Dolphin: Runtime Optimization for Distributed Machine Learning

Byung-Gon Chun[1], **Brian Cho**[2], **Beomyeol Jeon**[1], **Joo Seong Jeong**[1], **Gunhee Kim**[1], **Joo Yeon Kim**[1], **Woo-Yeon Lee**[1], **Yun Seong Lee**[1], **Markus Weimer**[3], **Youngseok Yang**[1], **Gyeong-In Yu**[1]
BGCHUN@SNU.AC.KR,BCHO@FB.COM,BEOMYEOLJ@GMAIL.COM,JOOSJEONG@GMAIL.COM,GUNHEE@SNU.AC.KR,
JOOYKIM@SNU.AC.KR,WOOYEONLEE0@GMAIL.COM,YUNSEONG@SNU.AC.KR,MWEIMER@MICROSOFT.COM,
JOHNYANGK@GMAIL.COM,GYEONGIN@SNU.AC.KR
[1]Seoul National University, [2]Facebook, [3]Microsoft

Abstract

Large-scale machine learning (ML) systems are becoming widely used. Typically, these ML systems run on fixed resources, but it is difficult to find their optimal configurations (e.g., how many nodes to use, how to distribute data) since they depend on multiple factors such as hardware environments, ML algorithms, input datasets, etc. Furthermore, optimal configurations often can change over time due to fluctuating cluster resources and changing ML algorithm patterns. In this paper, we present Dolphin, an elastic machine learning framework that addresses the configuration problem at runtime. Dolphin solves a cost-based optimization problem to find an optimal configuration and reconfigures the system dynamically at runtime. Dolphin introduces a new distributed memory abstraction to change resource and data configurations based on the optimizer plan transparently and efficiently.

1. Introduction

Large-scale machine learning (ML) is a key ingredient to realize the value proposition of the Big Data revolution. Successful applications include topic modeling, recommendation, classification and deep learning for sensory data (Smola & Narayanamurthy, 2010; Li et al., 2014; Chilimbi et al., 2014; Dean et al., 2012). Machine learning algorithms differ from most other Big Data applications in their iterative-convergent behavior: they iteratively update the model until the model converges or a maximum number of iterations is reached. Parameter Servers (Smola & Narayanamurthy, 2010) have been introduced to overcome the scalability challenges in such a system. They facilitate *asynchronous* parallel computation. Training data is distributed across *workers*, which synchronize model parameters via a set of *servers* by pushing updates to them and pulling up-to-date parameter values from them.

The computational performance of a machine learning program on a given data set can be quantified by the time required to reach model convergence. In particular, the computational performance of the Parameter Server approach is influenced by the set of workers and servers as well as the distribution of data across the workers, which together form the configuration of the Parameter Server. Prior work not only validates this configuration-dependent performance, but it also provides ways to perform a static optimization of this configuration (Yan et al., 2015). However, it is difficult to find a fixed optimal configuration both from a systems perspective and from a machine learning perspective. Parameter update frequencies are not uniform; neither over time nor across parameters. Similarly, the available cluster resources are not static for the duration of the training, which leads the resource manager (e.g., YARN (Vavilapalli et al., 2013) or Mesos (Hindman et al., 2011)) to either offer more (free) resources or preempt already allocated resources during training. Lastly, performance is not uniform over time for a single compute node and across nodes, which necessitates workload rebalancing.

This gives rise to the need for *runtime optimizations* of the Parameter Server configuration. The runtime optimization in turn requires solutions to two technical challenges: (1) an optimizer and an associated data collection scheme to decide when and how to change the configuration, and (2) a Parameter Server system that can update its configuration during runtime.

Dolphin is a framework that supports runtime configuration optimization in the Parameter Server architecture by addressing these two challenges. Dolphin performs cost-based optimization at runtime. Dolphin monitors the computation and communication time in the system, estimates the iteration time based on a cost model, and computes

ML Systems Workshop - 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016.

an optimal configuration to maximize overall performance. When Dolphin decides to change its configuration, Dolphin relies on an underlying distributed memory system, called Elastic Memory Store, to execute the reconfiguration actions efficiently.

