Big Data and Information Analytics
©American Institute of Mathematical Sciences
Volume **1**, Number **4**, October **2016**
doi:10.3934/bdia.2016015
pp. **349–376**

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

Yaguang Huangfu, Guanqing Liang and Jiannong Cao*

Department of Computing
The Hong Kong Polytechnic University
Hong Kong, China

(Communicated by Zhouchen Lin)

Abstract. The computation core of many big data applications can be expressed as general matrix computations, including linear algebra operations and irregular matrix operations. However, existing parallel programming systems such as Spark do not have programming abstraction and efficient implementation for general matrix computations. In this paper, we present 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 programming interface defined by matrix patterns. Users can easily load data into bulk key matrices and program algorithms into parallel matrix patterns. 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 context-aware 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. The experiment results show that MatrixMap is 12 times faster than Spark.

1. **Introduction.** Many common machine learning and graph algorithms which are extensively used in big data applications have been implemented by general matrix computations, including both linear algebra operations and irregular matrix operations. However, existing data-parallel programming systems (e.g., MapReduce, Spark) do not have programming abstraction and efficient implementation for general matrix computations. And graph-parallel programming systems such as Pregel and GraphLab are designed for graph algorithms rather than matrix computations. Large scale matrix computation programming systems (e.g., ScaLAPACK, MadLINQ), however, are limited to linear algebra operations, without providing support for some complex machine learning and graph algorithms. Hence, a unified and efficient programming model for general matrix computation is highly desirable.

2010 *Mathematics Subject Classification.* Primary: 68N15, 68Nxx; Secondary: 68Txx.
Key words and phrases. Big data, parallel programming, matrix computation, machine learning, graph processing.
* Corresponding author: Jiannong Cao.

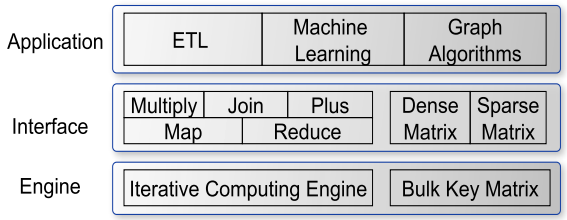


FIGURE 1. MatrixMap Framework

This paper presents MatrixMap, a unified and efficient data-parallel programming framework for general matrix computations (see Fig.1). Nevertheless, designing and implementing such programming system entails two major challenges. The first challenge is how to abstract a unified programming interface for general matrix computations in big data applications. Here, MatrixMap provides a powerful yet simple abstraction. In particular, MatrixMap defines a data structure called bulk key matrix and a programming interface using matrix patterns. In MatrixMap, data are loaded into bulk key matrices and algorithms are formulated as a series of matrix patterns.

Bulk key matrix is a scalable and constant distributed shared memory data structure [25], which stores vector oriented data indexed by key and can keep data across matrix patterns. Bulk key matrix is a high-level abstraction and mathematical matrix can be considered as a special case with key and value in digits. We define two kinds of matrix patterns, which can be programmed by user-defined lambda function. One is the unary pattern: Map, Reduce; the other is the binary pattern: Plus, Multiply, Join. Note that linear algebra operations are special cases of matrix patterns with specific lambda functions.

The second challenge lies in how to implement the unified interface with efficient runtime support. MatrixMap is implemented on a shared nothing cluster with multi-cores support. It follows Bulk Synchronous Parallel(BSP) model [32] to compute each pattern and form an asynchronous computation pipeline to get, compute and save data with context-aware data shuffling strategies. Furthermore, we leverage Compressed Sparse Matrices(CSR) and BLAS (Basic Linear Algebra Subprograms) to speed up in-memory matrix computations. And we introduce Key-CSR data format and frequently used graph operations for graph algorithms.

MatrixMap with unary matrix patterns supports Extract-Transform-Load [33] 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 context-aware data shuffling strategies for specific matrix patterns and in-memory data structures reusing data in iterations. To evaluate the 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.

In summary, our key contributions are:

- A unified and efficient programming framework for big data applications in general matrix computations.

- A scalable matrix data structure across memory and out-of-core storage with vector-oriented cache algorithm for the massive amount of data.
- Matrix patterns with context-aware data shuffling strategies and asynchronous computation pipeline for algorithm parallelization.

