithm 3 Data partitioning algorithm for achieving load balance for large sparse matrix

INPUT: 1. Np: total number of processes;
2. n: matrix size ;
3. $startCol$: 1: start column;
4. $TotalEn$: non-zero entries in the matrix;
5. $Load : TotalEn/(Np-1)$: avg load/process;
6. $myStep$: move steps;
7. $avgCol : floor(n/(Np-1))$: avg cols/process

OUTPUT: $PartitionTable$: (Np-2) number of *ticks* to indicate the last row id of matrix for the corresponding process id.

1: **for** $ticks = 1 : Np-2$ **do**
2: $endCol = [floor(startCol/avgCol) + 1] * avgCol$
3: $CurrentLoad = Sum(startCol : endCol)$
4: **if** $CurrentLoad > Load$ **then**
5: **while** $CurrentLoad > Load$ **do**
6: $CurrentLoad = CurrentLoad - Sum((endCol - myStep) : endCol)$
7: $endCol = endCol - myStep$
8: **else**
9: **while** $CurrentLoad < Load$ **do**
10: $CurrentLoad = CurrentLoad + Sum(endCol : (endCol + myStep))$
11: $endCol = endCol + myStep$
12: $startCol = endCol + 1$
13: Save endCol to current *ticks*

as the baseline of a naive partition approach, in which we simply divide the matrix into equal number of columns per process. This naive approach is most likely to lead to imbalance load because of the sparsity of the matrix. We adjust the load per process based on the naive approach, and update $CurrentLoad$ based on the non-zero entries in the updated range. A range is defined by the $startCol$ and $endCol$. $myStep$ is the adjusting rate to control how fast we update the range. By comparing $CurrentLoad$ and $Load$, we can update the $endCol$ of current range to make sure $CurrentLoad$ is more or less the same as $Load$.

Algorithm 3 achieves balanced workload in terms of co-location data and com-

putation. The number of partitions equal to the number of processes spawned in the cluster, maximizing the resource usage, e.g. CPU, memory utilization, minimizing the network I/O. Computation skew is not likely to happen in SpMV because all process will share almost the same amount of workload. At last, for real-time applications, we can easily run a monitoring process to watch the change of $TedgeDeg$ to update the partitions. However, in this work, we have not yet explored real-time applications.

Algorithm 3 demonstrates how to arrange rows of input matrix to all processes in worker nodes to achieve load balancing. The output of algorithm 3 is a partition table which shows how the sparse matrix is partitioned among all processes. All working processes will cache their input to Alluxio according to the partition table. The time complexity for Algorithm 3 is $O((n_p)$ where n_p is total number of processes.

Fig.5.2 lists an example of how yInMem generates a partition table according to Algorithm 3. *Load* in this example is $17/4 = 5$, 5 entries per process. Therefore, process 1 will work on row 1 while process 3 will work on both row 3 and row 4. Each process starts caching the corresponding rows of matrix to the RAM of hosting worker node by reading this partition table. Benefits of this partition algorithm are:

1. Data is co-located with computation. In SpMV, it is ideal to save partitions of rows of matrix into each worker process. For MapReduce, same partitioning purpose can be achieved by carefully choosing the *key* and *hash* function. However, yInMem offers a data partition algorithm prior to computation.

2. Workload balance is ensured by assigning same *Load* to each worker. MapRe-

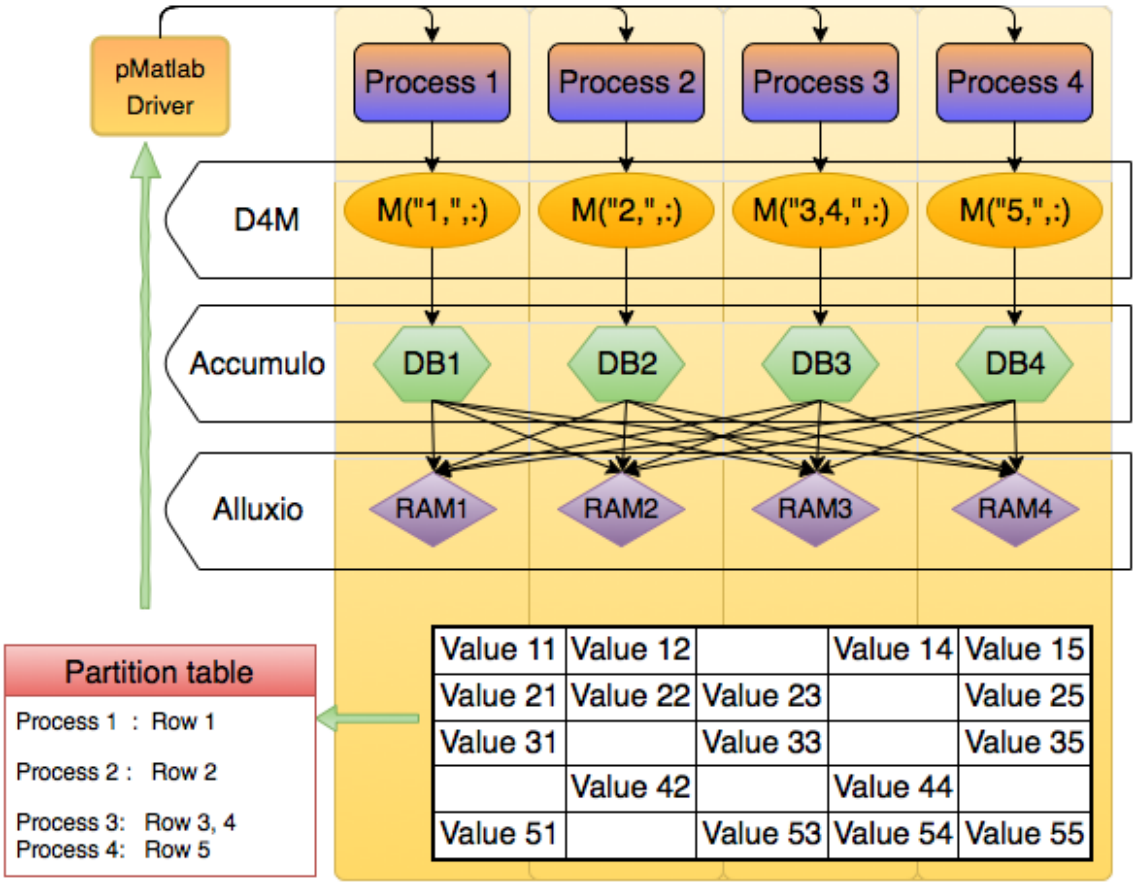


Figure 5.2: Example of data partition for a sparse matrix. Each process spawned by the driver program will cache their corresponding rows of matrix to the RAM of hosting worker node by reading the partition table generated from Algorithm 3.

duce based approach is generally hard to achieve this goal. To simplify the discussion, we assume that all workers have almost the same computing power and the computation time is proportional to its working load.

3. Maximization of computing resources. yInMem guarantees that each worker node will share the same workload in cluster computers.

5.2.1 Memory management

After generating the partitioning table, the next step is to query the database and cache these entries into Alluxio. Memory management problem arises naturally. For example, how much RAM should be assigned for caching? Since yInMem collects the global information about the input data, it becomes easy to estimate the memory footprint for each application.

The average load for each process can be estimated in the following equation:

$$Avg = \frac{Num\_Of\_Entries}{Num\_Of\_Processes} \tag{5.1}$$

where $Num\_Of\_Entries$ is the total number of entries in the matrix which can be obtained from Accumulo degree table and $Num\_Of\_Process$ the total number of processes. Moreover, average main memory footage (MM) can be evaluated by the following formula:

$$MM = (\frac{\alpha * S^2}{N} + S) \times 3B \tag{5.2}$$

Where α is the sparseness of the matrix, S is the size of the matrix, N means the number of machines, 3 Byte for an associative array. For matrix with size 262,144, the main memory requirement is around $240MB$ with sparseness 1% and 8 machines. Such fine control of data elements provide another opportunity for scheduling in-memory distributed computation, for example Spark [29] uses Resilient Distributed Datasets [23] for such computation on large clusters. One known problem for RDD is when the size of RDD exceeds the memory capacity, and it is hard for Spark to

estimate the size prior to computation since RDDs are constructed by transforming files in HDFS using operators like *map*, *filter* etc. Programmers typically have no idea how files are distributed in HDFS.

5.2.2 Evaluation

To evaluate the partition algorithm, we test Lanczos-SO 4 with our synthetic dataset matrix size 1 million without in-memory file system. Algorithm 4 shows the major code of Lanczos-SO algorithm for computing top k eigenvalues and eigenvectors of a large sparse matrix.

Algorithm 4 Major part of Lanczos-SO

Input:
Matrix: $A^{n \times n}$, random $n - vector$ b, iteration steps m, error threshold ϵ
$\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$

```
1: for i ← 1, n do
2:     v ← A × v_i
3:     α_i ← v_i^T v
4:     v ← v − β_{i−1} v_{i−1} − α_i v_i
5:     β_i ← ||v||
6:     v_{i+1} ← v/β_i
```

MV:	matrix-vector multiplication	Line 2
Alpha:	dot product of two vectors to compute α	Line 3
OrtV:	orthogonalize against two previous basis vectors	Line 4
Beta:	normalization of a vector to compute β	Line 5
UpdateV:	Update vector v_{i+1}	Line 6

Table 5.4: Operation definition in Lanczos-SO algorithm

Table 5.4 shows the definition of each operation in Lanczos-SO algorithm. SpMV dominates the computation. Other operations mainly involve vector-vector

multiplication, scalar-vector multiplication and normalization of a vector, which are computationally easier than SpMV (Fig. 5.3).

