Brook: An Easy and Efficient Framework for Distributed Machine Learning

Chao Ma, Yan Ni, and Zhen Xiao*
Department of Computer Science and Technology
Peking University
{machao, niyan, xiaozhen}@net.pku.edu.cn

Abstract

We present Brook, a new framework for distributed machine learning problems. Like some previous frameworks, Brook adopts the parameter server paradigm that simplifies the task of distributed programming. Unlike these frameworks, we build a novel system component called parameter agent that masks the communication details between workers and servers by mapping remote servers to local in-memory file. In this way, Brook provides a simple and platform-independent interface called RWW, where users can migrate existing single-machine programs, written in any programming language, to the distributed environment with minimal modification. In addition, to achieve system efficiency and scalability, Brook is designed to minimize network traffic, maximize CPU and memory utilizations, and support flexible fault-tolerance strategies. Our evaluation results show that Brook has the highly competitive performance and scalability, while providing enhanced ease of use compared to existing frameworks.

1 Introduction

Machine learning (ML) is becoming the primary mechanism to extract useful knowledge from Big Data. To improve the accuracy, ML methods tend to use models with more parameters trained on large numbers of examples. However, due to the computation and storage limit of a single machine, executed in a distributed manner has become a prerequisite for solving large-scale ML problems.

The data-flow frameworks such as Hadoop [4] and Spark [2], have significantly simplified the task of building large-scale data processing on commodity clusters. Based on these frameworks, the distributed ML libraries such as Mahout [5] and MLI [6], have been widely used in both academia and industry. However, most of these frameworks adopt the iterative MapReduce [1] paradigm that mandates synchronous and coarse-grained computation and communication. This inherent system design for batch tasks incurs great inefficiency and is often inadequate when building the "Big Model" ML applications such as large-scale sparse logistic regression, massive topic model and deep networks. Parameter server paradigm [7] has recently emerged as an efficient approach to resolve the "Big Model" ML challenge. Under this paradigm, both the training data and workloads are spread across worker nodes, while the server nodes maintain the globally shared parameters. In contrast to the iterative MapReduce paradigm, computation and communication in parameter server can be asynchronous and fine-grained, and hence can improve the CPU utilization and reduce the communication cost dramatically.

The aforementioned frameworks have proven to be tremendously useful for simplifying the task of building distributed ML applications. However, almost all of them force users to re-write their existing code in a new software stack, many of which expose an unfamiliar programming environment to

*The contact aurthor is Zhen Xiao.

users. For example, on one hand, many ML developers are accustomed to using powerful, high productivity array-languages such as Matlab, R and Numpy. For these users, especially inexperienced ones, the steep learning curve of the new programming platforms, including both the programming language and the programming model, is a major obstacle to the adoption of the new frameworks. On the other hand, some skilled developers in ML domain prefer to use more efficient programming languages such as C/C++, as well as the high-performance hardwares such as GPGPU to build ML applications which can be tuned to give extremely good performance. For these users, most of the popular frameworks, such as Spark, cannot satisfy their needs. So here comes a natural question: can we build a new framework where users can re-use their single-machine code easily and the framework is agnostic of the underlying programming platforms?

To answer this question, in this paper we propose Brook, a new framework which allows developers to completely get rid of the restrictions of the programming platforms. Using Brook, developers can migrate existing single-machine ML programs, potentially written in any programming language or executed on the specific underlying hardwares such as GPGPU, to a distributed environment for concurrent execution with little change, while achieving the similar fault-tolerance guarantees and the enhanced performance compared to existing frameworks. To achieve these goals, Brook extends the original parameter server paradigm by adding a novel component called parameter agent and its counterparts. Section 2 explains how these components work closely together and demonstrates how users build ML applications via the simple and language-agnostic interface RWW. Furthermore, to achieve high performance and scalability, both computational intensive workloads and the volume of data communication demand careful system design and optimization. In section 3, we discuss the optimization approaches used in Brook, including the vector store, message compression, and flexible fault-tolerance strategies. In the last section, we show a preliminary performance evaluation of our system.

