MATRIXMAP: PROGRAMMING ABSTRACTION AND IMPLEMENTATION OF MATRIX COMPUTATION FOR BIG DATA APPLICATIONS

YAGUANG HUANGFU

M.Phil

The Hong Kong Polytechnic University

2016

THE HONG KONG POLYTECHNIC UNIVERSITY
DEPARTMENT OF COMPUTING

MATRIXMAP: PROGRAMMING ABSTRACTION AND IMPLEMENTATION OF MATRIX COMPUTATION FOR BIG DATA APPLICATIONS

Yaguang Huangfu

A thesis submitted in partial fulfillment of the requirements for
the degree of Master of Philosophy

November 2015

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____ (Signed)

Huangfu Yaguang (Name of Student)

Abstract

Big data refers to information that exceeds the processing capacity of conventional database systems and is characterized by its volume, velocity and variety. Big data programming requires parallel programming systems to implement parallel programming models to scale up with flexibility. A parallel programming model is an abstraction which expresses the application logic, defines how to load data into data structures, and perform parallel operations on the structure.

The computation core of many big data applications can be expressed as general matrix computations, including linear algebra operations and irregular matrix operations. Many common machine learning algorithms and graph algorithms can be implemented by matrix operations. However, existing parallel programming systems do not provide programming abstraction and efficient implementation for general matrix computations. For example, Data-Parallel programming systems such as Spark are inefficient to support matrix operations. Graph-Parallel programming systems such as GraphLab are for graph algorithms but do not support matrix operations. Large-scale matrix computation systems such as MadLINQ are specified for linear algebra operations, but do not support irregular matrix operations.

In this thesis, we describe the design and implementation of MatrixMap, a unified and efficient data-parallel programming framework for general matrix computations. MatrixMap provides powerful yet simple abstraction, consisting of a distributed in-memory data structure called bulk key matrix and a computation interface defined by matrix patterns. Users can easily load data into bulk key matrices and program algorithms into parallel matrix patterns. Bulk key matrix is the fundamental data structure of MatrixMap, a scalable and constant distributed shared memory data structure, which stores vector-oriented data indexed by key and can keep data across matrix patterns. Matrix patterns can be programmed by user-defined lambda function. Mathematical matrix is the special case with

key and value in number.

We implement MatrixMap on a shared nothing cluster with multi-cores support. The BSP model is used to compute each pattern and to form an asynchronous computation pipeline of getting, computing and saving data. Furthermore, we leverage sparse matrices and BLAS (Basic Linear Algebra Subprograms) to speed up in-memory matrix computations. MatrixMap outperforms current state-of-the-art systems by employing three key techniques: matrix patterns with lambda functions for irregular and linear algebra matrix operations, asynchronous computation pipeline with optimized data shuffling strategies for specific matrix patterns, and in-memory data structure reusing data in iterations. Moreover, it can automatically handle the parallelization and distribute execution of programs on a large cluster. Based on MatrixMap, many example applications have been implemented and tested. The experiment results show that MatrixMap can be 12 times faster than Spark.

Publications

Conference Paper

1. **Yaguang Huangfu, Jiannong Cao, Hongliang Lu, Guanqing Liang, "MatrixMap: Programming Abstraction and Implementation of Matrix Computation for Big Data Applications, IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2015**

Journal Paper

1. **Yaguang Huangfu, Jiannong Cao, Hongliang Lu, Guanqing Liang, "MatrixMap: a Big Data Programming Model for Machine Learning and Graph Algorithms, IEEE Transactions on Computers, 2016 (submitted)**

Acknowledgements

I would like to express my gratitude to all those who helped me during my MPhil. study. My deepest gratitude goes first and foremost to Prof. Jiannong Cao, my supervisor, for his systematic guidance and valuable suggestions. He has broad knowledge, keen insight, and enormous enthusiasm for the research, which inspire me and encourage me to keep going in my study. He teaches me to be a good researcher, not only about the research methods and presentation skills but also rigorous work attitude. I appreciate all these and will definitely benefit from him in my future work.

I would also like to thank my parents. They always encourage and support me when I encounter difficulties in different aspects. Their love is the most powerful motivation I can make progress in my work. I would like to thank my colleagues Dr. GuanQing Liang, Dr. Hongliang Lu, Dr. Lei Yang, all other members of our research group that I cannot enumerate here. Thank you for your help in these years. We learn from each other, share our joyfulness and sadness, and have an unforgettable memory together. I wish all of you a brilliant future.

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis aims to design and implement a big data programming framework to solve issues of big data programming. The core of many big data applications can be expressed as general matrix computation, including linear algebra operations and irregular matrix operations. Irregular matrix operations are similar to linear algebra operations, for example, irregular matrix plus in Graph Merge is not plus but to compare two matrices. However, existing data-parallel systems lack the support of abstractions for programming and efficient implementation of general matrix computation. Here, we propose MatrixMap, a unified and efficient data-parallel system for general matrix computation. MatrixMap provides powerful yet simple abstraction, a data structure, called bulk key matrices with computation interface, matrix patterns. Based on MatrixMap, we have implemented typical example applications.

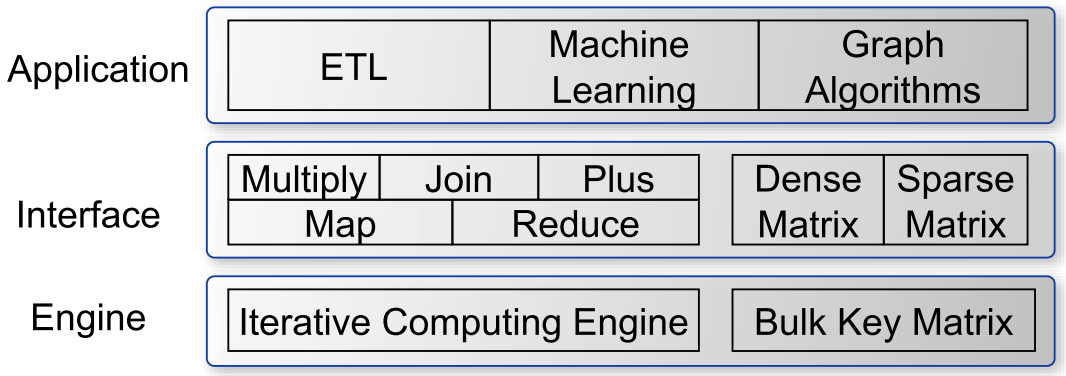

Fig. 1.1: MatrixMap Framework

MatrixMap can support varieties of big data applications, Extract-Transform-Load, machine learning and graph algorithms. Figure1.1 illustrates whole picture of MatrixMap. There are three layers in the MatrixMap project. The middle layer is the interface layer which provides five parallel patterns and matrix data formats, dense matrices and sparse matrices. At the bottom layer, iterative computing engine supports matrix patterns and bulk key matrix supports dense and sparse matrix. At the application layer, MatrixMap supports Extract-Transform-Load, machine learning, and graph algorithms. MatrixMap plays important roles in big data research community and big data in industries.

In this chapter, we first describe the background knowledge of big data in Section 1.1. Then we introduce problems of existing systems in Section 1.2. After that, we introduce the approach in Section 1.3. In Section 1.4, we summarize the main contributions of this thesis. Finally, we outline the organization of this thesis in Section 1.5.

1.1 Big Data Appications

Big data is characterized by its volume, velocity, variety, and applies to information that exceeds the processing capacity of conventional database systems. Cloud computing provides abundant computing resources and massively parallel processing capabilities that can support the management and processing of big data. In recent years, many parallel programming systems have been carried out on coordinating the cloud computing resources to support different types of big data applications.

A parallel programming model is an abstraction which expresses the logic of the application, defines how to divide data into structure and element, and performs parallel operations on the structure. A framework is the implementation of the corresponding model: architecture and runtime mechanisms that map them onto the processing units, coordinate the processing units to form a data flow, and scale up data to huge volume flexibly. Frameworks provide API in libraries or language extensions. Such high-level support enables users to process big data without being involved in many low-level things, for example, load balancing and failover.

Many machine learning algorithms are based on matrix operations. Matrix parameters can be used to learn interrelations between features: The (i,j)th element of the parameter matrix represents how feature vector i is related to feature vector j. The prediction is a function of a dot product between the parameter vector and the feature vector. Fo example, PCA and collaborative filtering are to infer matrix parameters.

Additionally, matrix operations provide computation engine for the majority of machine learning algorithms. Such algorithms will not be effective unless they are trained and are operating on large data sets, ranging from hundreds of training examples to millions of testing data. For example, KMeans [Kan02] is a typical unsupervised learning clustering algorithms. The multiple dimensions points are formulated into matrices. In an iteration, the point will multiply centroids and get the distance. Points will be classified into new clusters according to distance. And logistic regression [HLS00] is a simple and typical supervised learning algorithm which is an algorithm for learning a binary classifier: a function that maps its input vector to a binary output.

Many common graph algorithms can be implemented by operations on the adjacency matrix: Breadth-first or depth-first search can be formulated into matrix-vector multiplication; Graph merge can be formulated into matrix-matrix plus; Breadth-first or depth-first search to or from multiply vertices simultaneously can be formulated into matrix-matrix multiplication. [KG11] This approach can provide a variety of benefits:

- Graph algorithms expressed in matrices are more compact and are easier to figure out.

- Graph algorithms expressed in matrices are easy to be implemented by the existing tools for parallel computations.

- Graph algorithms expressed in matrices have clear data access patterns which can be readily optimized.

1.2 Limitations of Existing Matrix Computations Systems

Many common machine learning and graph algorithms which are extensively used in big data applications can be implemented by general matrix computations, including linear algebra operations and irregular matrix operations. General matrix operations define a sequence of operations on matrices. Irregular matrix operations extending linear algebra operations have the same sequences of operations as their corresponding linear algebra operations but have different operations. For example, operations of the irregular matrix plus in Graph Merge is not the addition but the or operation, but they perform the same sequence of operations on matrices.

However, existing parallel programming systems do not have programming abstraction and efficient implementation for general matrix computation in machine learning and graph algorithms. Data-Parallel programming systems are inefficient to support matrix operations. In MapReduce [DG08] and Spark [ZCF+10], matrix multiplication has to be implemented into several Map and Reduce, which is cumbersome and not efficient. Graph-Parallel programming systems are for graph algorithms but do not support matrix operations. Large-scale matrix computation systems are specified for linear algebra operations but do not support irregular matrix operations such as graph merge and all pair shortest paths.

To facilitate the processing of big data applications, a unified and efficient programming model for general matrix computation is highly desirable. Nevertheless, designing and implementing such programming system entails two major challenges.

The first challenge lies in abstracting a unified interface for both machine learning and graph algorithms. Many algorithms, for example, PageRank, can be formulated into linear algebra operations. So we need to support linear algebra operations. On the other hand, many algorithms such as graph algorithms, can be formulated into irregular matrix operations. For example, graph merge is to merge two graphs. It is similar to linear algebra matrix plus. It is not to plus each element in the matrix but to do or operation. All pair shortest paths is similar to matrix multiplication. It is not to sum corresponding row and

column in two matrices, but to get the minimum value between elements in the first matrix and sum of two elements of two matrices. We also need to support these irregular matrix operations.

The second challenge is how to implement a unified interface to support general matrix computations. The input of logistic regression is the dense matrix, while the input of PageRank is a sparse matrix. Thus, we need to support both dense matrix and sparse matrix. Irregular matrix operations, especially in graph algorithms, are different from linear algebra operations. For example, Breadth-First search can be formulated into irregular multiply operations, but needs level synchronization. Additionally, many algorithms require to reuse data in iterations. For example, PageRank can be formulated as matrix and vector multiplication. In each iteration, the matrix keeps the same. We need to reuse data in iterations algorithms.

1.3 MatrixMap Programming Framework

We present MatrixMap, a unified and efficient data-parallel programming framework for general matrix computations. MatrixMap provides a powerful yet simple abstraction. MatrixMap defines a data structure called bulk key matrix and a computation interface using matrix patterns. Bulk key matrix is the fundamental data structure, a scalable and constant distributed shared memory data structure [PTM98], which stores vector-oriented data indexed by key and can keep data across matrix patterns. Mathematical matrix is the special case with key and value in digits. Matrix patterns can be programmed by user-defined lambda function. Particularly, linear algebra operations are special cases of matrix patterns with specific lambda functions. There are two kinds of matrix patterns. One is the unary pattern: Map, Reduce; the other is the binary pattern: Plus, Multiply, Join.

In MatrixMap, data are loaded into bulk key matrices and algorithms are formulated as a series of matrix patterns. MatrixMap is implemented on a shared nothing cluster with multi-cores support. It follows BSP model [Val90] to compute each pattern and form an

asynchronous computation pipeline to get, compute and save data. Furthermore, we leverage sparse matrices and BLAS (Basic Linear Algebra Subprograms) to speed up in-memory matrix computations. MatrixMap with unary matrix patterns supports Extract-Transform-Load [Vas] operations like MapReduce on data and MatrixMap with binary matrix patterns supports complex and dense matrix computations. Users can easily program sequential algorithms to parallel codes without considering parallel issues. It outperforms current systems by employing three key techniques: matrix patterns with lambda functions for irregular and linear algebra matrix operations, asynchronous computation pipeline with optimized data shuffling strategies for specific matrix patterns and in-memory data structures reusing data in iterations. To evaluate performance, several typical algorithms such as PageRank are expressed in this framework. The experiment results show that MatrixMap is up to 12 times faster than Spark, especially for iterative computation. Based on MatrixMap framework, we implement several typical algorithms, such as Breadth-First Search and PageRank, as algorithm library in this framework and we can see codes in MatrixMap are similar to sequential codes.