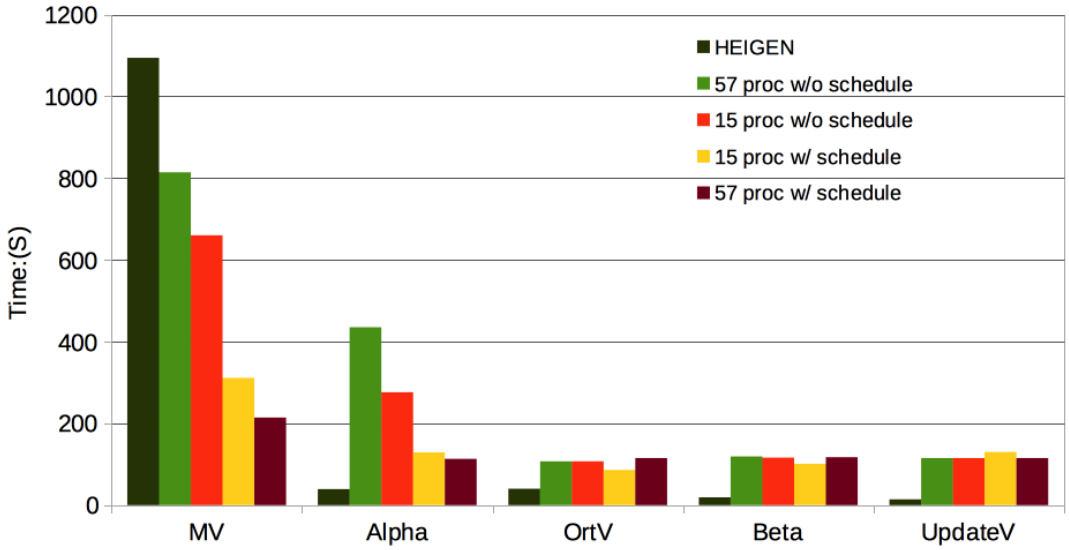


Figure 5.3: Average running time for different Lanczos-SO operations on HEIGEN and proposed architecture for matrix with size of one million

Fig. 5.3 lists average running time of operations in Table 5.4. On average, **MV** is the most expensive operation among all of them. We compared 5 sets of experiments, HEIGEN is a MapReduce based approach, while the rest 4 sets are conducted with yInMem, in which 2 are tested without data partitioning and the other 2 are tested with data partitioning. Because this chapter focuses on how data partitioning impact the performance, we focus on the most expensive operation **MV**. In pMatlab, we implement SpMV with the following steps in Table 5.5.

The average running time of SpMV without caching without data partition is listed in Fig. 5.4. The matrix entries distribution is listed in Fig. 5.5 for matrix with size of 1 million. The average running time of SpMV with data partition is

RV:	Reading Vector
RM:	Reading Matrix
PP:	Post Processing: convert string into matrix
MUL:	Matrix Vector Multiplication
WB:	Writing result Back

Table 5.5: Operation definition for MV implementation

listed in Fig. 5.6.

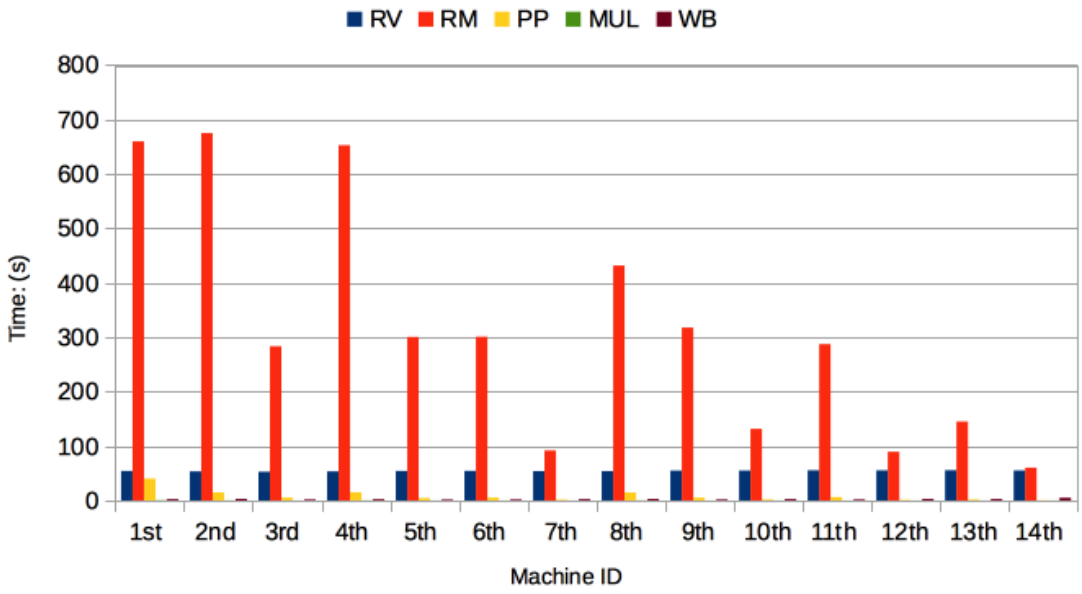


Figure 5.4: Average running time of operations in MV by distributing columns equally into 14 machines for matrix with size of one million.

Naive Data Partition: Fig. 5.4 shows the average running time of SpMV using a naive data partitioning algorithm. A naive data partitioning algorithm simply divides the matrix into chunks with equal number of columns. And each worker work on the same number of ranges of chunks. Due to the skewed nature of the input data, naive data partitioning also results in skewed data distribution.

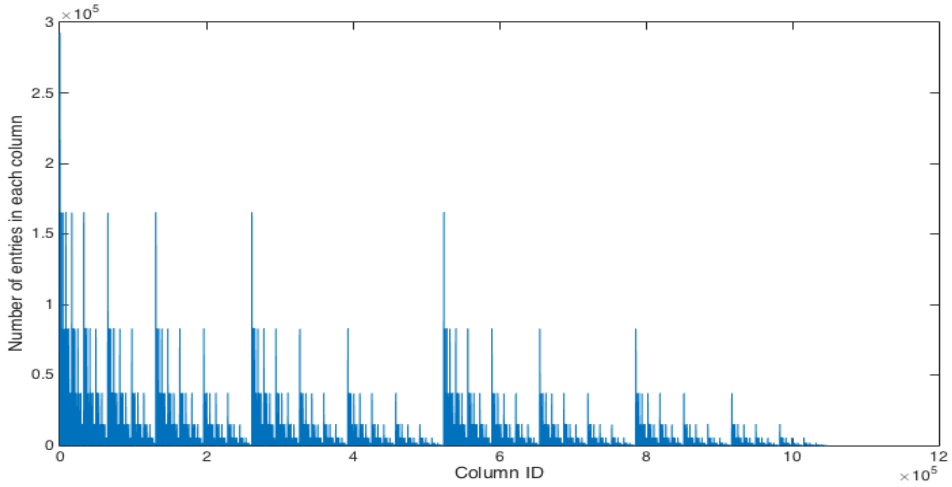


Figure 5.5: Statistical information obtained from Accumulo table for non-zero entries distribution for matrix with size 1048576*1048576

In Fig. 5.4, x-axis is the machine ID while y-axis is the running time. Among all of the operations, **RM** (reading matrix) is the most expensive operation, which is proportional to the input size. Machine 2 has the highest cost of reading matrix, the reason being input data in this machine has the highest density (evidenced in Fig. 5.5). Fig. 5.5 shows the statistical information about matrix with size of one million: x-axis shows the column number and y-axis the total number of non-zero entries in each column. Other operations are much faster than **RM**. For iterative algorithms, the leader process will synchronize the parallel computations in the cluster, which means it will always wait for the slowest machine to finish (Machine 2 in this case).

In addition, the bar heights distribution in both Fig. 5.4 and Fig. 5.5 is similar due to naive partitioning mechanism.

yInMem Data Partition: Fig. 5.6 shows the average running time of SpMV with data partitioning on the same matrix. Fig. 5.6 shows 3X faster for RM because

we have scheduled almost the same work load across the working processes. In addition, other operations like **PP**, **MUL**, and **WB** also cost more or less the same amount of time.

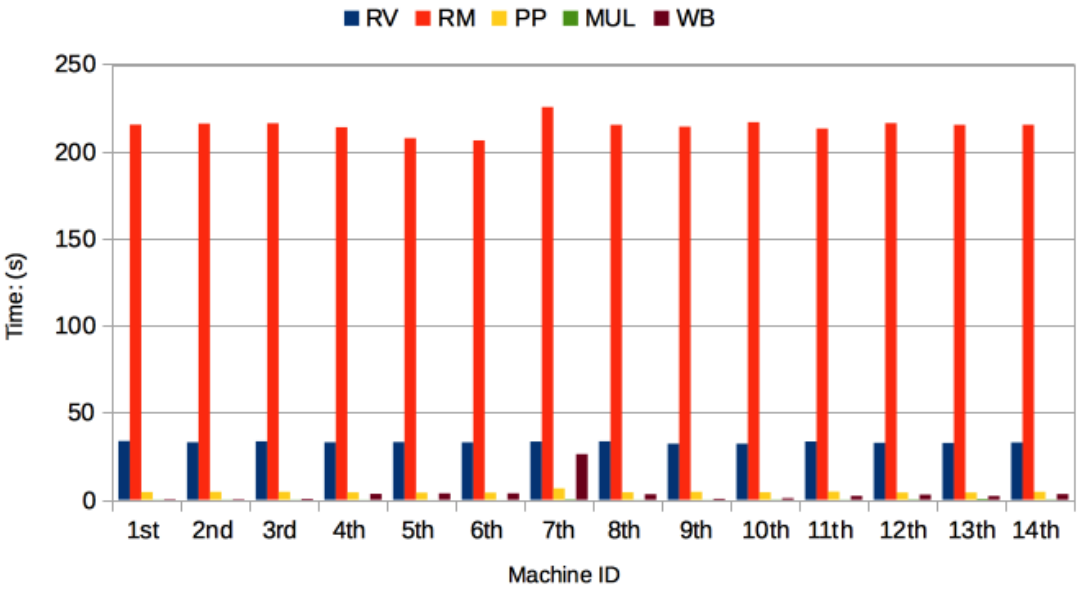


Figure 5.6: Average running time of operations in MV by distributing work loads equally to working machines for matrix with size of one million.