2 System Architecture

This section describes our system architecture. In our presentation, we first provide an overview of Brook (§2.1) and then introduce the programming model based on the RWW interface (§2.2). After that, we make a comparison between Brook and the existing solutions for the cross-language programming in ML systems, and show the advantages of Brook (§2.3). Finally, we discuss the implementation of our system (§2.4).

2.1 Overview

Brook's execution environment consists of one master process and many server and worker processes, each executing on a potentially different machine. Figure 1 illustrates the overall interactions among the master, servers and workers when executing a Brook program. As Figure 1 shows, a server node stores assigned parameters partition in its memory and handles the aggregation and update operations associated with that partition. Each worker node is responsible for storing a portion of the training data to compute local updates such as gradient. The master node maintains the bookkeeping of each worker and server process, which can be recovered without interrupting the computation when it crashes by non-catastrophic machine failures. Similar to existing frameworks, we assume that the master node failures are rare and hence provide no protection for that. Note that workers communicate only with the servers and the master, not among themselves.

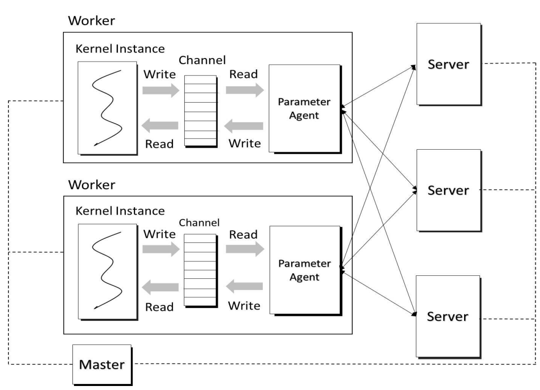

Figure 1: Brook Architecture

In Figure 1, a ML task is divided among all of the workers, which jointly learn the globally shared parameters by computing the local updates based on its own training data. These local updates can then be sent to servers via a specific partition algorithm in the worker nodes. Servers aggregate all of these updates before applying them to the corresponding parameters and later send these updated parameters back to the corresponding workers, as the response of their requests. This series of processes is similar to that in the original parameter server.

The main difference between Brook and the original parameter server model is that the client process in Brook dose not communicate with the server nodes directly. Instead, at run-time each worker process derives two child processes, including the client (also called kernel instance in Brook) and the parameter agent. Workers communicate with the servers through the parameter agent processes and Brook combines the kernel instance and the parameter agent with the communication channel, which is a platform-independent abstraction for local data transport and synchronization. We now describe the details of these components in terms of the basic functions and the interactions of them.

Kernel Instance. Kernel instance is provided by the application developer and started as an independent process that can be executed on any language platform. The effect of a kernel instance is to repeatedly read new parameters from channel, compute local updates, and write both the updates and the request to channel. As will be discussed later, because the parameter agent abstraction hides the communication details, the distributed shared parameters appear to be local. This allows the kernel instance to easily retro-fit existing single-machine implementations with minimal modification.

Parameter Agent. Parameter agent acts as a middleman between the client and the servers. Brook makes use of the abstraction of the parameter agent to simplify the logic of the client process. The cumbersome system work such as network communication, message queue and serialization will be taken over by the parameter agent process. By doing so, a client process can focus only on the implementation of the core algorithm in ML, based on any underlying programming platform, and just read from and write to channel via the simple and language-agnostic interface to share data with the parameter agent process.

Channel. Channel consists of a data channel which is responsible for local data transport, and a signal channel which manages the synchronization. The concrete implementation of the channel abstraction is file, since all programming languages can access it in a consistent way. There are two types of files used in a channel. For the data channel, we use the in-memory file which is like a regular file but based on Ramfs with at least two orders of magnitude improvement in performance. For the signal channel, we use FIFO, which is easy to perform blocking I/O between two processes.

Based on the foregoing descriptions, in the next section we demonstrate how users build their distributed ML applications on Brook by using the RWW interface.

