Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Stochastic Gradient Descent Based Support Vector Machines Training Optimization on Big Data and HPC Frameworks

Vibhatha Abeykoon*[1] | Geoffrey Fox[1] | Minje Kim[1] | Saliya Ekanayake[2] | Supun Kamburugamuve[1] | Kannan Govindarajan[1] | Pulasthi Wickramasinghe[1] | Niranda Perera[1] | Chathura Widanage[1] | Ahmet Uyar[1] | Gurhan Gunduz[1] | Selahatin Akkas[1]

[1]Intelligent Systems Engineering, Indiana University Bloomington, Indiana, United States

[2]Performance and Algorithm Research, Lawrence Berkeley National Laboratory, California, United States

Correspondence
*Vibhatha Abeykoon. School of Informatics, Computing and Engineering, Indiana University Bloomington, Email: vlabeyko@iu.edu

Summary

Support Vector Machines is one of the widely used lightweight machine learning algorithm which can do efficient training on smaller data sets. In this research, we focused on highly scaleable gradient descent based approach. In providing a scalable solution, we propose to use high-performance computing model and big data computing model (dataflow). Designing this algorithm with MPI like programming model has been widely used. In this paper, our, objective is to enhance a training model designed by us with math kernels and analyze how C++ and Java programming languages can be used to design optimized algorithms. We also discuss the overheads in the applications and optimization techniques used to improve the performance. In this research, our objective is to build this algorithm with the support of multiple dataflow design mechanisms involving iterative and ensemble training models. For this purpose, we use Twister2, a big data tool kit which provides the basic infrastructure to address this kind of problems. And also we compare the performance of Twister2-APIs with Spark RDD and MPI. In our research, we showcase how the high-performance computing stack and big data programming stack can be used to optimize the training of SGD-based SVM algorithm in distributed environments.

KEYWORDS:
Svm, Dataflow, High-Performance Computing, Machine Learning

1 | INTRODUCTION

Support vector machines are one of the most used classification algorithms in the machine learning domain. In designing a highly scale-able algorithm, the most important thing is to identify the bottlenecks in the existing implementations. The batch size is a very sensitive number which affects the accuracy and the execution time of the algorithm. For various use cases, there is a requirement of designing appropriate programming models to support these requirements. In an early research [1], we thoroughly analyzed how the batch size can affect the performance and accuracy of the application. In this research, our main purpose is to optimize our existing model and scale the training process on a much higher scale in a cluster. In order to achieve higher performance at the process level, we also consider the usage of math kernels especially considering BLAS level optimization. In application development for machine learning, another important concept is to consider the nature of application development

based on programming languages. In this research, we analyze how C++ and Java programming languages can be used to design optimized applications. Performance improvement under optimized library usage and default compiler optimization are discussed in this paper. In terms of data, there can be streaming data and already existing batch data. In terms of programming models, high-performance programming model and dataflow programming models are used by most of the researchers.

For batch data, high-performance computing model has been used over a couple of decades with great success. But the application development overhead involved with big data processing and data pipeline design needs to be handled with effective solutions. In order to do effective computations each of these data types, dataflow methods have to be used. In the high-performance computing world, MPI[2] programming model is the most prominent methods adopted in solving computationally intensive problems on large datasets. In Dataflow community, Hadoop[3],[4], Apache Spark[5], Apache Flink[6], Apache Storm[7], Twitter Heron[8] and Google Dataflow[9] are prominent tools. The dataflow programming models also have different types of abstractions. The low-level dataflow programming model involves designing a task graph and designing the full dataflow model using basic building blocks. This model is very flexible and easy to add optimization. But the programming model abstraction is important in designing better design patters for the ease in designing complex models. Providing all these programming abstractions along with high-performance model compatibility is vital in designing scientific applications. Twister2[10] is such a big data tool kit which allows designing applications supporting both HPC and dataflow model applications. Twister2 implicitly supports MPI applications to run within its programming model. Aforementioned communication level APIs, task-level APIs and higher-level data abstraction APIs are available in Twister2. In this paper, first, we discuss how the SGD-based SVM algorithm can be scaled in a single node and multiple nodes with basic MPI programming models. We design these models on Java and C++ programming languages. On top of these models, we try to optimize the process level performance by using BLAS level operations on vector vs vector dot products and vector vs scalar products. In parallel to this, we also observe how each language is behaving with this optimization to provide optimum performance. In addition to this, we also analyze how ensemble model SGD-based SVM can be implemented in Twister2, Spark and MPI and compare how each framework provides optimization.

