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 (SVM) is a widely used machine learning algorithm. With the increasing amount of research data nowadays, understanding how to do efficient training is more important than ever. This paper discusses the performance optimizations and benchmarks related to providing high-performance support for SVM training. In this research, we have focused on a highly scalable gradient descent-based approach to implementing the core SVM algorithm. In providing a scalable solution, we have designed optimized high-performance computing and dataflow-oriented SVM models. A high-performance computing approach means the algorithm is implemented with the bulk synchronous parallel (BSP) model. In addition, we analysed the language level optimizations and math kernel optimizations on a prominent HPC modelling programming language (C++) and dataflow modelling programming language (Java). In the experiments we compared the performance of classic HPC models, classic dataflow models, and hybrid models designed on classic HPC and dataflow programming models. Our research illustrates a scientific approach in designing the SVM algorithm at scale in classic HPC, dataflow and hybrid systems.

KEYWORDS:
SVM, Dataflow, High-Performance Computing, Machine Learning, Hybrid Systems

1 | INTRODUCTION

Support vector machines (SVM) are one of the most popular classification algorithms among the machine learning algorithms. Many applications are depending on SVM in various scientific disciplines. In these applications, a major requirement is to understand what is the best way to implement the SVM algorithm to support various requirements. There are many research work done scaling SVM with algorithm optimizations and distributed strategies. Depending on the application, multiple system design disciplines can be adopted. Depending on the eco-system to which the distributed SVM application is being developed decides the required disciplines. In the high-performance computing (HPC) disciplines, MPI-based software development is very common. For these developments, C++ can be considered as the commonly used programming language. But when it comes to scientific applications developed as a part of an existing user library, the usage of Java programming language can be commonly seen in many use cases. There are many reasons to support many languages. Apart from the HPC community, the big data community also has a different practice in solving similar problems associated with big data.

In the big data frameworks, the most commonly used language is Java. In the Big data world, the most common programming model used is the dataflow model. Here the classic collective communication libraries in HPC are not commonly used. Instead of similar operators are developed within the dataflow model. In the dataflow model, programmability is one of the key requirements. Apache Spark[1] is one of the prominent tools supporting these requirements. But a known fact is that HPC software stack provides much better performance. In bringing better performance to the big data stack, Twister2[2] has merged some of the valuable concepts from HPC programming model into the big data world by using MPI as a backend to provide better performance in the dataflow model. This clearly shows the variety of choices a scientist can take in developing applications. Most of these usages for a high-performance SVM implementation is not well understood. Our research focuses on discussing these in detail with a set of experiments in showcasing how each discipline can be used to implement an efficient distributed SVM algorithm.

In the existing work with high-performance computing and dataflow-based computing, there have been efforts made towards unifying the capabilities of HPC and dataflow systems. Or using high-performance computing concepts to develop better systems to support developing efficient SVM implementation. In high-performance application development, message passing interface (MPI[3]) is a prominent way to write scalable high-performance applications. Many standards in the dataflow model derive from the Apache software foundation. Hadoop[4], an extension to the original work on map-reduce[5] architecture, is one of the earliest examples of developing advanced dataflow systems. Extending from Hadoop are dataflow community-produced examples like Apache Spark[1], Apache Flink[6], Apache Storm[7], Twitter Heron[8] and Google Dataflow[9]. HPC community-produced versions of MPI standards include OpenMPI[10], MPICH[11] and MVAPICH[12]. Twister2:Net[13] introduces a unified communication collective API, along with a unified data analytics platform in Twister2 big data toolkit[2] and Twister2 TSet[14] APIs.

The main objective of our research is understanding and analyzing ways to implement a scalable SVM algorithm in HPC (MPI), dataflow (Spark) and HPC-dataflow hybrid systems (Twister2). With multiple software development approaches available in multi-disciplined research communities, investigating and providing better solutions to scale SVM is a vital goal to enhance the scientific application development process. In analysing this issue, there are a few areas that must be addressed before designing a system to support a distributed high-performance SVM implementation. This includes the identification of issues in training the algorithm. Mini-batch training is one of the primary methods used inefficient training for modern-day machine learning research. The classic SVM algorithm defined with a Lagrangian equation system is a difficult algorithm to scale. This classical approach couples with sequential minimal optimization (SMO). SMO algorithm contains a classical approach of searching for suitable Lagrangian coefficients which can formulate the weight vector. In this approach, search always involves two such coefficients. This search is an exhaustive search which couples with very high accuracy. But as far as performance is considered it becomes a slow approach.

