JBösen: A Java Bounded Asynchronous Key-Value Store for Big ML

Yihua Fang

CMU-CS-15-122

July 21st 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Eric P. Xing, Chair
Kayvon Fatahalian

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Copyright © 2015 Yihua Fang

Keywords: Distributed System, Machine Learning, Parameter Server, Stale Synchronous Parallel Consistency Model, Big Data, Big Model, Big ML, Data-Parallelism

Abstract

To effectively use distributed systems in Machine Learning (ML) applications, practitioners are required to possess a considerable amount of expertise in the area. Although highly abstracted distributed system frameworks such as Hadoop can help to reduce the complexity of writing code for distributed systems, their performances are incomparable to that of specialized implementations. Other efforts such as Spark and GraphLab each has it's own downsides. In light of this observation, the Petuum Project is indented to provide a new framework for implementing highly efficient distributed ML applications through a high-level programming interface.

JBösen is a Java key value store system in Petuum using the Parameter Server (PS) paradigm and it aims to extend the Petuum project to Java as almost a quarter of the programmer population use Java. It provides an iterative-convergent programming model that covers a wide range of ML algorithms and an easy-to-use programming interface. JBösen, unlike other platforms, exploits the error tolerance property of ML algorithms to improve performance with a Stale Synchronous Parallel (SSP) consistency model to relax the overall consistency of the system by allowing the workers to access older and more staled values.

Contents

List of Figures

Chapter 1

Introduction

In recent years, Machine Learning (ML) practitioners are turning to distributed systems to satisfy the growing demand of computational power in ML applications. As we advanced into an age of Big Data (terabytes to petabytes of data) and Big Model (well beyond billions of parameters), a single machine can no longer solve ML problems in an efficient and timely manner. However, to implement ML algorithms on top of a distributed system is proven to be no easy task. One needs to possess a considerable amount of knowledge in both distributed systems and ML in order to write an efficient piece of distributed ML application code. In today's world, there is a only a small percentage of ML practitioners can fit in this description. The apparent solution is an abstracted distributed platform that hides away the "difficult" system work to allow users to focus on ML algorithms.

In the past, there have been efforts to build highly abstract general purpose distributed frameworks for ML, but these existing systems present a variety of trade-offs on efficiency, correctness, programmability, and generality[9]. Hadoop [8] is an popular example of such a platform with an easy-to-use MapReduce programming model. Yet, the MapReduce programming model makes it difficult to optimize using many properties that are specific to ML algorithms such as the iterative nature, and its performance is not competitive to other existing systems. Spark [10], while remaining the scalability and fault tolerance of MapReduce, is an alternative platform to Hadoop that is optimized for iterative ML algorithms. What Spark does not offer in the platform is the fine-grained scheduling of computation and communication which is considered to be essential for a fast and correct implementation of distributed ML algorithms [1]. GraphLab [4] and other graph-oriented platforms distribute workload through clever partitioning of the graph based models, but many advanced ML algorithms cannot be easily or efficiently represented in graphs such as topic modeling. The remaining category of systems [3, 6] offers powerful and versatile low-level programming interface but do not offer an easy-to-use building blocks for implementing many ML algorithms.

The Petuum project [9] aims to address this issue by providing a new framework for implementing highly efficient distributed machine learning applications through a high-level programming interface. The framework proposes that a wide spectrum of ML algorithm can be represented by an *iterative convergent* [9] programming model, treating the ML algorithm as a series of iterative updates to the ML model. In this way, the system exploits two types of parallelism: (1) data parallelism - partition and distribute the training data across a cluster of

machines; (2) model partitioning - models are partitioned and assigned to different machines and then are updated in parallel. The system also exploits a statistical properties that is specific to ML algorithms: error tolerance - ML algorithms can tolerate limited amount of inconsistency in intermediate calculations. It takes advantages of the error tolerance property through a relaxed consistency model, a Stale Synchronous Parallel (SSP) consistency model [7] and it's variant SSP Push consistency model. [7] shows that the SSP and SSP Push consistency model may improves significantly the throughput of the system.

The key component in the Petuum project to realize the above described designs and optimization is a parameter server [3] system, named Bösen, for accessing the shared model parameters from any machine through a key-value storage APIs that resembles single machine programming interfaces. What is missing from the Petuum project, however, is a Java PS system as Bösen is implemented in C++. With figures from a variety of unofficial sources suggested on [5], it is estimated that there are between eight to ten million Java programmers in the world today. In addition, there are far more ML researchers and data scientists who are familiar with Java than those who are familiar with C++ which is required to use Bösen.

In light of this observation, we present in this thesis a Java version of the PS system, JBösen [1] for the Petuum project. While we incorporated all of the above mentioned design points from Bösen into the JBösen, we implemented JBösen independent of Bösen as a stand alone system. We begin by a theoretical explanation in details of the design points JBösen incorporated. We then illustrate how different components of the JBösen come together to realize the aforementioned design ideas. Followed after, we demonstrate how a ML algorithm can be implemented using JBösen's interfaces as an iterative convergent programming model. In this thesis, we use a matrix factorization algorithm using Stochastic Gradient Descent (SGD) as demonstration, but the Petuum ML library also includes more algorithms on JBösen not explored in this paper as well as more are planned in the future.

[1] JBösen is available as part of the Petuum project open sourced on `http://petuum.org`

Chapter 2

Key Ideas Overview

We begin with a theoretical discussion on the key ideas used in the JBösen system, which differentiate JBösen from other distributed ML frameworks. These ideas are manifested as the fundamental features of JBösen and each exploits some unique properties of ML algorithms. We start by illustrating the iterative convergence programming model in representing a wide range of ML algorithms with data parallelism. Then we discuss how the parameter server architecture is suitable for implementing data parallelism. Following that, we explain how SSP and SSP Push consistency model can exploit the error tolerance nature of ML algorithms to improve throughput of the system.