1.4 Contributions

Machine learning and graph algorithms have been extensively used in big data analytic applications, including social network analysis, recommendation system, etc. However, directly applying existing data-parallel models (e.g., Spark, Presto) to machine learning and graph algorithms can be cumbersome and inefficient. This thesis makes contributions on designing and implementing novel programming model, MatrixMap, for big data. MatrixMap framework plays important roles in big data research community and big data industries.

We propose and implement a unified and abstract programming model for big data processing. We abstract five patterns, Map, Reduce, Join and Multiply, Plus, from machine learning and graph algorithms, especially for Multiply pattern and Plus pattern, which can support a wide range of frequently used algorithms. Map, Reduce and Join patterns can support Extract-Transform-Load and database operations, Multiply and Plus patterns can

support algorithms with general matrix computation. Users with five patterns can cover most of the work in big data processing. Our contributions are:

- A unified and efficient programming framework for general matrix computations.
- A scalable matrix data structure across memory and out-of-core storage for the massive amount of data.
- Matrix patterns with optimized data shuffling strategies and asynchronous computation pipeline for algorithm parallelization.

1.5 Organization of Thesis

Chapter 1 is the introduction of this thesis. Chapter 2 reviews related works in the literature. The main body of this thesis is Chapter 3, Chapter 4 and Chapter 5.

In Chapter 3, we present the design of MatrixMap parallel programming model for big data. In Chapter 4, we show the details of implementation of the MatrixMap framework. In Chapter 5, we introduce example algorithms and carry out the evaluation of typical algorithms on MatrixMap. Finally, we conclude the thesis and discuss the future works in Chapter 6.

Chapter 2

Literature Review

Big data programming environments are different from sequential programming environments, which have to coordinate computation in an unstable and distributed environment. In this chapter, we will introduce the definition of parallel programming models in the big data programming environments, categorize and illustrate typical models. A parallel programming model is an abstraction which expresses the logic of the application, defines how to divide data into structure and element, and perform parallel operations on the structure.

As the complexity of big data, current models can be categorized according to two dimensions: process and application. In terms of process dimension, models can be categorized into message passing model and shared memory model. From the perspective of application, models can be categorized into data parallel models, graph parallel models and matrix computation models.

2.1 Process-oriented Programming Systems

Process-based programming systems make a wrapper of low-level sockets of operating system and can map tasks on a distributed cluster. In the parallel computing parts, users make use of sequential languages, while in the communication and synchronization parts, users take advantage of the message API of the library above the raw sockets. In the beginning, these models are pure message passing model with message method, but gradually some models append shared memory model [PTM98]. These models are general purposes, they are not specified for certain applications. These application programming models are

not automatic, as programmers must manually decompose data, partition tasks, manage tasks, map tasks, communication and make synchronization.

We divide these models into two groups. The first message passing group is a collection of processes, each running on a separate processor and communicating using SEND and RECEIVE primitives that send messages over a network. The typical example is MPI. The second shared memory group has the same logical multiple process structure while their shared memory is simulated by implementing message passing. The typical example is PVM.

2.1.1 Message Passing Systems

MPI (Message Passing Interface) is a specification for a standard library for message passing. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77 or the C programming language. Its interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features.

Since MPI is a general programming model, users are free to express any kinds of applications, which has pros and cons. Although MPI is more advanced than the raw sockets, yet programmers have to manually make communication and synchronization in the distributed environment.

2.1.2 Distributed Shared Memory Systems

The Parallel Virtual Machine (PVM) [Sun90] is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor. Thus, large computational problems can be solved more effectively by using the aggregate power and memory of many computers. The overall objective of this project is to permit concurrent computation resources.

Munin [BCZ90] allows programmers to annotate variables with the access pattern to choose an optimal consistency protocol for them. Linda [CGL86] provides a tuple space programming model that may be implemented in a fault-tolerant fashion. Thor [LAC+96] provides an interface to persistent shared objects.

Parallel language has been embedded with distributed shared memory programming model into the specification of language which shares a global address, such as UPC [CDC+99], X10 [CGS+05]. Users have to manually decompose task mapping into threads or process. In the virtue of these models, users do not need to communicate with message methods. These models try to hide the physical distribution by making the system look like it has shared memory. Users can acquire variable just like in the local programming language. After finishing the program, it can automatically run across the clusters and can dispatch tasks in different nodes.

A language may present a programming model that is higher level, more abstract, than the message passing model supported by the most operating system. There are three main characteristics that distinguish distributed programming languages from traditional sequential languages, in terms of how they deal with parallelism, communication, and partial failures.

X10 [CGS+05] is a modern object-oriented programming language for high performance, high productivity programming of NUCC (Non-Uniform Cluster Computing) systems. It is a new generation of parallel programming language. It aims to deliver new adaptable, scalable systems that will provide a 10× improvement in development productivity for parallel applications.

2.2 Application-oriented Programming Systems

Compared to above-mentioned programming model, these application programming models are more automatic, as they can decompose data, partition tasks, manage tasks, map tasks, communication and synchronize without instructive and redundant codes. It not only manages the resource of a cluster but also can express algorithms. However, in

order to efficiently program applications, these models cannot manipulate concrete task and cannot achieve optimal efficiency. Furthermore, higher abstraction means less generalization. Application programming models are limited to a specific area, for instance, Pregel for the graph. They cannot be applied to every application like MPI.

2.2.1 Data-Parallel Programming Systems

MapReduce, a data parallel programming model, is the dominant programming model in big data. MapReduce [DG08] programming model makes it possible to easily parallelize a number of common batch data processing tasks and operates in large clusters without worrying about system issues like failover management. Most computations in MapReduce are conceptually straightforward and are not iterative jobs. If you want to do iterative job, you have to redirect your data to a file in secondary storage. MapReduce is suitable for offline analysis, for example, ETL operations of large data sets. This abstraction is inspired by the map and reduces primitives presented in Lisp and many other functional languages. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. MapReduce groups together all intermediate values associated with the same intermediate key and passes them to the reduce function. The reduce function, also written by the user, accepts an intermediate key and a set of values for that key. It merges these values together. Typically there is just zero or one output value produced in each reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to be fitted in memory.

Twister [ELZ10], a MapReduce framework that allows long-lived map tasks to keep static data in memory between jobs, which extends MapReduce to support iterative jobs. Abstraction of MatrixMap is more general than iterative MapReduce. A MatrixMap program can define multiple BKMs and alternate between running patterns on them, whereas a Twister program has only one map function and one reduce function.

Dryad [IBY+07] allows a more general application model than MapReduce. It allows programmers to write acyclic graphs of sequential processing modules spanning many computers without compiling any code which refers to existing executables such as Perl or grep,

which are likely to be a generalization of the Unix piping mechanism. More than two stages, map and reduce, are able to be specified by users. Dryad is suitable for offline analysis of large data sets (batch computation system). A Dryad application combines computational vertices with communication channels to form a data flow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers. The vertices provided by the application developer are quite simple and are usually written as sequential programs with no thread creation or locking.

Naiad is a distributed system for executing data parallel, cyclic data flow programs. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform the iterative and incremental computation. It enriches data flow computation with time stamps. It offers low-level primitive data flow for users but does not support high-level matrix operations. [MMI+13]

Spark [ZCF+10] is designed for iterative algorithms (machine learning, graphs) and interactive data mining. It provides two main abstractions for parallel programming: resilient distributed datasets(RDD) and parallel operations on these datasets (invoked by passing a function to be applied on a dataset). Users can explicitly cache an RDD [ZCDD12] in memory across machines and reuse it in multiple MapReduce-like parallel operations. Spark cannot directly support matrix operations, which is the basic operations in machine learning and graph algorithms. For example, matrix multiplication must be formulated into a series of the map and reduce. Its RDD cache algorithm is LRU, which does not consider the context of algorithms to improve efficiency. The main abstraction in Spark is a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. [Spark] In bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance. Thus, RDDs are best suited

for batch applications that apply the same operation to all elements of a dataset.

2.2.2 Graph-Parallel Programming Systems

GraphX [XGF+13] is a resilient distributed graph system on Spark. In GraphX, distributed graph representation is to efficiently distribute graphs as tabular data structures. GraphX model, however, is limited to graph algorithms. Although they are in the same system, GraphX model totally different from Spark model. When programming graph algorithm, programmers should use graph parallel model, GraphX; when programming machine learning algorithms, programmers have to switch to data-parallel model, Spark. So users would be easily confused when they are programming these two kinds of algorithms with separate models.

Pregel [MAB10] is a programming model for processing large graphs in a distributed environment. It is a vertex-centric model. The programs are expressed in an iterative vertex which can send and receive messages to other vertices in the iterations. Vertices iteratively process data and send messages to neighboring vertices. It is easy to adapt typical graphic algorithms into a vertex-based program.

GraphLab [LGK11] is a graph-based, high performance, distributed computation programming model in machine learning. It solves the dependency of the graph. Different from Pregel, it does not work in bulk-synchronous steps, but rather allows the vertices to be processed in asynchronous steps [LGK+12]. The recent PowerGraph [GLG+12] framework combines the shared-memory and asynchronous properties of GraphLab with the associative combining concept of Pregel. It follows GAS Decomposition: gather, apply, scatter. Vertex approach will flood messages in the graph, which is inefficient to converge.

GraphChi [Aap12] which adopts GraphLab model is a system for handling graph computations using just a PC. It uses a novel parallel sliding windows method for processing graphs from disk. Ligra [SB13] is a lightweight graph processing framework for shared-memory, which makes graph traversal algorithms easy to write. The framework consists of two simple routines, one for mapping over edges and one for mapping over vertices. But vertex floods messages through the graph. These can only run on a single computer, which

cannot take advantage of distributed computing to process large-scale data.

X-Stream is a system for processing both in-memory and out-of-core graphs on a single shared-memory machine. A large number of graph algorithms can be expressed using the edge-centric scatter-gather model. This system is for graphs and on a single shared-memory machine [RMZ13].

2.2.3 Matrix Computation Systems

ScaLAPACK [CDPW92] is a library of high-performance linear algebra routines for parallel distributed memory machines. It is used for linear algebra matrix operations. But it does not supprt irregular matrix operations and lacks in-memory data structure to reuse data in iterations.

Presto [VBR+13] extends R programming language to support linear algebra operations. Its data structure is a mathematical matrix, which cannot support key value data. So they cannot manipulate data like MapReduce or Spark, for instance, data join or data aggregation. Its interface limited to matrix multiplication is different from matrix operations and not flexible to support irregular matrix operations. It cannot reuse data in iterations.

Piccolo [PL10] provides key-value partitioned tables which allow computation running on different machines to share distributed mutable state. It is costly to support the immutable state. Piccolo does not support parallel patterns like Map and Reduce with optimized shuffling strategies.

MadLINQ [QCKC12] is a highly scalable, efficient and fault-tolerant matrix computation system integrated with Dryad. Without lambda function on its interface, MadLINQ is less flexible to support algorithms in irregular matrix operations and its DAG engine does not take advantage of in-memory data structure and matrix operations with optimized shuffling strategies.

Blitz [Vel98] and Eigen [JG12] provide optimized matrix computation. Although they have many computation operations, they do not have parallel programming patterns which can be programmable. They cannot run matrix operations in parallel either. Matrix Template 4 [SL98] can do parallel matrix operations, but it cannot program function into the

matrix data operations.[SYK+10]

HAMA [SYK+10] is a framework supporting linear algebra operations based on MapReduce for Big Data analytics which uses the Bulk Synchronous Parallel (BSP) computing model. Its disadvantages are the same as MapReduce.

Chapter 3

The MatrixMap Programming Model

Many common machine learning and graph algorithms which are extensively used in big data applications can be implemented by general matrix computations, including linear algebra operations and irregular matrix operations. We present MatrixMap, a unified and efficient data-parallel system for general matrix computations. MatrixMap provides a powerful yet simple abstraction. MatrixMap defines a data structure called bulk key matrix and a computation interface using matrix patterns. Bulk key matrix is the fundamental data structure, a scalable and constant distributed shared memory data structure [PTM98], which stores vector-oriented data indexed by key and can keep data across matrix patterns. Mathematical matrix is the special case with key and value in digits. Matrix patterns can be programmed by user-defined lambda function. Particularly, linear algebra operations are special cases of matrix patterns with specific lambda functions. There are two kinds of matrix patterns. One is the unary pattern: Map, Reduce; the other is the binary pattern: Plus, Multiply, Join.