In scaling SVM better, the stochastic gradient descent (SGD) or gradient descent becomes a far suitable approach due to two main reasons. Gradient descent is a highly scalable algorithm and also with some optimizations to learning rate it can be made as accurate as of the SMO solution. Adding to why SGD becomes a scalable algorithm, it's parameter searching is based on a less constrained manner, unlike SMO-based approach. In SMO, programming-wise the number of highly synchronous program units are much lesser than that of SMO. Due to the nature of the optimization problem, to scale well a stochastic gradient descent (SGD)-oriented method has been adopted by many fields of research to improve the performance of SVM. The usage of SGD-based algorithm provides to create a highly scalable SVM algorithm. So our main focus is to design a highly scalable SVM algorithm and we used the SGD-based approach to continue with our research.

With an SGD-based method, one of the most sensitive hyper-parameters in training is the batch size. The batch size determines the number of elements considered when calculating the gradient (weight vector update). Furthermore, the batch size is a very sensitive number, affecting the accuracy and execution time of the algorithm. In a parallel algorithm, the batch size becomes even more sensitive, as the model synchronization or global gradient calculation involves. For various use cases, it becomes necessary to design appropriate programming models to support these requirements. Our earlier research[15] thoroughly analysed how the batch size can influence the performance and accuracy of the application. For this paper, our main purpose is to optimize the existing model and scale the training process on a much higher scale in a cluster. We also investigate multiple application development disciplines to improve the analysis. To achieve the best performance, we also consider the usage of math kernels (BLAS-routines). In application development for machine learning, another important consideration is the nature

of application development based on programming languages. We analyse how C++ and Java programming languages can be used to design an optimized SVM implementation. Performance improvement under optimized library usage and default compiler optimization are also discussed in this paper.

Our distributed SVM algorithm is based on distributed batch processing. For distributed batch processing, the bulk synchronous parallel model (BSP) has been used with great success for decades. BSP models are widely designed with various MPI implementations. This application development process is mostly focused on program execution design using threads, MPI processes and MPI collectives. In referring to the dataflow model, the programming logic is written by just considering the flow of the data in different segments of the data pipeline. This programming logic is bounded towards data loading, data partition, data transformation and data analytics. When it is compared to conventional HPC implementations, programming logic is much simpler in the dataflow model. When considering both HPC and dataflow models, understanding the overheads involved with big data processing allows one to design efficient and effective systems. Most of these BSP models are associated with C++ or Fortran programming interfaces. But there are some extensions developed to support Java virtual machine (JVM)-oriented languages as well. In fashioning scalable application development strategies, focusing on efficient programmable interfaces or languages is essential. Modern-day big data systems deal mainly with JVM-based programming languages due to the efficient programmable interface provided. Also, dataflow modelling frameworks offer an efficient programming interface with rich APIs. Understanding the strengths of both systems helps to design efficient machine learning algorithms.

Another focus in our paper is to evaluate the overlap of HPC and big data programming models in implementing an efficient distributed SVM implementation. Twister2 is one such framework created by unifying strong attributes from each programming models. In Twister2, there are two methods of writing programs. The first uses the dataflow programming model and writes conventional dataflow programs like with similar state-of-the-art dataflow systems [1,6,7]. The other methodology is to use a BSP programming model supported with an MPI backend. Both have a common API abstraction to handle data and design tasks. To enable streamlined programming in Twister2, data management APIs, task management APIs, communication APIs and task scheduling APIs are included. Thus providing a unified API collection to handle both dataflow and BSP applications with or without MPI capability. The objective of our research is to undergo a deeper analysis of scaling SVM algorithms in a broader application development paradigm.

This paper focuses on analysing major aspects in designing efficient distributed SVM implementation for multi-discipline programming development associated with HPC and dataflow communities. The paper's layout is organized as follows: Section 2 will analyse the work related to this field. Section 3 reveals the methodology adopted for the research. This includes a discussion on both HPC and dataflow model-oriented distributed SVM implementation. In section 4, the experiment configurations and details are explained. Section 5 relates the results obtained in the experiments, and in section 6, the conclusions of our research is discussed.

2 | RELATED WORK

SVM is one of the lightweight algorithms in the machine learning domain for supervised learning-based classification problems. In the machine learning domain, SVM by Cortes and Vapnik [16] can be considered the groundwork for developing a far successful classification algorithm. Libsvm [17] was initial work done on developing a complete software library for SVM. LibSVM-based SVM library collection includes numerous programming languages and dataflow frameworks. It also supports multiple kernels and optimization algorithms. One of the most important works done on optimizing the sequential SVM algorithm is the sequential minimal optimization-based SVM by Platt [18], as it is a prominent sequential optimization for SVM. A simplified version of SMO [19] has also been widely used to develop a lightweight version of this algorithm, but low-accuracy is a notable drawback in such implementations. Through improving performance by means of sequential optimization, DC-SVM [20], a divide and conquer-based sequential model, was developed with K-Means clustering. Sequential level optimization can provide performance improvement to a certain extent. When the data size increases, a distributed version of this algorithm is necessary to provide the required performance. PSVM [21 22] is one of the most prominent examples done on a parallel version of SVM algorithm. However with PSVM-based implementation, the traditional Lagrangian multiplier-based optimization is not used. Instead a matrix-based decomposition method for factorization can find the solution to the optimization problem.