2 | RELATED WORK

In the machine learning domain, Support Vector Machines (SVM) by Cortes and Vapnik[11] can be considered as the groundwork done on developing this classification algorithm. SVM is one of the lightweight algorithms in machine learning domain for supervised learning-based classification problems. Inspired by this work, Libsvm[12] one of the initial work done on developing a complete library on providing a wide range of programming tools to do SVM based classifications for multi-class classifications. It also supports multiple kernels and multiple optimization algorithms. One of the most important works done on optimizing the sequential SVM algorithm is the Sequential Minimal Optimization-based SVM by Platt[13]. This is one of the prominent sequential optimization done. Followed by this work, simplified versions of SMO[14] has also been widely used to develop a lightweight version of this algorithm. But the main issue with the simplified model is the lesser accuracy. In improving performance by means of sequential optimization, DC-SVM[15] a divide and conquer based sequential model was developed with K-Means clustering involved. Sequential level optimization can provide performance improvement to a certain level. When the data size is increasing, a distributed version of this algorithm is vital to provide the required performance. PSVM[16],[17] is one of the prominent work done on a parallel version of SVM algorithm. But in this approach, the traditional Lagrangian multiplier based optimization is not used, but a matrix-based decomposition method is used to do factorization to find the solution to the optimization problem. SMO based parallel applications have also been developed by Keerthi et al[18]. In the progression of SVM optimization, stochastic gradient descent based approach is heavily discussed in Pegasos[19]. Using an adaptive learning rate to provide an efficient training model is discussed by solving the optimization problem using stochastic gradient descent (SGD) based approach. For distributed SVM with SGD-based approach, P-PackSVM[20] and parallel stochastic gradient descent[21] can be recognized as prominent research work done to influence optimized distributed models. In addition to that fast feature extraction based SVM models have also been developed to provide efficient training to SVM algorithm[22]. In a distributed application, the main goal is to make sure the communication overhead caused by model synchronization is lesser compared to the performance gain compared by the division of computation load. Distributed application development to solve this problem can be done in multiple ways. MPI[2] model is the prominent solution when the problem needs to be solved by means of adopting high-performance computing. In application development with MPI, collective communications like reduce, allreduce, gather, allgather, broadcast and scatter can be used to synchronize models in a distributed environment. MPI programming model supports distributed data and does a process level performance improvement. In addition to this, it is vital to improving performance within a process. In order to

obtain performance boost within a process, BLAS[23],[24],[25],[26],[27] level operators are vital to do vector-based calculations in an efficient manner. In improving SMO-based SVM, BLAS operations have been used in previous research as well as[28]. On the other hand, it is very important to see how compiler level optimization involves providing performance improvement.

Java-related JIT(Just-In-Time)[29] is a runtime performance improving compiler. In C++ similarly, compile-time optimization is done using O3, O2 and OFast level optimization[30]. In big data related frameworks like Apache Spark[5], Apache Flink[6], Google Dataflow[9], Apache Storm[7],[31], Heron[8] dataflow based solutions has been provided to solve big data-related problems in both streaming and batch mode datasets. In each of these frameworks, a well-defined data pipeline is provided for the application users to design big data applications on distributed environments. All these frameworks provide big data-oriented solutions. Twister2[32],[10] a big data tool kit designed to provide a variety of functionality to both HPC and Big Data world, provides a set of programming abstractions to design distributed applications. Twister2: Net[33] an optimized communication library contains an MPI-like communication style with MPI-like communication and TCP-based communication. Application development abstractions are important to design applications with efficiency. TSet[34] API in Twister2 is another big data programming abstraction to develop optimized applications similar to Spark RDD format. SGD-based SVM is a very flexible model that can be developed in both the HPC model and dataflow model with less development complexity.