Notice the result obtained in Fig. 5.6 is conducted without caching. By caching the data into in-memory file system, we can significantly improve the performance by reducing expensive disk I/O.

Recall the evaluation model discussed in last chapter, we argue that yInMem has offered an optimal partition solution.

$$T = T_{preparation} + T_{partition} + T_{computation} \tag{5.3}$$

$$T_{computation} = \sum_{i=1}^{n}(t_i + t^i_{agg} + t^i_{broadcast} + t^i_{syncrhonization}) \tag{5.4}$$

$$t_{agg} = n_p * S_{v_{partial}}/S_{network} \tag{5.5}$$

$$t_{broadcast} = n_p * S_{v_{agg}}/S_{network} \tag{5.6}$$

First, $t_{syncrhonization}$ has been optimized since all processes will finish at around the same time, there is no extra waiting for the slowest process. Second, t_{agg} and t_i have also been optimized because of balanced workload. $t_{broadcast}$ is typically not impacted because it is only proportional to the vector size. In addition, $T_{partition}$ is the running time of Algorithm 3 with the time complexity of $O(n_p)$.

5.3 Related work

FiDoop-DP [58] was developed to use the Voronoi diagram-based data partitioning technique to boost the performance of parallel Frequent Itemset Mining on Hadoop clusters. More formally, FiDoop-DP aggregates highly similar transactions into a data partition to improve locality. LBP(Locality Based Partitioning) [59] clusters data blocks from a same node into a single partition, avoiding the spoil time for slot reallocation and reducing the initialization time for multiple tasks. They also provide a LBP-SA(LBP Skew Aware) to partition the data file according their record and computation skews. The fundamental difference of yInMem is that

yInMem uses NoSQL distributed data management system to partition the data and also ensures load balance.

Pronto [60] extends R to a distributed system for iterative algorithms with a focus on matrix operations. To conquer the imbalance and computation skew, Pronto adapts a dynamic repartition method using the concept of distributed arrays. Pronto tracks the number of elements in a partition (e_i which is similar to $avgLoad$) and execution time fo the task (t_i), and then dynamically repartition data to reduce load imbalance. This mechanism works well in a shared memory environment, but it becomes hard and expensive in non-shared environment.

5.4 Summary

This chapter discusses the impact of data partition in a cluster computer environment. We have presented the challenges with current systems and introduced the data partitioning algorithm deployed in yInMem. By exploring the usage of associative arrays in a NoSQL database, data partitioning algorithm presented in yInMem offers an optimal solution. We have also verified the efficiency of yInMem by comparing the naive partitioning method and our method.

The data partitioning algorithm in yInMem has the following benefits:

1. Minimizing the inter-node communication by breaking data dependencies for embarrassing parallelization. In SpMV example, yInMem caches the rows of matrix into different machines since row is the smallest unit for such operation. There is no communication among rows.

2. The data partitioning algorithm also ensures all machines will bear almost the same amount of work. This guarantees the maximal utilization of cluster resources.

Chapter 6: Evaluation

This chapter presents most of the experimental results to evaluate the performance of yInMem. We have implemented the three following algorithms: (1) K-Means clustering (2) PageRank and (3) Spectral clustering mainly using the parallel SpMV APIs and data partitioning algorithm introduced in chapter 5. Experiments are conducted in our Bluewave cluster using both synthetic datasets and real-world data sets. We have also compared the performance of Hadoop, Spark, and MEM-HDFS with yInMem. Hadoop is a HDD-based approach, serving as the baseline. MEM-HDFS caches the both the initial input data and intermediate results to speed up the computation. And Spark caches input and exchange intermediate results in RDDs. Machine learning libraries have been developed to support both Hadoop and Spark. Mahout [31] is a machine learning library based on Hadoop, while MLlib [30] is Apache Spark's scalable machine learning library. We directly use existing libraries to assess the performance of these algorithms on each parallel system respectively.

6.1 System configuration

This section describes the cluster computer configurations in Bluewave at our CHMPR lab. Our experiments use a 32-nodes cluster connected with 10Gbps switch with the following configuration: each node has 8 processors each with Quad-Core AMD Opteron(tm) processor 2376, 25GB of RAM, L1 cache 512KB, L2 cache 2MB. Hadoop 2.2.0. Accumulo 1.5.2. Zookeeper 3.4.6. D4M, pMatlab, and Matlab 2010bSP2. CentOS 6.5 64-bit. Apache Spark 1.6.1.

Out of all 32-nodes, 1 node is designated as the *namenode* and another node as the leader node for *pMatlab*. So there are 14 nodes as the worker nodes. All input data have been uploaded into Accumulo table in adjacency matrix format , and $TedgeDeg$ table included for each application. Remember $TedgeDeg$ table maintains the metadata about the non-zero elements in each row of sparse matrix.

Datasets: Table 6.1 shows the graphs used for evaluation. Friendster is the largest network graph we can find in [53] with 65 million vertices and 1.8 billion edges. Twitter graph comes next in terms of size. Road graph and Orkut graph have 1.9 and 3 million vertices respectively , but Orkut has a much higher degree than Road. All graphs are real world-data downloaded from [53] except Kron which is generated by Graph500 benchmark.

Table 6.2 is the large synthetic dataset generated to fill the memory capacity of our cluster. The largest dataset is the sparse matrix with 16 million vertices and 24 billion edges, which is around 48GB.

Graph	Description	Vertices(m)	Edges(m)	Degree	Directed
Kron	Synthetic	4k-16	0.2-24000	Varies	N
Friendster	social network	65	1800	27.7	N
Twitter	social network	17	476	28	Y
Road	USA road network	1.9	2	1.05	N
Orkut	social network	3	117	39	Y

Table 6.1: Graphs used for evaluation. All graphs are real-world data [53] except Kron which is generated by Graph500 benchmark.

6.2 Data partitioning

In chapter 5, we have discussed the significance of data partition in cluster computers. We have also presented the result on synthetic dataset with the matrix size 1 million. In this section, we will cover not only the synthetic dataset, but also the real-world graph to strike a comparison. Table 6.3 lists the two datasets we are comparing.

Fig. 6.3 compares the per worker execution time for *SpMV* with synthetic 1-million scale matrix and Friendster graph without Alluxio both before and after data partitioning. By comparing both execution time, we can see the parallelism

Matrix size	Edges	Data file size	Degree
4,096	0.2 million	4MB	48.8
8,192	0.7 million	14MB	85.5
16,384	2 million	40MB	122
65,536	43 million	860MB	6,561
262,144	0.6 billion	1.2GB	22,888
524,288	1.4 billion	2.8GB	23,121
1,048,576	5 billion	10GB	26,122
16,777,216	24 billion	48GB	14,305

Table 6.2: Synthetic graph generated using Kronnecker generator

Graph	Description	Vertices	Edges	Degree	Directed
Kron	Synthetic	1 million	5 billion	2.6 million	N
Friendster	social network	65 million	1.8 billion	27.7	N

Table 6.3: Dataset for data partitioning

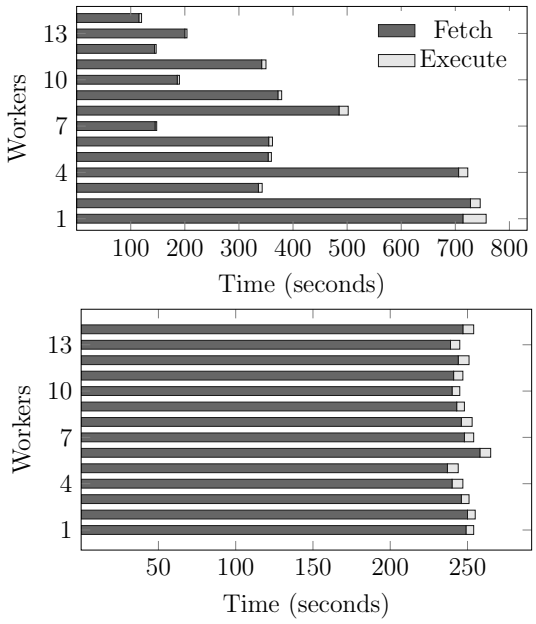


Figure 6.1: synthetic matrix(1 million)

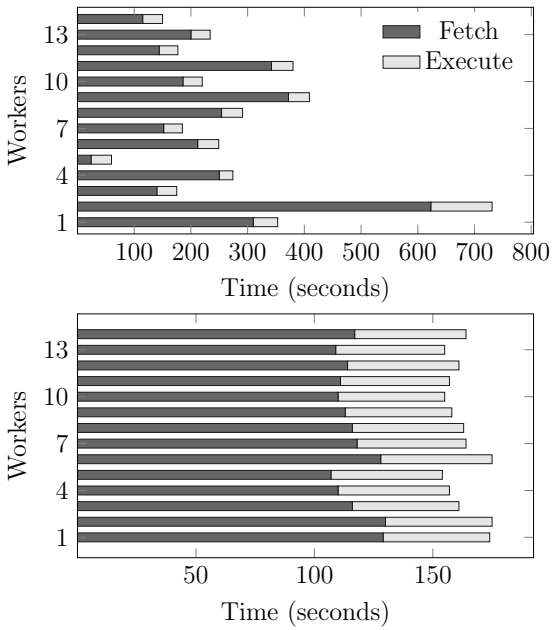


Figure 6.2: Friendster

Figure 6.3: Comparison of per worker execution time for *sparseMV*. Left: synthetic 1 million scale matrix. Right: Friendster graph. (top: before partitioning; down: after partitioning

between our synthetic dataset and the Friendster (real-world application). Top figures indicate the inherent imbalanced distribution of the data. After running data partition, both running time get significantly reduced and all workers in both complete at around the same time, proving the efficiency of data partition. One difference between these two sets is the execute time. The Friendster takes longer to execute, the reason is it has more vertices than Kron (65 times more), meanwhile the fetch time is slightly less than Kron, because the total number of edges is less

than Kron. Fig. 6.6 compares the per worker execution time for Road and Orkut

Graph	Description	Vertices	Edges	Degree	Directed
Road	USA road network	1.9 million	2 million	1.05	N
Orkut	social network	3 million	117 million	39	Y