Furthermore, SMO-based parallel applications have also been developed by Keerthi et al[23].

The stochastic gradient descent-based approach for SVM optimization is heavily discussed in the work of Shai et al[24] where they use an adaptive learning rate to provide an efficient training model. For distributed SVM with an SGD-based approach, P-PackSVM[25] and parallel stochastic gradient descent[26] can be considered prominent research done to influence optimized distributed models. In addition, fast feature extraction-based SVM models have also been developed to provide efficient training to SVM algorithms[27]. In a distributed application, the main goal is to make sure the communication overhead caused by model synchronization is less than the performance gained by the computation workload distribution. Distributed application development addresses multiple ways to solve this problem. MPI[10] model is the prominent solution when 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. It is vital to improving performance within a process. In order to obtain a performance boost within a process, BLAS[28 29 30 31 32] level operators are required to perform vector-based calculations in an efficient manner. BLAS operations have been used in previous research for improving SMO-based SVM, as well as[33]. On the other hand, it is very important to see how compiler level optimization provides performance improvement.

Referring to language level optimizations, Java-related JIT(Just-In-Time)[34] is a runtime performance-improving compiler. Similarly, compile-time optimization is used in C++ (in some implementations it is recognized using -O3, -O2 and -Ofast level optimization[35]). Dataflow frameworks like Apache Spark[1], Apache Flink[6], Google Dataflow[9], Apache Storm[7], Storm @Twitter[36], and Heron[8] have dataflow-based solutions to solve big data-related problems in both streaming and batch mode datasets. Each of these frameworks sports a well-defined data pipeline for application users to develop big data applications on distributed environments. Twister2[37,2] is a big data toolkit designed to provide a variety of functionalities to both HPC and dataflow application developers. Twister2: Net[13] is an optimized communication library that contains an MPI-like communication style with TCP-based communication. Application development abstractions are important for creating applications with efficiency. TSet[14] API in Twister2 is another big data programming abstraction geared towards fashioning optimized applications similar to Spark RDD format.

3 | METHODOLOGY

The current section discusses the methodologies needed to scale SVM in HPC and dataflow systems. Here we consider these methodologies under two sections. HPC-stack based implementations and related implementation improvements. The next section focuses on state of the tools in the big-data stack. HPC-related application development and optimization deal with the language-level optimizations on Java and C++, BLAS routine-based optimizations and single-node multi-core parallel programs. The first step for implementation is understanding the anatomy of the SVM algorithm (3.1). The methodology adopted in this research is divided into two main components. The first is HPC model-based performance analysis (3.2) with and without BLAS routines3.2.1. The second conducts experiments on distributed dataflow models on the big data stack (3.3). In the HPC model, we discuss how MPI-based parallel processing improves performance. We observe how process performance can be improved with BLAS operations. Following this, we scale the application from single-node multi-core to multiple-node multi-core. For the distributed dataflow model, we consider an ensemble model of the distributed algorithm using the same core algorithm but only focusing on model design and scale-up. Spark-RDD, Twister2-Task and Twister2-TSet frameworks are used to develop the application. We also fashion the same ensemble model with MPI designed with OpenMPI 3.1.2 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 iterative training is done for a considered amount of iterations (or iterations until convergence).

Algorithm 1 Gradient Descent SVM

$INPUT : [x, y] \in S, w \in R^d, t \in R^+, b \in R^+$
$OUTPUT : w \in R^d$
procedure GRADIENT DESCENT(S, w, t, b)
  for $i = 0$ to n **do**
    if $(g(w; (x_i, y_i)) == 0)$ **then**
      $\nabla J^t = w$
    else
      $\nabla J^t = w - Cx_i y_i$
    $w = w - \alpha \nabla J^t$
  return w

FIGURE 1 SGD SVM Algorithm

Algorithm 2 Parallel Gradient Descent SVM

$INPUT : [X, Y] \in S, w \in R^d, b \in R^d$
$OUTPUT : w \in R^d$
procedure PARALLEL GRADIENT DESCENT(S, w, b)
  Parallel in K Machines $[S_1, ...S_k] \in S$
  for $t = 0$ to T **do**
    procedure GRADIENT DESCENT(S, w, t, b)
    w = MPI_AllReduce(w) / K
  return w