2.2 Programming model

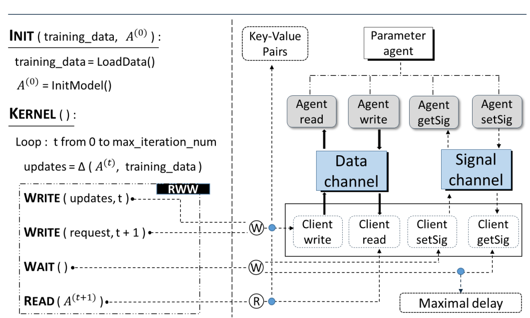

Figure 2: Programming model

Although ML algorithms come in many forms, almost all of them seek a set of parameters to a global model A that best summarizes or explains the input data $\mathscr{D}$. Such problems are usually solved by iterative-convergent algorithms, many of which can be abstracted as the additive form: $A^{(t+1)} = A^{(t)} + \Delta(A^{(t)}, \mathscr{D})$, where $A^{(t)}$ is the state of model parameters at iteration t, $\mathscr{D}$ is the input data, and the kernel function Δ computes the model updates from $\mathscr{D}$. Using Brook, developers focus on the implementation of the kernel instance, where the program will be distributed over worker nodes and run concurrently at run-time for computing their local updates such as gradients. As shown in Figure 2, we see that developers can easily migrate existing single-machine code to the Brook environment by invoking a set of functions called RWW at the end of each iteration in a ML task. These functions include the Write, the Read and the Wait. As we mentioned before, the RWW interface gives developers an easy and platform-independent way to communicate with the parameter agent process. Figure 2 illustrates that, the Write and the Read functions are responsible for local data transport by writing to and reading from data channel, while the Wait function manages the process synchronization through the signal channel by getting and setting the iteration timestamps (also called vector clock) during Brook's run-time. We note that, the

RWW interface could be implemented either by users or by system developers, since the protocol of data transport and synchronism that used in this interface are extremely simple.

Several features about the RWW interface are also worth noting. First, Brook supports a number of data types for the cross-platform data transport, such as the text data, the 3rd-party message libraries such as protobuf and Thrift, as well as the native binary data. Second, developers can configure the value of the maximal delay in the Wait function, which could give developers the opportunity to implement different consistency models such as the BSP, the SSP [24] and the ASP. The core technique under the flexible consistency model mechanism is a signal queue. Furthermore, we support user-defined update mechanism on server nodes by using the expression template, a simple and efficient programming language trick of C++.

2.3 Related solutions

In contrast to our system, there are two common ways of the cross-language programming that have been used in existing ML frameworks. The first way, many ML frameworks make use of the languages wrapper around their original API. Such as a python wrapper for the original C++ API by using the Boost or the SWIG [9] library. In contrast to Brook, the disadvantages of this solution are obvious. First, building the languages wrapper is not an easy job, since we have to modify the source code of the original framework and it might be inadequate for the inexperienced users who need to do this by themselves. A real example in the open source community is the SparkR [3], which is published in the latest version of Spark recently. So, in other words, it is unrealistic to build the wrappers for every programming languages which might be used by existing single-machine ML applications. However in Brook, the RWW interface is totally language-agnostic. Apart from this, building languages wrapper naively is often lack of both efficiency and flexibility.

The second way, that is, the Hadoop streaming [10], which is widely used as it frees programmers from Java language, which makes developers use power of Hadoop more easily. Similar to Brook, Hadoop streaming uses the 3rd-party medium (standard input/output) to transport data over processes. However, as we mentioned before, the inherent system design of Hadoop is not suitable for the ML applications in both the programing model and the system performance. In addition, Hadoop streaming transports all of the training data through the standard IO, which can incurs great system overhead. By contrast, using Brook, we only transport the data of parameters trough the channel.