Table 6.4: Dataset of Road and Orkut

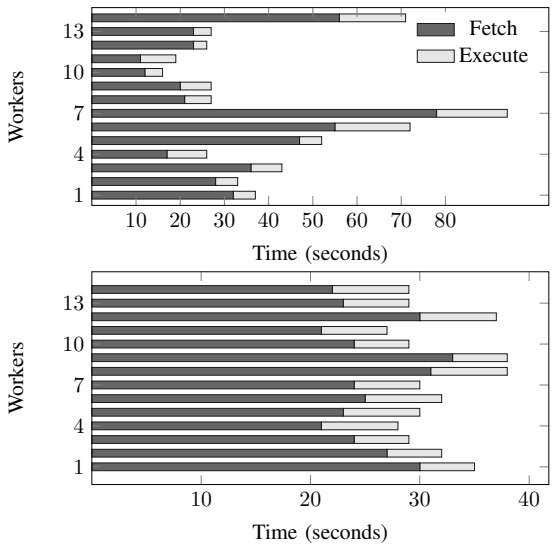


Figure 6.4: synthetic matrix(1 million)

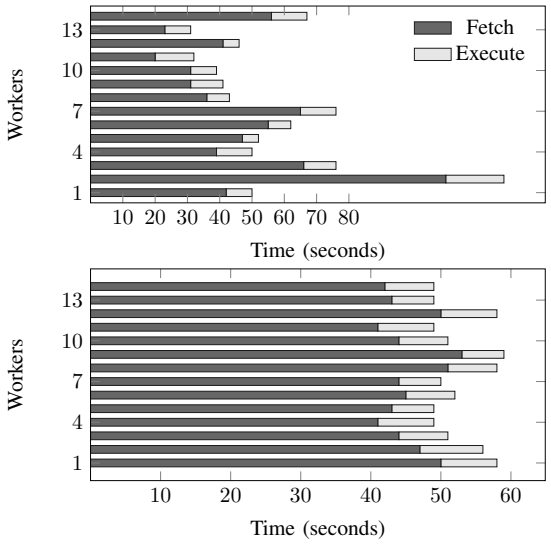


Figure 6.5: Friendster

Figure 6.6: Comparison of per worker execution time for *sparseMV*. Left: Road. Right: Orkut. (top: before partitioning; down: after partitioning

without Alluxio both before and after data partitioning. We see similar result as Fig. 6.3. Fig. 6.7 shows the result for Twitter graph.

Above listed results prove the optimal data partition with yInMem for both synthetic and real-world graphs. We also observe that each process completes the assigned tasks at around the same time, reducing the synchronization overhead for iterative algorithms. In addition, the fetch time is equivalent to the input data size, indicating the workload balance has been achieved for all cases. This is essential to maximize the computing resources in the cluster.

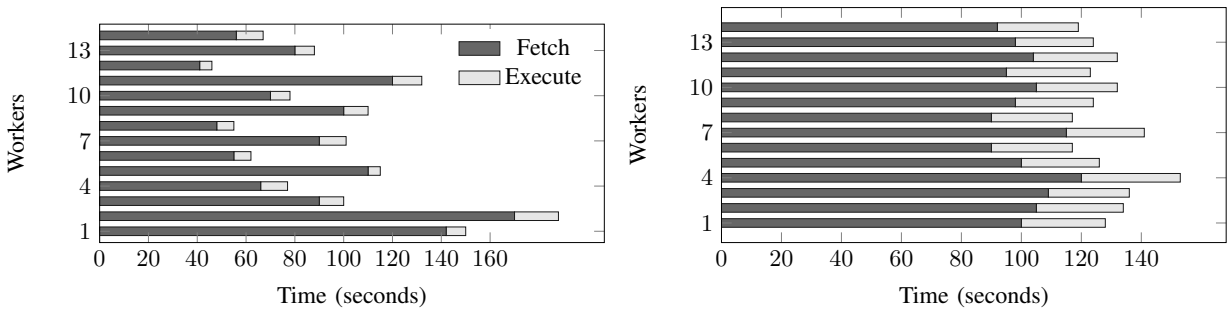


Figure 6.7: Twitter

While all of above experiments are conducted on HDD level, the performance gain is also applicable when data splits are cached in the in-memory file system. First, the computation time is, in general, linear to the input computing data size for large sparse data. Second, Such data partitioning also balances the memory usage across the cluster. This creates new opportunities for memory management in a cluster computing system. At last, it also benefits asynchronized computation because the programmer has fine control over how to achieve load balance on a memory level.

6.3 Performance with iterative algorithms

This section demonstrates the overall running time of iterative algorithms: (1) Spectral clustering algorithms (2) PageRank and (3) K-Means clustering on both synthetic dataset and real world graph.

6.3.1 Spectral clustering

We have implemented an YinEigen algorithm 5 to compute top k eigenvalues and eigenvectors for large sparse graph. We argue that YinEigen is a good representation of iterative machine learning algorithms because: first, it involves the sparse matrix-dense vector multiplication (line 7) which dominates every single iteration, load balancing should be taken into consideration for maximum performance; second, data sharing occurs at each iteration when the dense vector is updated and broadcast across the cluster to all worker nodes(line 12). Compared to PageRank, YinEigen has more extra operations other than SpMV and vector update.

During the initialization stage, the leader first generates a normalized random vector $v_i^{M\times 1}$ and also creates α and β table. The input of YinEigen includes the partition table generated from Algorithm 3. And it assumes that all worker nodes have already cached corresponding data partition in Alluxio. YinEigen takes a power method to calculate α and β which are used to construct tridiagonal matrix T_{mm} whose eigenvalues and eigenvectors are approximation to input matrix M (line 3 to 10). Each function inside the iteration is executed in parallel using the following syntax pRUN('function_name ', Total_number_of_processes, Nodes_list). The first step in the iteration is sparse matrix-dense vector multiplication($SpMV$) which dominates each iteration. The dense vector v_i gets updated after each iteration and v_i needs to be shared across the cluster since we do the multiplication based on rows (line 9). The result vector v has the same partition as input matrix M, and partial result R_{i+1} of v are computed and cached in the main memory of individual worker.

Algorithm 5 YinEigen algorithm: eigenvalue decomposition for large sparse matrix on top of yInMem.

INPUT: 1. Np: total number of processes;
2. $Machines$: nodes in the cluster ;
3. $MaxIt$: number of iterations;
4. k: top k eigenvalues ;
5. Partition table from Algorithm 3 ;
6. M: input matrix $M^{m \times m}$ stored in Accumulo;

OUTPUT: Top k eigenvalues $\lambda[1..k]$ and eigenvectors $Y^{m \times k}$.

#Initialization:
1. Leader initializes a normalized random vector v_1
2. Leader creates α and β table
3. $Np - 1$ workers cache rows of M and v_1 to Alluxio
#Iteration procedure: calculates α_i and β_i to construct tridiagonal matrix T_{mm} whose eigenvalue and eigenvectors are approximations to M, pRUN means run in parallel with Np processes in $Machines$.
for $i = 1 : MaxIt$ **do**
    pRUN **func1**: $v \leftarrow Mv_i$
    pRUN **func2**: $\alpha_i \leftarrow v_i^T v$
    pRUN **func3**: $v \leftarrow v - \beta_{i-1} v_{i-1} - \alpha_i v_i$
    pRUN **func4**: $\beta_i \leftarrow ||v||$;
    pRUN **(Selectively orthogonalization)** [21]
    pRUN **func5**: $v_{i+1} \leftarrow v/\beta_i$
$T_{mm} \leftarrow$ build tridiagonal matrix from α and β
$QDQ^T \leftarrow EIG(T)$; // Eigen decomposition of T
$\lambda[1..k] \leftarrow$ top k diagonal elements of D
$Y \leftarrow V_m Q_k$; Q_k is the columns of Q corresponding to λ

To collate the result, the driver program makes a copy of all these partial results which are saved as files in Alluxio, and then caches through the collated file into HDD. All workers runs *scp* to copy and load the updated v_{i+1} into Alluxio.

Fig. 6.8 shows the average running time of eigenvalue decomposition of various input graphs (Table 6.1 6.2) with the following frameworks: 1. HEIGEN-PLAIN, MapReduce based approach without in-memory support. 2. yInMem. 3. Spark. 4. MEM-HDFS. The input graph is arranged in ascending order of matrix size.