3.1 Overview

We present MatrixMap, a unified and efficient data-parallel system for general matrix computations. MatrixMap provides powerful yet simple abstraction, matrix data structure, bulk key matrix and matrix patterns. Bulk key matrix is a constant and scalable distributed shared memory, which stores vector-oriented data indexed by key and can keep data across matrix patterns. Specifically, matrix patterns can be programmed by user-defined lambda function and mathematical matrix operations are special cases of matrix patterns with specific lambda functions. In MatrixMap, data are loaded into bulk key matrices and algorithms are formulated as a series of matrix patterns.

3.2 Programming Interface

MatrixMap provides object-oriented interfaces in C++, a BKM data structure and its patterns with lambda functions as input parameters, illustrated in codes 3.1. MatrixMap supports multiple kinds of data types by C++ template: int, float and other user-defined data type. In order to simplify elaboration, the interface in the example only contains float and string.

Each pattern has its corresponding lambda function. For Map pattern, the map lambda function receives a string and then insert processed key-value pairs into the context. For Reduce, the reduce lambda function receives a string and iterable object and write key-value pairs into the context. For Multiply and Plus patterns, their lambda functions receive two numbers and return another number. For Join pattern, its lambda function receives two keys in numbers or string from each row in each matrix. If users want to reserve two input rows, the lambda function should return true.

Listing 3.1: Matrix Interface

```
class BKM {
 // Matrix Patterns
 Map(MapLambda);
 Reduce(ReduceLambda);
 Multiply(MultiplyLambda);
 Plus(PlusLambda);
 Join(JoinLambda);

 // Matrix supporting method
 BKM(string file_name);
 Load(string file_name);
 Cahe();
 Save();
};
void map(string, string, Context);
void reduce(string, Iterable<int>, Context);
float multiply(float, float);
float plus(float, float);
bool join(float, float);
```

To use MatrixMap, users should write a driver program that implements the high-level control flow of their application and launches various patterns in parallel. It can directly implement acyclic data flow and cyclic data flow with native C++ control flow clause, for example, if-else clause or while clause. Besides matrix patterns, MatrixMap provides supporting methods. For example, users can use Load method to load data, use Cache method to cache data in the memory, and use Save method to dump all data into disks.

3.3 Bulk Key Matrix

Bulk key matrix (BKM) is the fundamental data structure. BKM can be viewed as an abstraction for distributed shared memory which spreads data in the whole cluster and

provides an integrated interface for usage. It can achieve the balance between cost and performance. Mathematical matrix is the special case of BKM with key and value in digits. In the processing, data will be loaded into BKM and the lambda function in patterns will operate on this data structure in parallel. It has following features:

BKM is a constant data structure. After initiation, it cannot be changed. If users want to change data in BKM, users should create a new one. If users want to update several rows in BKM, it may be inefficient to reconstruct a new data structure. But in big data analytic, users do not care about the concrete individual element but the whole data set.

BKM can keep data across matrix patterns, so users can save the result and reuse the result conveniently. In many cases, algorithms have to reuse input data or temporal results, for example, PageRank which has to compute input data in each iteration. Keeping data in the BKM can be more efficient for the usage of next time than to read data from files again. Because it will preserve data in memory and in a good format. For unary patterns, patterns will directly perform functions on BKM in memory.

BKM can reuse data in iterations. For binary patterns, MatrixMap will keep large matrix and shuffle the small matrix. For example, in the iteration, PageRank can reuse the matrix. In each iteration, BKM can keep the large matrix in the iteration. In the processing, it does not need to transmit all data.

BKM is a vector-oriented data structure. Users cannot randomly slice the individual element in the matrix, for example, fetch element located in row 1, column 1. So it does not encourage to do algorithms involved matrix slice, for example, matrix inversion. The data structure is vector-oriented. Users must fetch bulk rows or columns. Although slice may be convenient in some cases, but most of the algorithms and data operations are vector-oriented and power method [CW93] in vectors can be used to solve matrix inversion. In big data, it is costly and rare to slice single element. If users want to slice element, users can

write map pattern on BKM to get the concrete element.

BKM supports key-value data, one key with multiple values in the same data type. One key with one value is the special case. The key can be string or digit. It uses keys to index row or column. Key in strings is more readable and friendly than key in digits. Mathematical matrix is the special case with key and value in digits. It makes indexes of the data by hash functions. Although indexing will make extra costs, it is flexible to query data according to keys. In the case that users do not need to process all data, if the data is indexed, users can filter data according to keys, which is useful in database operations. We store the numerical data in binary formats, so there is no need to do serialization and deserialization.

BKM supports massive data beyond memory. The data structure can automatically manage data between memory and storage. It can form a pipeline which asynchronously fetches data, asynchronously computes data and asynchronously stores data between memory and secondary storage. It preferentially reserves data in memory. If the size of data is more than physical memory, the data structure will distribute parts of data into secondary storage and memorize the data location into location index. When fetching data, BKM gets data location from the index, if data location is not in memory, it will read data from secondary storage.

3.4 Matrix Patterns

MatrixMap provides powerful yet simple parallel matrix patterns. Parallel matrix patterns define the sequence of operations on elements in matrices. Matrix patterns can be programmed by user-defined lambda functions which configure operations of the pattern and will be applied to elements according to their pattern. Mathematical matrix operations are special cases of matrix patterns filled with specific lambda functions. MatrixMap

supports frequently used parallel patterns such as map, reduce and also abstracts parallel patterns from machine learning and graph algorithms, for example, multiply pattern. There are two kinds of matrix patterns. .

- Unary Matrix Pattern: Map, Reduce
- Binary Matrix Pattern: Multiply, Plus, Join

Unary matrix patterns operate on a single matrix, which are frequently used and basic patterns. These patterns apply associate lambda functions to every element on a matrix. Unary matrix patterns like Map and Reduce are well suited for ETL data operations. The typical example is WordCount. Firstly, each line of input is mapped to key-value data, (word, 1), then they will be reduced to numbers of each word.

- *Map* patterns apply a function to every element in each vector of a matrix. The input of Map in the MapReduce is the special case, one key with a vector in one element. This pattern can both map one element to one element or to multiple elements.
- *Reduce* patterns combine all elements in a vector of a matrix into a single element using an associative combiner function. The reduce pattern will be operated on every vector and output is the single element.

MatrixMap provides single Map and single Reduce and optional global Sort. But MapReduce includes Map and Reduce, and a global sort between Map and Reduce, which is a bottleneck in the process. Often, Map or Reduce is needed, but MapReduce forces users to run whole MapReduce model, Map, Sort and Reduce. So Users are more flexible to use unary matrix patterns in MatrixMap.

Listing 3.2: WordCount Code

```
BKM m("wordcount.txt");
m.Map([](string key, string word, Context c){
c.Insert(word, 1);})
  .Sort().Reduce([](string key, Iterable<Int> i, Context c) {
    int sum = 0;
    for (int e: r) {
          sum += e;
    }
    context.Insert(key, sum);
    }
  );
```

The above code defines a BKM, m, to load WordCount data. Then it invokes a Map pattern to map data and sort results. Finally, a Reduce pattern is invoked to count numbers of each word.

Binary matrix patterns operate on two matrices. These matrix patterns are similar but are not limited to mathematical matrix operations. Actually, these are parallel patterns, which define the sequence of combinations of each element between two matrices. The lambda function should be defined into parallel patterns and will be applied to elements according to their patterns.

$Matrix + Matrix$ patterns will apply user-defined lambda functions to every two elements in the same position of two matrices, similar to matrix plus. Mathematical Plus is to plus two corresponding elements, illustrated in Figure 3.1. It maps each vector from each matrix to a computing node, then run lambda function on each element in the two vectors. Users can define lambda function to perform mathematical plus in this pattern. Users can write other lambda functions in this pattern instead. For example, lambda functions can compare two elements to merge two graphs or minus two elements to implement matrix

minus operation.

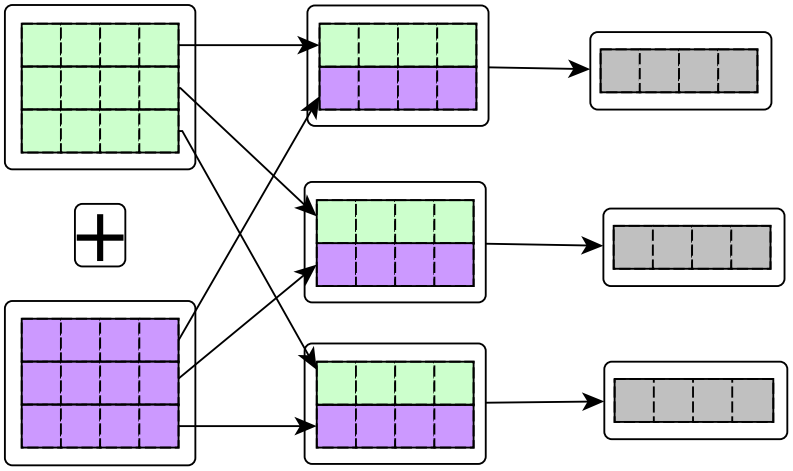

Fig. 3.1: Matrix Plus Pattern

Graph merge algorithm is to merge graph A and graph B to create graph C: Edges are created in graph C, if any of its vertices exist in graph A or graph B. We formulate the algorithm as matrix plus, $C = A + B$ with a lambda function to compare two elements in graphs.

Listing 3.3: Graph Merge

```
BKM A("a.data");
BKM B("b.data");
BKM C = A.Plus(B,
  [](float a, float b){
  if (a != 0) return a;
  else if(b != 0)return b;
  else return 0;
  }
);
```

Codes define BKM of graph A and graph B, and load their data. The BKM A uses plus pattern to merge graph B. The lambda function receives two input, if one element is not zero, then return the element, otherwise it returns 0.

$Matrix \times Matrix$ patterns will apply user-defined lambda functions to combinations of every row and every column from two matrices, similar to mathematical matrix multiplication, illustrated in Figure 3.2. The pattern is in two stages. Firstly, map this pair of row and column to a vector. Then reduce result vector to a single element. Mathematical multiplication is the special case that is to add each pair of elements in a pair of row and column and sum up the result. It is a common operation on large graphs, used in graph contraction, peer pressure clustering, all-pairs shortest path algorithms, and breadth-first search from multiple source vertices. Particularly, in all pair shortest path algorithm, the lambda function is to find the minimum value of the sum of two elements.

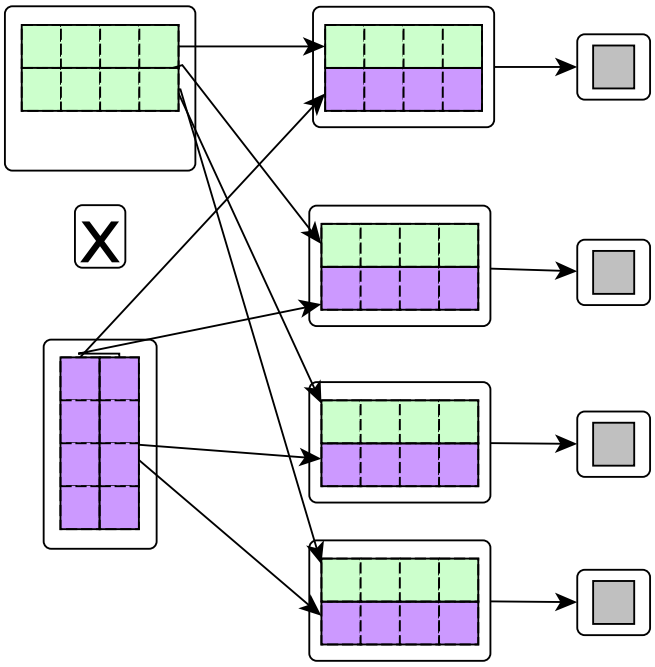

Fig. 3.2: Matrix Multiply Pattern

All Pair Shortest Path [Cor09] is to find the shortest paths between all pairs of vertices in a graph. The all-pairs shortest paths problem for unweighted directed graphs could be solved by a linear number of matrix-matrix multiplications. This is a dynamic-programming algorithm. Each major loop of the dynamic program will invoke an operation that is similar

to matrix-matrix multiplication. The lambda function is to find the minimum value between the sum of the two elements and input element.

Listing 3.4: All Pair Shortest Path

```
BKM W("graph.data");
int iteration = W.GetRows();
for(int i = 0; i < n - 1 ; i = 2*i){
 W = W.Multiply(W,
  [](float x, float y) {
   return min(x+y, x);
  }
  );
}
```

Codes define BKM W and initialize with graph data name. The data will be automatically loaded. Then the iteration number is set in the loop. In the for loop, the BKM W will multiply itself. The lambda function in the multiply pattern returns the smaller value between the addition of the two input and first input.

$Matrix \times Vector$ patterns are special cases of the matrix and matrix multiplication pattern. The right part does not limit to a single vector, but can be small matrices. It is the most widely used matrix operation, since it is the workhorse of iterative linear equation solvers and eigenvalue computations. Many algorithms can be formulated into Matrix Vector Multiplication. For example, PageRank algorithm, Breadth-first search algorithm, Bellman-Ford shortest paths algorithm, and Prim's minimum spanning tree algorithm. [KG11]

PageRank [L P99] is an algorithm used by Google Search to rank websites in their search engine results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. Each link's vote is proportional to the importance of its source page. Given a web graph with N nodes, where the nodes are

pages and edges are hyperlinks. We load the data into adjacency matrix M. We have a rank vector r with an entry per page. We can formulate the PageRank into the flow equation in the matrix form: $r = M * r$.