2. **Background.** In this section, we firstly provide the related works in data-parallel programming systems, graph-parallel programming systems and matrix programming systems. Then we describe the limitations of related works and highlight the challenges of designing and implementing a unified and efficient programming system for general matrix computations.

Data-Parallel Programming System: MapReduce [6] 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 such 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. MapReduce cannot directly support matrix operations and reuse data via in-memory data structures.

Twister [7], which allows long-lived map tasks to keep static data in memory between jobs, extends MapReduce to support iterative jobs. Twister program with one map function and one reduce function is inefficient to support flexible matrix operations.

Dryad [12] 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 the users. A Dryad application combines computational vertices with communication channels to form a dataflow graph. Dryad is suitable for offline analysis of large data sets (batch computation system).

Naiad is a distributed system for executing data parallel, cyclic dataflow programs. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform iterative and incremental computation. It enriches dataflow computation with time stamps. It offers low-level primitive dataflow for users, but does not support high-level matrix operations [21].

Spark [36] is 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 to a dataset). Users can explicitly cache an RDD [37] in memory across machines and reuse it in multiple MapReduce-like parallel operations. Spark cannot directly support matrix operations, which are basic operations in machine learning and graph algorithms. Matrix multiplication must be formulated into a series of map and reduce. Its RDD cache algorithm is LRU, which does not consider the context of algorithms to improve efficiency.

Graph-Parallel Programming System: GraphX [35] is a resilient distributed graph system on Spark. In GraphX, distributed graph representation is to efficiently distribute graphs as tabular data structures. GraphX model following vertex thinking limits to graph algorithms. GraphX flooding message is slow to converge to answer and is slower than matrix computations as which graph algorithms formulate.

Pregel [20] is a programming model for processing large graphs in a distributed environment. The programs are expressed in an iterative vertex which can send and receive messages to other vertices in the iterations. Vertices [16] iteratively process data and send messages to neighboring vertices. It is easy to adapt typical graphic algorithms into a vertex-based program.

GraphLab [18] is a graph-based, high performance, distributed computation programming model in machine learning. It solves the dependency of the graph. It differs from Pregel in that it does not work in bulk-synchronous steps, but rather allows the vertices to be processed asynchronous steps. [19]. The recent PowerGraph [8] 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 [17] adapting 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 [30] 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. This framework can only run on a single computer, which cannot take advantage of distributed computing to process large-scale data.

X-Stream [28] 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 graph and on a single shared-memory machine.

Matrix Programming System: ScaLAPACK [2] is a library of high-performance linear algebra routines for parallel distributed memory machines. It is for linear algebra matrix operations, not for irregular matrix operations and lacks in-memory data structure to reuse data in iterations.

Presto [34] 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.

Piccolo [24] 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 [26] 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, its DAG engine does not take advantage of in-memory data structure and matrix operations with optimized shuffling strategies.

HAMA [29] 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 that it should transform matrix computations into MapReduce jobs.

Above-mentioned parallel programming systems do not support general matrix computation in machine learning and graph algorithms. Data-Parallel programming

systems are inefficient to support matrix operations. For example, in MapReduce [6] and Spark [36], matrix multiplication has to be implemented into several Map and Reduce jobs, which is cumbersome and not efficient [3]. 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. Besides these linear algebra operations, many algorithms, especially in graph algorithms, are 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 does or operation. All pair shortest paths is similar to matrix multiplication. It is not to sum corresponding row and column in two matrices, but get the minimum value between elements in the first matrix and sum of two elements of two matrices. We 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. The input of PageRank is a sparse matrix. 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 need level synchronization. Additionally, many algorithms will 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.

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 [25], 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 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 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 [4] 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.

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 2. 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.

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.

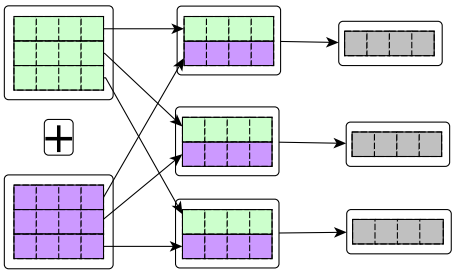


FIGURE 2. Matrix Plus Pattern

We formulate the algorithm as matrix plus, $C = A + B$ with a lambda function to compare two elements in graphs.

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. 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.