2.1 Iterative Convergent ML programming model

[9] formalized the Iterative Convergent ML programming model as executing a update function iteratively until the model state reaches some state matching some stopping criteria. Mathematically, it can be expressed as

$$A^t = F(A^{t-1}, \Delta(A^{t-1}, D))$$

where t denotes the iteration, A denote the model state and given the training data D. The update function

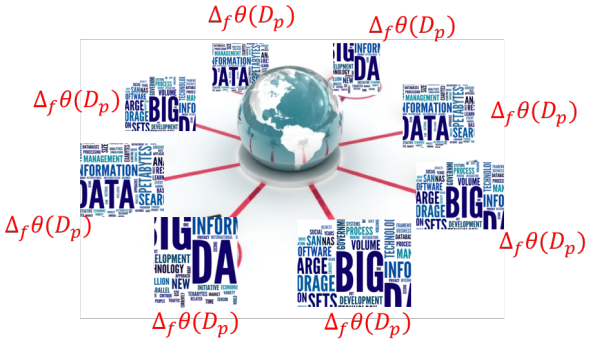
$$\Delta()$$

which improves a loss function performs computation on the training data D and model state A and the results are aggregated by some function F. In the context of the data parallelism, the iterative convergent programming model becomes:

$$A^t = F(A^{t-1}, \sum_p = 1^P \Delta(A^{t-1}, D_p))$$

where P is the number of partitions or the number of workers. The training data is partitioned and assigned to each of the workers and the intermediate results computed by each of the workers are aggregated together before applying to the model parameters. The important observation is the property of additive updates where updates $\Delta()$ are associative and commutative and are allowed to be aggregated through summation both locally and over the network. This property also allows

Figure 2.1: Data Parallelism



that no matter which partition of the training data is assigned to a particular worker, the worker can in a statistical sense contribute to the convergence of a ML algorithm via $\Delta()$ equally [9]. This is commonly assumed in many ML applications.

2.2 Parameter Server

The primary challenge to the data parallelism approach is how to design the software system to share a gigantic ML model across all nodes in a cluster with correctness, performance and easy to program interfaces at the same time. The parameter server is a popular paradigm that ML practitioners have been turning to in recent years[1].

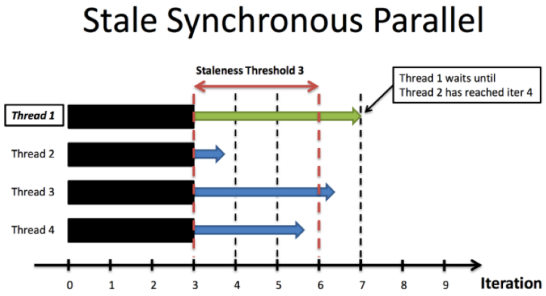
Without limiting to a specific implementation, the concept of the Parameter Server is a shared key-value storage with a centralized storage model and an easy to use programming model. It is intended as a software system to share the model parameters in ML applications across a cluster of commodity machines. The nodes are partitioned into two classes of nodes. The client nodes perform the bulk part of the computation for the ML algorithm. Usually, each client node will retain a portion of the training data and perform statistical computation such as Stochastic Gradient Descent (SGD) to train the shared model. The server nodes are responsible for synchronizing parameters of the shared model to provide a global view of the shared ML model to all clients. The model is partitioned and assigned to different servers and therefore updates to the model can be processed in parallel. The clients communicate with the server nodes to retrieve or update the shared parameter values.

2.3 Stale Synchronous Parallel Consistency Model

ML applications possess the iterative-convergence characteristics where the algorithm has a limited error-tolerance in the parameter state. In another word, the ML parameters at each iterations need not to use the most up to date parameters for computations. This presents an unique opportunity for optimization by relaxing the consistency model of the software system to achieve higher throughput compared to a strong consistency model. The SSP consistency model is based on this observation that the ML parameters are allowed to be stale and it aims to maximizes the

time each worker spends on useful computation (versus communication with the server) while still providing the algorithm correctness guarantees.

Figure 2.2: Demonstration of the SSP consistency. Worker threads cannot be more than s iterations apart where s is user defined.



In SSP, each worker make an update ε_i to an parameter where the update is associative and commutative. Hence, the freshest version of the parameter value δ is a sum of updates from all workers on δ.

$$\delta = \sum_i \varepsilon_i + \delta' \text{ where } \delta' \text{ denote the parameter from last iteration}$$

When a worker asks for the δ, the SSP consistency model allows the server to present to the clients a staled version of δ that may or may not included all of the updates up to a limit. As the example in 2.3, the limit of the staleness of the parameters can be user defined as a staleness threshold. In this way, we limit the system's staleness to a bounded value, the staleness threshold. In another word, informally, if c is the iteration of the fastest client in the system, it may not make further progress until all other clients are progressed to at least $c - s - 1$ iterations, where s is some user defined threshold. In this way, by using SSP, we relaxed the consistency of the entire system. The clients can perform more computations instead of spending time waiting for fresh parameters from the servers. [1] presents a formal proof of the correctness of the SSP consistency model.

