# Fast and Accurate Support Vector Machines on Large Scale Systems

Abhinav Vishnu[1] Jeyanthi Narasimhan[2] Lawrence Holder[3] Darren Kerbyson[4] Adolfy Hoisie[5]

[2,3] *School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164*
[1,4,5] *Advanced Computing, Mathematics and Data Division, Pacific Northwest National Laboratory,*
*902 Battelle Blvd, Richland, WA 99352*

[1] abhinav.vishnu@pnnl.gov
[2] jsalemna@eecs.wsu.edu
[3] holder@wsu.edu
[4] darren.kerbyson@pnl.gov [5] adolfy.hoisie@pnl.gov

***Abstract*—Support Vector Machines (SVM) is a supervised Machine Learning and Data Mining (MLDM) algorithm, which has become ubiquitous largely due to its high accuracy and obliviousness to dimensionality. The objective of SVM is to find an optimal boundary — also known as hyperplane — which separates the samples (examples in a dataset) of different classes by a maximum margin. Usually, very few samples contribute to the definition of the boundary. However, existing parallel algorithms use the entire dataset for finding the boundary, which is sub-optimal for performance reasons.**

**In this paper, we propose a novel distributed memory algorithm to eliminate the samples which do not contribute to the boundary definition in SVM. We propose several heuristics, which range from early (aggressive) to late (conservative) elimination of the samples, such that the overall time for generating the boundary is reduced considerably. In a few cases, a sample may be eliminated (shrunk) pre-emptively — potentially resulting in an incorrect boundary. We propose a scalable approach to synchronize the necessary data structures such that the proposed algorithm maintains its accuracy. We consider the necessary trade-offs of single/multiple synchronization using in-depth time-space complexity analysis. We implement the proposed algorithm using MPI and compare it with `libsvm` — *de facto* sequential SVM software — which we enhance with OpenMP for multi-core/many-core parallelism. Our proposed approach shows excellent efficiency using up to 4096 processes on several large datasets such as UCI HIGGS Boson dataset and Offending URL dataset.**

## I. Introduction

Machine Learning and Data Mining (MLDM) algorithms are becoming increasingly popular for analyzing the exorbitant volume of data produced by simulations and instruments [1], [2]. MLDM algorithms are widely applied for modeling, classification and clustering in many science domains [3]–[5]. Wu *et al.* have identified the top 10 MLDM algorithms [6]. Support Vector Machines (SVM) is one such algorithm which is used for non-parametric modeling, classification and regression. SVM has become the *de facto* supervised learning algorithm due to its excellent accuracy and obliviousness to dimensionality. Due to its excellent theoretical foundation and accuracy, several SVM algorithms have been proposed in literature and practice [7]–[11]. A majority of these algorithms are either sequential or scale to a single compute node [11].

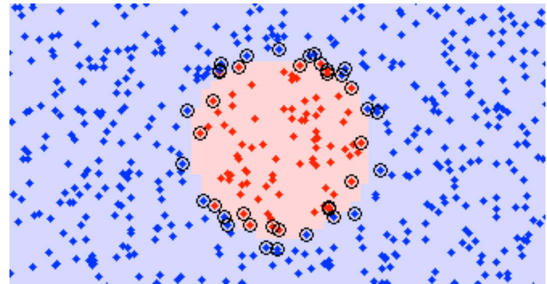

Fig. 1: A two-class dataset with support vectors — samples which contribute to boundary definition (encircled)

A few other researchers have proposed distributed memory SVM algorithms. Parallel SVM (PSVM) [12] is an example which uses a kernel-matrix to save important computations, such that they can be re-used during the iterative procedure in SVM. However, it ceases to scale due to its prohibitive space complexity ($\Theta(n^2)$ for $n$ samples). Other parallel approaches such as `MLLib` [13] also primary rely on large kernel-cache for scaling SVM algorithm.

Figure 1 shows an example of a two-class dataset (marked with blue and orange points). *Support vectors* — samples which contribute to the definition of the boundary are shown as encircled. We observe that a very small fraction of the samples are support vectors. SVM algorithms (the most popular example is Sequential Minimal Optimization (SMO) [14]) use an iterative procedure to find these support vectors. Existing distributed memory SVM algorithms (such as distributed SMO) use the entire dataset for finding the support vectors, potentially wasting significant compute time in processing the samples, which could have been *shrunk* (eliminated from consideration).

Hence there are several important questions to be considered: 1) What are the properties of a sample, such that it could be shrunk? 2) When should the sample be shrunk? 3) How to maintain the accuracy of the proposed shrinking based algorithm — since it can falsely eliminate a few samples — and what are the space-time complexity tradeoffs?