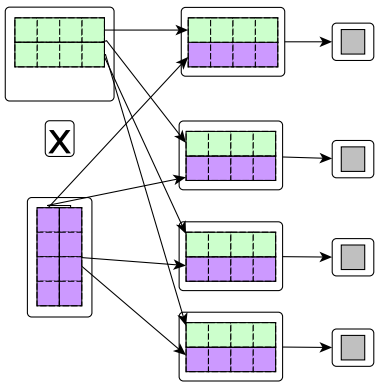


FIGURE 3. Matrix Multiply Pattern

All Pair Shortest Path [5] 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.

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 [14].

PageRank [23] 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: $r = M * r$.

Matrix Join Matrix patterns combine vectors from two matrices, illustrated in Figure 4. 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.

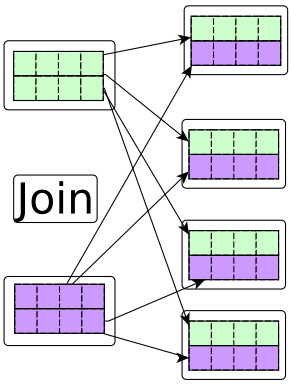


FIGURE 4. Matrix Join Pattern

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) [11] to implement matrix patterns. TBB helps programmers easily write parallel C++ programs that take full advantage of multi-core performance. We use ZeroMQ [10] 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 5. As C++ is a compiled language, which cannot dynamically load codes like interpreted languages, so MatrixMap framework cannot dynamically

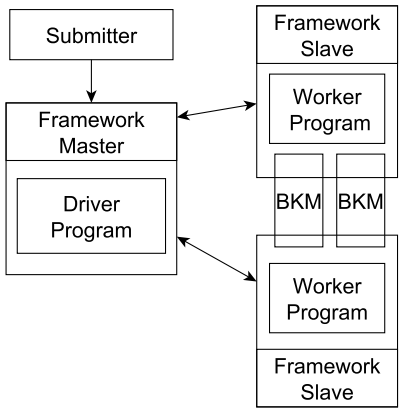


FIGURE 5. System Architecture

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 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.

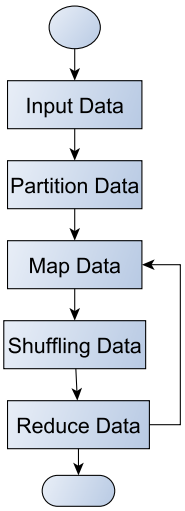


FIGURE 6. Data flowchart of MatrixMap framework

Figure 6 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. 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 [27]. 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) [22] 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 [32]. 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 7 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.

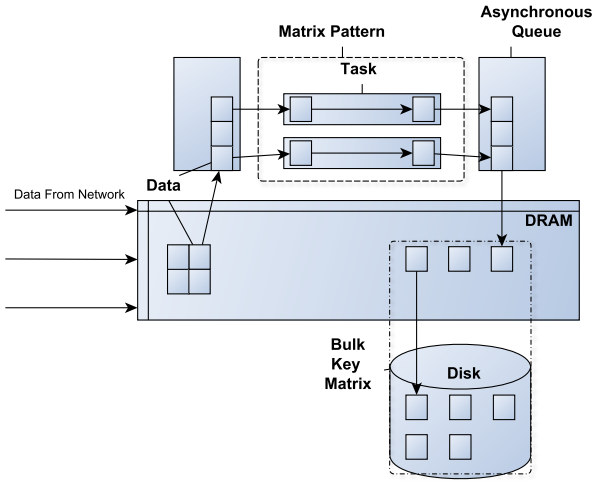


FIGURE 7. Asynchronous Computing Process

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)[15] 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.

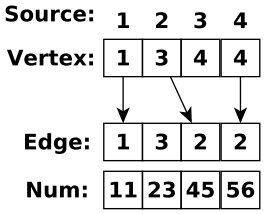


FIGURE 8. CSR Format

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, 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.

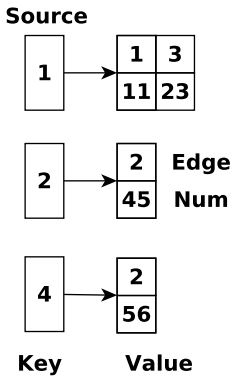


FIGURE 9. Key-CSR Format

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. 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.

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 [33] 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.