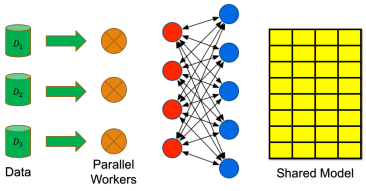
2.4 SSP Push Consistency Model

SSP Push consistency model is a variant of the SSP consistency model. In SSP, whenever the clients are in need of a more fresh parameter, they will request the parameter from the servers and be forced to wait for the responses from the servers. In SSP Push, the servers, instead of waiting for the clients to request parameter values, will try to push the updated parameters to the clients. If a client requested the parameter before, the server will push the parameters to the client after collecting all updates from other clients. This is taking the advantage that if the client access a parameter, it is likely that it will access it again in future iterations. In this way, we reduce the time needed for clients to wait for the requests to come back. The other benefit of SSP Push is improving the convergence rate of ML algorithms because the the average staleness of the parameters used in computation is significantly reduced [1].

Chapter 3

JBösen System Design

Figure 3.1: JBösen overall architecture



The aims of the JBösen system are a distributed ML platforms that allows ML practitioners to and to exploit unique properties of ML algorithms and to write data parallel ML programs that scales to Big Data and Big Model in Java. It offers to users a table-like interface with key-value storage APIs for ease of programming. It follows a parameter server paradigm which partition the the system into two classes of nodes, clients and servers. The clients are responsible for the ML computations and the servers are responsible for synchronizing and share model parameters across the cluster. By adopting a consistency controller, the system implements the SSP and it's variant SSP Push consistency model and frees users from explicit network synchronizations.

To further the usability of the JBösen system, we integrated the system with YARN/Hadoop environment. YARN is a popular resource manager and scheduler in the Hadoop environment and has become quite popular among a large scale clusters. Therefore, from a usability stand point, it is useful to integrate the JBösen with the YARN.

3.1 Table Interface

The JBösen provides a table-like interface for the shared parameters. Each shared parameter in a ML model is represented as a row ID and column ID. Rows are the minimum unit of communication between the servers and the clients. The rows are implemented in two different types: dense and sparse. The dense row is implemented using arrays where each row ID and column ID is present in the memory. The sparse row is implemented as map where only non-zero row ID and column ID pair exist in memory. The rows types also comes with different value size such as Integer vs Long and Float vs Double. The differentiation is designed to be flexible for users to tailor to their own application needs. More about row interface is explained in 4.4.

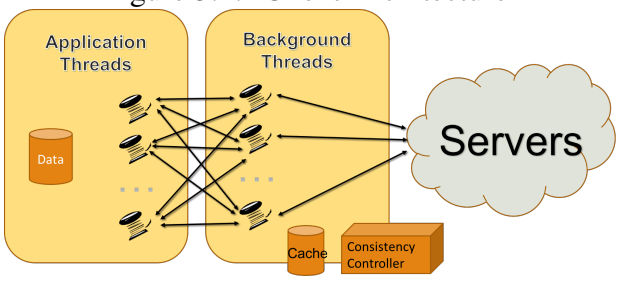
In the table interface, each application thread in the client is regarded as a worker by the system. The application threads execute the computation and access the shared parameters stored

in the table through a key value store interface via GET and INC. In context of the JBösen, the GET will obtain a row while INC will update the values of a row. Once the application threads complete an iteration, it notifies the system via CLOCK, which increment the clock of the application thread clock by 1. The clock is used to keep track of the progress of the system where clock c represent iteration c. The SSP consistency model will satisfy the READ request generated at clock c will observe all updates from 0 upto $c-s-1$ where s is the staleness threshold.

3.2 Clients

The clients consists of four different components: application threads, background threads (JBösen system threads on the client), the consistency controller and process cache storage. The application threads contains the ML algorithm logic which will generate requests to the background threads through the JBösen interface. The background threads are responsible for communication between clients and servers, and between application threads, maintaining the process cache storage using data from the servers and book-keeping with the servers. The background threads use a consistency model controller to control the staleness of data based on SSP and free the application threads from synchronizing explicitly.

Figure 3.2: Client Architecture



3.2.1 Application Threads

The application threads are user-implemented application logic with data partitions. The application thread would first load the training data from either memory, from disk or over a distributed file-system such as HDFS or NFS. The data loading is intentionally left to the application programmers since Java has native supports to most of the storage system. Then, the application threads can touch the data in any order desired by the application programmers based on the implemented algorithm. The application can choose to sample one data point at a time or choose to pass through all data points in one iteration. At the end of iteration, updates are pushed to the servers and the parameters are synchronized with the servers.

3.2.2 Back Ground Threads

The background threads are the system threads on the client nodes and they have three functions.

The first function of the background thread is to handle the communications within clients and between clients and servers. The application threads communicate with the background threads through network messages using intra-process channels. Each of the messages represents an instructions to the background threads. The background threads, upon receiving the

instruction, will sent the appropriate network messages to the servers. The response from the server arrives at background threads first. The background threads would handle the message for any book keeping and then respond to the application threads through intra-process channels.

The second function of the background thread is to handle the network synchronizations with the servers. Note, the JBösen API only offers GET and INC methods in the interfaces. The network synchronizations with the servers are done in the background threads abstracted away from the users and the synchronization happens each time the system clock advance. The system clock advances when all application threads that the background thread is responsible for advanced to a new clock. In another word, the system clock is defined as the minimum clock for a group of application threads and advances when the minimum clock advanced by 1. Each time the system clock advances, the background threads will send the updates in this clock along with the clock information to the servers.

The third function of the background thread is to maintain a process level cache for all the application threads on a node and maintain the staleness of the data based on the SSP consistency model. Whenever, servers send parameter data to the clients, the background threads would update the process cache storage. This can happen when both clients request for rows from the servers or the servers push row data to the clients. When the application threads requests for rows, the background thread uses a consistency controller to first checks to make sure to only request rows from server if the cached rows are too staled based on the SSP consistency.