3 | METHODOLOGY

In this research, we develop our scope of analysis in the following way. The methodology adopted in this research is divided into two main components. The first one is the HPC model-based performance analysis with BLAS routines and without BLAS routines. The other model is to conduct experiments on distributed dataflow model on Big Data stack. In the HPC model, we discuss how MPI based parallel processing improves performance. Then we observe how within process performance can be improved with BLAS operations. For the distributed applications, we consider an ensemble model of the distributed algorithm. In this, we use the same core algorithm but we only focus on model design and scale-up. For this, we use Spark-RDD, Twister2-Task and Twister2-TSet frameworks to develop the application. On top of this, we also design the same ensemble model with MPI designed with OpenMPI 3.0.0 and compare HPC vs Dataflow model performance on a distributed scale.

3.1 | Anatomy of the Algorithm

The core optimizer iterates through the data points and does the optimization to calculate the weights. The distributed algorithm shows how the iterative training is done for a considered amount of iterations (or iterations until convergence).

$$S = \{x_i, y_i\}$$
$$\text{where } i = [1, 2, 3, ..., n],\ x_i \in R^d,\ y_i \in [+1, -1] \tag{1}$$

$$\alpha \in (0, 1) \tag{2}$$
$$g(w; (x, y)) = \max(0, 1 - y\langle w|x\rangle) \tag{3}$$

$$J^t = \min_{w \in R^d} \frac{1}{2} w^2 + C \sum_{x,y \in S} g(w; (x, y)) \tag{4}$$

The sample space of the data distribution is defined in 1. Learning rate α range is defined in 2. In the SGD approach, minimize the objective function in (4) with the constraint on the optimization defined in (3).

3.2 | HPC Model Based Performance Analysis

In designing an HPC model, the tool we selected is MPI with OpenMPI 3.0.0 standard. In the first step, we design a methodology to test the performance of Java and C++ applications developed to solve the SVM optimization problem. The compiler level optimization-based performance tuning is the first aspect that is evaluated on MPI based C++ and Java applications. Along with this we also research how each language is sensitive to providing performance boost with BLAS on variable data sizes and

Algorithm 1 SGD SVM

1: $INPUT : [x, y] \in S, w \in R^d, t \in R^+$
2: $OUTPUT : w \in R^d$
3: **procedure** SGDSVM(S, w, t)
4: **for** $i = 0$ to n **do**
5: **if** $(g(w; (x, y)) == 0)$ **then**
6: $\nabla J^t = w$
7: **else**
8: $\nabla J^t = w - Cx_i y_i$
9: **end if**
10: $w = w - \alpha \nabla J^t$
11: **end for**
return w
12: **end procedure**

FIGURE 1 SGD SVM Algorithm

Algorithm 2 PSGD SVM

1: $INPUT : X, Y, w$
2: $OUTPUT : w \in R^d$
3: **procedure** PSGDSVM(S, w)
4: **In Parallel in K Machines** $[S_1, ..., S_b] \subset S$
5: $w_{local} = w$
6: **for** $i = t$ to T **do**
7: **procedure** SGDSVM(S_m, w_{local}, t)
8: **end procedure**
9: $w_{global} =$ `MPI_AllReduce`(w_{local})
10: $w = w_{global}/K$
11: **end for**
12: **end procedure**
return w

FIGURE 2 PSGD SVM Algorithm

feature sizes, especially with Java. From the conclusions obtained from the work done in the parallel stochastic gradient descent with model synchronization research[1], algorithm in figure 2 was designed by calling the core optimizer algorithm in figure 1.