HEIGEN-PLAIN is a MapReduce based approach without caching any data,

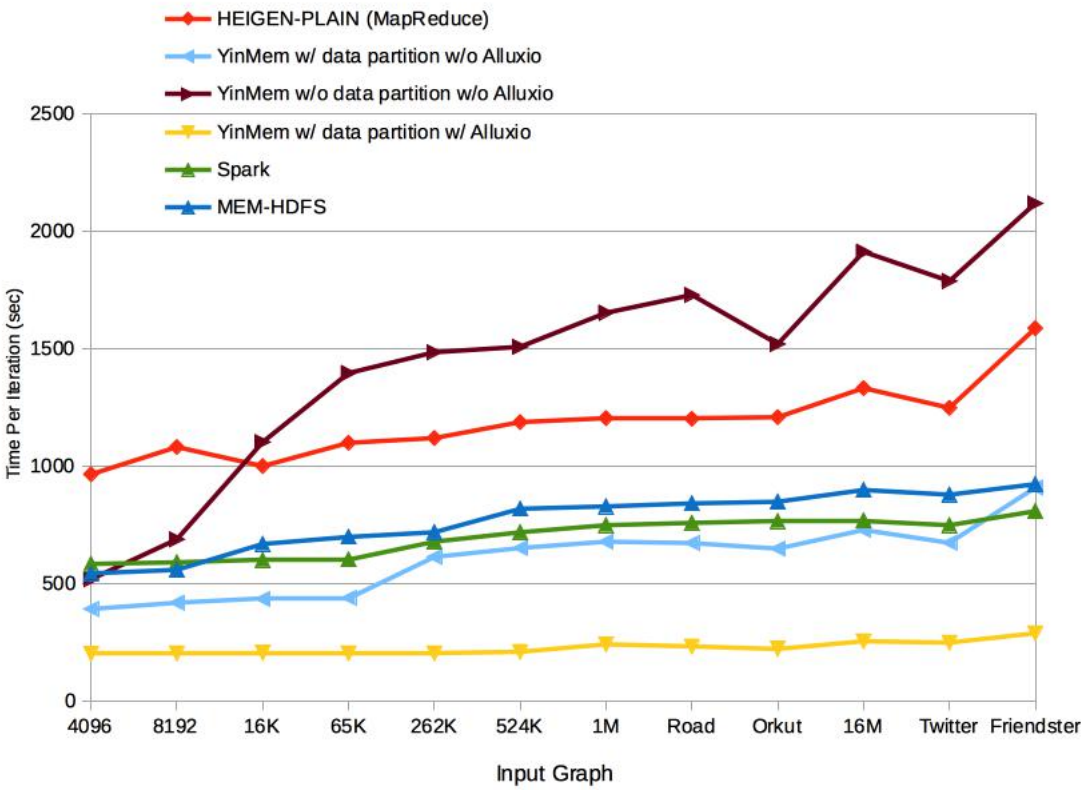


Figure 6.8: Average running time per iteration for eigenvalue decomposition of various input graphs. yInMem with Alluxio (yellow line) shows 6X speedup as compared to HEIGEN and 3X speedup as compared to Spark.

MEM-HDFS is also based on MapReduce with data caching. The difference between MEM-HDFS and Spark is the data replication on memory level. Spark uses lineage to track the computing logics for generating RDDs, removing the data replication for fault-tolerance. MEM-HDFS, however, is slower than Spark when caching the intermediate result because of memory level replication. For yInMem, we run three different sets of experiment, the best performance is when both data partitioning and Alluxio are used.

The top line is when naive data partitioning algorithm is used to split the input matrix without Alluxio. Due to the power-law distribution of the input graph,

every iteration is waiting for the slowest process to complete. As a result, it has the worst performance among all. HEIGEN-PLAIN runs consistently around 1300s per iteration, the reason is first data is very sparse, the shuffling cost is not much affected by the number of vertices. The following three lines are: Spark, MEM-HDFS and yInMem with data partition and without Alluxio. yInMem is slightly faster than previous two, proving the importance of an optimal data partition. Spark is better than MEM-HDFS, because Spark removes data replication. And yInMem with data partition and Alluxio achieves almost 5X speedup than HEIGEN, and 3X faster than Spark.

For Orkut dataset, even it has more vertices than Road, the number of edges is 100X smaller than Road. That explains why all frameworks show almost the same performance. The same reason applies to Twitter, even though Twitter has more vertices, it has much less edges than our 16 million-scale synthetic graph.

We test the accuracy of our spectral clustering algorithm using a smaller matrix size, 10,000 for example. And we then verify the top k eigenvalues and eigenvectors with other eigenvalue decomposition packages like Matlab.

yInMem not only demonstrates the fastest running time for all input graphs but also yields consistent results. That is because of the optimal data partitioning which ensures data locality and workload balance. All workers in the cluster are assigned almost the same amount of work. In addition, data sharing mechanism offers linear performance which outperforms MapReduce based approach.

6.3.2 PageRank

The essential part of power method for PageRank is computing the principal eigenvector of a Markov matrix representing the structured graph. Algorithm 6.3.2 shows the power method for PageRank, which is very similar to YinEigen, both of which consist of SpMV iteratively until convergence.

Algorithm 6 Power method for PageRank

Input:
Matrix $\mathbf{G}$, $k = -1$, pick $x^{(0)} > 0$, $||x^{(0)}||_1 = 1$

1: **Repeat**
2: $k = k + 1$
3: $[x^{(k+1)}]^T = [x^{(k)}]^T G$
4: **until** $||x^{(k+1)} - x^{(k)}|| \leq \epsilon$

Fig. 6.9 shows the average running time per iteration for PageRank of various input graphs. The result is almost the same as Fig. 6.8 except yInMem has gained almost 2X speedup, reducing from 250s per iteration to 110s per iteration. The main reason for this speedup is because of the synchronization cost for multiple parallel operations. PageRank has only one simple parallel operation for each iteration, while YinEigen includes extra operations. Since pMatlab synchronize all processes at each operation, it accumulates this synchronization cost per operation. MapReduce-based frameworks only improve slightly in terms of speed, the reason being there is no extra synchronization cost. Unlike pMatlab which will terminate all worker processes, MapReduce maintains the whole life cycle until the end of the program.

Even though PageRank generates reasonable results for web applications, we benchmark the performance using both synthetic and real-world user applications other than web applications.

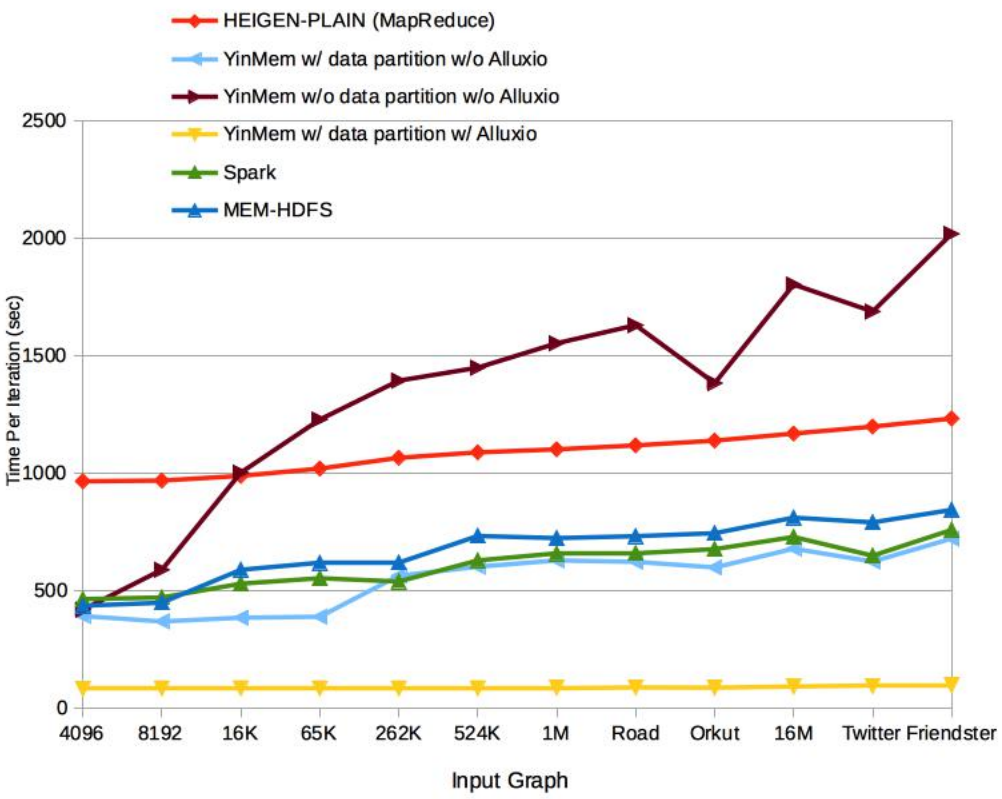


Figure 6.9: Average running time per iteration for PageRank of various input graphs.

6.3.3 K-Means clustering

The parallelization of K-Means is different from previous two algorithms. We deploy the data parallelism for K-Means clustering. That is, we can use naive data partition to split the data points across the cluster. Since each data point is the smallest unit for computation, we first equally divide the data into N_p chunks. Algorithm 6.3.3 is the serial K-Means clustering algorithm. This algorithm is computation intensive as compared to previous two, $O(nk)$ complexity because all data points will calculate the distance to potential centroids. And the data exchange amount is much less than previous two algorithms, since they only broadcast the

local means for each cluster, that is a vector size of k. Our implementation of

Algorithm 7 Serial K-Means clustering algorithm

Input:
D: training examples , k clusters and ϵ convergence rate, t=0
Randomly initialize k centroids: ${\mu_1}^t, {\mu_2}^t, ..., {\mu_k}^t$
1: **Repeat**
2: $t \leftarrow t + 1$
3: $C_j \leftarrow \emptyset$ for all $j = 1, ..., k$
4: **for** $x_j \in$ **D do**
5: $j^* \leftarrow argmin_i ||x_j - {\mu_i}^t||^2$ // assign x_j to closet centroid
6: $C_{j*} \leftarrow C_{j*} \cup \{x_j\}$
7: //Centroid update step
8: **for** $i = 1$ to k **do**
9: ${\mu_i}^t \leftarrow \frac{1}{||C_i||} \sum_{x_j \in C_i} X_j$
10: **until** $\sum_{i=1}^{k} ||{\mu_i}^t - {\mu_i}^{t-1^2}|| \leq \epsilon$

K-Means are as follows:

1. Partition N/P data points to each node.
2. Leader node randomly choose K points and assigns them as the cluster means and broadcast.
3. Each node finds membership for their local data point using the cluster mean.
4. Each node updates local means for each cluster.
5. Leader node collects these local means and broadcast the global mean.