2.4 Implementation

Brook is implemented in C++ and requires no change to the underlying OS or compiler. Our implementation re-uses a number of existing libraries such as MPICH2 for communication (supports both the Ethernet and the InfiniBand network), Google's protobuf for object serialization, and Snappy for data compression. Brook can run over the cluster resource manager such as Yarn [11] or Mesos [12], and also provides an easy way for deployment on clusters by using the docker [13] container.

3 System Optimization

Implementing an efficient and scalable distributed framework is not an easy job, because both the volume of data communication and the intensive computational workloads demand careful system design and optimization. In this paper, we focus on three optimization techniques:

Message compression. Since distributed machine learning problems typically require high bandwidth, message compression is desirable. We use several compression approaches in our system to reduce the network traffic as mush as possible. First, we avoid sending single items because both the communication overhead that caused by TCP/IP package header and the serialization overhead are horrible. Hence we pack all of the single items into a batched message form. Second, instead of using (key, value) pair to represent each item, the message consists of a list of (start-key, value-list), where the value-list is a sequence of consecutive values. This simple modification can greatly reduce the size of each message especially when message tends to be dense. Next, since many ML problems may use the same training data in different iterations, we cache the key lists in the receiving nodes. Later the senders can send only a hash value of this list rather than the list itself. Finally, we use the Snappy compression library to compress the serialized message.

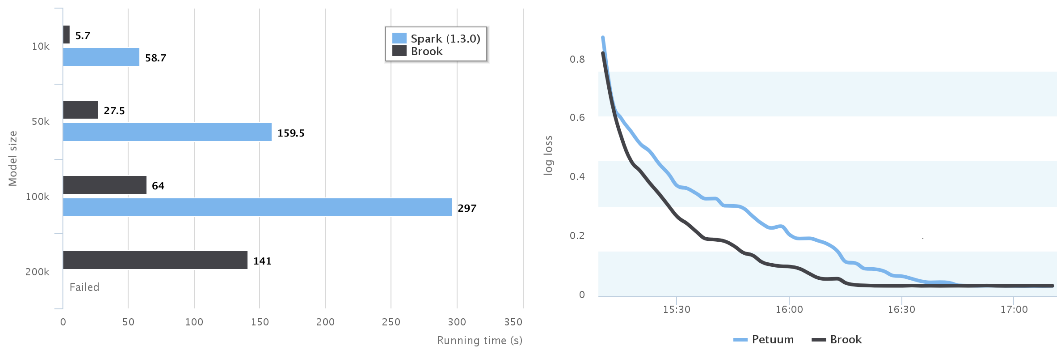

Figure 3: **Left:** System performance under the BSP consistency, versus Spark (1.3.0 version, 100 iterations), on logistic regression. **Right:** Convergence rate under the SSP consistency, versus Petuum (800-million parameters, maximal delay = 4), on large-scale sparse logistic regression. All datasets come from the Criteo CTR [19]. We note that, the left diagram shows that, the Spark task failed when we scale-up the mode size to 200k.

Vector store. Many previous systems use key-value table to store the globally shared state during the run-time [23]. However, using this abstraction naively is inefficient in both the memory utilization and the computation. We found that operations in server nodes are typically represented as linear algebra computations. Hence in Brook, we use the vector store where the underlying implementation is the contiguous region of memory that stores only the consecutive values ordered by its potential index, while the non-existing items are associated with zeros automatically. Thus, we can save at least half of the memory cost and achieve efficiency by leveraging the high-performance multithreaded linear algebra libraries such as the OpenBLAS [15]. It also simplifies the system design for user-defined update mechanism on the server nodes. A similar approach is also used in the previous work [8].

Flexible level of fault-tolerance. At scale, fault-tolerance is critical. However, most of the existing systems can only support one fixed fault-tolerance strategy, which is inflexible and often incurs much unnecessary overhead when system is deployed in a small, well-controlled cluster. Fortunately, in Brook, we give developers the opportunity to configure the fault-tolerance level, which could range from L0 to L3 and covers the cluster types from the small platform such as desktop PCs and lab-clusters, to the big, less predicable platform such as large data center or the cloud. Brook will choose the different backup strategies such as backup on remote nodes or backup locally, according to the different fault-tolerance level.