Listing 3.5: PageRank

```
BKM M("web.graph");
BKM r_new, r_old;
int iterations = 100;
for(int i = 0; i < iterations; ++i){
 r_new = M.Multiply(r_old);
 r_old = r_new;
}
```

Codes define BKM M and initialize with graph data name. r_new, and r_old are rank vectors in $1 \times N$. We define iteration number as 100. We set the multiplication pattern in the for loop.

Matrix Join Matrix patterns combine vectors from two matrices, illustrated in Figure 3.3. It is abstracted from database operation Join, which is a common operation in the database. Join operations include inner join, left join, outer join, right outer join, full outer join and cross join. Users can implement different Join operation with different lambda functions. This pattern selects return result from element combinations between two matrices.

An inner join [Wik15] requires each record in the two joined tables to have matching records and is a commonly used join operation in applications. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. The result of the join can be defined as the outcome of first taking the Cartesian product of all records in the tables (combining every record in table A with every record in table B) and then returning all records which satisfy the join

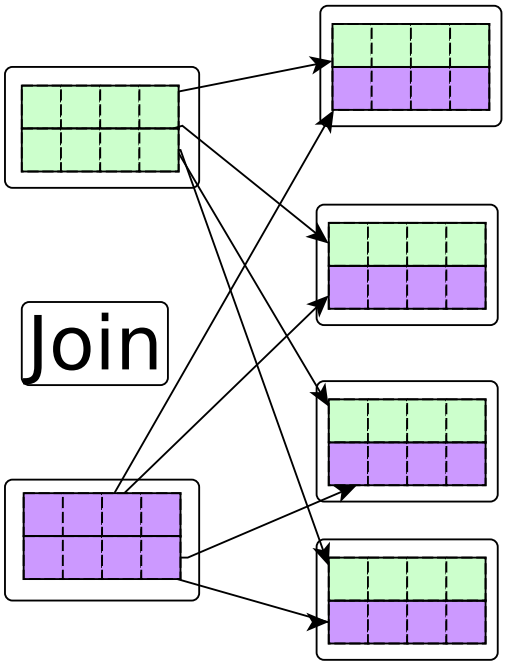

Fig. 3.3: Matrix Join Pattern

predicate.

Listing 3.6: Inner Join

```
BKM matrix1("matrix1.data");
BKM matrix2("matrix2.data");
matrix1.Join(matrix2,
  [](float key1, float key2) {
  return key1 == key2;
  }
);
```

Codes define BKM matrix1, matrix2 and initialize with matrix data name. The data will be automatically loaded. Matrix1 uses join pattern to join matrix2. The lambda function of Join pattern is to return true, if the two inputs are the same.

3.5 Summary

Existing systems are cumbersome and inefficient to program general matrix computation in parallel manner for big data analytics. We present MatrixMap, a unified and efficient data-parallel system for general matrix computations. MatrixMap provides powerful yet simple abstraction, consisting of a distributed data structure called bulk key matrix and a computation interface defined by matrix patterns. Users can easily load data into bulk key matrices and program algorithms into parallel matrix patterns.

Chapter 4

The MatrixMap Framework

MatrixMap framework is the implementation of the corresponding MatrixMap programming model. This framework can process data in parallel on a shared nothing cluster with multi-cores support. We use Intel Threading Building Blocks (TBB) [tbb14] to implement matrix patterns. TBB helps programmers easily write parallel C++ programs that take full advantage of multi-core performance. We use ZeroMQ [Hin13] to do communication between machines. We implement the framework in C++ language and take advantages of lambda functions, a new feature in C++ 11. We use template and metaprogramming technique to support multiple kinds of data.

4.1 System Architecture

The framework is a typical master and slave system as depicted in Figure 4.1. As C++ is a compiled language, which cannot dynamically load codes like interpreted languages, so MatrixMap framework cannot dynamically load users' program code. Users' driver program codes should be compiled before submitting. The driver program is also in slave and master mode. Framework master will run driver program and framework slaves will run worker program.

Data of BKM cross around machines in the cluster. When a parallel pattern in the users' program is invoked on a BKM, MatrixMap creates and sends tasks to process each

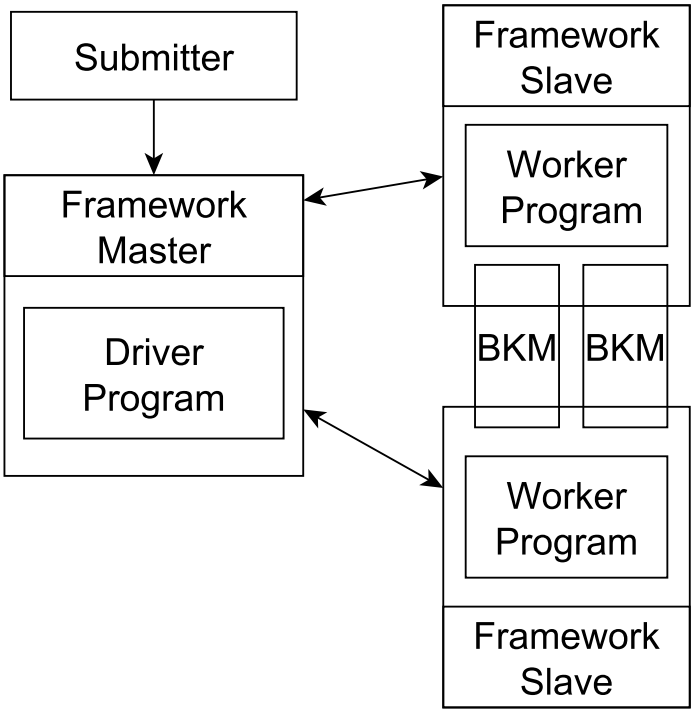

Fig. 4.1: Distributed Framework

partition of the BKM on slaves. Slaves will compute each partition of data in parallel. After all computation of a pattern, the master will start another pattern.

Figure 4.2 illustrates the flowchart of the MatrixMap framework. The framework will input data and partition data for the Map phase. Map phase takes an input pair and produces a set of intermediate key/value pairs. The framework collects all intermediate values associated with the same intermediate key and passes them to the Reduce phase. The Reduce phase accepts an intermediate key and a set of values for that key and merges these values together.

4.2 Bulk Key Matrix Implementation

BKM reserves data into memory and store unused data into secondary storage. If there is not enough memory for one BKM, it will store parts of matrix data into secondary storage.

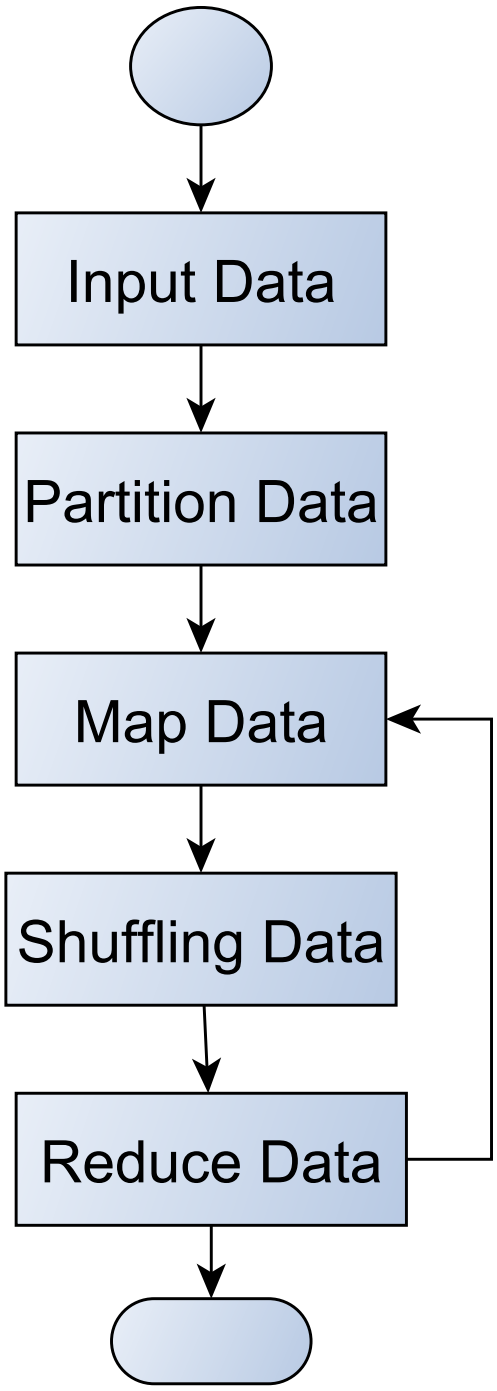

Fig. 4.2: Distributed Framework

If data is stored into secondary storage, data will never be deleted. It assumes that there is infinite capacity for secondary storage and is costly to write data into secondary storage. It will remember keys of data both in memory and storage with location information in a map. BKM supports all kinds of data types, if the data types do not belong to basic data types, users should write serialization function.

BKM makes **efficient memory management**. It prefers to store data into memory. In memory part, there is an object manager. It will create continuous memory block for each vector in BKM. So it is efficient to apply parallel functions on BKM with memory coherence. When there are not enough memory for data, it will save data in the storage via RocksDB. We do not write our own persistent part, but use RocksDB instead. RocksDB is an embeddable persistent key-value store for fast storage. MatrixMap will asynchronously store data into secondary storage via RocksDB [Roc15]. It will set up a database for the data. Each BKM will be stored into each column file in the database. Although a random query is faster in hash-based tree format, but BKM often iterate all data. So MatrixMap uses tree-based file format to store data.

BKM makes **efficient cache** across memory and secondary storage according to matrix patterns. The cache algorithm is vector-oriented, not Least Recently Used(LRU) [OOW93] in many systems. Many matrix operations are based on rows or columns. So our cache prefetches the following vectors and those not frequently-used elements.

4.3 Matrix Patterns Implementation

For the implementation of matrix patterns, since TBB directly provides some basic patterns, like Map, Reduce, Sort. We directly use these patterns in TBB. Plus pattern is built on top of Map pattern of TBB. One dimension plus pattern is to map two arrays simultaneously. The two dimensions plus pattern is based on one dimension, which iterates all rows

data with one dimension Plus pattern. Multiply pattern is built on top of Map pattern and Reduce pattern of TBB. One dimension multiplication pattern is dot multiplication which maps two arrays to a vector, then reduces result vector. Two dimensions pattern is to map each pair of row and column with one dimension multiplication pattern.

MatrixMap conforms to the **synchronous computation mode**, the bulk synchronous parallel (BSP) model [Val90]. The BSP model proceeds in a series of global super steps which consist of three ordered stages: Computation, Communication and Barrier synchronization. One matrix pattern is a kind of super step. In one matrix pattern, it will create tasks to computes each element in parallel. The length of the matrix is the barrier. Without finishing all rows in matrices during a pattern, it can not process another matrix patterns. Since matrices have enough rows for computation, synchronous model also can fully utilize computation resources.

MatrixMap has the **asynchronous computation pipeline** to asynchronously fetch data, compute data and store data, although MatrixMap follows synchronized BSP model. We use asynchronous input queues to decouple data fetch and data computation and use asynchronous output queue to decouple data computation and data storage. Asynchronous queues are lock-free concurrent data structures. MatrixMap will fetch data from asynchronous input queues and create parallel tasks to compute data in parallel. Then store results to asynchronous output queues. Consequently, MatrixMap fully leverages multi-core CPUs and other computation resources.

Figure 4.3 is an asynchronous computation pipeline of a Map pattern. Firstly, the Map pattern can asynchronously get data from an asynchronous input queue. Then the Map pattern does parallel tasks on these data. Thirdly, Map pattern outputs the result to an asynchronous output queue. If there is not enough memory for data, BKM will move parts of data to disk. In the whole process, data are transmitted by pointers.