## A. Contributions

Specifically, in this paper, we make the following contributions:

1) We propose a novel distributed memory SVM algorithm, which adaptively shrinks the non-contributing samples during the iterative procedure. We propose and implement several heuristics, ranging from early (aggressive), average and late (conservative), while considering single and multiple eliminations of samples during the procedure.
2) We propose a distributed memory algorithm to synchronize the important data structures — gradient reconstruction — of falsely eliminated samples, such that **the accuracy of the proposed solution remains intact**. In practice, the cardinality of falsely eliminated samples is a very small fraction of the entire dataset — which keeps the overhead of this step low.
3) We use a compressed representation of data samples and avoid the kernel cache completely. The proposed SVM algorithm is effective for memory-restricted many-cores and large scale systems on the horizon.
4) We enhance `libsvm` [15] — the *de facto* sequential SVM software — for multi-core parallelism by using OpenMP. Unlike existing literature — which uses sequential `libsvm` as the baseline — the enhanced version provides a fairer baseline for performance comparison with the proposed SVM algorithm.
5) We implement the proposed SVM algorithm using MPI and evaluate it with ten different datasets using up to 4096 cores. Our proposed SVM algorithm is able to train the URL dataset (2.3M samples) on 4096 processes in 8 minutes, while the enhanced `libsvm` (using 16 cores on one node) takes 39 hours to complete the training. On UCI HIGGS dataset using 2.6M samples, the proposed algorithm is able to complete the training in 2 hours, while the Original algorithm takes more than 3 hours. The proposed algorithm and its implementation are available with Machine Learning Toolkit for Extreme Scale (MaTEx) [16].

The rest of the paper is organized as follows: In section II, we provide a brief introduction to SVM. In section III, we present a solution space of the algorithms and associated heuristics in section IV. In section V, we present a performance evaluation and analysis of the proposed SVM algorithm. In section VI we present the related work and present the conclusions in section VIII.

# II. Support Vector Machines (SVM)

## A. Sequential Minimal Optimization (SMO)

SVM training, which solves the dual problem, is typically conducted by splitting a large optimization problem into a series of smaller sub-problems [17]. The SMO algorithm [14], [18] uses exactly two samples at each optimization step. With two samples, it is possible to generate an analytical solution for the quadratic minimization at each step.

*1) Gradient updates:* In the SMO algorithm [14], several data structures are read/updated at each step. Gradient ($\gamma$ in Table I) is a data structure, which is maintained for each sample. It is calculated as follows:

$$\gamma_i = \textstyle\sum_j \alpha_j y_j \Phi(\mathbf{x}_i, \mathbf{x}_j) - y_i \tag{1}$$

The gradient of the dual objective function for each sample is updated after every iteration. The gradient for i-th sample ($\gamma_i$) is updated as:

$$\begin{aligned} \gamma_i^{new} &= \gamma_i^{old} + \\ y_{up} * (\alpha_{up}^{new} - \alpha_{up}^{old}) &* (\Phi(x_{up}, x_i)) + \\ y_{low} * (\alpha_{low}^{new} - \alpha_{low}^{old}) &* (\Phi(x_{low}, x_i)) \end{aligned} \tag{2}$$

*2) Working Set Selection:* For faster convergence, an SVM algorithm (like SMO) selects a few samples (two in SMO algorithm) for gradient calculation of remaining samples (Eq. (2)). Keerthi *et al.* have proposed several possibilities for working set selection [18]. They are shown in Algorithm 1. In one case, the algorithm iterates over all samples in $I_0$. In the second case, the algorithm evaluates only the *worst KKT violators* — $\beta_{up}$ and $\beta_{low}$ — which are calculated at each step using Eq. (3). The other data structures are updated as follows:

$$\begin{aligned} \beta_{up} &= min\{\gamma_i : i \in I_0 \cup I_1 \cup I_2\} \\ \beta_{low} &= max\{\gamma_i : i \in I_0 \cup I_3 \cup I_4\} \end{aligned} \tag{3}$$

where

$$\begin{aligned} i \in I_0 \cup I_1 \cup I_2 \cup I_3 \cup I_4. \quad I_0 = \{i : 0 < \alpha_i < C\}, \\ I_1 = \{i : y_i = 1, \alpha_i = 0\}, I_2 = \{i : y = -1, \alpha_i = C\}, \\ I_3 = \{i : y_i = 1, \alpha_i = C\}, I_4 = \{i : y_i = -1, \alpha_i = 0\} \end{aligned} \tag{4}$$

$\beta_{up}$ and $\beta_{low}$ are are two threshold parameters, which are discussed in detail by Keerthi *et al.* [18]. The optimality condition for termination is:

$$\beta_{up} + 2 * \epsilon \geq \beta_{low} \tag{5}$$

where $\epsilon$ is a user-specified tolerance parameter. We also observe from Eq. (2) and Eq. (3), that the worst violators for the next iteration are gathered by considering all the samples, which keeps the accuracy of the solution intact.

## B. Message Passing Interface

Message Passing Interface (MPI) [19], [20] supports mail-box style of communication. MPI provides a rich set of interfaces for point-to-point operations, collective communication operations (group operations) and one-sided primitives (for remote memory access operations). In this paper, we specifically use several MPI primitives : `MPI_Send`, `MPI_Recv` for exchanging dataset during gradient-reconstruction, `MPI_Allreduce` (collective operation for reduction on data structures) for exchanging $\beta_{up}$ and $\beta_{low}$.