3.3 Consistency Controllers

The JBösen implements both SSP and SSP Push consistency models through a consistency controller. The main purpose of the consistency controller is to manage the data supplied to the application threads' GET request to comply with the staleness requirements of the SSP consistency model.

3.3.1 Ensuring SSP consistency

The clients in the JBösen system cache previously accessed rows via the process cache storage on each client. When the application thread issues a GET request, it first check the local cache for the requested rows. If the rows is not found in the local cache, a request is sent to the server to get the rows. In addition, each row in the process cache storage is associated with a clock. A row with clock c means that updates generate by all workers before clock c has been applied. If the row has a clock c and $c > c_p - s$ where c_p is the worker's clock and s is the staleness threshold, then the row is considered fresh and will be returned to the worker. Otherwise, the row is considered to be too stale and the worker will send a request to server and block until the server send a fresh row.

3.3.2 SSP Push consistency

After each advance to the system clock, the updates are sent to the server. Since the updates are associative and commutative, the order of which are applied to the rows are not a concern.

The servers will then push the updated parameters through call-backs. When a client request a row for the first time, it registers a call back on the server. As the server clock advances from collecting all workers' clock tick, it push the rows to the registered clients. This is different from the SSP consistency where the server passively sends out the updates as clients request them. This happens each time the client does not have a fresh value in its process cache storage. This mechanism exploits the fact that application often revisit the same rows in iterative-convergence algorithms.

3.4 Server

The servers are a centralized key-value storage to share Big ML Models through the table interface. We designed the server to partition the table in a simple hash ring fashion where rows are shard across the server machines. The sharding is based on the hashing of the key and in this way each server maintains a portion of the shared tables. There are two primary methods in the server interface, row requests and push rows.

To synchronize in application logic, we implemented a light weight name node running on one of the server nodes to provide a globalBarrier() interface for the applications. The globalBarrier() allow the application thread to synchronize globally to the same state. The name node is also used to synchronize both the clients and servers initialization phase.

3.4.1 Row Requests

The client will sent a row request to the server if there is no fresh enough rows in it's local cache based on the SSP consistency. The server, upon receiving the row request, will examine if it has the row in a fresh enough version. If so, the server will send the row to the client immediately.

In the case that the server does not have a fresh enough row indicates that some other clients have not been progressed to the server clock, where server clock is defined as the min clock of all clients. For example, if client $client_1$ requests for a row at clock c_1, and there is another client $client_2$ currently running at clock $c_1 - s - 1$ where s is the staleness threshold, the request to $client_1$ cannot be satisfied because not all clients have updated the requested row in $c_1 - s$ iterations yet. The request will be stored at the server and the client requesting the row will hang to wait for other client to catch up.

3.4.2 Push Rows

At the end of each system clock, the client will send to the server the updates for that iteration. As the result, the server will apply the updates to the rows and maintain the iteration information. When all clients advanced at least 1 clock, the server clock is said to be advanced. Each time the server clock advances, the server will push the current rows to the client if the client have requested the rows or the client have updates for this row in the last iteration.

3.4.3 Name Node

To synchronize the states globally in the JBösen system, we implemented a light weight name node as a central server independent of the servers. In the beginning, the name node is used to synchronizing the initialization of all the components of the JBösen system. At the start, each background thread and each server thread are registered with the name node using a simple hand shake. Then, name node will synchronize the table creation process and signal all components that the system is ready to start.

The global barrier is implemented with the name node. The globalBarrier() method would send a global barrier message to the name node and wait for the name node to reply. The name node would wait for the same message to come from all clients before sending off reply to all the clients.

Since the name node is quite lightweight, it is reasonable to choose a centralized architecture as a single point of failure. In addition, the name node would run on one of the server nodes which does not affect the performance since the name node is quite light weight.

3.5 Network

The network communication of JBösen is implemented using the Netty project and Blocking Queue. The Netty project is a NIO framework that allows easy implementation of network application. It is asynchronous and event-driven which is suitable for highly scaled applications such as JBösen. In the JBösen datapath, the application threads first communicate with the background threads and the background threads would communicate with the servers. The network relationship between application threads and background threads is a many-to-many relationships. Following the PS architecture, each clients will be connected to all the servers and maintain a connection through out the life time of the application. The communication between the back ground threads on the clients and the application threads are done through a blocking queue where each massage is directly queued onto the destination application threads' queue. When communication between the server and the client, the Netty threads will automatically received the message and queues onto the destinations' queue.

Each of the message communicated over the network would have a header associated with it. The reason we need to encode the size of the message in the header is because Netty would fragment the message and the receiver would need to assemble the fragmented message to decode the message.

3.6 Integration with YARN

The integration of JBösen into the YARN environment is through writing a YARN application. To run on YARN, we need a YARN client and an application master. The YARN client is responsible for setting up the YARN application master and copying all local files onto HDFS if users do not copy them manually. The node running the application master is logically considered to be separate from the local machine running the client. The application master is responsible for setting up the containers and distribute necessary files. The training data used for the application

are usually so large that users should move the file to HDFS manually and it is not reasonable to copy data files every time the system runs.

The YARN client is much simpler compared to the application master as it consists of mostly boiler plate code from YARN's user manual. One extra task is to use the HDFS APIs to copy files from local file system to HDFS, since program running on YARN can only be assume to have access to HDFS. Note, differ from most YARN applications, we choose to make the client to fail fast in the case the application master failed instead of resubmitting the application master. This will be explained more in sections below.