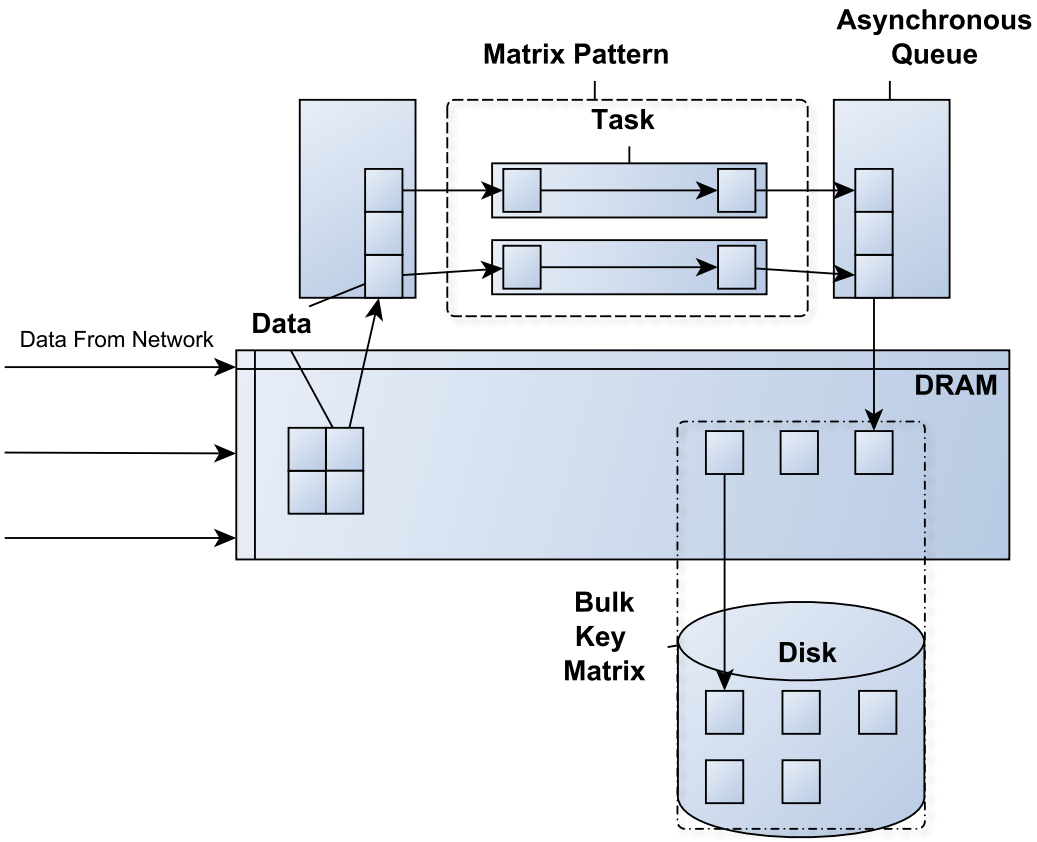

Fig. 4.3: Asynchronous Computing Process

MatrixMap will make **parallel data partition**. For unary patterns, there is one input and it is easy to partition data in vectors into asynchronous input queues, then send data to each slaves nodes for computation. For binary patterns, MatrixMap will simultaneously partition two inputs and has two kinds of cases for sending data. In matrix and vector case, it will send the vector first and then start computation when one row of a matrix is coming. In matrix and matrix case, MatrixMap will send pairs of vectors in each matrix to each slave and compute each pair of vectors.

MatrixMap has **efficient data shuffling strategies** for different matrix patterns, which are much more efficient than the strategy of MapReduce. It will send important data first and send less redundant data in the data shuffling.

For Matrix and Vector Multiplication pattern, which has M rows matrix and 1 column vector and runs on C machines, it transmits C copies of the column vector and M rows matrix data separately assuming that C is much smaller than M. In MapReduce, it will send M rows and M columns in data shuffling.

For Matrix and Matrix Multiplication pattern, we simultaneously send data from two matrices. For M rows matrix and N columns matrix with D computing nodes, it sends a couple of columns to D computing nodes, then it sends D copies of first matrix data to each computing nodes separately. We only need to send $D * M + N$, which is much less than those in MapReduce. MapReduce needs to send $M \times N$ data.

For Matrix Plus pattern, it will send pairs of vectors in the same position in each matrix simultaneously, so it can start computing immediately. For Matrix Join pattern, it will send pairs of vectors in each matrix simultaneously.

4.4 Fault Tolerance

Since MatrixMap is designed to process huge amounts of data using hundreds or thousands of commodity machines, the framework must tolerate machine failures gracefully. The framework has implemented fault tolerance mechanism and can quickly recover from the former matrix patterns. When encountering failures, it does not need to repeat the application at the start. It can reserve temporal accomplishment and start at the former matrix patterns.

To make fault tolerance, matrix data and matrix patterns will be updated into secondary storage periodically. If MatrixMap fails, it can recover data and data patterns from secondary storage. MatrixMap will write pattern logs into secondary storage after finishing a pattern. When recovering from a failure, it reads logs and start from current pattern.

4.5 Optimization for Sparse Matrix Computation

A sparse matrix is a matrix in which most of the elements are zero. Large sparse matrices often appear in big data applications. MatrixMap adopts compressed sparse row format(CSR)[KKGK11] for sparse matrix and optimizes computing engine for CSR matrix computation. So MatrixMap can support both dense matrices and sparse matrices.

Compressed Sparse Row(CSR) consists of value, column index, row pointer, where value is an array of the (left-to-right, then top-to-bottom) non-zero values of the matrix; column index is the column indices corresponding to the values; and, row pointer is the list of value indexes where each row starts. This format is efficient for arithmetic operations, row slicing, and matrix-vector products.

SIMD Computation via BLAS has been introduced to speed up numerical computation. The Basic Linear Algebra Subprograms (BLAS) are a specified set of low-level subroutines that perform common linear algebra operations such as copying, vector scaling,

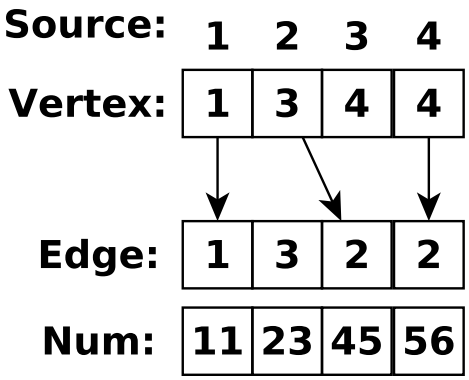

Fig. 4.4: CSR Format

vector dot products, linear combinations, and matrix multiplication via Single Instruction Multiple Data instruction(SIMD) of CPUs. It uses BLAS subroutine as lambda function on corresponding matrix patterns, for example, Multiply. Users do not need to write specific lambda function for mathematical matrix operations. MatrixMap supports such operations in default.

4.6 Optimization for Graph Algorithms

To support graph algorithms, we introduce a new data format, Key-CSR to improve the performance of the framework when processing graphs. And we introduce frequently used graph operations for users.

CSR (Compressed Sparse Row) format is almost identical to the adjacency matrix representation of a directed graph [SC96]. However, it has much less overhead and much better cache efficiency. Instead of storing an array of linked lists as in the adjacency list representation, CSR is composed of three arrays that store whole rows continuously. The first array, vertex array, stores the row pointers as explicit integer values, the second array, edge array, stores the column indices, and the last array, number array, stores the actual numerical values. Those column indices stored in the edge array indeed come from concatenating the

edge indices of the adjacency lists.

We introduce **Key-CSR** format into our BKM, vertex array as key, edge array and value array as values in the matrix. It is similar to adjacency list, but there is no link between elements in the value of the key matrix. It is more efficient than CSR. It does not need to store the end position of each vertex and does not need to store a vertex with no neighboring vertices. There is no vertex array to store the beginning of edges. The element can be easily located like locating the element in the array. So it can take advantage of memory cohesion.

Besides basic matrix patterns for graph algorithms, MatrixMap provides frequently used graph operations for graph algorithms, which is the combination of basic matrix patterns with specific lambda functions.

Graph contraction can be formulated into sparse triple product. The contraction of a pair of vertices v_i and v_j of a graph produces a graph in which the two nodes v_1 and v_2 are replaced with a single node v such that v is adjacent to the union of the nodes to which v_1 and v_2 were originally adjacent. [KG11]

Subgraph extraction can be formulated into a sparse triple product. A subgraph, H, of a graph, G, is a graph whose vertices are a subset of the vertex set of G, and whose edges are a subset of the edge set of G. [KG11]

Graph join (or complete join) of two graphs is their graph union with all the edges that connect the vertices of the first graph with the vertices of the second graph. It is a commutative operation (for unlabelled graphs) [KG11]

4.7 Summary

MatrixMap framework is the implementation of the corresponding MatrixMap programming model. This framework can process data in parallel on a shared nothing cluster with

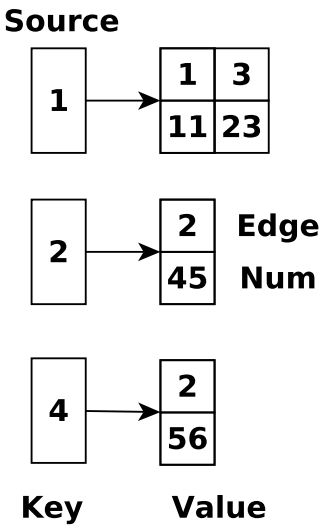

Fig. 4.5: Key-CSR Format

multi-cores support. It outperforms currents systems by employing three key techniques: matrix patterns with lambda functions for irregular and linear algebra matrix operations, asynchronous computation pipeline with optimized data shuffling strategies for specific matrix patterns and in-memory data structures reusing data in iterations.

Chapter 5

Implementation of Example Applications on MatrixMap

Many machine learning and graph algorithms can be implemented by operations on matrices. MatrixMap not only supports Extract-Transform-Load [Vas] like MapReduce, but also supports complex and dense computation algorithms, machine learning and graph algorithms. Users can easily express various algorithms in MatrixMap that are difficult or inefficient to implement in current models. We implement several typical algorithms in this framework and we can see codes in MatrixMap are similar to sequential codes from listings below.

5.1 Extract-Transform-Load

It is important to clean data with ETL. In computing, Extract, Transform and Load (ETL) refers to a process in database usage:

- Extracts data from homogeneous or heterogeneous data sources
- Transforms the data for storing it in proper format or structure for querying and analysis purpose
- Loads it into the final target.

Unary patterns and Join pattern can support various kinds of ETL operations.

5.1.1 Word Count

Although word count is simple, yet it is a frequently used algorithms in the search engine to build inverted index. The following codes define a BKM, m, to load WordCount data. Then invoke a Map pattern to map data and sort results. Finally, invoke a Reduce pattern to count numbers of each word.

Listing 5.1: WordCount Code

```
BKM m("wordcount.txt");
m.Map([](string key, string word, Context c){
 c.Insert(word, 1);})
 .Sort().Reduce([](string key, Iterable<Int> i, Context c) {
 int sum = 0;
 for (int e: r) {
  sum += e;
 }
 context.Insert(key, sum);
 }
);
```

5.1.2 Inner Join

An inner join requires each record in the two joined tables to have matching records and is a commonly used join operation in applications. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. The result of the join can be defined as the outcome of first taking the Cartesian product of all records in the tables (combining every record in table A with every record in table B) and then returning all records which satisfy the join predicate.

Listing 5.2: Inner Join

```
BKM matrix1("matrix1.data");
BKM matrix2("matrix2.data");
matrix1.Join(matrix2,
 [](float key1, float key2) {
 return key1 == key2;
});
```

Codes define BKM matrix1, matrix2 and initialize with matrix data name. The data will be automatically loaded. Matrix1 uses join pattern to join matrix2. The lambda function of Join pattern is to return true when the two inputs are the same.

5.2 Machine Learning Algorithms

Many machine learning algorithms are based on matrix operations. Matrix parameters can be used to learn interrelations between features. Matrix operations provide computation engine for the majority of machine learning algorithms.

5.2.1 Logistic Regression

Logistic regression is to predict a binary response from a binary predictor, used for predicting the outcome of a categorical dependent variable (i.e., a class label) based on one or more predictor variables (features). It will take all features and multiply each one by a weight and then add them up. This result will be put into the sigmoid function, and it will get a number between 0 and 1. If the number is above 0.5, it will get a 1, else it will get a 0. [Har12]

Listing 5.3: Logistic Regression

```
BKM data("points.data");
BKM weights,label,error;
BKM temp;
int iterations = 100;
for (int i = 0; i < iterations; ++i) {
 temp = data.Multiply(weights)
 float h = sigmoid(temp);
 error = label.Plus(h);
 temp = data.Multiply(error);
 temp = temp.Multiply(alpha);
 weights = temp.Plus(weights);
}
```

Codes define BKM data and initialize with data name. The data will be automatically loaded. Define vector weights, label, error by BKM. Then defines the iteration number of the loop. In the loop, we formulate the algorithm into matrix-vector multiplication.

5.2.2 K-Means

K-means clustering aims to partition n points into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. It can be formulated into Map and Reduce in iterations. The input matrix with each point as a row. Center matrix with each central point as a column. Since the centroid data is much smaller than point data, we store the centroid data into shared variable. In the Map, we each point multiplies the centroid. Then send centroid with the smallest distance and corresponding key. Finally, find the means of clusters to get new centroids.