3.2.1 | BLAS Optimization

In considering the core algorithm 1, to optimize the dot products and vector scalar multiplications, BLASlevel operations can be applied. For this research, we selected the OpenBlas standard of Blas implementations on Red Hat Enterprise Linux Server 7.6 (Maipo) operating system. In order to implement Blas level operations, the equation and BLAS operation mappings are in 5, 6, 7 and 8.

$$g(w; (x, y)) \Rightarrow max(0, 1 - y\langle w|x\rangle) \Rightarrow max(0, 1 - ddot(d, x, inc_x, w, inc_y)); \tag{5}$$

$$\langle X_j, y_i\rangle \Rightarrow daxpy(d, y_i, X_j, inc_x, xiyi, inc_y); \tag{6}$$

$$w = w - \alpha C X_i y_i \Rightarrow daxpy(d, \alpha, xiyi, inc_x, w, inc_y) \tag{7}$$

$$w = w - \alpha w \Rightarrow daxpy(d, \alpha, w, inc_x, w, inc_y); \tag{8}$$

3.3 | Big Data Frameworks for SVM

In previous sections, we described how HPC based solution can work on different application development platforms. In considering big data frameworks, developed to create well-defined data pipelines and computation channels, it is important to identify how big data stack can fit in solving similar problems that can be solved using HPC based solutions. In referring to the big data stack, Spark can be identified as one of the most prominent tools used by many data scientists and big data application developers. In Big data application stack, the problem we are trying to optimize comes under the iterative batch applications. To do iterative computations Spark provides a data level abstraction called RDD (resilient distributed data). To support big data stack Digital Science Center in Indiana University Bloomington has produced a framework, Twister2 which supports both HPC and Big data stack application development on a task level API and a TSet level API. The task-level API in Twister2 refers to a higher-level abstraction on top of communication level API and TSet is a data level abstraction on top of task API which is similar to the RDD API in Spark. We developed the SVM algorithm in an ensemble way in MPI using Java, Spark RDD, Twister2 Task API and Twister2 TSet API.

TABLE 1 Datasets

DataSet	Training Data (80%)	Testing Data (80%)	Sparsity	Features
Ijcnn1	39992	9998	40.91	22
Webspam	280000	70000	99.9	254
Epsilon	320000	80000	44.9	2000

4 | EXPERIMENTS

The experiments in this research were conducted in the Victor and Juliet cluster group in the Future Systems cluster at Indiana University Bloomington. For the single node experiments, we used a maximum of 32 slots in a node and for distributed mode experiments, we used an equal number of processes per machine over a group of 16 nodes. Victor cluster nodes comprises of Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz configuration. Juliet cluster nodes comprises of Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz configuration. For the single node experiments, we considered a range of parallelisms from 2 to 32 with powers of 2. In distributed experiments, parallelism from 2 to 256 was used among 16 nodes such that for each parallelism each machine gets an equal number of processes. From parallelism 2 to 8, a single process per machine among 2 to 8 machines. And for parallelism 16 to 256, each machine gets an equal number of processes among all 16 machines. For the big data stack based applications we considered the processor affinity and configured Twister2 and Spark such a way that a single process is run per core in the distributed mode experiments. And the MPI Java application also consumed a single process per core.

4.1 | Dataset Configuration

For the experiments, we used 3 data sets considering the features per data point, sparsity and data size as shown in table 1.

4.2 | Java and C++ Performance Benchmark

For both Java and C++ based MPI applications. Both applications use the compiler level optimization done by each compiler. C++ one uses the 03 level optimization while Java uses compiler level optimization along with JIT (Just in Time) compiler optimization at runtime. The purpose of this experiment is to see how the same algorithm developed in each language performs to the scaling done within a node. Here we carried out two sets of experiments. The first one just uses compiler optimization from each language. In the other setting, each language uses the Blas level optimization for vector vs vector and vector vs scalar multiplications. In these settings, we analyzed how each implementation from each language behaves for three data sets with various sparsity, features and size.

4.3 | Blas Configurations

For Blas optimizations, we used OpenBlas as the Blas standard for our experiments. C++ applications have complied with the support of this version directly. To provide support to Java applications, we use netlib-java library which supports Blas level operations on a Blas installed system.