The application master does a few more things extra on top of the boiler plate code. First, since JBösen system needs a host file containing the IP addresses of the cluster, the application master will requests the containers and generate a host file based on the containers obtained. Second, the application master needs to distribute Jar file for JBösen onto containers. YARN has a local resource interface to automatically distribute the files onto the containers, which requires careful set ups.

Chapter 4

JBösen Programming Interface and APIs

The JBösen system provided three main methods in the programming interface for users to express their iterative convergent ML algorithms. As explain in 3.1, the shared model is represented in the JBösen as a table interface with rows and columns. A Get() methods is used to request rows from the table interface, a Inc() methods to alter the values in the table, and last but not least, a clock() method to indicate the iteration progress of the application logic. Below is a basic JBösen program using the three methods in the interface.

```
table = PSTableGroup.getTable(tableId);
for (int iter = 0; iter < totalIteration; iter++) {
  // Request row data from the table
  row = table.get(rowId);

  // Using the row data to perform some computation and
  // compute the changes to the row data.
  change = compute(row);

  // Write the changes to the row data.
  table.inc(rowId, change);

  // At the end of this iteration, use clock to indicate the advance
  // of the iteration.
  PsTableGroup.clock();
}
```

The tables are stored on the servers and are identified through tableId. The reference of a table is accessed through the PsTableGroup object and usually obtained after the table creation. The table object contains two methods: table.get(rowId) to read the row data specified by the rowId, and table.inc(rowId, changes) to write the changes to the row specified by the rowId. With these two methods, the JBösen system will automatically ensure the SSP consistency across all components of JBösen and no addition attention is required by the users.

There are two different sets of APIs for setting up the JBösen. One interface is to manually set up the JBösen system with many boiler plate function calls that needs to be called in each

application but gives the programmers a greater flexibility to write their applications. The other is a simpler PS application interface where a lot of the boiler plate code for setting the JBösen is automated. Although the PS Application interface is simper, it imposes a specific programming model for using the APIs.

4.1 Complex APIs

In general, in order, a JBösen application needs to:

- Initialize the global PsTableGroup instance.
- Use PsTableGroup to create tables that are needed.
- Spawn worker threads and complete computation.
- De-initialize the global PsTableGroup instance.

4.1.1 Initialize PsTableGroup

PsTableGroup.init(PsConfig config) function must be called before any other methods from PsTableGroup interface are invoked. The PsConfig class contains several public fields that the application must set:

- clientId: The ID of this client, from 0, inclusive, to the number of clients, exclusive. This value must be set.
- hostFile: The absolute path to the host file in the local file system. This value must be set.
- numLocalWorkerThreads: The number of application threads for every client.
- numLocalCommChannels: The number of background threads per client. The recommended value is one per 4 to 8 worker threads.

For example, an application may begin as follows:

```
public static void main(String[] args) {
   PsConfig config = new PsConfig();
   config.clientId = 0;
   config.hostFile = "/home/username/cluster_machines";
   config.numLocalWorkerThreads = 16;
   config.numLocalCommChannels = 2;
   PsTableGroup.init(config);
   ...
}
```

4.1.2 Creating tables

After PsTableGroup has been initialized, it is time to create the distributed tables needed by the application. This is done using the createTable method, which takes several parameters:

- int tableId: The identifier for this table, which must be unique. This identifier is used later to access this table.
- int staleness: The SSP staleness parameter, which bounds the asynchronous operation of the table. Briefly, a worker thread that has called PsTableGroup.clock() n times will see all updates by workers that have called PsTableGroup.clock() up to n - staleness times.
- RowFactory rowFactory: A factory object that produces rows stored by this table. The implementation of the row is up to the application.
- RowUpdateFactory rowUpdateFactory: A factory object that produces row updates stored by this table. Essentially, JBsen tables store Row objects, which are updated by applying RowUpdate objects to them. The implementation of the row update is up to the application.

Since implementing these rows and row updates can be a lot of work, JBsen provides some common pre-implemented tables. For example, createDenseDoubleTable() creates a table that contains rows and row updates that are essentially fixed-length arrays of double values, and applying an update means element-wise addition.

4.1.3 Spawning application threads

After all the necessary tables are created, the application can spawn its worker threads. JBösen expects the number of worker threads to be exactly the number set in PsConfig.numLocalWorkerThreads. To inform JBösen that a thread is a worker thread, it must call PsTableGroup.registerWorkerThread() before accessing any tables. After doing so, the worker can use PsTableGroup.getTable(int tableId) to retrieve the previously created tables, and begin to use them. When the worker has completed running the application, it must call PsTableGroup.deregisterWorkerThread() to inform JBösen. Each worker thread may look something like the following:

```
class WorkerThread extends Thread {
   @Override
   public void run() {
      PsTableGroup.registerWorkerThread();
      Table table = PsTableGroup.getTable(0);
      ...
      PsTableGroup.deregisterWorkerThread();
   }
}
```

4.1.4 Shutdown JBösen

When all worker threads have been de-registered, the last thing to do is to shut down JBösen. This is done simply by calling PsTableGroup.shutdown().

4.2 PS Application

The motivation behind the PS Application interface is to hide the boiler plate code in order to simplify the application implementation efforts. In the previous interface, there are quite a few functions that needs to be called in every applications. The init() functions needs to be called before other methods in the PsTableGroup interfaces can be invoked. Each worker thread needs to register and de-register with the system at the beginning and the end of thread execution. The programmers needs to spawn the application threads by themselves instead of letting the system spawning the threads automatically. It is quite reasonable to assume that most application code will launch more than one application threads due to data parallelism. Ultimately, we try to abstract away the system interfaces and allow ML programmers to only focused on the algorithm implementations.