The input graph for this experiment set is primarily synthetic datasets because real-world user application graphs used in previous two experiments are not high dimension. The input data points are generated with 16 dimensions to reflect real-world applications. Admittedly, some applications render even higher dimensions than 16, our intention is to investigate the system performance. We compare our performance with HEIGEN-PLAIN, yInMem, Spark and MEM-HDFS.

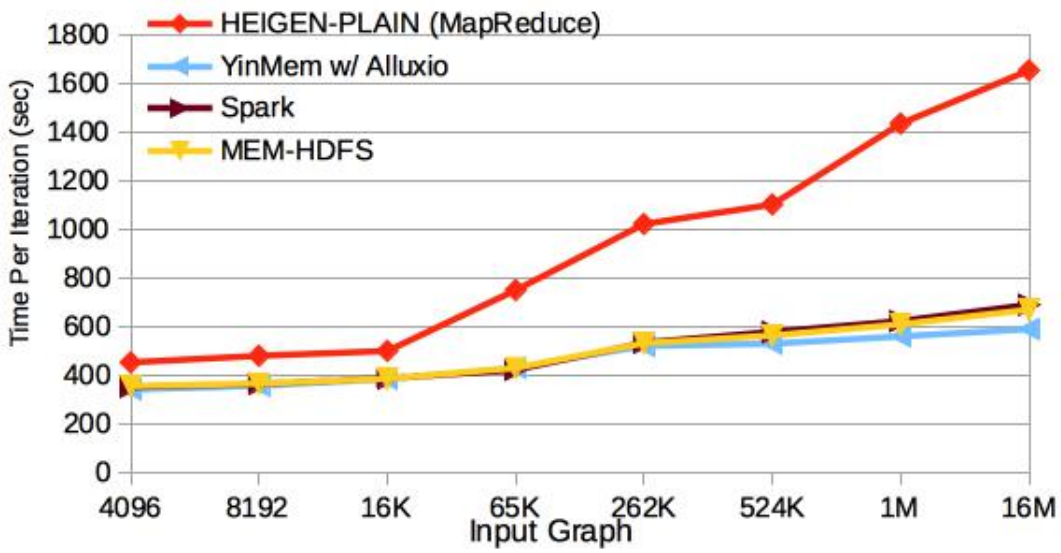


Figure 6.10: Average running time per iteration for K-Means of synthetic data.

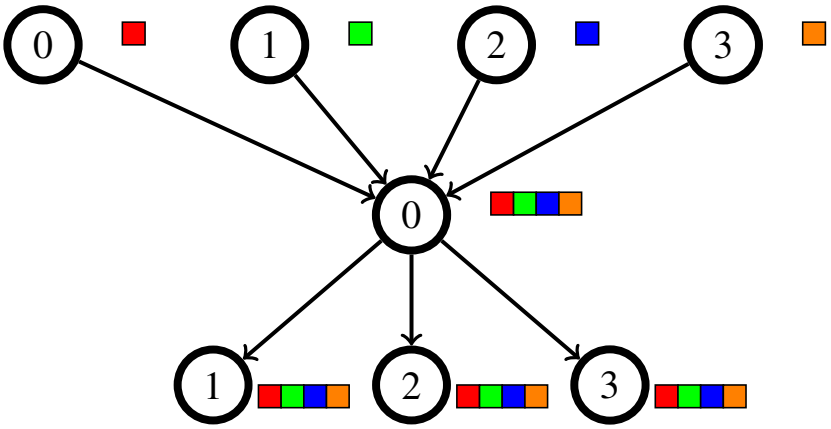
Fig. 6.10 shows the average running time per iteration for K-Means of synthetic dataset. Data partition does affect the converge rate (which is also dependent on the initialization of k centroids) but has no impact on the average running time per iteration. To simplify the analysis, we choose the same k centroids for each test case. Since we are trying to evaluate the framework for computation, this test case actually eliminates the bias resulting from data partition. The three in-memory computing frameworks are very close for K-Means. Because there is no computation/data skew within K-Means, every process will generate a local means based on the data points. As a result, Spark, MEM-HDFS and yInMem are consistent in terms of computing speed.

6.4 Data sharing

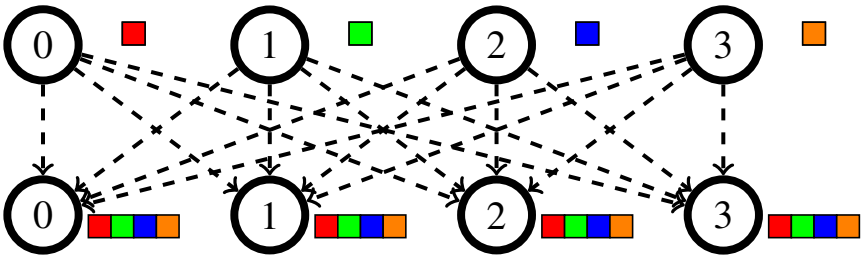
While data partitioning impact most of the system, data sharing is also important for iterative algorithms. And this also explains why yInMem outperforms other MapReduce based in-memory systems. Existing MapReduce based platform inherits the limitation of data sort and data shuffling for exchanging intermediate results. yInMem enables data sharing by copying intermediate results to Alluxio worker nodes. In short, MapReduce follows an all-to-all strategy while yInMem follows point-to-point.

One way to mitigate the effects of communication on performance is to reduce the total number of individual messages by sending a few large messages rather than sending many small messages. All networking protocols incur a fixed amount of overhead when sending or receiving a message. Rather than sending multiple messages, it is often worthwhile to wait and send a single large message, thus incurring the network overhead once.

Consier, iterative SpMV operations, for example. The partial results (vector v) computed from all worker processes have to be aggregated and updated into one dense vector (vector v_{i+1}) which will be multiplied by the input sparse matrix in a parallel manner. That means the output from each iteration will be aggregated and then broadcast for next iteration. With a comparison of yInMem and MapReduce, it is clear that yInMem handles the data sharing more efficiently than MapReduce. First, less communication channels are established. Second, the aggregated result will be assembled once.



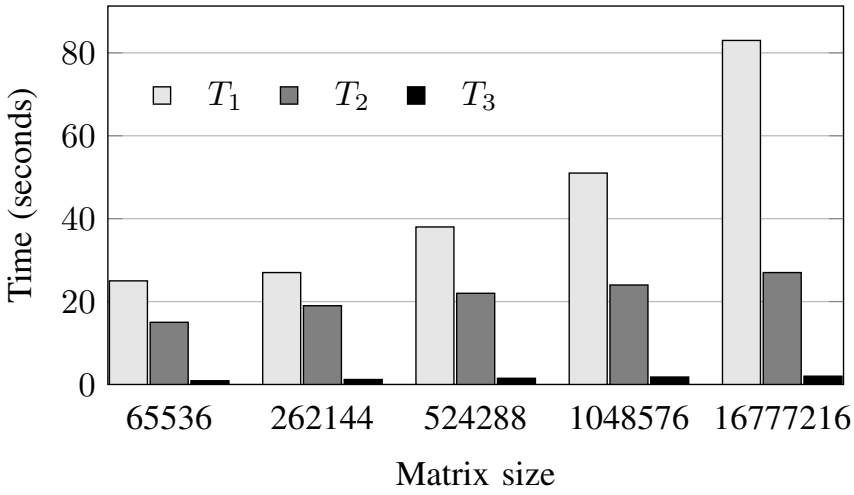
(a) YinMem: data sharing by all-to-one and then one-to-all



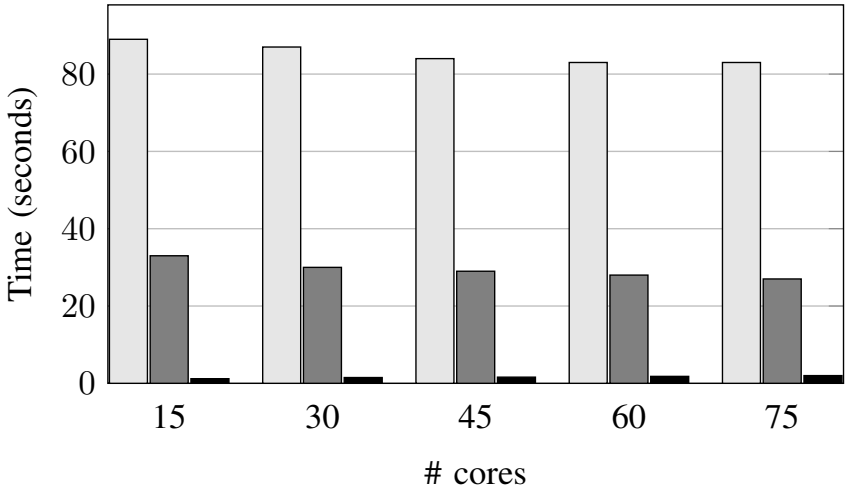
(b) MapReduce: data sharing by all-to-all

Figure 6.11: yInMem VS MapReduce for data sharing

There are three steps in yInMem to share these partial results (6.11(a)). The time for the leader process to receive partial results from workers or T_1 is proportional to the matrix size. That's because partial result vector v_i size is proportional to the matrix size. T_2, the time to broadcast v_i to all workers, is less proportional to the matrix size as compared to T_1 because it is mostly determined by the network connection speed for the cluster. T_3, the time to upload v_{i+1} to DRAM is almost consistent within a few seconds (6.12). MapReduce has its own *shuffling* mechanism, which is essentially a *hash* function. We estimate the shuffling time based on the running time of *mapper* for each iteration 6.13. Fig. 6.12(a) illustrates the time for all these three steps with various synthetic matrix sizes. T_1 appears to be linear