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 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);
}
);
```

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 3. Inner Join

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

Codes 3 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.

LISTING 4. 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);
}
```

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. [9]

Codes 4 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.

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. Code 5 illustrates the implementation.

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 (see Code 6 below):

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.

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.

LISTING 5. 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();
});
}
```

LISTING 6. 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);
}
```

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 (see Fig. 10). The implementation in the MatrixMap is shown in code 7. 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 [30].

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

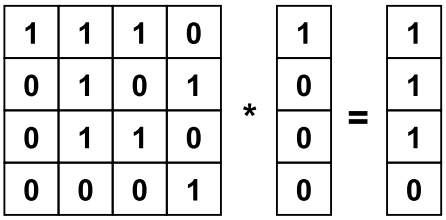


FIGURE 10. Breadth-First Search in Matrix Operations

or graph B. We formulate the algorithm as matrix plus, $C = A + B$ with a lambda function to compare two elements in graphs.

Codes 8 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.

LISTING 7. Breadth-first Search

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

LISTING 8. 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;
}
);
```

All Pair Shortest Path All Pair Shortest Path [5] 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

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

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.

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

Listing 9. 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 9 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.

PageRank PageRank [23] 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 10 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 10. 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;
}
```

6. **Evaluation.** Although our implementation of MatrixMap is still at an early stage, we relate the experiment results that show its promise as a 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.

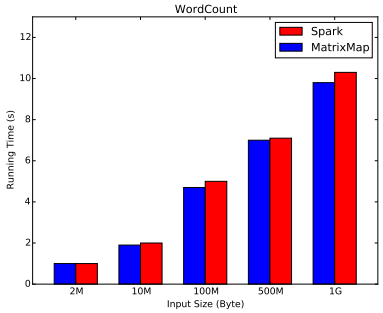
We compare MatrixMap with a data-parallel system called Spark (version 1.2), a graph-parallel system called GraphX (version 1.2) and a matrix computation system called ScaLAPACK (version 2.0.0). Apache Spark is a fast and general engine for large-scale data processing, which runs programs up to 100x 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 [2] is a library of high-performance linear algebra routines for parallel distributed memory machines.

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

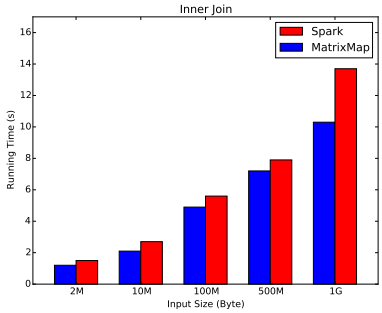
We evaluate framework in different data size with non-iterative algorithms, wordcount, inner join, graph merge and all pair shortest path with one iteration. They have input stages and no computation stages. We use GraphX's implementation to compare their performance with MatrixMap. We test algorithms with data size in 4k(2M bytes), 81k(6.5M bytes), 875k(62M bytes), 1.9m(71M bytes), 3.9m(203.5M bytes) graph nodes [1]. GraphX in Spark project follows vertex programming, thinking in vertex. Actually, it is a message passing method [31] 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, Floyd-Warshall algorithm. Additionally, its vertices will flood messages through the graph and not all vertices need all their neighbor vertices' information. Noise information may interfere the computation. So vertex as processing unit will slowly converge to answer. 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 cannot take advantage of BLAS to speed up computation. Graph algorithms in the matrix are more compact and are easier to figure out and have clear data access patterns, and MatrixMap takes advantage of matrix operations via BLAS.

WordCount, which is a typical application, can be formulated into map pattern and reduce pattern. We compare it with the implementation of Spark (see Fig. 11(a)). Since the map pattern and reduce pattern of MatrixMap are similar to those of Spark, whereas MatrixMap is faster than Spark, due to its implementation in C++.