We present related work and then describe the Dolphin framework in the rest of the paper.

2. Related Work

Much research has been done on distributed machine learning systems. We briefly summarize works that are most relevant to Dolphin.

Spark MLlib (The Apache Software Foundation, 2016) is an ML library built on Spark. Spark MLlib performs synchronous execution, i.e., there is a barrier to synchronize model updates at every iteration. Mahout (The Apache Software Foundation, 2015) is an ML library targeting H2O, Flink and Spark as backend execution engines. GraphLab (Low et al., 2012) represents ML algorithms as graphs and executes them with a graph processing engine. TensorFlow (Abadi et al., 2015) is an ML framework that is built on the DAG processing of tensors.

In recent years, the Parameter Server architecture has been successfully used for large-scale machine learning systems: e.g., Petuum (Xing et al., 2015), ParameterServer (Li et al., 2014), Yahoo! LDA (Smola & Narayanamurthy, 2010), Adam (Chilimbi et al., 2014), and DistBelief (Dean et al., 2012). Petuum and Parameter Server provide ways to trade between convergence and system efficiency by bounding the staleness of model updates. Adam and DistBelief use parameter servers to train large-scale deep neural networks. All of these frameworks use a static set of workers and servers, focusing on how to improve the performance of ML algorithms for a given static configuration.

There has been a few proposals to find a good configuration to use based on a cost model. The performance model for Adam (Yan et al., 2015) captures the training time of DNNs by taking advantage of the algorithm characteristics of DNNs. The work uses the model to provision optimal resources for DNN training statically. SystemML (Ghoting et al., 2011; Huang et al., 2015) compiles a declarative ML program into a control program and MapReduce jobs. The hybrid runtime provides an optimizer that can tune memory configurations of the control program and MapReduce jobs by dynamically recompiling the ML program. TUPAQ (Sparks et al., 2015) tackles the problem of model search for Spark MLlib: finding a supervised learning model that provides good prediction accuracy by selecting an ML algorithm and its hyper-parameters. TUPAQ also has an estimator that decides the number of nodes to use statically based on a model of cluster job execution time. Unlike prior work, Dolphin is the first system that performs *runtime* optimization of the Parameter Server ML framework. Dolphin can reconfigure the system transparently and efficiently by introducing a new distributed key-value memory store tailored to processing data.

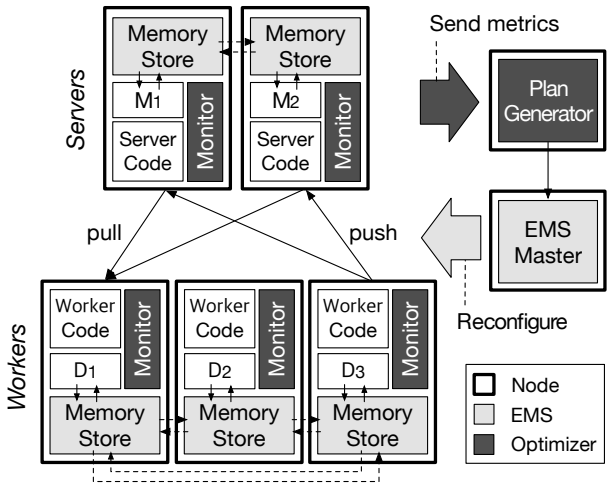

Figure 1. The Dolphin Framework that supports runtime configuration optimization. Dolphin adds Optimizer and Elastic Memory Store to the static parameter server ML system architecture.

3. The Dolphin Framework

We propose Dolphin, a framework that supports runtime configuration optimization. It approaches the system optimization problem in two steps. In the first step, it finds the optimal configuration for a running Parameter Server system. In the second step, it applies that configuration to the running system in the most efficient way. To do so, we extend the existing Parameter Server architecture with two new components, Optimizer and Elastic Memory Store, as depicted in Figure 1. Optimizer finds the optimal configuration based on real-time system metrics. Reconfiguration requests from the Optimizer are sent to Elastic Memory Store, which applies them with zero downtime.