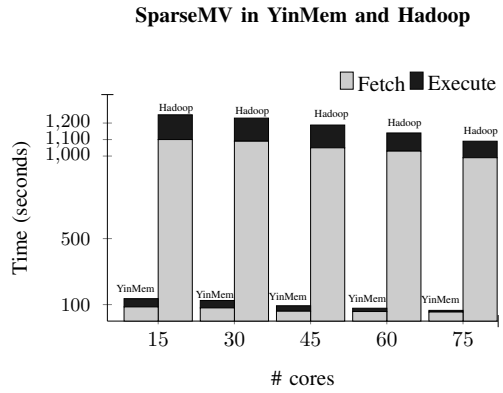
(a) YinMem data sharing performance with varied matrix sizes



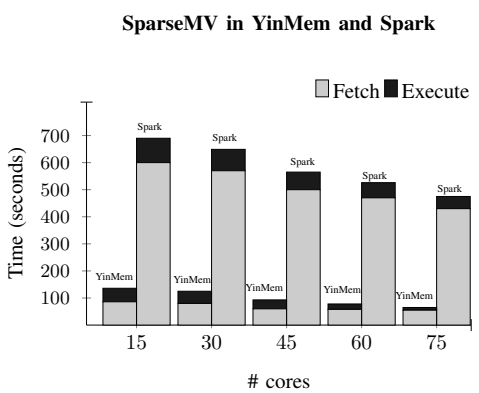
(b) YinMem data sharing performance for 16-million scale matrix with varied number of cores

Figure 6.12: yInMem data sharing

to the matrix size. T_2 remains almost constant, largely because it is determined by network topology and network speed. T_3 is constant. Fig. 6.12(b) shows the three times for 16 million size matrix. T_1 and T_2 is decreasing very slowly, that's because each process will generate smaller output within larger number of cores per worker. However, there is an overhead with increasing number of processes spawned in the cluster. This overhead compensates the performance gains from more processes per



(a) SparseMV performance comparison between Hadoop HEIGEN-PLAIN and YinMem for 16-million scale matrix



(b) SparseMV performance comparison between Spark and YinMem for 16-million scale matrix

Figure 6.13: Performance advantage of yInMem over (a) Hadoop and (b) Spark for *SparseMV*

worker. That is why T_1 and T_2 are not decreasing linearly.

Fig. 6.13 depicts the performance advantage of yInMem over Hadoop(a) and Spark(b) for *SparseMV*. For Hadoop(a), we measure the running time of *mapper* as the fetch time which indicates both the time for reading matrix and vector from HDFS and also the time of shuffling the intermediate results, which is around 1000s for 16 million size matrix using 75 cores. yInMem is around 10X faster both for caching intermediate results and efficient data sharing mechanism. To better understand the performance gain of data sharing, the comparison with Spark(b) shows that yInMem is around 6X faster than Spark. This performance gain is attributed to the data/computation skew within Spark.

6.5 Summary

This chapter presents the experimental results of iterative machine learning algorithms to compare the performance among yInMem, Spark, and Hadoop. yInMem

has achieved consistent speedup over the latter two frameworks for both synthetic and real-world graphs. The performance gain is largely due to data partition with workload balance and data sharing.

Three algorithms under investigation are: (1) K-Means clustering (2) PageRank (3) Spectral clustering. Data partition plays a more important role in PageRank and Spectral clustering than K-Means clustering. However, K-Means is slightly more computation-intensive compared to the other two algorithms. As for data sharing, all of these algorithms generate a vector output which should be aggregated and broadcast to all workers. Both PageRank and Spectral clustering output a larger vector than K-Means. And both run a SpMV kernel iteratively to achieve an object function. The subtle difference between PageRank and Spectral clustering is Spectral clustering includes extra cheap parallel operations besides SpMV, while SpMV is the only operation in PageRank.

Chapter 7: Conclusion

This dissertation presents yInMem: a parallel distributed indexed in-memory computation system for big data analytics with a focus on iterative machine learning algorithms, more specifically Sparse Matrix-Vector Multiplication kernel. The novelty results from introducing the associative arrays to index sparse matrix entries stored in a NoSQL database. This not only separates the data management from the computation engine, but also enables fine control for data partition. As a result, yInMem can achieve state-of-the-art performance in a wide range workloads by bridging the gap between HPC and Hadoop community.

With the rapid development of cluster computing systems, we hope that yInMem presented here can offer, at the very least, a useful insight for next-generation computing system. Even though we mainly focus on iterative algorithms with a SpMV kernel, we believe that the idea of first collecting global information about the input data using database and then devising an optimal data partition solution can maximize the overall system performance. This also benefits the memory management for in-memory computation.

In the rest of this chapter, we summarize a few of the lessons that influenced this work. Finally, we sketch areas for future work.

7.0.1 Lessons learned

The importance of data partitioning. The main thread underlying our work is how important data partitioning is to performance within a single computation (e.g., an iterative algorithm). For big data applications, social network graph analysis in particular, an optimal data partitioning solution is crucial to performance. yInMem utilizes the state-of-the-art distributed data management system to co-locate the computation with the data splits cached in the local RAM. Whereas existing systems have mostly focused on devising hash/partitioning techniques tailored to specific applications, yInMem integrates the data management system with the more general applications. With the continuing gaps between network bandwidth, storage bandwidth and computing power on each machine, we believe that data partitioning will remain a major concern in most distributed applications.

The importance of workload balance. A balanced workload across the cluster is also important for maximizing resource usage. For parallel computing models with synchronization, an unbalanced workload will lead to performance degeneration because of the waiting time for the slowest (with heaviest load) worker. However, some applications naturally lead to data skews during computation. Even though such cases are not explored in this work, we believe a dynamic data partition strategy can be designed to re-balance the workload within yInMem. Because of the associative arrays, it is easy to move data around and cache intermediate results to local RAM. Meanwhile, for most in-memory computing systems, it is also crucial to be able to estimate or predict the memory usage during runtime. Recent

research focuses on statistically analyzing the workload to optimize the memory usage. yInMem might actually shed some light on optimizing the memory usage by both analyzing the input data and also the computation logics.

The importance of minimizing data shuffling. MapReduce is notoriously infamous for the expensive data shuffling for complex algorithms. This limitation is inherently rooted in the computation framework. Even though the simplicity comes at the cost of performance, the right choice of algorithm can also contribute to minimizing data shuffling. One example is to pick the algorithm that can be easily parallelized with minimal intermediate results to share, SpMV is one example. MPI based approach has the advantage of fine control over which machine runs which process but also inherits the limitation of data sharing for large input applications. yInMem conquers this problem by deploying the in-memory file system to share intermediate results while maintaining the fine control of parallel computation.

7.0.2 Future work

Our implementation of yInMem focuses on iterative machine learning algorithms with SpMV kernel. In the future, it is important to devise Sparse Matrix-and-Matrix(SpMM) kernel for a wider range of applications. In addition, current implementation uses a row-based decomposition approach, it is viable to support block algorithms for SpMM. Since we can query associative arrays in a row and column manner, it is straightforward to design interfaces to support block queries.

To make yInMem more extensible, another important future work is to get

rid of the dependency of pMatlab. As a result, it is imperative to re-write the associative arrays to support other languages, C, C++, java e.g.

In addition, yInMem is prone to single point of failure (SPOF) which is a common problem for most master-slave architecture. Since we use pMatlab as the parallel computation engine, if the master process fails, for whatever reason, the programmer has to restart the whole application from the beginning. This is particularly true for iterative algorithms, since the same computation will run multiple rounds. One potential solution is to add a monitor process in the leader node. The monitor process serves two major tasks: 1. keeping track of computing tasks for slaves. 2. watching the progress of each slave process. A task log will be generated on the monitor process, which lists the task for each sub-process. All work processes also maintain a connection to the monitor process by sending heartbeat messages. When the monitor process identifies a dead/crashed machine, it will try to assign the unfinished task to a new machine.

Finally, yInMem makes a homogeneous assumption about the worker machines in the cluster. That means all worker machines share the same computing power, for example, CPU, memory, and network etc. This assumption simplifies the overall architecture of yInMem because the parallel computing model follows the integration of OpenMP and MPI. When this assumption is violated, however, a new running time scheduler might be devised for yInMem to handle the heterogeneity of the cluster. HTGS [66] is such an example of running time scheduler that offers a hybrid task graph for scheduling tasks for multi-cores and multi-GPUs system, with the goal of overlapping data move and data computation. Another future work is,

therefore, to integrate yInMem with HTGS to handle task scheduling in a hybrid cluster environment.

Bibliography

[1] Apache Hadoop. http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In The 6th Symposium on Operating Systems Design and Implementation (OSDI 04), pages 137150, December 2004.

[3] Shvachko, K., Kuang, H., Radia, S. and Chansler, R., 2010, May. The hadoop distributed file system. In Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on (pp. 1-10). IEEE.

[4] B.A. Miller, N. Arcolano, M.S. Bear d, N.T. Bliss, J. Kepner, M.C. Schmidt, and P.J. Wolfe, A Scalable Signal Processing Architecture for Massive Graph Analysis, 37th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, Mar 2012

[5] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D., 2007, March. Dryad: distributed data-parallel programs from sequential building blocks. In ACM SIGOPS operating systems review (Vol. 41, No. 3, pp. 59-72). ACM.

[6] Cheatham, T., Fahmy, A., Stefanescu, D. and Valiant, L., 1996. Bulk synchronous parallel computinga paradigm for transportable software. In Tools and Environments for Parallel and Distributed Systems (pp. 61-76). Springer US.

[7] 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 Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[8] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, and B. Rinaldi, YCSB++: benchmarking and performance debugging

advanced features in scalable table stores, in Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011, p. 9.

[9] N. Bliss, R. Bond, H. Kim, A. Reuther, and J. Kepner, Interactive Grid Computing at Lincoln Laboratory, Lincoln Laboratory Journal, vol. 16, no. 1, 2006.