4.4 | Big Data Benchmark Configurations

For the experiments in Java, Java(TM) SE Runtime Environment (build 1.8.0_101-b13) was used and for C++ OpenMPI 3.0.0 was used for Java and C++ comparisons. In big data stack related benchmarks, for Twister2 experiments, OpenMPI 3.1.2 was used as it is a required dependency for Twister2. For Spark, 2.4.0 version was used while MPI 3.1.2 was used for Java-based MPI applications.

5 | RESULTS

5.1 | Java and C++ based MPI Application

In single-node experiments, the parallelisms 2 to 32 was used in the same machine in the cluster for both Java and C++ experiments. In considering the performance concerning the compiler level optimization in both Java and C++ we can see a clear pattern concerning each dataset. In figure 6, the results show that compiler level optimization is done by Java and Just In Time compiler optimization in runtime provides better performance over C++ compiler level performance with 03 level optimization. Java performance better over C++ language level optimization in single node experiments. In referring to the results from Blas based optimization in figure 5, in C++ for all three datasets Blas based optimization always do better for all three datasets in both distributed and single node experiments in figure 3. This behaviour is expected as Blas level operation provides a boost for vector vs vector and vector vs scalar calculations involved with the SVM algorithm. In Java, the behaviour is entirely different concerning the figure 4. For Ijcnn1 and Webspam datasets the Blas optimization is always slower than the just compiler level optimized version. But with the epsilon dataset, we can observe that the Blas level optimization has provided a performance boost. The reason for this behaviour is a part of the process Java using to get this performance boost. To obtain the Blas level optimization, Java cannot directly obtain it from the Blas level operators. The reason is these optimization libraries are written in C++/C by accessing native operating system level functionality. Java cannot obtain the native operating system level functionality as Java is running on a virtual machine called JVM (Java Virtual Machine). To obtain the library functionality written in C++/C, Java needs to use an interface called JNI (Java Native Interface) which provides the ability to call a native function or library or to be called by such one. To this to happen, how JNI works have to be understood. JNI native functions are implemented in C++/C medium. When JVM invokes a native function, a JNIEnv pointer is being passed along with a jobject pointer and any Java arguments declared by the corresponding Java method. The env pointer holds interface to JVM. In these function calls going from either side, JNI functions are converting native arrays to and from Java arrays when the vector vs vector or scalar vs vector computations are called in Blas level operations. To gain a performance improvement, the data conversion time and the computation time all together must be lesser than the just compiler optimized code in Java. For smaller data sets the conversion time with computation time is lesser as the array sizes involved in dataset Ijcnn1 is 22 and in the dataset, Webspam is 254. But in dataset epsilon, the array size is 2000 and the computation is 100 to 10 times higher than earlier scenarios. So the computation advantage obtained with Blas level operations provides better performance. From this observation, we decided to carry out the distributed experiments with Blas support in both Java and C++ applications. But the main observation is to keep track of the performance obtained per Epsilon dataset related experiments. Because it is the common dataset which provides a better performance improvement over Blas optimization. Referring to figure 5, it is clear that Epsilon dataset performs better with C++ language than Java. The reason for this issue is the Blas level optimization improvement is diluted by the data conversion to either side from native to Java and Java to native due to the iterative nature of the application over a large dataset. In analyzing this fact further, we conducted experiments on the sequential version of the algorithm by considering a variable feature size and constant data size by using random data generated using a Gaussian distribution. The results in figure 9, shows that the Blas level performance is becoming better than compiler level optimized code when the array size grows up. The data conversion overhead makes C++ a better platform to develop scalable solutions on larger datasets with higher dimensions. To improve the performance of Java-based developed applications the JNI function calls have to be minimized meaning the iterative algorithm can be run in C++ end and do the data conversion just once before algorithm starts and the algorithm ends.

In evaluating the performance boost obtain from Blas optimizations, we observed the ratio of training time with BLAS level operations and without BLAS level operations on dataset epsilon. These experiments were carried out for both single-node and distributed domain experiments. The performance improvement for Java was not as significant as that of C++. Figures 7 and 8 depicts the aforementioned fact.