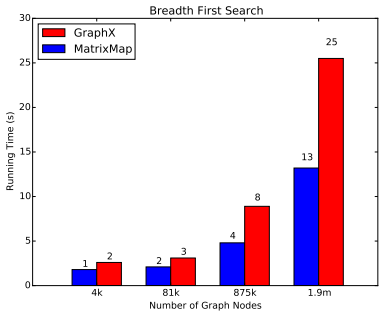
Inner Join can be formulated into matrix join. Spark provides join operations. The join pattern is formulated into map and reduce phase. In map phase, the two matrices will be map into pairs of vectors, in reduce phase, it will execute lambda function and output corresponding pairs. In the data shuffling between map phase



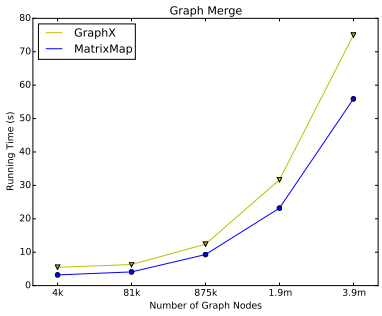
(a) WordCount Run Time



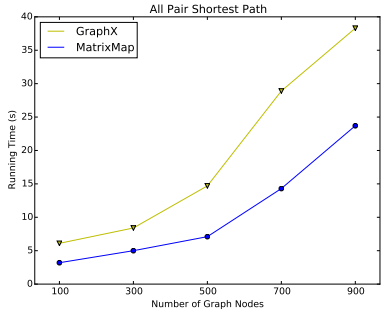
(b) Inner Join Run Time



(c) Breadth-First Search Run Time



(d) Graph Merge Run Time



(e) All Pair Shortest Path Run Time

FIGURE 11. Runtime comparison between MatrixMap and other programming models in non-iterative algorithms

and reduce phase, MatrixMap can asynchronously output each pair of data. As shown in Fig. 11 (b),MatrixMap is faster than Spark.

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.

Figure 11 (c) show that MatrixMap performs better than GraphX in executing Breadth-First Search .

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 find the corresponding key in each graph. MatrixMap costs much less time than GraphX (see Fig.11 (d)).

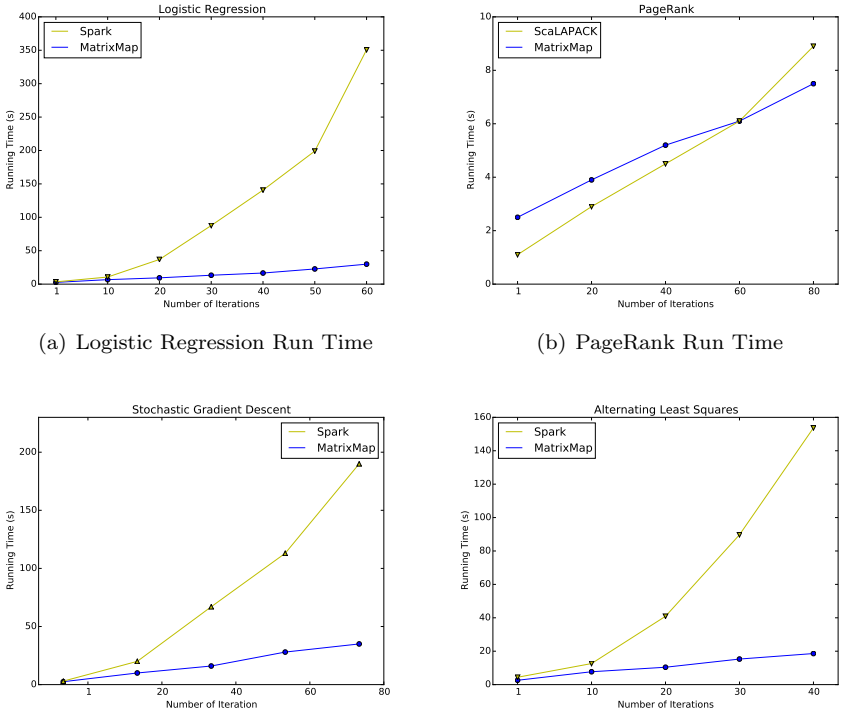
All Pair Shortest Path can be formulated into matrix-matrix multiplication. We use GraphX's implementation to compare its performance with MatrixMap. GraphX cannot implement all pair shortest path in dynamic algorithm paradigm, but runs single shortest path on every vertex. From algorithm analysis, the dynamic algorithm in MatrixMap costs less than the algorithm in GraphX. This algorithm needs to shuffle matrix data in each iteration. The result shown in Fig.11 (e) demonstrates that MatrxiMap is faster than GraphX.