[10] U Kang, Breandan Meeder, Evangelos E. Papalexakis, and Christos Faloutsos, HEigen: Spectral Analysis for Billion-Scale graphs, IEEE Transactions on knowledge and data engineering, VOL. 26, No.2, Feb 2014.

[11] Ankur Dave, Wei Lu, Jared Jackson, Roger Barga, Cloudclustering: Toward an iterative data processing pattern on the cloud.

[12] Apache Giraph http://giraph.apache.org/

[13] Apache Hama https://hama.apache.org/

[14] V. Hernandez, J.E. Roman, A. Tomas, and V. Vidal, A Survey of Software for Sparse Eigenvalue Problems, technical report, Universidad Politecnica de Valencia, 2005

[15] Nathan P. Halko, Randomized methods for computing low-rank approximations of matrices, Ph. D., Department of Applied Mathematics, University of Colorado. 2012

[16] B.A. Miller, N. Arcolano, M.S. Bear d, N.T. Bliss, J. Kepner, M.C. Schmidt, and P.J. Wolfe, A Scalable Signal Processing Architecture for Massive Graph Analysis, 37th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, Mar 2012

[17] Travinin Bliss, N., Kepner, J. pMatlab parallel Matlab library. Int.J. High Perform. Comput. Appl. 21(3), 336-359 (2007)

[18] Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In 19th Symposium on Operating Systems Principles. Lake George, NY. 29-43

[19] R. Buyya (ed.), High Performance Cluster Computing: Architectures and Systems, vol. 1, Prentice Hall, 1999

[20] B. Wilkinson and M. Allen, Parallel Programming (New Jersey: Prentice Hall, 1999).

[21] B. N. Parlett and D. S. Scott, The Lanczos algorithm with selective orthogonalization, Mathematics of Computation, 33:217-238, 1979.

[22] Apache Spark https://spark.apache.org/

[23] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.

[24] D. C. Liu , J. Nocedal, On the limited memory BFGS method for large scale optimization, Mathematical Programming: Series A and B, v.45 n.3, p.503-528, Dec. 1989

[25] H.P. Crowder and P. Wolfe, Linear convergence of the conjugate gradient method, IBM Journal of Research and Development 16 (1972) 431433.

[26] J. Kepner et al., "Dynamic distributed dimensional data model (D4M) database and computation system," 37th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, Mar 2012.

[27] Apache HBase http://hbase.apache.org/

[28] Apache Accumulo https://accumulo.apache.org/

[29] Apache Spark https://spark.apache.org/

[30] MLlib http://spark.apache.org/mllib/

[31] Apache Hama https://mahout.apache.org/

[32] U Kang , Brendan Meeder , Evangelos Papalexakis , Christos Faloutsos, HEigen: Spectral Analysis for Billion-Scale Graphs, IEEE Transactions on Knowledge and Data Engineering, v.26 n.2, p.350-362, February 2014

[33] Li, Haoyuan, et al. "Tachyon: Reliable, memory speed storage for cluster computing frameworks." Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[34] Zaharia, Matei Alexandru. An architecture for fast and general data processing on large clusters. Diss. University of California, Berkeley, 2013.

[35] Ghoting, Amol, et al. "SystemML: Declarative machine learning on MapReduce." Data Engineering (ICDE), 2011 IEEE 27th International Conference on. IEEE, 2011.

[36] Huang, Botong, Shivnath Babu, and Jun Yang. "Cumulon: Optimizing statistical data analysis in the cloud." Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013.

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[38] N. S. Islam, X. Lu, M. W. Rahman, R. Rajachandrasekar, D. K. Panda, "In-Memory I/O and Replication for HDFS with Memcached: Early Experiences", 2014 IEEE International Conference on Big Data (IEEE BigData), 2014.

[39] B. Fitzpatrick, "Distributed Caching with Memcached" Linux Journal, 2004.

[40] N. S. Islam, X. Lu, M. W. Rahman, D. Shankar, and D. K. Panda. Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture. In 15th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid) , 2015.

[41] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In NSDI, 2011.

[42] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In PLDI, 2010.

[43] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In HPDC 10, 2010.

[44] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. Proc. VLDB Endow., 3:285 296, September 2010.

[45] GrzegorzMalewicz,MatthewH.Austern,AartJ.CBik,JamesC.Dehnert,Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In SIGMOD, 2010.

[46] Russel Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In Proc. OSDI 2010, 2010.

[47] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. Computer, 24(8):52 60, Aug 1991.

[48] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In SOSP, 2011.

[49] Weale, T., Gadepally, V., Hutchison, D. and Kepner, J., 2016, September. Benchmarking the graphulo processing framework. In High Performance Extreme Computing Conference (HPEC), 2016 IEEE (pp. 1-5). IEEE.

[50] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013, p. 2.

[51] https://github.com/GovernmentCommunicationsHeadquarters/Gaffer

[52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," Proceedings of the VLDB Endowment, vol. 5, no. 8, pp. 716727, 2012.

[53] https://snap.stanford.edu/data/

[54] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. European Conference on Principles and Practice of Knowledge Discovery in Databases, 2005.

[55] Yin Huang, Han Dong, Yelena Yesha, Shujia Zhou, "A Scalable System for Community Discovery in Twitter During Hurricane Sandy", 14th IEEE/ACM International Symposium Cluster Cloud and Grid Computing (CCGrid), May 2014.

[56] Ipsen, I.C. and Wills, R.S., 2006. Mathematical properties and analysis of Googles PageRank. Bol. Soc. Esp. Mat. Apl, 34, pp.191-196.

[57] Yin Huang, Yelena Yesha, and Shujia Zhou. 2015. A database-based distributed computation architecture with Accumulo and D4M: An application of eigensolver for large sparse matrix. In Proceedings of the 2015 IEEE International Conference on Big Data (Big Data) (BIG DATA '15).

[58] Xun, Y., Zhang, J., Qin, X. and Zhao, X., 2017. FiDoop-DP: Data Partitioning in Frequent Itemset Mining on Hadoop Clusters. IEEE Transactions on Parallel and Distributed Systems, 28(1), pp.101-114.

[59] Wang, C., Wu, Q., Tan, Y., Wang, W. and Wu, Q., 2013, December. Locality Based Data Partitioning in MapReduce. In Computational Science and Engi-

neering (CSE), 2013 IEEE 16th International Conference on (pp. 1310-1317). IEEE.

[60] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A. and Schreiber, R.S., 2013, April. Presto: distributed machine learning and graph processing with sparse matrices. In Proceedings of the 8th ACM European Conference on Computer Systems (pp. 197-210). ACM.

[61] Gadepally, V., Kepner, J., Arcand, W., Bestor, D., Bergeron, B., Byun, C., Edwards, L., Hubbell, M., Michaleas, P., Mullen, J. and Prout, A., 2015, September. D4m: Bringing associative arrays to database engines. In High Performance Extreme Computing Conference (HPEC), 2015 IEEE (pp. 1-6). IEEE.

[62] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In OSDI 2012.

[63] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M.Le on,Y.Li,A.Lloyd, and V.Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In CIDR, volume 11, pages 223234, 2011.

[64] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. ACM SIGCOMM Computer Communication Review, 42(4):2536, 2012.

[65] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Lev- erich, D. Mazie‘res, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. Communications of the ACM, 54(7):121130, 2011.

[66] Tim Blattner (2017). The Hybrid Task Graph Scheduler (Doctoral dissertation).

[67] L.N. Trefethen and D. Bau III, Numberical Linear Algebra, SIAM, 1997

[68] Shi, Juwei, et al. “Clash of the titans: MapReduce vs. Spark for large scale data analytics.” Proceedings of the VLDB Endowment 8.13 (2015): 2110-2121.

[69] Reyes-Ortiz, Jorge L., Luca Oneto, and Davide Anguita. “Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf.” Procedia Computer Science 53 (2015): 121-130.

[70] Zhang, Y., Gao, Q., Gao, L., AND Wang, C. 2012. IMapReduce: A Distributed Computing Framework for Iterative Computation. J. Grid Comput. 10, 1, 4768

[71] Aly,A.M., Sallam, A., Gnanasekaran, B.M., Nguyen-Dinh, L.V.,Aref, W. G., Ouzzani, M., Ghafoor, A. (2012, April). M3: Stream processing on main-memory mapreduce. In 2012 IEEE 28th International Conference on Data Engineering (pp. 1253-1256). IEEE.

[72] Yoo, R. M., Romano, A., Kozyrakis, C. (2009, October). Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on (pp. 198-207). IEEE.

[73] Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J., McPherson, J. (2010, June). Ricardo: integrating R and Hadoop. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (pp. 987-998). ACM.

[74] Davis, Timothy A., Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. "A survey of direct methods for sparse linear systems." Acta Numerica 25 (2016): 383-566.

[75] Beamer, Scott, and Krste Asanovic David Patterson. "Reducing Pagerank Communication via Propagation Blocking."

[76] Monteiro, Steena, Forrest Iandola, and Daniel Wong. "STOMP: Statistical Techniques for Optimizing and Modeling Performance of blocked sparse matrix vector multiplication." Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on. IEEE, 2016.

[77] Samfass, Philipp Johannes. Towards a deeper understanding of hybrid programming. Diss. 2016.

[78] Azad, Ariful, et al. "The Reverse Cuthill-McKee Algorithm in Distributed-Memory." arXiv preprint arXiv:1610.08128 (2016).

[79] Li, Sicheng, et al. "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel." Proceedings of the 35th International Conference on Computer-Aided Design. ACM, 2016.

[80] Fischer, Peter. "Density-Aware Linear Algebra in a Column-Oriented In-Memory Database System."

[81] AbuBaker, Nabil. Reordering methods for exploiting spatial and temporal localities in parallel sparse matrix-vector multiplication. Diss. Bilkent University, 2016.