We will demonstrate at the workshop the concrete method about the flexible fault-tolerance strategies. We also plan to demo some other optimizations such as skip-list buffer, memory zero-copy and signal queue. Each of them has improved our system performance considerably.

4 Preliminary Evaluation

We evaluated Brook on a cluster of 15 machines in our own laboratory. Each machine has a 8 cores Intel Xeon E5620 (2.40GHz) processor with 26GB memory. We compared Brook with Spark and Petuum under the BSP and the SSP consistency models, respectively. Our evaluation results show that Brook can outperform its competitors in both system efficiency and scalability . The highlights of our results can be found in Figure 3. During the workshop, we plan to demo more detailed evaluation of our system.

5 Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was supported by the National Grand Fundamental Research 973 Program of China under Grant No.2014CB340405, and the National Natural Science Foundation of China under Grant No.61170056 and No.61572044.

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

[2] Zaharia, Matei, et al. "Spark: cluster computing with working sets." *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing.* Vol. 10. 2010.

[3] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* USENIX Association, 2012.

[4] White, Tom. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[5] Apache. Apache mahout: Scalable machine learning and data mining. `http://mahout.apache.org`, October 2015.

[6] Sparks, Evan R., et al. "MLI: An API for distributed machine learning." *IEEE 13th International Conference on Data Mining (ICDM), IEEE, 2013.*

[7] Dean, Jeffrey, et al. "Large scale distributed deep networks." *Advances in Neural Information Processing Systems.* 2012.

[8] Li, Mu, et al. "Scaling distributed machine learning with the parameter server." *Proc. OSDI.* 2014.

[9] Beazley, D.M. Automated scientific software scripting with SWIG. Future Gener. Comput. Syst. 19 (July 2003), 599-609.

[10] Ding, Mengwei, et al. "More convenient more overhead: the performance evaluation of Hadoop streaming." *Proceedings of the 2011 ACM Symposium on Research in Applied Computation.* ACM, 2011.

[11] The Apache Software Foundation. Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/.

[12] Hindman, Benjamin, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." *NSDI.* Vol. 11. 2011.

[13] Docker: http://www.docker.com

[14] Eric P. Xing, Qirong Ho, et al. "Petuum: A New Platform for Distributed Machine Learning on Big Data" *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2015.

[15] J.J.Dongarra, J.Du Croz, S.Hammarling, and R.J.Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18-32, 1988.

[16] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. "Spartan: A Distributed Array Framework with Smart Tiling." *Proc. of the USENIX Annual Technical Conference (ATC 2015),* July 2015.

[17] Qi Chen, Jinyu Yao, and Zhen Xiao. "LIBRA: Lightweight Data Skew Mitigation in MapReduce." *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, September 2015.

[18] Qi Chen, Cheng Liu, and Zhen Xiao. "Improving MapReduce Performance Using Smart Speculative Execution Strategy." *IEEE Transactions on Computers (TC)*, April 2014.

[19] Zhen Xiao, Qi Chen, and Haipeng Luo. "Automatic Scaling of Internet Applications for Cloud Computing Services." *IEEE Transactions on Computers (TC)*, May 2014.

[20] Zhen Xiao, Weijia Song, and Qi Chen. "Dynamic Resource Allocation using Virtual Machines for Cloud Computing Environment." *IEEE Transactions on Parallel and Distributed Systems (TPDS),* June 2013.

[21] Li, Hao, et al. "MALT: distributed data-parallelism for existing ML applications." *Proceedings of the Tenth European Conference on Computer Systems.* ACM, 2015.

[22] Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." *ACM SIGOPS Operating Systems Review.* Vol. 41. No. 3. ACM, 2007.

[23] Chilimbi, Trishul, et al. "Project adam: Building an efficient and scalable deep learning training system." *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* 2014.

[24] Qirong Ho, James Cipar, Eric P.Xing. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server.", *Neural Information Processing Systems*, 2013. (NIPS 2013)