Listing 5.4: K-Means

```
BKM point("points.data");
BKM centroids;
int iterations = 100;
for (int i = 0; i < iteraions; ++i) {
 point.Map([](string key, vector<double> point, Context c) {
 BKM temp = point.Multiply(centroids);
 int index = min_index(temp);
 c.Insert(index, point);
}).Reduce([](string key, Iterable<double> i, Contex c){
 centroids = c.insert(key, average(point)).Dump();
});
}
```

5.2.3 Alternating Least Squares

Alternating Least squares (ALS) decomposes matrix for collaborative filter problems, such as predicting users' ratings for movies according to other users' historical ratings. ALS is computation-intensive rather than data-intensive. $R = U * M + E$. We have to decompose R into U and M. We use EM method to compute U and M in iteration: [ZWSP08]

1. Initialize M with a random value.

2. Solve U given M to minimize error on R

3. Solve M given U to minimize error on R

4. Repeat steps 2 and 3 until a stopping criterion is satisfied.

Listing 5.5: Alternating Least Squares

```
BKM m("r.data");
BKM u, r, error;
int iteration 100;
for (int i = 0; i < iterations; ++i ) {
 BKM temp = m.Multiply(u);
  error = r.Plus(temp);
  temp = m.Multiply(error);
  temp = temp.Multiply(alpha);
  u = temp.Plus(u);

  temp = u.Multiply(m);
  error = r.Plus(temp);
  temp = u.Multiply(error);
  temp = temp.Multiply(alpha);
 m = temp.Plus(m);
}
```

5.3 Graph Algorithms

Many graph algorithms can be implemented by operations on the adjacency matrix: Breadth-first or depth-first search can be formulated into matrix-vector multiplication; Graph merge can be formulated into matrix-matrix plus; Breadth-first or Depth-first search to or from multiply vertices simultaneously can be formulated into matrix-matrix multiplication.

5.3.1 Breadth-First Search

Breadth-first search can be performed by multiplying a sparse matrix G with a sparse vector x. To search from node i, we begin with $x(i) = 1$ and $x(j) = 0$ for $j \neq i$. Then $y = G^T * x$ picks out row i of G, which contains the neighbors of node i. Then multiplying

y by GG^T gives nodes two steps away, and so on. We solve the BFS problem using level synchronization. BFS traverses the graph in levels; once a level is visited it is not again and processes each level of the BFS in parallel. The number of iterations required is equal to the (unweighted) distance of the furthest node reachable from the starting vertex, and the algorithm processes each edge at most once.[SB13]

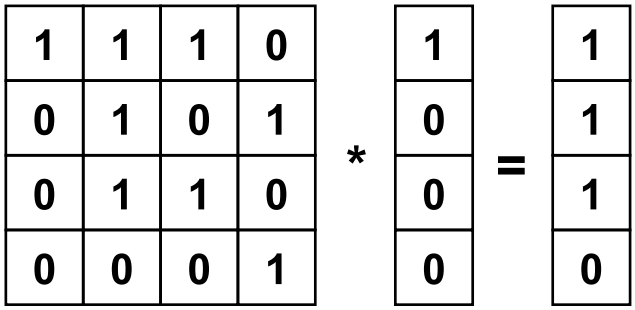

Fig. 5.1: Breadth-First Search in Matrix Operations

Listing 5.6: Breadth-first Search

```
BKM graph("graph.data");
BKM trace;
graph.Multiply(trace);
```

5.3.2 Graph Merge

Graph merge algorithm is to merge graph A and graph B to create graph C: Edges are created in graph C if any of its vertices exist in graph A or graph B. We formulate the algorithm as matrix plus, $C = A + B$ with a lambda function to compare two elements in graphs.

Listing 5.7: Graph Merge

```
BKM A("a.data");
BKM B("b.data");
BKM C = A.Plus(B,
  [](float a, float b){
   if (a != 0) return a;
   else if(b != 0)return b;
   else return 0;
  }
);
```

Codes 5.7 define BKM of graph A and graph B, and load their data. The BKM A uses plus pattern to merge graph B. The lambda function receives two input, if one element is not zero, then return the element, else return 0.

5.3.3 All Pair Shortest Path

All Pair Shortest Path [Cor09] is to find the shortest paths between all pairs of vertices in a graph. The all-pairs shortest paths problem for unweighted directed graphs could be solved by a linear number of matrix-matrix multiplications. This is a dynamic-programming algorithm. Each major loop of the dynamic program will invoke an operation that is very similar to matrix-matrix multiplication. The lambda function is to find the minimum value between the sum of the two elements and input element. So the algorithm will look like repeated matrix multiplication. Now we can see the relation to matrix multiplication. Suppose we wish to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for i, j = 1,2,..., n, we compute

$$
\begin{aligned}
l_{ij}^{(m)} &= min(l_{ij}^{(m-1)}, \min_{1 \le k \le n})\{l_{ij}^{(m-1)} + w_{kj}\} \\
l_{ij}^{(m)} &= \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{ik}\}
\end{aligned}
\tag{5.1}
$$

Taking as our input the matrix $W = (w_{ij})$, we now compute a series of matrices $L^{(1)}, L^{(2)}, ..., L^{n-1}$, where for m = 1,2,...,n - 1, we have $L^m = (l_{ij}^{(m)})$. The final matrix $L^{(m-1)}$ contains the actual shortest-path weights. Observe that $l_{ij}^{(1)} = w_{ij}$ for all vertices $i, j \in V$, and so $L^{(1)} = W$. The core of the algorithm is the following procedure, which, given matrices $L^{(m-1)}$ and W, returns the matrix $L^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

$$
\begin{aligned}
&L^{(1)} = L^{(0)}.W = W, \\
&L^{(2)} = L^{(1)}.W = W^{(2)} \\
&L^{(n-1)} = L^{(n-2)}.W = W^{(n-1)}
\end{aligned}
\tag{5.2}
$$

Listing 5.8: All Pair Shortest Path

```
BKM W("graph.data");
int iteration = W.GetRows();
for(int i = 0; i < n - 1 ; i = 2*i){
 W = W.Multiply(W,
   [](float x, float y) {
          return min(x+y, x);
   }
  );
}
```

Codes 5.8 define BKM W and initialize with graph data name. The data will be automatically loaded. Then set the iteration number in the loop. In the for loop, the BKM W will multiply itself. The lambda function in the multiply pattern is to return the smaller value between the addition of the two input and first input.

5.3.4 PageRank

PageRank [L P99] is an algorithm used by Google Search to rank websites in their search engine results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. Each link's vote is proportional to the importance of its source page. Given a web graph with n nodes, where the nodes are pages and edges are hyperlinks. We load the data into adjacency matrix M. We have a rank vector r with an entry per page. We can formulate the PageRank into the flow equation in the matrix form: $r = M * r$.

Codes define BKM M and initialize with graph data name. r_new, and r_old is rank vector in $1 \times N$. We define iteration number as 100. We set the multiplication pattern in the for loop.

Listing 5.9: PageRank

```
BKM M("web.data");
BKM r_new, r_old;
int iterations = 100;
for(int i = 0; i < iterations; ++i){
 r_new = M.Multiply(r_old);
 r_old = r_new;
}
```

5.4 Evaluation

Although our implementation of MatrixMap is still at an early stage, the experiment results demonstrate that it is a promising cluster computing framework. All of the experiments were performed on a cluster of 10 nodes machines with CPU Intel Xeon E5-2630 v2 2.6G, RAM 8G, HDD 150G. The parallel programs were compiled with Intel's TBB (version 4.2). The programs were compiled using g++ 4.8.3 with the -O2 flag.

We compare MatrixMap with a data-parallel system called Spark, a graph-parallel system called GraphX and a matrix computation system called ScaLAPACK. Apache Spark is a fast and general engine for large-scale data processing, which run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Because Spark is faster than Hadoop MapReduce, we compare MatrixMap with Spark for machine learning algorithms. GraphX is Apache Spark's API for graphs and graph-parallel computation. Because GraphX is faster than Graphlab, we compare MatrixMap with GraphX for graph algorithms. ScaLAPACK [CDPW92] is a library of high-performance linear algebra routines for parallel distributed memory machines.

According to the experiment result, MatrixMap is 12 times faster than Spark and outperforms ScaLAPACK by 30%, especially for iterative algorithms which reuse data. We roughly cut the time into two stages, input stage and computation stage. The input stage is the first iteration and computation stage is the remain iterations. Due to the asynchronous computation pipeline, both stages include data input and data computation.

We evaluate framework in different data size with non-iterative algorithms, graph merge and all pair shortest path with one iteration. They have input stages and no computation stages. We use GraphX's implementation as a baseline. We test algorithms with data size in 4k(2M byte), 81k(6.5M byte), 875k(62M byte), 1.9m(71M byte), 3.9m(203.5M byte) graph nodes. GraphX in Spark project is a message passing method [Sni95] with fixed topology in the graph. GraphX has nothing to do with graph algorithms, which do not send messages through graph topology, for example, all pair shortest path algorithm and Floyd-Warshall algorithm. Additionally, vertices will flood messages through the graph, but not all vertices need all their neighbor vertices' information. The problem is that it takes longer time to converge. GraphX uses three RDDs to represent a graph, which will cost much memory. Before processing, it has to take much time to cut graph into vertex-cut. With vertex-cut, it

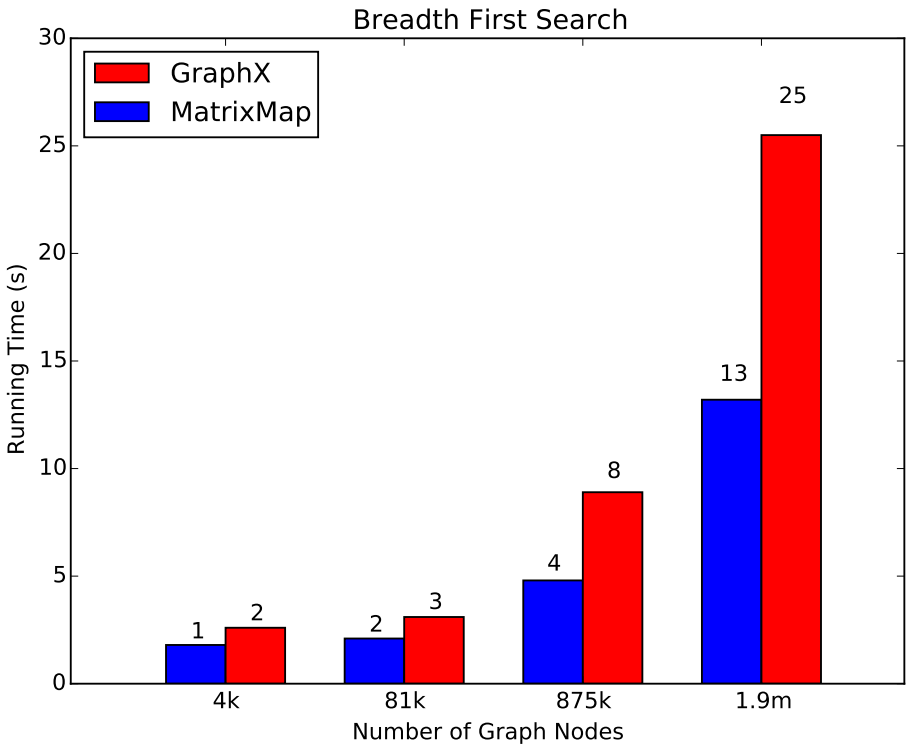

Fig. 5.2: Breadth-First Search Run time

cannot take advantage of BLAS to speed up computation. Using BLAS, Graph algorithms in the matrix are more compact and are easier to figure out and have clear data access patterns. Here, MatrixMap takes advantage of BLAS for matrix operations.

Breadth-First Search can be formulated into matrix-vector multiplication, but not a typical mathematical matrix operation. In each iteration, parts of rows will be visited, do one dot multiplication. In one dot multiplication, it can visit a couple of vertices at one time. But GraphX has to visit each vertex through its edge separately. Additionally, GraphX will cost much time to cut graph into partitions.

Graph Merge can be formulated into matrix-matrix plus. MatrixMap will send data in each matrix simultaneously and asynchronously compute each row. GraphX uses the outerJoinVertices operation to merge two graphs. It will cost much time to compare and

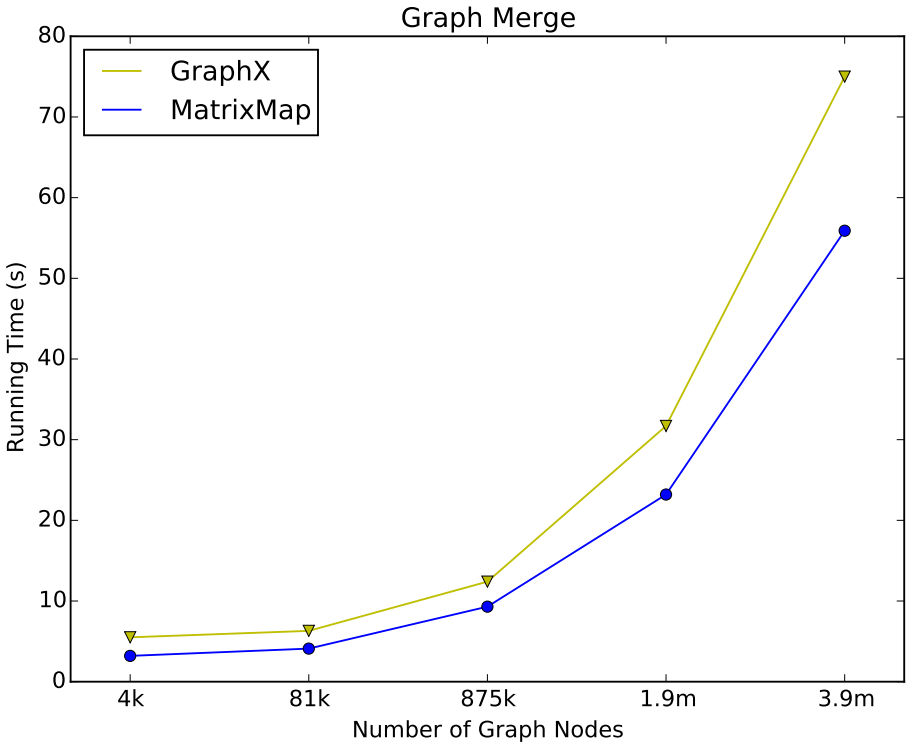

Fig. 5.3: Graph Merge Run Time

find the corresponding key in each graph. MatrixMap costs much less time than GraphX.