5.2 | Big Data Ensemble Model for SVM Training

For the Big data domain and HPC domain performance comparison for distributed SGD-based SVM algorithm, we used parallelism 16 to 256 among 16 nodes in Juliet cluster such that each machine gets an equal number of processes. For these experiments, we used Epsilon dataset. Figure 10 refers to the performance comparison for MPI-Java, Spark-RDD, Twister2-Task and Twister2-TSet. It is clear from this experiment that Twister2 APIs provide similar performance concerning MPI implementation while being faster than the Spark RDD based application. In considering the anatomy of MPI-Java application, it uses MPI Allreduce communication to do the model synchronization across the processes. Twister2 also supports the allreduce based

FIGURE 3 C++ Single Node Experiments

FIGURE 4 Java Single Node Experiments

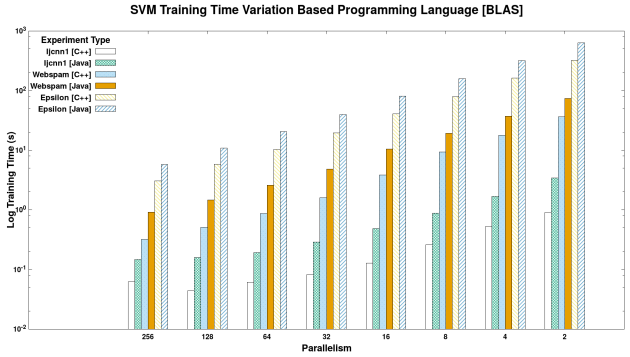

FIGURE 5 Java and C++ Distributed Node Experiments with Blas

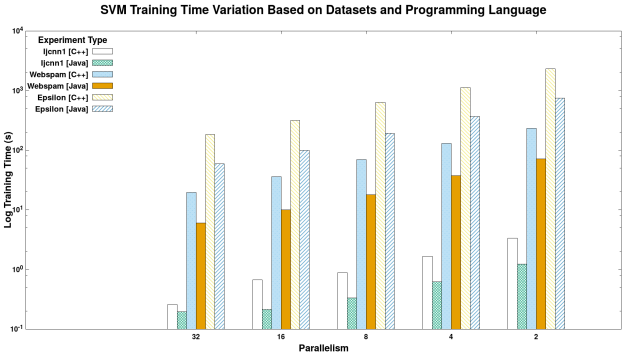

FIGURE 6 Java vs C++ Compiler Level Optimization Based Performance

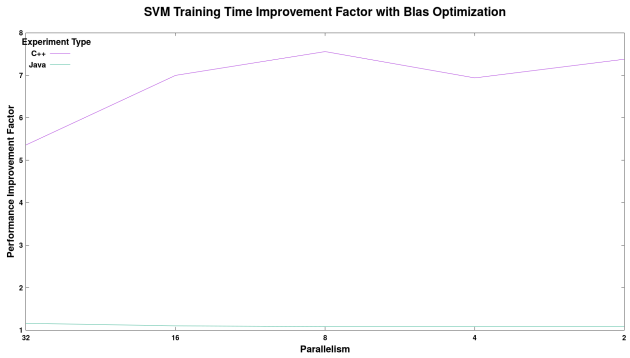

FIGURE 7 Performance Improvement on Single Node Experiments with Epsilon Dataset

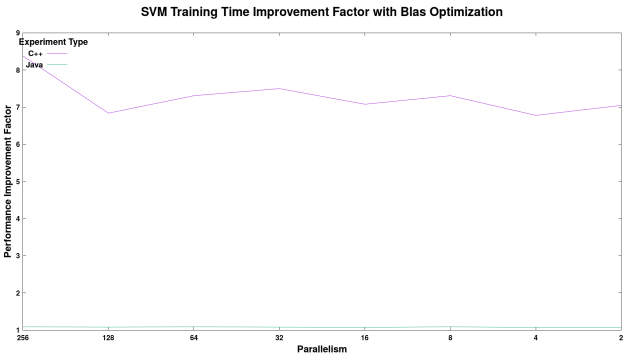

FIGURE 8 Performance Improvement on Distributed Node Experiments with Epsilon Dataset