The PS Application interface is simplified that all of the aforementioned methods are done by the system instead of the application code. The interface reduced the requirements the users need to implement two functions, initialize() and runWorkerThread(int threadId) in the PsApplication class. A complete PsApplication interface can be found in B.

4.2.1 Initialize

In the initialize(), the users are expected to create all the tables needed for the application and initialized any variables that is needed for creating the tables. It is also a good time for the application threads to load the data from the memory, local disk or distributed storage system. For example, a simple application that creates one table would look like this:

```
@Override
public void initialize() {
   int rowCapacity = PsTableGroup.getNumLocalWorkerThreads();
   PsTableGroup.createDenseIntTable(CLOCK_TABLE_ID, staleness,
      rowCapacity);
}
```

4.2.2 runWorkerThread

The runWorkerThread(int threadId) function contains the application logic. This assumes the application takes a data parallel model approach to partition the training data and computation to each application thread. The input threadId variable is a globally unique threadId. Usually, the application will perform computations using the local training data, push updates at the end

of iteration by calling PsTableGroup.clock(), and read off updates from the servers to reflect the updates of other clients. Here is an simple example that simply adding 1 to the parameters at each iterations:

```
@Override
public void runWorkerThread(int threadId) {
    ...

    // Get table by ID.
    IntTable clockTable = PsTableGroup.getIntTable(CLOCK_TABLE_ID);

    for (int iter = 0; iter < numIterations; iter++) {
        ...

        // Increment the clock for this thread.
        table.inc(clientId, threadId, 1);

        // Finally, finish this clock.
        PsTableGroup.clock();
    }

    // globalBarrier makes subsequent read all fresh.
    PsTableGroup.globalBarrier();

    // Read updates. After globalBarrier all updates should
    // be visible.
    for (int cliId = 0; cliId < numClients; cliId ++) {
        IntRow row = clockTable.get(cliId);
    }
}
```

4.3 Application Matrix Fact

We have included in C a Matrix Factorization implementation using the JBösen PsApplication interface to demonstrate the simplicity of the APIs. This implementation runs SGD on the local training data for a number of iterations.

4.4 The Row Interface and the User Defined Row Type

The table interface 3.1 used in the JBösen choose rows as the minimum unit of data. In JBösen, we implemented the rows for all the primitive types: Integer, Double, Long and Float, but we also provide a user defined row interface for those who wish to implement their own row type.

This feature is an extension to the table interface and can greatly expand the programability of the system. For example, consider the following row type where in addition to row types offered in the table interface, there is a value ε associated with each row and the values are automatically set to 0 if the value in the row become between $-\varepsilon$ and ε. To implement this example with the user defined row feature can require great engineering efforts to work around.

To implement a row type, there are four piece of information required, a row type with a row create factory, and a row update type with a row update factory, where the factory interface is used to create the respective row and row update. As explained in the previous sections, the JBösen uses rows and row updates separately where the value of the rows can be considered as the aggregate of a series of row updates. The users therefore is required to implement this aggregation through inc() interface. For example, in all the provided implementation of row type with the primitive types, the inc() aggregation operation is implemented as summation.

Note, besides the aggregation, creation, and serialize de-serialize, the row interface does not contain any other methods discussed before in the thesis such as the get() or methods that should be fundamental to the type of operation such as contain() (if a value is contained in the row). This is a design choice deliberately delegated to the users. Conceptually, from the perspective of the JBösen system, the table interface is a collection of rows with rowIds that can be aggregated, created, and packed for network communication. To create a table, the interface createTable() takes in only a row factory and a row update factory object. The other functionality of the rows are used in the application logic implemented by the users. In this way, the users have the full control to tailor the row types to their needs. A complete row interface is attached in appendix A.

Chapter 5

Performances

We emphasize that JBösen is for the moment about extending the Petuum Project to Java users. It focuses on incorporating design principles that exploit error tolerance and data parallelism from the existing Petuum project and an easy-to-use programming interface for Java users. Currently, there is only a limited number of applications implemented on top of the JBösen system, but more applications are coming in the near future. To demonstrate the performance potential of the JBösen system, we use the existing Matrix Factorization (MF) application implemented on top of JBösen.

Figure 5.1: Left: JBösen scaling performance using MF application. Right: JBösen speedup using MF application

We demonstrate the scale performance of the MF application implemented on top of JBösen. In Figure 5.1, we demonstrate using 1, 2, and 4 nodes clusters running a Netflix Rating dataset, the application achieved 3.8 speedup with 4 nodes and 1.9 speedup with 2 nodes. This result shows that the system has the potential to scale well with increasing number of machines. The testing cluster we used is a 4 nodes cluster, each machine with 16 Intel 2.1 GHz cores, 128 GB RAM and 10Gbps Ethernet. We use the Netflix Ratings dataset with a 480189×17770 matrix and $100,480,507$ ratings.

For cross-platform comparisons, we did not find a fair benchmark for the JBösen, although we compared the performance to an older version of the Petuum ML application on top of Bösen since it incorporates similar design principles such as PS and SSP. The ML application on top of JBösen performed around 5x faster and converged around 1.5x faster using the same Netflix

Ratings dataset and the 4 nodes cluster, but this result is not a rigorous comparison because the newest release of Bösen includes more improvements and performance optimization. We could not use the newest release for comparison because the most current Petuum MF application is implemented on top of another system STRADS [2] under the Petumm project, not Bösen. With more applications implemented on top of JBösen down the road, we will be able to evaluate the performance of the system in a more rigorous manner. We will have more benchmark to compare to whereas currently it's hard to find a fair comparison. We will also be able to evaluate the performance of the system separately from the performance of the application implementations.