Optimizer, as its name suggests, is in charge of obtaining the optimal configuration. Multiple factors such as the used algorithm, the data set, and the system environment can affect the iteration time and convergence per iteration and therefore the optimal configuration. The impacts of such factors cannot all be predicted in advance for finding the optimal configuration. We tackle our optimal configuration problem by reflecting the current status of the running ML system in the solution to the problem. Metrics related to computations and communications occurring at each node are used to find the optimal configuration of the running system.

Elastic Memory Store (EMS) is in charge of applying the optimal configuration to the running system efficiently. A naive method to apply the optimal configuration would be to checkpoint the current state, stop the job, and restart the job, recovering the learned state from the checkpoint. However, the overhead incurred with this method is substantial. Moreover, if the system's optimal configuration changes over time, we need to make frequent configuration modifications. Repeating restarts are inefficient. Thus, we need a method to apply the new configuration without stopping and restarting the system during runtime. EMS provides a mechanism to apply configurations during runtime so that reconfiguration cost is minimized.

3.1. Optimizer

Machine learning is an iterative convergent process, consisting of many iterations that perform similar computation. Therefore the wall clock training time of a machine learning job is determined by the time elapsed for each iteration and the number of iterations required to achieve model convergence. Optimizer models the iteration time as a cost function of several variables, namely the number of workers and servers nodes as well as the data and model distribution. Through cost-based optimization, Optimizer finds the optimal configuration that produces the lowest cost, i.e. the shortest iteration time, using metrics collected during runtime.

We first define a few notations to explain the cost terms. w and s denote the number of workers and the number of servers, respectively. $\mathbf{d}$ refers to the data distribution across workers, while D equals the total size of the entire dataset. Likewise, $\mathbf{m}$ indicates the model distribution across servers and M is size of the whole model.

During an iteration, a worker performs computation on its local partition of training data with a model given from the servers. The computation results are sent back to the servers via push requests, and the local model is updated with pull requests. Thus, the iteration time consists of the time spent on local computation, defined as computation cost, and the communication with servers, defined as communication cost.

Computation cost. For computation, each worker i does a pass on its local training data, d_i, per iteration. Using the notion $C^i_{w.proc}$ to indicate worker i's unit computation cost, the computation cost of worker i can be written as

$$C^i_{comp}(d_i) = C^i_{w.proc} d_i \tag{1}$$

As the given training data partition increases, the computation cost of worker i becomes greater as well. Note that $C^i_{w.proc}$, although not known yet, is a constant and is not a value Optimizer adjusts.

It is well known that slow workers act as computation bottlenecks in distributed computing (Cipar et al., 2013; Kwon et al., 2012; Yadwadkar et al., 2014). We set the total computation cost C_{comp} to be the largest computation cost of w workers.

$$\begin{aligned} C_{comp}(w, \mathbf{d}) &= \max_{i=1,\dots,w} \left[C^i_{comp}(d_i) \right] \\ &= \max_{i=1,\dots,w} \left[C^i_{w.proc} d_i \right] \end{aligned} \tag{2}$$

Communication cost. We model communication cost as the delay incurred on a worker's iteration when communicating with the server. The communication cost consists of pure network cost, C_{net}, and the processing cost for push and pull requests in servers, $C_{latency}$.

Similar to computation cost, each server j is presumed to have a unique unit processing cost, $C^j_{s.proc}$, and is considered to process partial models of size m_j. Then, the number of requests that are sent to a server per iteration is proportional to both m_j and w. A server that takes a large portion of partial models receives many push and pull requests from workers, which explains the m_j term, while placing many workers also increases the number of requests because requests issued from workers are oblivious of other workers, which leads to the w term.

$$C^j_{latency}(w, m_j) = C^j_{s.proc} m_j w \tag{3}$$

Since the pure network cost C_{net} is not dependent of any server's computation performance, it is simply set as a constant. As a result, the total communication cost can be estimated as

$$\begin{aligned} C_{comm}(w, s, \mathbf{m}) &= C_{net} + \max_{j=1,\dots,s} \left[C^j_{latency}(w, m_j) \right] \\ &= C_{net} + \max_{j=1,\dots,s} \left[C^j_{s.proc} m_j w \right] \end{aligned} \tag{4}$$