model synchronization with a binary tree based optimized communication. Spark-based application is based on the worker to driver and driver to worker based communication model which is another way of doing the model synchronization in iterative batch applications. The main difference between MPI, Twister2 implementations vs Spark implementation is the way the models are synchronized. When MPI and Twister2 do a tree-based reduce, Spark provides a reduce by calling back the models from each task back to the driver programme to do the model synchronization. For an iterative application, this model is costly. That is the main reason for the performance boost obtained by Twister2 and MPI based implementations over Spark.

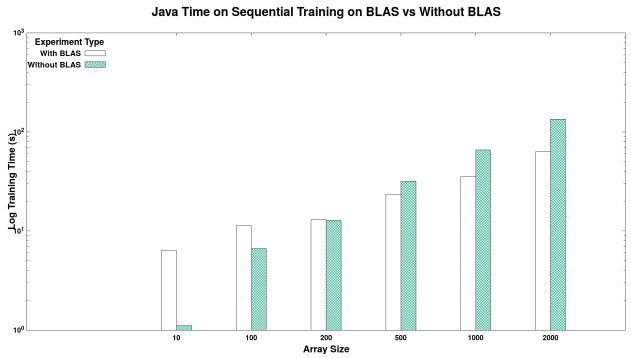

FIGURE 9 Java Blas Performance Against Vector Size Variation

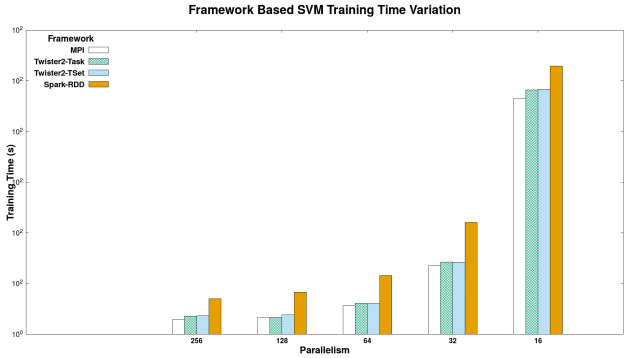

FIGURE 10 Big Data Stack vs HPC Benchmark on Distributed Ensemble SVM

6 | CONCLUSION

In obtaining better performance for distributed SGD-based SVM applications, there are multiple tools which can provide a wide range of benefits over multiple aspects. The performance gain is always a trade-off that has to be dealt with the level of programming abstractions we trying to adapt to the development model. To gain, the native performance it was evident that using Blas level operations on top of C++ based solution can provide much better performance with higher scalability. In comparing this with the same programming abstraction involved with Java, the barrier behind data transformation from the native environment to JVM and vice versa causes a performance drop down when it is compared to C++ solution. But it was evident that for smaller vectors the C++ or Java solution can be used to get close performance by using Blas with C++ and just use Java with compiler level optimization. This fact is not true when it comes to larger vector size and larger data size. In the development life cycle, managing a Java application is much easier than a C++ application and this trade-off has to be handled with good care when developing applications. But it was evident from the experiments that, the development abstraction in Java can be leveraged to design the data pipeline but using C++ based application to run the iterative algorithm and sending the final results back to Java-based data pipeline can gain performance and a better application development abstraction. In referring to the big data stack-based application benchmark, by using a similar optimization done in MPI for model synchronization, big data stack can also leverage a performance boost.

References

1. Abeykoon VL, Fox GC, Kim M. Performance Optimization on Model Synchronization in Parallel Stochastic Gradient Descent Based SVM. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). ; 2019: 508-517

2. Gabriel E, Fagg GE, Bosilca G, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Springer. ; 2004: 97–104.

3. Shvachko K, Kuang H, Radia S, Chansler R, others . The hadoop distributed file system.. In: . 10. ; 2010: 1–10.

4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; 51(1): 107–113.