All Pair Shortest Path can be formulated into matrix-matrix multiplication. We use GraphX's implementation to compare its performance with MatrixMap. GraphX cannot implement all pair shortest path in dynamic algorithm paradigm, but runs single shortest path on every vertex. From algorithm analysis, the dynamic algorithm in MatrixMap costs less than the algorithm in GraphX. This algorithm needs to shuffle matrix data in each iteration. The result shows that MatrxiMap is faster than GraphX.

We evaluate MatrixMap framework with iterative algorithms, logistic regression and PageRank. The input stage costs constant time and in computation stage, every iteration almost cost constant time. In input stage, ScaLAPACK uses less time on loading data than MatrixMap. In computation stage, MatrixMap is quicker than ScaLAPACK gradually.

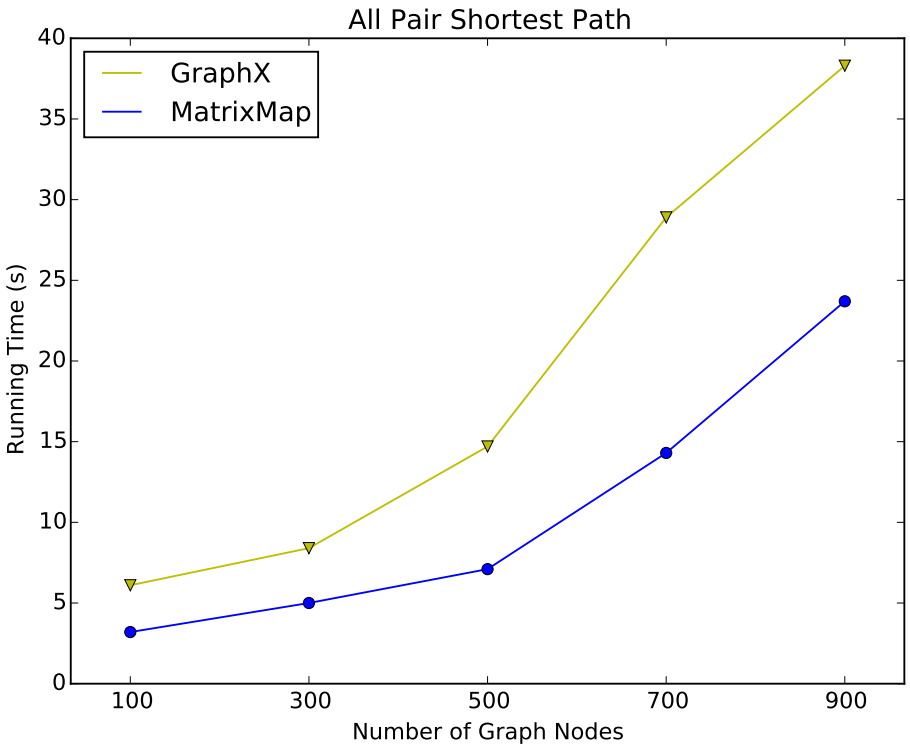

Fig. 5.4: All Pair Shortest Path Run Time

Therefore, MatrixMap is suitable to iterative algorithms. MatrixMap's running time in each iteration keeps constant. BKM can efficiently save and reuse the temporal result and data, so MatrixMap can be gradually faster than ScaLAPACK, especially for iterative algorithms which reuse data.

Logistic regression is formulated into matrix-vector multiplication in iterations. MatrixMap will firstly send the matrix to computing nodes, then send rows of the first matrix and asynchronously compute each row. We use Spark's implementation to compare its performance with MatrixMap. Spark formulates the algorithm into Map and Reduce in iterations. In MapReduce, it will shuffle data between Map and Reduce, which will cost much time. In MatrixMap, after the first iteration which sends data to cluster, the following iterations only need to reconstruct and transmit a vector and can reuse matrix data in the

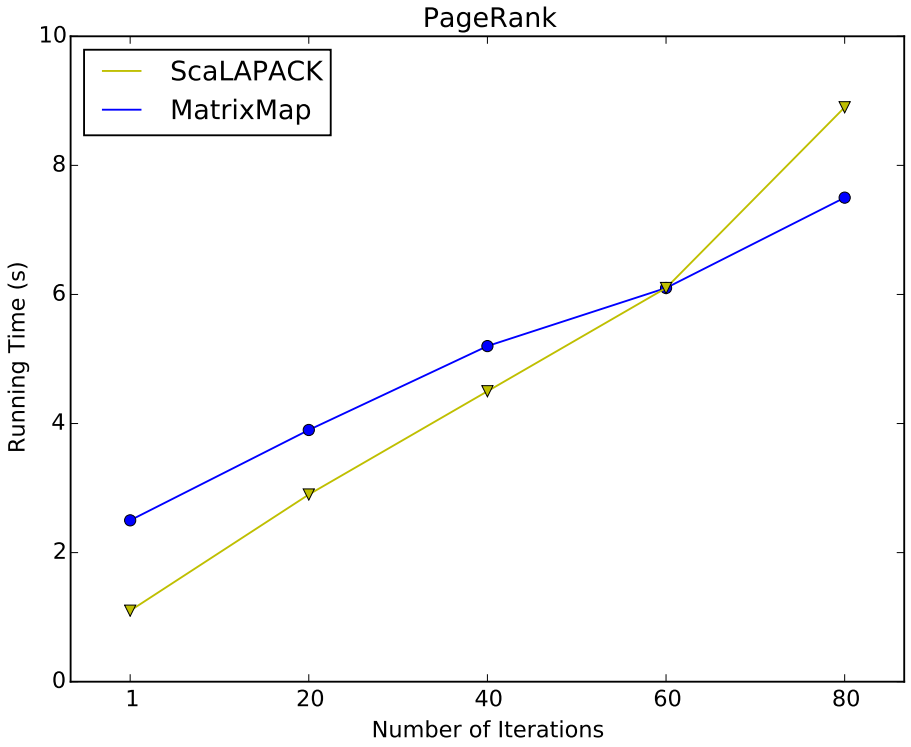

Fig. 5.5: PageRank Run Time

local machine. We run algorithms in several iterations. Experiments show that MatrixMap is much faster than Spark after several iterations for 12 times.

Alternating Least Squares is formulated into two matrix matrix multiplications in iterations. We use Spark's implementation to compare its performance with MatrixMap. In iterations, parts of matrix can be reserved and resued in the local machines. The result shows that MatrixMap is faster than Spark.

We evaluate the computation efficiency of MatrixMap. We use ScaLAPACK's implementation on PageRank as a benchmark. At the beginning of iteration phase, ScaLAPACK is faster than MatrixMap. After 60 iterations, MatrixMap gradually outperforms ScaLAPACK by 30% with the help of optimized data shuffling strategies, asynchronous pipeline directly and in-memory data structure for data reuse. The input matrix in the PageRank

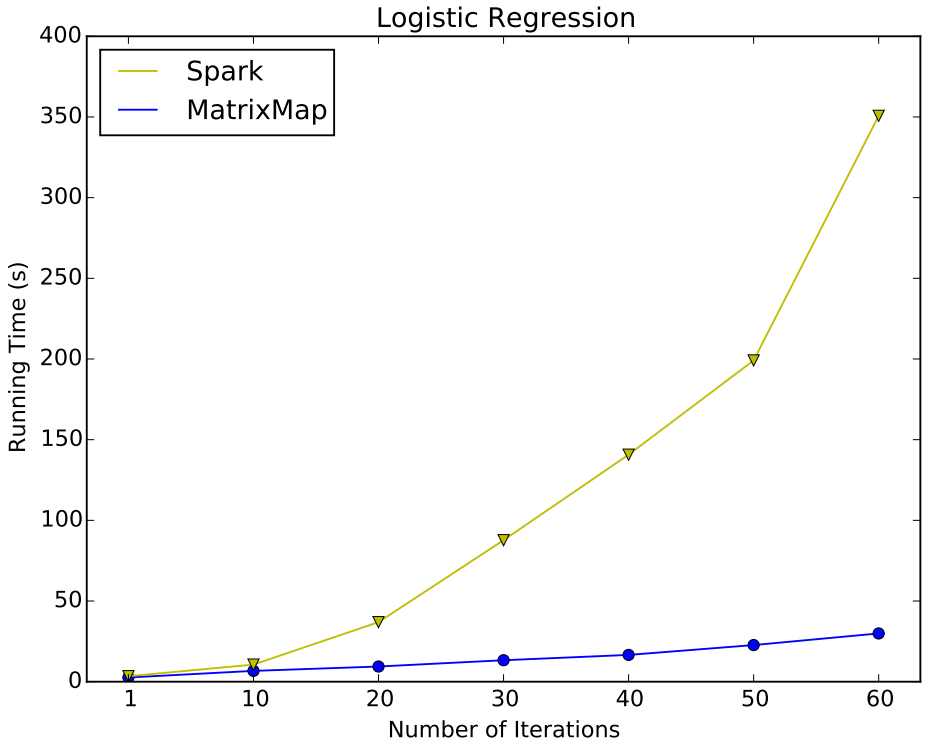

Fig. 5.6: Logistic Regression Run Time

can be reserved in the memory of computing nodes.

We further evaluate the scalability of MatrixMap. In particular, we compare MatrixMap framework with ScaLAPACK in PageRank algorithm with 80 iterations based on different numbers of machines. The experiment shows the running time of both Spark and ScaLAPACK will decrease linearly with respect to the number of machine. They both have good scalability and performance, because matrix data can be easily partitioned linearly according to different computing nodes.

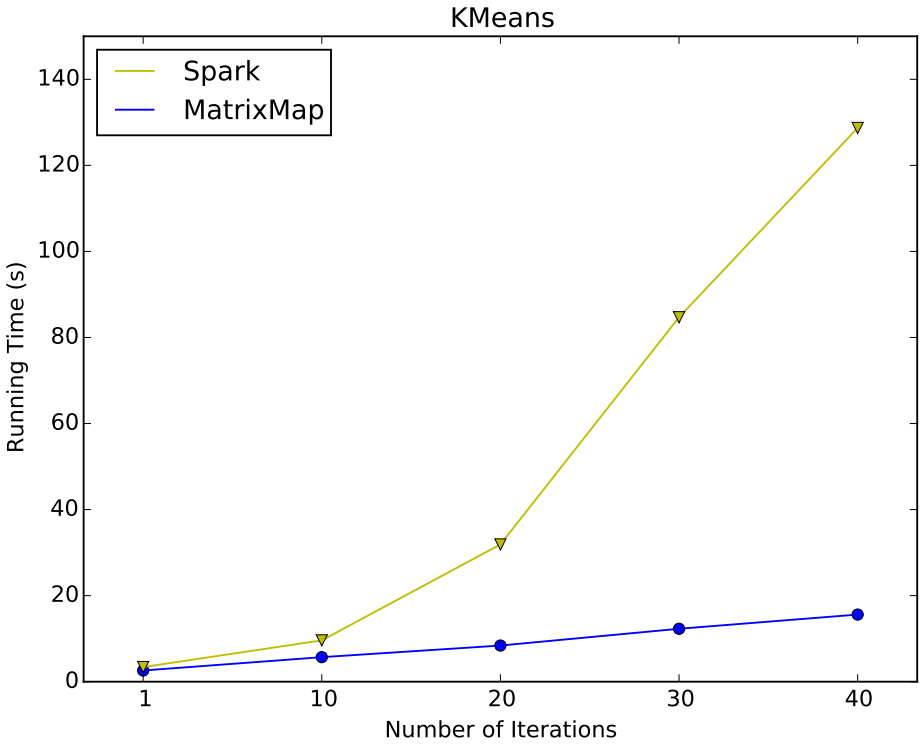

Fig. 5.7: KMeans Run Time

5.5 Discussion

MatrixMap is faster than Spark, GraphX and ScaLAPACK in iterative algorithms and MatrixMap can automatically handle the parallelization and distribute execution of programs on a large cluster. From the above code listings, we can see that codes in MatrixMap are similar to sequential codes. Users can easily load data into bulk key matrices and program algorithms into parallel matrix patterns without considering low-level issues.

MatrxMap not only provides the MapReduce model, also provides parallel matrix patterns for efficient process algorithms. MatrixMap directly provides matrix operations. It sends each row and column to slaves and computes each row and column. In MapReduce and Spark, matrix operations have to be implemented into several Map and Reduce, cutting matrix into key-value data and shuffling data with global sort between Map and Reduce.