FIGURE 2 PSGD SVM Algorithm

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

$$\alpha \in (0, 1) \tag{2}$$

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

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

We implemented the SVM using a SGD-based optimization. The classification algorithm considered in this research is focused on binary classification. In Equation (1), the sample space S is defined with x with feature vector and y with the label. Each data point x_i contains a set of d features with an associated label being $+1$ or -1. In Equation (2), the learning rate used in the stochastic gradient descent is denoted. The learning rate is a floating point value which in generally chosen between $0 < \alpha < 1$. Equation (3) denotes the conditional statement which selects the gradient update function in 1. In equation 4, the constant C is a hyper-parameter in the objective function. Additionally, this objective function generally has a regularization part and a hinge loss part. We omitted the hinge loss section as in this paper we are mostly focused on the implementation performance improvement rather than convergence. Also in 2, we use the term K to refer the number of machines/processes involved in the distributed setting. Here we use the allreduce operator to synchronize the weights across the K machines using the sum operator in MPI. Here we divide the weight vector w by K to get the average of the summation. Also b in the distributed algorithm refers to the batch size. If the stochastic approach is used $b = 1$ and in batch gradient descent approach $b > 1$.

3.2 | HPC Model Implementation

The framework we chose in designing an HPC model is the MPI standard. We selected the MPI implementation OpenMPI 3.1.2 as the backend in writing BSP models. The first step was developing a methodology to test the performance of Java and C++ applications made to solve the SVM optimization problem. An important point to note is OpenMPI supports Java-enabled compilation to provide a Java programming abstraction on top of the core C++ OpenMPI implementation. 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 feature sizes. With the conclusions obtained from researching the parallel stochastic gradient descent with model synchronization [15], the algorithm in Figure 2 was designed using the core optimizer algorithm in Figure 1.

3.2.1 | BLAS Optimization

When considering the core algorithm 1, to optimize the dot products and vector scalar multiplications, the BLAS level routines can be applied. For this research we selected the OpenBlas implementation on Red Hat Enterprise Linux Server 7.6 (Maipo)

operating system. The mathematical equation and corresponding BLAS operation mappings for implementation of BLAS level operations are shown in Equations (5), (6), (7) and (8).

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

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

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

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

In equation 5, the $ddot$ signature refers to a BLAS operation which performs dot product of two vectors[38]. In equations 6, 7 and 8 refers to $daxpy$ BLAS operation which performs constant times a vector plus a vector[39]. Additionally, the inc_x and inc_y refers to the storage space between the elements in the x and y arguments of the $daxpy$ notation. Where x and y refers to two vectors of similar length.

3.3 | Dataflow Model Implementation

Considering the dataflow model, specifically for batch data processing, our analysis has been focused on using Apache Spark and Twister2[13],[2],[14]. Apache Spark is one of the most prominent tools for dataflow frameworks used by many data scientists and big data application developers[1,40,41,42,43]. 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 resilient distributed data (RDD). For the purpose of improving dataflow models, the Digital Science Center in Indiana University Bloomington has produced a framework called Twister2. This big data toolkit 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, while TSet is a data-level abstraction on top of task API which is similar to the RDD API in Spark. TSet[14] is a programming abstraction which allows the users to create data transformations in a chained manner. This involves the use of map, filter, for each, and many other custom transformations, which facilitates developing applications in a much easier way. Apache Spark RDD is an equivalent data API to write dataflow programs. Here we have developed the SVM algorithm in an ensemble way in MPI using Java, Spark RDD, Twister2 Task API and Twister2 TSet API. Referring to the ensemble method, here what we do is initially models are being trained in parallel processes without doing synchronization till the end of all epochs (or convergence of the algorithm). Then we call an allreduce to take the sum of the weights and calculate a mean from that vector summation.

4 | EXPERIMENTS

The experiments in this research were conducted in the Victor and Juliet cluster group in the FutureSystems cluster at Indiana University Bloomington. For the single node experiments, we used a maximum of 32 processes in a node, and for distributed mode experiments we used an equal number of processes per machine over a group of 16 nodes. For instance, when the expected parallelism is 32, each node will run two processes (configured using *OpenMPI*). Victor cluster nodes are comprised of *Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz* configuration. Juliet cluster nodes include *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 every parallelism each machine gets an equal number of processes. For the parallelisms in the range 2 to 8, a single process was scheduled in one machine. For example, 4 parallelisms were designed such that they were scheduled in 4 machines. For parallelisms in the range 16 to 256, each machine gets an equal number of processes among all 16 machines. For the C++/Java based experiments on single node and distributed modes we used the Juliet cluster. For C++ oriented experiments we used the *g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)* compiler version and we used the following Java configuration *java version "1.8.0_221", Java(TM) SE Runtime Environment (build 1.8.0_221-b11),Java HotSpot(TM) 64-Bit Server VM (build 25.221-b11, mixed mode)*. For the big data stack-based applications, we considered the processor affinity and configured Twister2 and Spark in such a way that a single process is run per core in the distributed mode experiments. The affinity here refers to the idea that when we schedule a job with

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

M processes among K nodes, we make sure equal number of cores are used in each machine. For these experiments we used the Victor cluster.