We evaluate MatrixMap framework with iterative algorithms, logistic regression, PageRank, Stochastic Gradient Descent and Alternating Least Squares, in different iterations. The test results follow linear relation between time and iterations number. The input stage costs certain 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. Therefore, MatrixMap is suitable for iterative algorithms. MatrixMap's running time in each iteration keeps constant. Because 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 this 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, next iterations only need to reconstruct and transmit a vector and can reuse matrix data in the local machine. We run algorithms in several iterations. Experiments show that MatrixMap is much faster than Spark after several iterations for 12 times (see Fig.12 (a)).

PageRank can be formulated into matrix-matrix multiplication. We use ScaLAPACK's implementation to compare its performance with MatrixMap. Since ScaLAPACK is highly optimized, at the beginning of iterations, it is faster than MatrixMap. After 60 iterations, MatrixMap gradually outperforms ScaLAPACK for 30% (see FIg. 12 (b)), due to Matrix patterns with optimized data shuffling strategies and asynchronous pipeline directly and in-memory data structure for data reuse. The input matrix in the PageRank can be reserved in the memory of computing nodes, the vector is shuffled, although block-partitioned algorithms in ScaLAPACK can ensure high levels of data reuse.

Stochastic Gradient Descent can be formulated to matrix-vector multiplication. Each vector in the left matrix and the right vector will be mapped to computing machines, reduced each pair of vectors to a vector and reduced these vectors to single vector. It is faster than the implementation with broadcast variable in Spark. Broadcast variable has to update data in the cluster after computing each vector in the matrix each time. MatrixMap reduces the right vector data in the data shuffling



(a) Logistic Regression Run Time

(b) PageRank Run Time

(c) Stochastic Gradient Descent Run Time

(d) Alternating Least Squares Run Time

FIGURE 12. Runtime comparison between MatrixMap and other programming models in iterative algorithms

for one time. The result shown in Fig. 12 (c) demonstrates that MatrixMap is much faster than Spark.

Alternating Least Squares is formulated into two matrix and vector multiplications in iterations. We use Spark's implementation to compare its performance with MatrixMap. In iterations, parts of the matrix can be reserved and reused in the local machines. The result in Fig.12 (d) shows that MatrixMap is faster than Spark.

We further evaluate the scalability of MatrixMap (see Fig. 13). 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.

7. **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

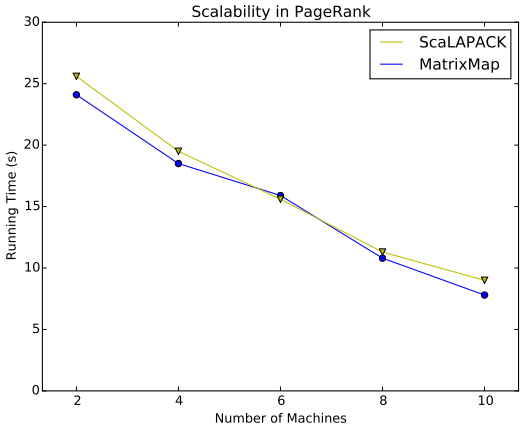


FIGURE 13. Scalability in PageRank Run Time

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. 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 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.

8. **Conclusion.** 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.

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.

Acknowledgments. This work is financially supported by Hong Kong RGC under GRF Grant PolyU 5104/13E.

REFERENCES

[1] C.-C. Chang and Chih-Jen, libsvm dataset url: http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/news20.binary.bz2, 2015.

[2] J. Choi, J. J. Dongarra, R. Pozo and D. W. Walker, ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers, in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, IEEE, (1992), 120–127.

[3] Chu, Cheng-Tao and Kim, Sang Kyun and Lin, Yi-An and Yu, YuanYuan and Bradski, Gary and Ng, Andrew Y and Olukotun, Kunle, Map-Reduce for Machine Learning on Multicore, in *Neural Information Processing Systems*, 2007.

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

[5] T. H. Cormen, *Introduction to Algorithms*, MIT press, 2009.

[6] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Communications of the ACM*, **51** (2008), 107–113.

[7] J. Ekanayake, H. Li and B. Zhang, Twister: A runtime for iterative MapReduce, in *HPDC '10 Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, (2010), 810–818.

[8] J. Gonzalez, Y. Low, H. Gu, D. Bickson and C. Guestrin, PowerGraph: Distributed graph-parallel computation on natural graphs, in *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, (2012), 17–30.

[9] P. Harrington, *Machine Learning in Action*, Manning Publications, 2012.