Chapter 6

Conclusion and Future Work

The key contribution of this thesis is a Java version of the bounded-asynchronous key-value store system for distributed ML algorithms with Big Data. It provides a simple programming model that covers a range of ML algorithms focused on simple programming interfaces instead of performance; it exploits data parallelism for Big Data using the parameter server architecture; it exploits the error tolerance nature of the ML algorithms. As more than 25% of programmers in this world use Java, JBösen is a great contribution to the Petuum project as it expand the project to reach these Java programmers.

Our work is just the beginning of the JBösen project. It has many potentials to improve and expand in terms of both performance and functionality. The primary plans in the near future is to expand the library of ML applications implemented on JBösen. One of the vision of the Petuum project is to provide a complete software stack for Big ML, from the system to out-of-box ML applications. JBösen, as part of the Petuum project, shares the same vision. There have already been plans for a new Machine Learning Ranking and Latent Dirichlet Allocation application on JBösen and more will come soon.

JBösen will no doubt continue to improve on both its performance optimization and usability. We have experimented new optimization with a proof of concept implemented on top of Bösen, since JBösen at the time are still being developed. Under the Petuum project, based off on the idea of model parallelism, there is a model parallel framework, STRADS [2], designed to ensure efficient and correct model parallel execution of ML algorithms. While the Bösen allows ML algorithms to be expressed as a series of iterative updates to a shared model, STRADS allows these parameter updates to be scheduled by discovering and leveraging structure properties of ML algorithms. In the near future, there are plans to integrate these two systems together as their design goals complement each other. In anticipating the plan, we explored a derivation of the SSP Push consistency model where we use schedule information to improve the network performance of the system.

Finally, the other aspect of JBösen we did not discuss in this thesis is fault tolerant with auto-recovery. Currently, JBösen does not offer any auto-recovery features for node failure which is considered to be essential for systems to scale beyond thousands of machines. The emphasis of the JBösen and by extension the Petuum project is, for the moment, largely to allow ML practitioners to implement and experiment with new ML algorithms on small to medium clusters. In fact, the infrastructure already exists for JBösen to build simple fault tolerance. The Clients

is stateless with training data unchanged throughout the lifetime of the system. The servers can use simple replications of table data to increase the fault tolerance of the system. However, the performance implication of these features on both JBösen and Bösen are under studied. In the future, as the aim of JBösen and Petuum project expand to cover larger size clusters, fault tolerant would be the first feature to add to the system.

Appendices

Appendix A

Complete Row Interface

Below is the complete row interface.

A.1 Row Update

The JBösen is implemented against the following row update and row update factory interface:

```
/**
 * This interface specifies a row update for a B{\" o}sen PS table. An
 * implementation of this interface does not need to be thread-safe.
 */
public interface RowUpdate {

   /**
    * Increment this row update using a RowUpdate object.
    *
    * @param rowUpdate the RowUpdate object to apply to this row
    *              update.
    */
   void inc(RowUpdate rowUpdate);

   /**
    * Serialize this row update into a ByteBuffer object.
    *
    * @return a ByteBuffer object containing the serialized data.
    */
   ByteBuffer serialize();
}

/**
 * This interface specifies a factory for row updates. An
    implementation of
```

```
 * this interface must be thread-safe.
 */
public interface RowUpdateFactory {

   /**
    * Create and return a new row update.
    *
    * @return new row update constructed by this factory.
    */
   RowUpdate createRowUpdate();

   /**
    * Create a new row update from serialized data.
    *
    * @param buffer buffer containing serialized data for a row
       update.
    * @return new row update constructed from the serialized data.
    */
   RowUpdate deserializeRowUpdate(ByteBuffer buffer);
}
```

A.2 Row Create Factory

The JBösen is implemented against the following row and row create factory interface:

```
/**
 * This interface specifies a row in a JB{\" o}sen PS table. An
    implementation of
 * this interface does not need to be thread-safe.
 */
public interface Row {

   /**
    * Increment this row using a RowUpdate object.
    *
    * @param rowUpdate the RowUpdate object to apply to this row.
    */
   void inc(RowUpdate rowUpdate);

   /**
    * Serialize this row into a ByteBuffer object.
    *
    * @return a ByteBuffer object containing the serialized data.
```

```
     */
    ByteBuffer serialize();
}

/**
 * This interface specifies a factory for rows. An implementation of
     this
 * interface must be thread-safe.
 */
public interface RowFactory {

    /**
     * Create and return a new row.
     *
     * @return new row constructed by this factory.
     */
    Row createRow();

    /**
     * Create a new row from serialized data.
     *
     * @param buffer buffer containing serialized data for a row.
     * @return new row constructed from the serialized data.
     */
    Row deserializeRow(ByteBuffer buffer);
}
```