4.1 | Dataset Configuration

For the following experiments, we used 3 datasets considering the features in terms of data point, sparsity and data size as shown in Table 1. This table contains information on training data size, testing data size, the sparsity of the dataset and number of features per data point. The reasoning behind selecting these three datasets is due to the nature of variable feature size with 10× scale-up, variable sparsity and variable data size. The feature size variation allows us to conduct experiments including a clear communication overhead in scaling the applications.

4.2 | HPC Benchmark Configurations

HPC benchmarks were designed for both Java and C++ languages with the OpenMPI backend. These implementations employed the compiler level optimizations. C++ uses the *03* level optimization while Java uses compiler level optimization along with *JIT* compiler optimization at runtime. The purpose of this experiment is to see how the same algorithm developed for each language performs with the scaling done within a node or single machine. Here we carried out two sets of experiments. The first uses compiler optimization from each language. In the second, each language uses the *BLAS* routines-based optimization for vector vs. vector and vector vs. scalar multiplications. In these settings, we analysed how the implementations behave for three datasets with varying sparsity, features and size.

4.2.1 | 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.

4.3 | Dataflow Benchmark Configurations

For the experiments in Java, *Java(TM) SE Runtime Environment (build 1.8.0_101-b13)* was used, and for C++ *OpenMPI 3.1.2* 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 *Apache Spark, 2.4.0* version was used while OpenMPI 3.1.2 was assigned for Java-based MPI applications.

5 | RESULTS

This section details the results from our experiments conducted in Section 4. We discuss the results obtained from dataflow benchmarks and compare them with the corresponding HPC benchmarks.

5.1 | HPC Benchmarks

We determine the performance of HPC-based implementations under the following categories. First we analyse the single node experiments with and without BLAS routines for Java and C++ implementations. Then we expand the experiments to multi-node experiments with and without BLAS routines for Java and C++ implementations. Finally we discuss the reasoning behind the observations made with these experiments.

5.1.1 | Single Node Experiments

Figure 3 and Figure 4 show the experiments conducted on Java and C++ implementations. The plots contain log scale training time to showcase the training time loss with scale-up for all three datasets. These experiments explore implementations with and without BLAS routines. The suffix with BLAS in the legend shows the experiments with BLAS routines, while NO BLAS shows the experiments without BLAS routines. From the observations in Figure 3, it is clear that the scaling of the SVM algorithm is much better with the usage of BLAS routines for C++ implementation. But even without BLAS routines the application scales well. For the Epsilon dataset with the highest amount of dataset for training and the highest number of features per data point, the scale-up can be observed. The speedup obtained from 32× parallelism vs 2× parallelism is 11 when it is compared to the ideal speedup of 16. This was recorded with BLAS routines and MPI-based parallelism obtained with allreduce collective communication.

In referring to the Java-based implementation, the observations in 4 show that the BLAS routines do better only with Epsilon dataset, performing on average with Webspam dataset and relatively slow on Ijcnn1 dataset. In order to understand this performance drop with Java-based BLAS routines, we conducted a micro-benchmark on variable feature size on sequential training time of the SVM algorithm. Figure 9 shows the average training time on the sequential version of the SVM implementation on fixed data size and variable feature size. This shows that the BLAS routine support for Java using *Netlib* provides better performance on higher feature sizes.

Java cannot directly obtain the BLAS optimization from the BLAS routines. The reason is these optimization libraries are written in C++/C by accessing native operating system level functionality, while Java runs on a virtual machine called JVM (Java Virtual Machine). To access 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. For this to happen, the manner in which JNI works has to be understood. JNI native functions are implemented in C++/C medium. When JVM invokes a native function, a JNIEnv pointer is 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 less than the compiler optimized code in Java. For smaller datasets the conversion time with computation time is smaller as the array size involved in dataset Ijcnn1 is 22 and in the dataset Webspam it 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.

5.1.2 | Multi-Node Experiments

With the results of the single node experiments to go by, we focused on the scalability of the multi-node experiments. Here we used two categories of experiments, one with BLAS optimizations and the other with the compiler optimizations. Figure 7 shows the performance comparisons obtained in the BLAS routine-enabled multi-node experiments. Here the C++ implementation shows a slightly better performance than Java implementation. Figure 8 shows the performance comparisons obtained with compiler level optimizations in the multi-node experiments. The observation from this result set is that Java implementation outperforms C++. With the improvement of JIT compiler and many other optimizations in Java, the compiler level analysis is a possible result. But this result set does not harness the maximum capability of both languages under the vectorization and optimized math kernel usage. So understanding the results with the best optimization possible guarantees the highest scalability of distributed SVM algorithm. Additionally, the most important observation is that our SVM implementation scales well up to 256 parallelisms for Webspam and Epsilon datasets. Ijcnn1 scales only up to 128 parallelisms as it has less data in the training