The goal of Optimizer is to find values for the variables w, s, $\mathbf{d}$, and $\mathbf{m}$ that minimize the total cost.

$$\begin{aligned} &\text{Find } w^*, s^*, \mathbf{d}^*, \mathbf{m}^* \\ &= \operatorname*{argmin}_{w,s,\mathbf{d},\mathbf{m}} \left[C_{comp}(w, \mathbf{d}) + C_{comm}(w, s, \mathbf{m}) \right] \\ &= \operatorname*{argmin}_{w,s,\mathbf{d},\mathbf{m}} \Big[\max_i \left[C^i_{w.proc} d_i \right] + C_{net} \\ &\qquad + \max_j \left[C^j_{s.proc} m_j w \right] \Big] \end{aligned} \tag{5}$$

Dolphin runs monitors to collect metrics to compute the above cost terms. Monitors are run as a part of each server and worker node in order to reflect the status of all servers and workers in the machine learning computations. For each iteration, monitors gather metrics to compute optimal configurations. Such metrics include iteration time and other time durations contributing to finding the optimal configuration. Once the metrics are collected by monitors on each node, they must be sent to the plan generator for optimal configuration calculation.

The plan generator receives metrics from monitors according to the scheduling policy we choose to implement in the system. As soon as the metrics are received, the plan generator applies the collected metrics to the optimization problem formula to calculate the optimal configuration. After the optimal configuration calculation, the plan generator determines whether a reconfiguration is required or not. A reconfiguration would be required if the current system configuration differs from what has been calculated. The plan generator then maps the change in configuration to calls to the interfaces provided by Elastic Memory Store (EMS), generating an optimization plan, and executes the plan by triggering the EMS reconfiguration options. Since plan generation itself involves computations, we keep the plan generator in a separate node from worker and server nodes to minimize any unfavorable overhead slowing down the running ML algorithm.

3.2. Elastic Memory Store

Optimizer focuses on finding optimal configurations for ongoing machine learning jobs. However, the found configurations would be of no use if they cannot be executed and applied in time without halting the running job. Elastic Memory Store (EMS) provides an efficient way to change the system's configuration.

EMS is a distributed in-memory store abstraction that allows efficient reconfiguration during runtime. Clients of EMS can store values to and retrieve values from EMS, similar to what can be done in distributed key-value stores. When a reconfiguration is requested, EMS can add or remove resources by interacting with underlying resource managers (e.g., YARN and Mesos), and migrate data between the key-value stores on nodes without adversely affecting runtime performance.

EMS clients access and update each data item per key with simple and standard operations like `put` and `get`. EMS also provides interfaces which Optimizer can call to reconfigure the system dynamically during runtime: `add`, `remove`, and `move`.

EMS consists of the EMS master and a set of MemoryStores. The EMS master is the endpoint where Optimizer submits its generated optimization plan. It manages MemoryStores where algorithms can put/get data to/from by keeping track of data locations. MemoryStores reside in each worker and server nodes, serving as a distributed key-value store with key-value mappings on different kinds of data depending on which node it resides. On the worker side, training data is distributed among MemoryStores. On the server side, model parameters are distributed among MemoryStores. The master keeps a routing table of MemoryStores distributed among worker and server nodes to know on which nodes the reconfiguration should take place.

A *MemoryStore* is a local key-value store. *Key* is the unit of data access in MemoryStore, which is mapped to a single and mutable value. *Client*s, workers and servers of ML jobs in our context, can access data by specifying keys. *Block*, on the other hand, is the unit of management. For example, data migration is performed by blocks, not by a single key-value tuple. A block consists of the data whose key is within a range that divides the entire key-space. When data is put in a MemoryStore, the data is stored in the corresponding block.

EMS provides two types of reconfigurations. EMS ensures that all operation to the existing data in MemoryStores by running ML algorithms is unaffected - i.e., the migration is transparent - for both reconfiguration types. First, the EMS master can be asked to add more resources to the system. In this case, it requests for the resources to a lower level system (e.g., the cluster resource manager) and registers them as a part of the EMS. At the same level, the EMS master can be asked to release its resources, applying the necessary modifications to the EMS.