Appendix B

Complete PsApplication Interface

```
/**
 * This class abstracts away some of the boilerplate code common to
    application
 * initialization and execution. In particular, this class takes care
    of
 * PsTableGroup initialization and shutdown, as well as worker thread
 * spawning, registering, and de-registering. An application using
    this class
 * should extend from it and implement initialize() and
 * runWorkerThread(int).
 */
public abstract class PsApplication {

   /**
    * This method is called after PsTableGroup is initialized and
    * before any worker threads are spawned. Initialization tasks
       such as
    * table creation and data loading should be done here.
       Initializing
    * PsTableGroup is not required.
    */
   public abstract void initialize();

   /**
    * This method executes the code for a single worker thread. The
    * appropriate number of threads are spawned and run this method.
    * Worker thread register and de-register are not required.
    *
    * @param threadId the local thread ID to run.
    */
   public abstract void runWorkerThread(int threadId);
```

```
/**
 * This method runs the application. It first initializes
    PsTableGroup,
 * then calls initialize(), and finally spawns
 * numLocalWorkerThreads threads, each of which runs a single
 * instance of runWorkerThread(int). This method returns after all
 * worker threads have completed and PsTableGroup has been shut
 * down.
 *
 * @param config the config object.
 */
public void run(PsConfig config) {
   PsTableGroup.init(config);
   initialize();
   int numLocalWorkerThreads =
      PsTableGroup.getNumLocalWorkerThreads();
   Thread[] threads = new Thread[numLocalWorkerThreads];
   for (int i = 0; i < numLocalWorkerThreads; i++) {
      threads[i] = new PsWorkerThread();
      threads[i].start();
   }
   for (int i = 0; i < numLocalWorkerThreads; i++) {
      try {
         threads[i].join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
   PsTableGroup.shutdown();
}
}
```

Appendix C

Matrix Factorization implemented Using JBösen APIs

Some code are omitted for clarity purposes.

```
// Main worker thread logic.
  @Override
  public void run() {
     // Get table by ID.
     DoubleTable LTable = PsTableGroup.getDoubleTable(LTableId);
     DoubleTable RTable = PsTableGroup.getDoubleTable(RTableId);

     ...

     int evalCounter = 0;
     for (int epoch = 1; epoch <= numEpochs; epoch++) {
        double learningRate = learningRateEta0 *
           Math.pow(learningRateDecay, epoch - 1);
        for (int batch = 0; batch < numMiniBatchesPerEpoch; ++batch)
           {
           for (int ratingId = elemMiniBatchBegin;
                 ratingId < elemMiniBatchEnd; ratingId++) {
              Rating r = ratings.get(ratingId);
              MatrixFactCore.sgdOneRating(r, learningRate, LTable,
                    RTable, K, lambda);
           }
           PsTableGroup.clock(); // clock every miniBatch.
        }

        evalCounter++;
     }
     PsTableGroup.globalBarrier();
```

```
    // Print all results.

    ...

}

// Perform a single SGD on a rating and update LTable and RTable
// accordingly.
public static void sgdOneRating(Rating r, double learningRate,
      DoubleTable LTable, DoubleTable RTable, int K, double
         lambda) {
    int i = r.userId;
    int j = r.prodId;
    // Cache a row for DenseRow bulk-read to avoid locking at each
       read
    // of entry.
    DoubleRow LiCache = LTable.get(i);
    DoubleRow RjCache = RTable.get(j);

    double pred = 0; // <Li, Rj> dot product is the predicted val
    for (int k = 0; k < K; ++k) {
       pred += LiCache.get(k) * RjCache.get(k);
       assert !Double.isNaN(pred) : "i: " + i + " j: " + j + " k: "
          + k;
    }

    // Now update L(i,:) and R(:,j) based on the loss function at
    // X(i,j).
    // The loss function at X(i,j) is ( X(i,j) - L(i,:)*R(:,j) )^2.
    //
    // The gradient w.r.t. L(i,k) is -2*X(i,j)R(k,j) +
    // 2*L(i,:)*R(:,j)*R(k,j).
    // The gradient w.r.t. R(k,j) is -2*X(i,j)L(i,k) +
    // 2*L(i,:)*R(:,j)*L(i,k).
    DoubleRowUpdate LiUpdate = new DenseDoubleRowUpdate(K);
    DoubleRowUpdate RjUpdate = new DenseDoubleRowUpdate(K);
    double grad_coeff = -2 * (r.rating - pred);
    double nnzRowi = LiCache.get(K);
    double nnzColj = RjCache.get(K);
    for (int k = 0; k < K; ++k) {
       // Compute update for L(i,k)
       double LikGradient = grad_coeff * RjCache.get(k)
          + 2 * lambda / nnzRowi * LiCache.get(k);
       LiUpdate.set(k, -LikGradient * learningRate);
       // Compute update for R(k,j)
       double RkjGradient = grad_coeff * LiCache.get(k)
```

```
            + 2 * lambda / nnzColj * RjCache.get(k);
        RjUpdate.set(k, -RkjGradient * learningRate);
      }
      LTable.inc(i, LiUpdate);
      RTable.inc(j, RjUpdate);
   }
```

Bibliography

[1] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *AAAI Conference on Artificial Intelligence*, 2015. 1, 2.2, 2.3, 2.4

[2] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. *Advances in neural information processing systems*, 2014. 5, 6

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014. 1

[4] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A frame-work for machine learning and data mining in the cloud. In *PVLDB*, 2012. 1

[5] Priit Potter. How many java developers are there in the world?, 2012. URL `https://plumbr.eu/blog/java/how-many-java-developers-in-the-world`. 1

[6] R. Power and J. Li. Piccolo. Building fast, distributed programs with partitioned tables. In *OSDI*, 2010. 1

[7] Jinliang Wei, Wei Dai, Abhimanu Kumar, Xun Zheng, Qirong Ho, and E. P. Xing. Consistent bounded-asynchronous parameter servers for distributed ml. *Manuscript*, 2013. 1

[8] T. White. *Hadoop: The definitive guide*. O'Reilly Media,Inc., 2012. 1

[9] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2015. 1, 2.1, 2.1

[10] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud 2010*, 2010. 1