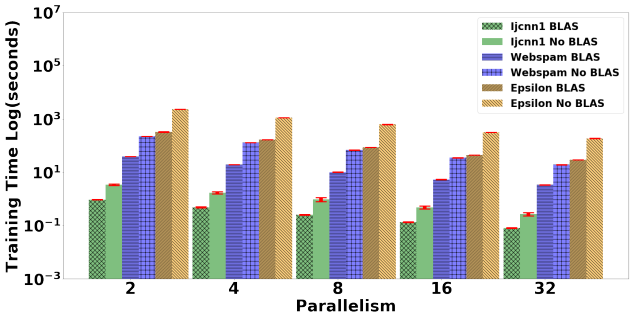

FIGURE 3 C++ Single Node Experiments

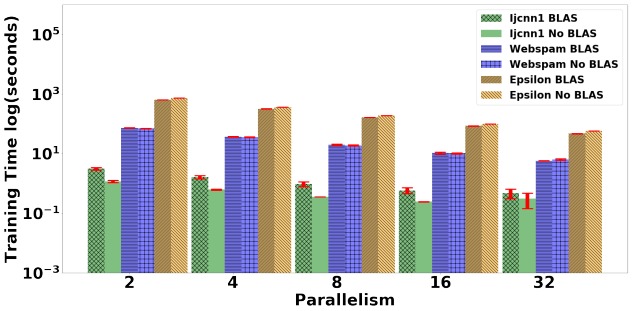

FIGURE 4 Java Single Node Experiments

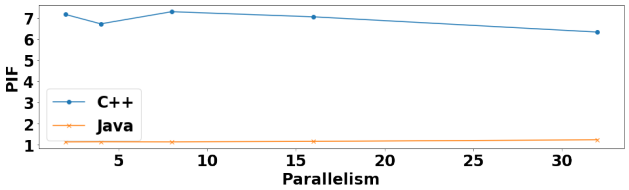

FIGURE 5 Performance Improvement on Single Node Experiments with Epsilon Dataset

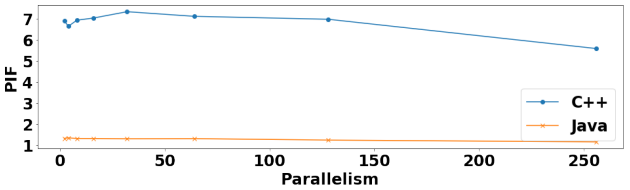

FIGURE 6 Performance Improvement on Distributed Node Experiments with Epsilon Dataset

dataset. This shows a similar scale-up compared to single node experiments.

To better understand the results in-depth, we conducted two other experiments for both single-node and multi-node experiments. The purpose was to observe the scalability of the algorithm with and without inter-node communication overhead. We used this approach because it can provide a better understanding of the results. Figure 5 and 6 show the performance evaluation with Java and C++ with BLAS optimizations in single-node and multi-node experiments. Here the 5 refers to the C++ and Java based single node experiments for variable parallelisms and it records the performance improvement of blas-based approach over non-blas-based approach. The 6 refers to the corresponding distributed node experiments. The y-axis in both figures shows the *performance improvement factor (PIF)*. PIF was evaluated as shown in (9).

$$PIF = \frac{Training\ Time\ with\ BLAS}{Training\ Time\ without\ BLAS} \tag{9}$$

We selected this metric because BLAS always provided a performance boost for all implementations in Java and C++. The results from these experiments show that the BLAS optimization on C++ offered a higher speed-up on the specific hardware. As explained in the single node experiments in Section 5.1.1, the overheads associated with obtaining BLAS performance with Java lead to a performance lag when compared to the pure native implementation in C++. The PIF factor on C++ is approximately 7.0, while it is approximately 1.3 for Java. In analyzing this further, a clear comparison can be seen with the Epsilon dataset. It has the highest number of features and the highest number of data for training. This includes a larger communication overhead and a higher number of iterative computations. The main reason for Java implementation low performance from BLAS operations is that the BLAS 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. This observation is further supported by the observations we made in Figure 9. With the increasing array size of feature size per data point, Java can obtain better performance. We observed this in-depth in the single-node experiment discussion5.1.1. For obtaining the highest performance, as C++ and associated kernels work well with BLAS, the data pre-processing logic can be handled in the Java end and high-performance code can be implemented by calling kernels designed with C++ implementations. So in high iterative models with higher communication overhead, the Java-based applications can be improved by optimizing JNI function calls with the usage of C++ kernels.