[10] P. Hintjens, *ZeroMQ: Messaging for Many Applications*, O'Reilly Media, Inc., 2013.
[11] Intel, Threading Building Blocks url: https://www.threadingbuildingblocks.org/, 2009.
[12] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, *ACM SIGOPS Operating Systems Review*, **41** (2007), 59–72.
[13] Join (SQL) url: https://en.wikipedia.org/wiki/Join, 2015.
[14] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, SIAM, 2011.
[15] K. Kourtis, V. Karakasis, G. Goumas and N. Koziris, CSX: An extended compression format for spmv on shared memory systems, in *ACM SIGPLAN Notices*, **46** (2011), 247–256.
[16] J. Kowalik, ACTORS: A model of concurrent computation in distributed systems (Gul Agha), *SIAM Review*, **30** (1988), 146–146.
[17] C. G. Aapo Kyrola and G. Blelloch, GraphChi: Large-scale graph computation on just a PC, in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, USENIX Association, (2012), 31–46.
[18] Y. Low, J. Gonzalez and A. Kyrola, Graphlab: A distributed framework for machine learning in the cloud, arXiv preprint, arXiv:1107.0922, **1107** (2011).
[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin and J. M. Hellerstein, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, in *Proceedings of the VLDB Endowment*, **5** (2012), 716–727.
[20] G. Malewicz, M. Austern and A. Bik, Pregel: A system for large-scale graph processing, *Proceedings of the the 2010 international conference on Management of data*, **114** (2010), 135–145.
[21] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham and M. Abadi, Naiad: A timely dataflow system, in *SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, (2013), 439–455.
[22] E. J. O'Neil, P. E. O'Neil and G. Weikum, The LRU-K page replacement algorithm for database disk buffering, in *ACM SIGMOD Record*, **22** (1993), 297–306.
[23] T. W. L Page, S Brin, R Motwani, *The PageRank Citation Ranking: Bringing Order to the Web*, tech. rep., Stanford InfoLab, 1999.
[24] R. Power and J. Li, Piccolo: Building fast, distributed programs with partitioned tables, *Proceedings of the 9th USENIX conference on Operating systems design and implementation - OSDI'10*, (2010), 1–14.
[25] J. Protic, M. Tomasevic and V. Milutinović, *Distributed Shared Memory: Concepts and Systems*, John Wiley & Sons, 1998.
[26] Z. Qian, X. 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*, (2012), 197–210,.
[27] RocksDB, http://rocksdb.org/, 2015.
[28] A. Roy, I. Mihailovic and W. Zwaenepoel, X-stream: edge-centric graph processing using streaming partitions, in *the Twenty-Fourth ACM Symposium on Operating Systems Principles*, (2013), 472–488.
[29] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim and S. Maeng, HAMA: An efficient matrix computation with the mapreduce framework, in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, (2010), 721–726.
[30] J. Shun and G. Blelloch, Ligra: A lightweight graph processing framework for shared memory, in *PPoPP*, (2013), 135–146.
[31] M. S. Snir, S. W. Otto, D. W. Walker, J. Dongarra and Huss-Lederman, *MPI: The Complete Reference*, MIT Press, 1995.
[32] L. Valiant, A bridging model for parallel computation, *Communications of the ACM*, **33** (1990), 103–111.
[33] P. Vassiliadis, A survey of extract-transform-load technology, *International Journal of Data Warehousing and Mining*, 5, 1–27.
[34] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, Presto, in *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, (2013), p197.
[35] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, and E. AMPLab, GraphX: A Resilient Distributed Graph System on Spark, in *First International Workshop on Graph Data Management Experiences and Systems*, p. 2, 2013.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, Spark: Cluster computing with working sets, *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, (2010), p10.

[37] M. Zaharia, M. Chowdhury, T. Das and A. Dave, *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, tech. rep., UCB/EECS-2011-82 UC Berkerly, 2012.

[38] T. Zhang, Solving large scale linear prediction problems using stochastic gradient descent algorithms, in *Proceedings of the twenty-first international conference on Machine learning*, ACM, (2004), p116.

[39] Y. Zhou, D. Wilkinson, R. Schreiber and R. 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)*, LNCS, **5034** (2008), 337–348.

E-mail address: lucienhuangfu@foxmail.com
E-mail address: csgliang@comp.polyu.edu.hk
E-mail address: csjcao@comp.polyu.edu.hk