5. Zaharia M, Xin RS, Wendell P, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM* 2016; 59(11): 56–65.

6. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 2015; 36(4).

7. Iqbal MH, Soomro TR. Big data analysis: Apache storm perspective. *International journal of computer trends and technology* 2015; 19(1): 9–14.

8. Kulkarni S, Bhagat N, Fu M, et al. Twitter heron: Stream processing at scale. In: ACM. ; 2015: 239–250.

9. Akidau T, Bradshaw R, Chambers C, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 2015; 8(12): 1792–1803.

10. Kamburugamuve S, Govindarajan K, Wickramasinghe P, Abeykoon V, Fox G. Twister2: Design of a big data toolkit. *Concurrency and Computation: Practice and Experience* 2017: e5189.

11. Cortes C, Vapnik V. Support-vector networks. *Machine learning* 1995; 20(3): 273–297.

12. Chang CC, Lin CJ. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2011; 2(3): 27.

13. Platt J. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

14. Yang J, YE Cz, Quan Y, CHEN Ny. Simplified SMO algorithm for support vector regression. *Infrared and Laser Engineering* 2004; 5.

15. Hsieh CJ, Si S, Dhillon I. A divide-and-conquer solver for kernel support vector machines. In: ; 2014: 566–574.

16. Chang EY. Psvm: Parallelizing support vector machines on distributed computers. In: Springer. 2011 (pp. 213–230).

17. Chang EY. Psvm: Parallelizing support vector machines on distributed computers. In: Springer. 2011 (pp. 213–230).

18. Cao LJ, Keerthi SS, Ong CJ, et al. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Trans. Neural Networks* 2006; 17(4): 1039–1049.

19. Shalev-Shwartz S, Singer Y, Srebro N, Cotter A. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming* 2011; 127(1): 3–30.

20. Zeyuan AZ, Weizhu C, Gang W, Chenguang Z, Zheng C. P-packSVM: Parallel primal gradient descent kernel SVM. In: IEEE. ; 2009: 677–686.

21. Zinkevich M, Weimer M, Li L, Smola AJ. Parallelized stochastic gradient descent. In: ; 2010: 2595–2603.

22. Lin Y, Lv F, Zhu S, et al. Large-scale image classification: fast feature extraction and svm training. In: IEEE. ; 2011: 1689–1696.

23. Kestur S, Davis JD, Williams O. Blas comparison on fpga, cpu and gpu. In: IEEE. ; 2010: 288–293.

24. Sam Halilday YYEA. Netlib-Java.

25. Open Blas.

26. Wang Q, Zhang X, Zhang Y, Yi Q. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In: IEEE. ; 2013: 1–12.

27. Xianyi Z, Qian W, Yunquan Z. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In: IEEE. ; 2012: 684–691.

28. Dong Jx, Krzyżak A, Suen C. A fast parallel optimization for training support vector machine. In: Springer. ; 2003: 96–105.

29. Cramer T, Friedman R, Miller T, Seberger D, Wilson R, Wolczko M. Compiling Java just in time. *IEEE Micro* 1997; 17(3): 36–43.

30. Kuhn BM, Binkley DW. An enabling optimization for C++ virtual functions. In: . 17. ; 1996: 420–428.

31. Toshniwal A, Taneja S, Shukla A, et al. Storm@ twitter. In: ACM. ; 2014: 147–156.

32. Kamburugamuve S, Govindarajan K, Wickramasinghe P, Abeykoon V, Fox G. Twister2: Design of a big data toolkit. *Concurrency and Computation: Practice and Experience* 2017: e5189.

33. Kamburugamuve S, Wickramasinghe P, Govindarajan K, et al. Twister: Net-communication library for big data processing in hpc and cloud environments. In: IEEE. ; 2018: 383–391.

34. Wickramasinghe P, Kamburugamuve S, Govindarajan K, et al. Twister2: TSet High-Performance Iterative Dataflow. In: IEEE. ; 2019: 55–60.