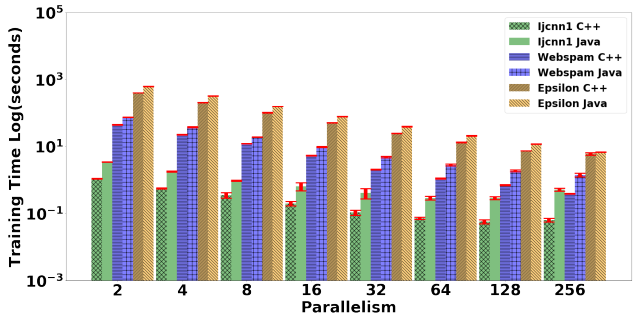

FIGURE 7 Java and C++ Distributed Node Experiments with BLAS

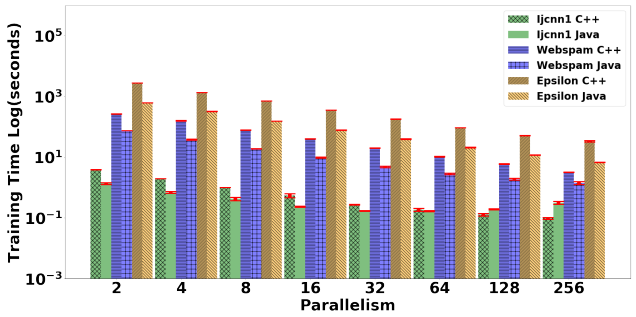

FIGURE 8 Java vs. C++ Compiler Level Optimization-Based Performance

5.2 | Dataflow Benchmarks

In expanding the benchmarks towards dataflow-oriented SVM implementation, we formulate a set of experiments in comparing HPC implementations along with the state-of-the-art dataflow models. For this experiment set, we include Spark-RDD-oriented implementation, Twister2-task implementation, Twister2-TSet implementation and corresponding HPC implementation with OpenMPI backend. For the big data domain and HPC domain performance comparison with distributed SGD-based SVM algorithms, 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 still being faster than the Spark RDD-based application. In considering the anatomy of MPI-Java application, MPI Allreduce communication performs the model synchronization across the processes. Twister2 also supports the allreduce-based 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 and Twister2 implementations vs. Spark implementation is the way the models are synchronized. While MPI and Twister2 perform a tree-based reduce, Spark synchronizes the modelsfrom each task back to the drive. 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. Clarifying Twister2 performance further, the existing literature on Twister2 communication model (Twister2:Net) shows that it performs better when compared to classic dataflow models. Furthermore, it ranks closely in terms of performance when compared to the state-of-the-art BSP models in the HPC domain. With the usage of an ensemble model, the noise in the experiments is slightly less than the iterative model-based experiments on HPC benchmarks discussed in Section 5.1. We designed the scale-up experiments using up to 256 parallel processes (16 nodes with 16 cores each) and 320,000 data samples. Here we didn't use thread parallelism, only used the process level parallelism with MPI-processes. The objective of the dataflow benchmark is to showcase the usage of optimized dataflow models to obtain better performance. From the dataflow model benchmarks, it is clear that our Twister2 implementation can be very effective in designing iterative and non-iterative workloads at scale with better performance compared to classic dataflow engines.

6 | CONCLUSION

Our research primarily focused on two aspects in designing scalable distributed SVM implementation. First, we investigated the high-performance computing implementations on two popular programming languages used in academic and industrial research. Then we investigated how state-of-the-art dataflow engines can be used to implement a scalable distributed SVM. We expanded our analysis to compare both HPC and dataflow models on a scalable SVM implementation. Our experiments resulted in several conclusions. Firstly, HPC-oriented application development on multiple programming languages like Java and C++ provide a variety of advantages. With Java implementations, the main objective is to obtain better programmability. With C++ implementations, the objective is to gain higher performance in computationally intensive cases. In supporting