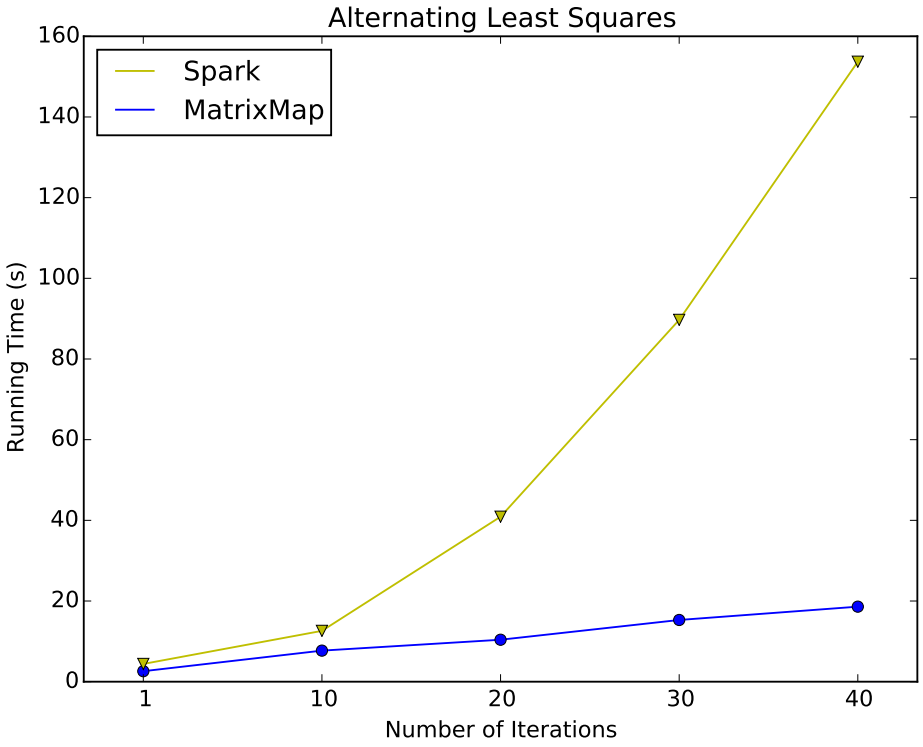

Fig. 5.8: Alternating Least Squares Run Time

It creates many temporal data in the process and has to recombine the data across cluster several times. If two matrices are both in large scale, firstly, it has to do Map and Reduce on each matrix separately into a same key value format in the same file. Then MapReduce has to Map each pair of row and column in the file and does multiplication in Reduce. For matrix-vector multiplication, it puts the vector into shared memory and Maps each row in the matrix with the shared variable. Since shared variables have to update values periodically, it is less efficient.

With **asynchronous computation pipeline**, MatrixMap can asynchronously compute data when there is a pair of row and column in the computing node of the reduce phase. It does not need to wait for the completion of all data shuffling after the map phase. In parallel data shuffling, MatrixMap transmits fewer data in iterations. Different matrix patterns

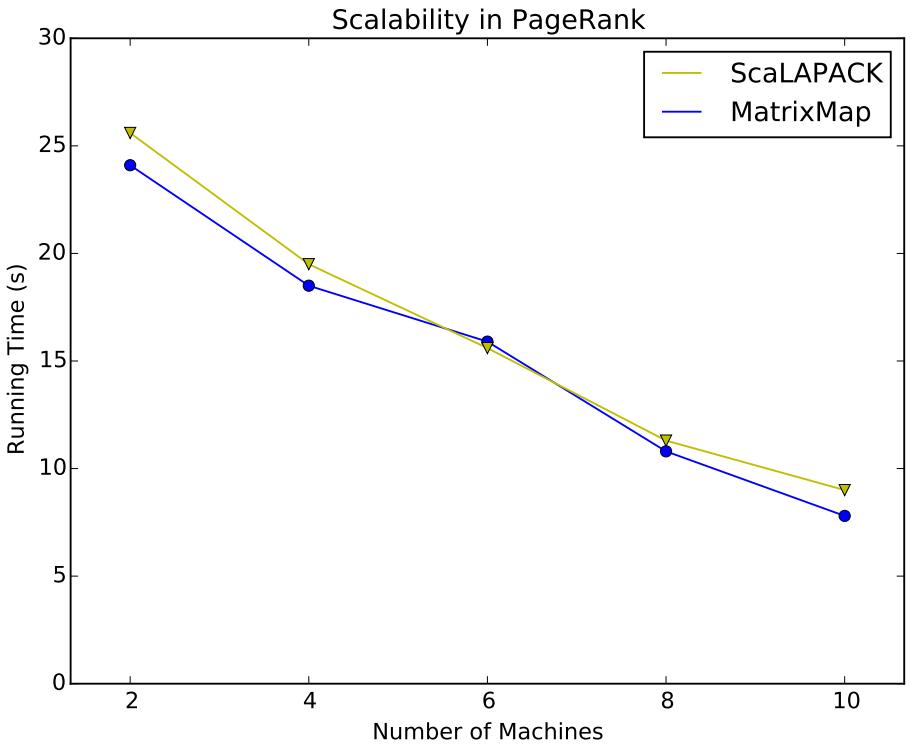

Fig. 5.9: Scalability in PageRank Run Time

have different data transmission strategies, which are much efficient than the strategy of MapReduce. It will first send important data and then send the less redundant data in the transmission.

With **in-memory data structures**, MatrixMap can reuse data in iterations. For example, in PageRank, it is unnecessary to resend input matrix in each iteration, because input matrix keeps constant and only the vector changes. After one iteration, it has already sent constant data across the cluster. Thus, it only needs to send the changed vector data in the iterations.

With **optimized framework** which is implemented in C++ language with lambda function and template to support multiple data type, MatrixMap runs faster than Scala, a interpreted language on JVM used in Spark. We utilize TBB to process data in parallel.

TBB has more advantage over raw threads used by Spark. We use Key-CSR matrix format to greatly boost the performance of graph algorithms, take advantage of BLAS to speed up density matrix operations and provide optimal data transmission strategy for different patterns.

5.6 Summary

We have implemented typical ETL, machine learning and graph algorithms in MatrixMap programming model. Users can easily express various algorithms in MatrixMap that are difficult or inefficient to implement in current models.

Chapter 6

Conclusions and Future Research

In this chapter, we conclude this thesis in Section 6.1 and outline some future works in Section 6.2.

6.1 Conclusions

Machine learning and graph algorithms can be formulated into matrix operations. Current models are cumbersome and inefficient to program such operations in parallel on big data. We introduce MatrixMap, an efficient parallel programming model which supports both machine learning and graph algorithms. It is easy for users to transform algorithms in matrix operations into parallel matrix patterns without handling issues such as fault tolerance. In the end, MatrixMap is able to allow users to process big data in an easy and unified way.

MatrixMap provides powerful yet simple matrix patterns and data structure, bulk key matrices. In MatrixMap, data are loaded into bulk key matrices and algorithms are formulated as a series of matrix patterns. It is implemented on a shared nothing cluster with multi-cores support. To evaluate performance, several typical algorithms are expressed in this framework. The experiment results show that MatrixMap is 12 times faster than Spark, especially for iterative computation.

6.2 Future Research

In the future, we need to run more complex algorithms such as deep learning algorithms, to get more information on big data. Although these intelligent algorithms are complex, they are made of basic matrix operations. MatrixMap which is designed for parallel matrix operations has the potential to support these complex algorithms.

To support more complex algorithms, MatrixMap needs to provide more matrix patterns, for example, matrix inversion which is common in PCA. To better support user' requirement, we plan to implement more typical machine learning algorithms in our algorithms library: SVM. To improve programming efficiency, it is better to provide domain specific language like SQL for users. It can help users like database administrator to use MatrixMap to process big data.

Overall, we believe that the following directions are worth further investigations.

- Provide more matrix patterns for users to program more algorithms.
- Implement more machine learning and graph algorithms in the library.
- Provide higher-level interactive interfaces on top of MatrixMap, such as SQL and R shells.

References

[Aap12] Carlos Guestrin Aapo Kyrola, Guy Blelloch. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 31–46. USENIX Association, 2012.

[BCZ90] John K Bennett, John B Carter, and Willy Zwaenepoel. *Munin: Distributed shared memory based on type-specific memory coherence*, volume 25. ACM, 1990.

[CDC+99] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification.* Center for Computing Sciences, Institute for Defense Analyses, 1999.

[CDPW92] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[CGL86] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in Linda. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 236–242. ACM, 1986.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[Cor09] THomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[CW93] Moody T Chu and J Loren Watterson. On a multivariate eigenvalue problem, Part I: Algebraic theory and a power method. *SIAM Journal on Scientific Computing*, 14(5):1089–1106, 1993.

[dat15] libsvm dataset url: www.csie.ntu.edu.tw/~cjlin/libsvmtools /datasets/binary.html#news20.binary, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[ELZ10] Jaliya Ekanayake, Hui Li, and Bingjing Zhang. Twister: a runtime for iterative mapreduce. In *HPDC '10 Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[GLG+12] JE Gonzalez, Y Low, H Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

[Har12] Peter Harrington. *machine learning in action*. Manning Publications, 2012.

[Hin13] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[HLS00] David W Hosmer, Stanley Lemeshow, and Rodney X Sturdivant. *Introduction to the logistic regression model*. Wiley Online Library, 2000.

[IBY+07] Michael Isard, M Budiu, Y Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[JG12] B Jacob and G Guennebaud. Eigen is a C++ template library for linear algebra: Matrices, vectors, numerical solvers, and related algorithms, 2012.

[Kan02] A. Y Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., & Wu. An efficient k-means clustering algorithm: analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, 2002.

[KG11] J Kepner and J Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[KKGK11] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. CSX: an extended compression format for spmv on shared memory systems. In *ACM SIGPLAN Notices*, volume 46, pages 247–256. ACM, 2011.

[L P99] T Winograd L Page, S Brin, R Motwani. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[LAC+96] Barbara Liskov, Atul Adya, Miguel Castro, S Ghemawat, R Gruber, U Maheshwari, A C Myers, M Day, and Liuba Shrira. Safe and efficient sharing of persistent objects in Thor. *ACM SIGMOD Record*, 25(2):318–329, 1996.

[LGK11] Yucheng Low, Joseph Gonzalez, and Aapo Kyrola. Graphlab: A distributed framework for machine learning in the cloud. *arXiv preprint arXiv:1107.0922*, 1107(0922), 2011.

[LGK+12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment*, pages 716–727, 2012.

[MAB10] Grzegorz Malewicz, MH Austern, and AJC Bik. Pregel: a system for large-scale graph processing. *Proceedings of the the 2010 international conference on Management of data*, 114(2):135–145, 2010.

[MMI+13] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Mart'n Abadi. Naiad: A Timely Dataflow System. In *SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*, volume 22, pages 297–306. ACM, 1993.

[PL10] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. *Proceedings of the 9th USENIX conference on Operating systems design and implementation - OSDI'10*, (1-14):1–14, 2010.

[PTM98] Jelica Protic, Milo Tomasevic, and Veljko Milutinović. *Distributed Shared Memory: Concepts and Systems*, volume 21. John Wiley & Sons, 1998.

[QCKC12] Zhengping Qian, Xiuwei Chen, N Kang, and M Chen. MadLINQ: large-scale distributed matrix computation for the cloud. *Proceedings of the 7th ACM european conference on Computer Systems. ACM*, pages 197–210, 2012.

[RMZ13] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472 —- 488, 2013.

[Roc15] RocksDB url:http://www.rocksdb.org/, 2015.

[SB13] Julian Shun and Ge Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[SC96] Y Saad and T.F. Chan. Iterative methods for sparse linear systems. *IEEE Computational Science and Engineering*, 3(4):88–88, 1996.

[SL98] Jeremy G Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *In International Symposium on Computing in Object-Oriented Parallel Environments*, pages 59–70. Springer, 1998.

[Sni95] Steven Snir, Marc and Otto, Steve W. and Walker, David W. and Dongarra, Jack and Huss-Lederman. *MPI: The complete reference*. MIT Press, 1995.

[Sun90] Vaidy S Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, 1990.

[SYK+10] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 721–726. Ieee, November 2010.

[tbb14] Threading Building Blocks url: https: //www.threadingbuildingblocks.org/, 2014.

[Val90] LG Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Vas] Panos Vassiliadis. A Survey of ExtractTransformLoad Technology. *International Journal of Data Warehousing and Mining*, 5(3):1–27.

[VBR+13] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Presto. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, page 197, 2013.

[Vel98] Tl Veldhuizen. Arrays in blitz++. In *Computing in Object-Oriented Parallel Environments*, pages 223–230. Springer, 1998.

[Wik15] Wikipedia. Join (SQL) url:https://en.wikipedia.org/wiki/Join_(SQL), 2015.

[XGF+13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, Ion Stoica, and Eecs AMPLab. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2, 2013.

[ZCDD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical report, UCB/EECS-2011-82 UC Berkerly, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10, 2010.

[ZWSP08] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5034 LNCS:337–348, 2008.