- `add (Integer numNodes)`: Adds `numNodes` new nodes. Other tasks such as application code transfer on the new node can take place.
- `delete (List<ID> ids)`: Deletes existing nodes whose IDs are in `ids`.

Second, the EMS master can be asked to redistribute data in MemoryStores. The EMS master uses the routing table to locate the source and the destination of data being migrated and then orders MemoryStores to transfer the data. This operation consists of (1) transferring the data blocks to the target MemoryStore, and (2) switching the ownership of the moved blocks for maintaining a synchronized view of block ownership across the entire system. To conduct such operation, EMS provides the interface below:

- `move (Integer numBlocks, ID src, ID dest)`: Moves the data in blocks from `src` to `dest`. The EMS Master chooses blocks in `src` to move

as many as `numBlocks` according to its ownership table.

This migration is transparent to the clients of EMS. We try to minimize the potential slowdown of client accesses to the moving data and we ensure that block ownership must be switched atomically. We combine on-demand pull for foreground data migration and asynchronous pull for background data migration.

4. Conclusion

Dolphin is an elastic machine learning framework that provides the runtime optimization of framework configurations. Dolphin's Optimizer finds optimal configurations by solving cost-based optimization problems formulated by the metrics collected from running systems. To facilitate reconfiguration, Dolphin manages data (both training data and model parameters) with Elastic Memory Store, a distributed, elastic key-value store. The reconfiguration actions are conducted through Elastic Memory Store API calls. Dolphin is built on Apache REEF (Chun et al., 2013; Weimer et al., 2015), a meta-framework for building distributed data processing applications. Dolphin is in active development. We plan to open source Dolphin in the future.

Acknowledgements

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0126-15-1093, (SW Star Lab) Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. Technical report, Preliminary White Paper, Google Research, 2015.

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

Chun, B.-G., Condie, T., Curino, C., Douglas, C., Matusevych, S., Myers, B., Narayanamurthy, S., Ramakrishnan, R., Rao, S., Rosen, J., Sears, R., and Weimer, M. Reef: Retainable evaluator execution framework. In *International Conference on Very Large Data Bases (VLDB)*, 2013.

Cipar, J., Ho, Q., Kim, J. K., Lee, S., Ganger, G. R., Gibson, G., Keeton, K., and Xing, E. Solving the straggler problem with bounded staleness. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Q. V. Le, M. Z. Mao, Ranzato, M.'A., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., and Vaithyanathan, S. SystemML: Declarative machine learning on mapreduce. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Katz, A. Josephand R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

Huang, B., Boehm, M., Tian, Y., Reinwald, B., Tatikonda, S., and Reiss, F. R. Resource elasticity for large-scale machine learning. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skewtune: mitigating skew in mapreduce applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB Endowment*, volume 5, pp. 716–727, 2012.

Smola, A. and Narayanamurthy, S. An architecture for parallel topic models. In *Proceedings of the VLDB Endowment*, volume 3, pp. 703–710, 2010.

Sparks, E. R., Talwalkar, A., Haas, D., Franklin, M. J., Jordan, M. I., and Kraska, T. Automating model search for large scale machine learning. In *ACM Symposium on Cloud Computing (SoCC)*, 2015.

The Apache Software Foundation. Apache Mahout, 2015. http://mahout.apache.org.

The Apache Software Foundation. Apache Spark MLlib, 2016. http://spark.apache.org/mllib.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

Weimer, M., Chen, Y., Chun, B.-G., Condie, T., Curino, C., Douglas, C., Lee, Y., Majestro, T., Malkhi, D., Matusevych, S., Myers, B., Narayanamurthy, S., Ramakrishnan, R., Rao, S., Sears, R., Sezgin, B., and Wang, J. Reef: Retainable evaluator execution framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. Petuum: A new platform for distributed machine learning on big data. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

Yadwadkar, N. J., Ananthanarayanan, G., and Katz, R. Wrangler: Predictable and faster jobs using fewer resources. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

Yan, F., Ruwase, O., He, Y., and Chilimbi, T. Performance modeling and scalability optimization of distributed deep learning systems. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.