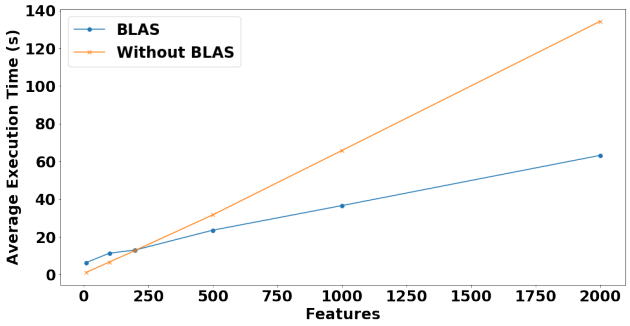

FIGURE 9 Java BLAS Performance Against Vector Size Variation

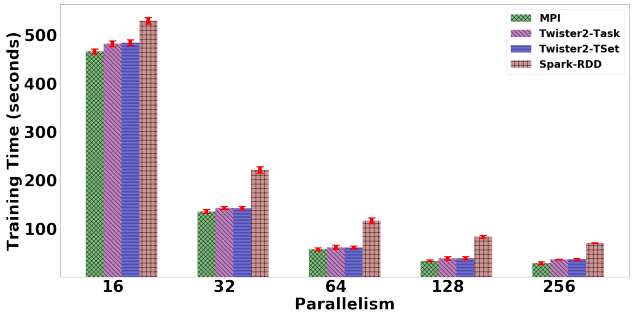

FIGURE 10 Big Data Stack vs HPC Benchmark on Distributed Ensemble SVM

this statement, we draw our conclusions as follows. Our observations from without-BLAS optimizations in Java shows that with-BLAS, the performance is slightly low when the number of features per data points is smaller. This fact is clear from the results gathered from Ijcnn1 (22 features) and Webspam (254 features) datasets. But with the Epsilon (2000 features) the BLAS implementation outperforms without-BLAS optimized implementation. We analyzed this furthermore and identified that with the increasing feature size BLAS-based Java implementation can outperform just-compiler optimized Java implementation. This is one of the highlights of our finding in showcasing where Java-based distributed SVM implementation can make the best use. Furthermore, this shows where better programmability can be used to gain better use in implementing distributed SVM.

Furthermore, we analyzed the major computational intensive cases where data size and a number of features per data-point is very high. To showcase this, the experiments were done on the Epsilon dataset which has a higher number of data-points and a higher number of features per data-point, compared to Ijcnn1 and Webspam datasets. This analysis done on C++ vs Java against BLAS and without BLAS implementations shows a performance improvement factor between 1 and 2 for Java and 6 and 7 for C++. This concludes that C++ outperforms Java when a dataset has a higher number of features. With the increasing data sizes, most of the cases fall into the higher dimensional cases where C++ based implementations with BLAS can be leveraged for better performance. This is another important conclusion drawn from our research findings. This finding confirms our claim that C++ allows gaining higher performance in computationally intensive scenarios. But for datasets with a smaller number of features can get better performance with both Java than C++ implementations. There are also a vast number of such applications in various disciplines of scientific research. Depending on the programmability factor and the nature of the data distribution, our in-depth analysis allows a user to select the most suitable developer configurations to get the better performance in implementing SGD-based distributed SVM in scientific and enterprise applications.

The conclusions drawn from the dataflow experiments demonstrate that optimized dataflow engines like Twister2 are promising in obtaining better scaling for classic dataflow engines like Apache Spark. Furthermore, the native MPI implementation of distributed SVM also showcases that Twister2 implementation provides decent performance compared to it. The usage of high-performance computing principles in the dataflow communication provides better performance in distributed training of the SVM algorithm at scale. This showcase that our dataflow-based distributed SVM model is a better choice when implementing distributed SVM for various scientific use-cases. Twister2 TSet includes an efficient dataflow API like Apache Spark RDD/-Dataset for easier programming. It also offers better scale-up compared to a corresponding HPC implementation. The research conducted in this paper shows the optimized design for the distributed SVM implementation at scale on high-performance computing and dataflow paradigm.

References

1. 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.

2. 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.

3. Walker DW, Dongarra JJ. MPI: a standard message passing interface. *Supercomputer* 1996; 12: 56–68.

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

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

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. 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.

11. MPICH | High-Performance Portable MPI. https://www.mpich.org/; . (Accessed on 01/05/2020).

12. MVAPICH :: Home. http://mvapich.cse.ohio-state.edu/; . (Accessed on 01/05/2020).

13. 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.

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

15. 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

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

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

18. Platt J. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Tech. Rep. MSR-TR-98-14, Microsoft Research; 1998.

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

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

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

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

23. 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.

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

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

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

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

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

29. Sam Halilday YYEA. Netlib-Java. *Github Open Source* 2013.

30. Xianyi Z, Kroeker M, Saar W, et al. Open Blas. *Github Open Source* 2019.

31. 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.

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

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

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

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

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

37. 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.

38. Dongara J. LAPACK: ddot. http://www.netlib.org/lapack/explore-html/de/da4/group__double__blas__level1_ga75066c4825cb6ff1c8ec4403ef8c843a.html#ga75066c4825cb6ff1c8ec4403ef8c843a; . (Accessed on 06/07/2020).

39. Dongara J. LAPACK: daxpy. http://www.netlib.org/lapack/explore-html/de/da4/group__double__blas__level1_ga8f99d6a644d3396aa32db472e0cfc91c.html; . (Accessed on 06/07/2020).

40. Meng X, Bradley J, Yavuz B, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 2016; 17(1): 1235–1241.

41. Shanahan JG, Dai L. Large scale distributed data science using apache spark. In: ACM. ; 2015: 2323–2324.

42. Shoro AG, Soomro TR. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology* 2015.

43. Reyes-Ortiz JL, Oneto L, Anguita D. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science* 2015; 53: 121–130.