Most modern supercomputers support MPI, since large scale applications use MPI directly/indirectly. High performance MPI implementations on modern interconnects such as InfiniBand, Cray and Blue Gene systems have become available [21]–[26]. .

## III. Solution Space: Preliminaries

TABLE I: Symbols used for modeling time-space complexity of SVM

| Name | Symbol |
|---|---|
| # of Processors | $p$ |
| # of Training Points | $\mathcal{N}$ |
| Class label | $y_k$ |
| Lagrange multiplier | $\alpha_k$ |
| Set of Support Vectors | $\zeta$ |
| Working set | $\pi$ |
| $\delta L_D/\delta\alpha_k$, $\gamma_k * y_k$ (1) | $\nabla_k$ |
| Hyperplane threshold | $\beta$ |
| Indices set $I_{0-4}$in (4) | $\varsigma$ |
| Shrinking threshold | $\delta$ |
| User-Specified Tolerance | $\epsilon$ |
| Avg $\langle, . , \rangle$ time | $\lambda$ |
| Average sample length $\|\boldsymbol{x_k}\|$ | $m$ |
| Network Latency | $l$ |
| Network Bandwidth | $\frac{1}{G}$ |

**Algorithm 1:** SVM training - Sequential

**Input**: $\mathcal{C}$, $\sigma$, $\mathcal{X} \in \Re^{\mathcal{N}\times d}$, $y_i \in \{+1, -1\}, i = 1, 2, \dots \mathcal{N}$
**Data**: $\boldsymbol{\alpha} \in \Re^{\mathcal{N}\times 1}$
**Result**: $\zeta$

1 $\forall i, \gamma_i = -y_i, \alpha_i = 0$;
2 $i_{low} \in \{j \mid y_j = 1, j \in \{1, 2, \dots \mathcal{N}\}\}$;
3 $i_{up} \in \{k \mid y_k = -1, k \in \{1, 2, \dots \mathcal{N}\}\}$ ;
4 **while** $\beta_{up} + 2 * \epsilon \leq \beta_{low}$ **do**
5 $\quad\alpha_{i_{low}} \leftarrow$ (6), $\alpha_{i_{low}} \leftarrow$ (6);
6 $\quad i_{low} \leftarrow \varsigma$, $i_{up} \leftarrow \varsigma$ (4);
7 $\quad\forall i$, $\gamma_i \leftarrow$ (2);
8 $\quad$Calculate new $\beta_{low}$ and $\beta_{up}$ using (3);

Table I shows the symbols which we have used to model the time and space complexity of the proposed algorithms and heuristics. Algorithm 1 shows the key steps in the sequential SVM algorithm. At each iteration, $\forall i$, $\alpha_i$ is calculated using Eq. (6). When the objective function is positive definite ($\rho < 0$) (the objective function is always positive semi-definite, and positive definite for Gaussian kernel ($e^{-\gamma \cdot ||x_i - x_j||^2}$)) , Eq. (7) is used for the updates. We use the approach proposed by Platt *et al.* [14] to update the equations, when $\rho > 0$.

$$\begin{aligned} \alpha_{i_{low}}^{new} &= \alpha_{i_{low}} - y_{i_{low}} * (\gamma_{i_{up}} - \gamma_{i_{low}})/\rho \\ \alpha_{i_{up}}^{new} &= \alpha_{i_{up}} + y_{i_{low}} * y_{i_{up}} * (\alpha_{i_{low}} - \alpha_{i_{low}}^{new}) \end{aligned} \tag{6}$$

where

$$\begin{aligned} \rho = 2 * \Phi(\mathbf{x}_{i_{low}}, \mathbf{x}_{i_{up}}) \\ -\Phi(\mathbf{x}_{i_{up}}, \mathbf{x}_{i_{up}}) - \Phi(\mathbf{x}_{i_{low}}, \mathbf{x}_{i_{low}}) \end{aligned} \tag{7}$$

At the terminating condition (5), $\beta$ is calculated as:

$$\beta = \begin{cases} \sum_{i \in I_0} \gamma_i / |I_0| & \text{if } |I_0| \neq 0 \\ (\beta_{low} + \beta_{up})/2 & \text{otherwise} \end{cases}$$

### A. Data Structures Organization and Kernel Cache

*1) Data Structures Organization:* As shown in Table I, there are several important data structures which are used in algorithms 2 , 3, 4 and 5. These data structures include $\mathcal{X}$, the input dataset; $y$, the sample label; $\boldsymbol{\alpha}$, the Lagrange multipliers; $\gamma$, gradient and $\varsigma$, the set information. The data structures — $\boldsymbol{\alpha}$, $\gamma$, $\varsigma$ and $y$ — are associated with each sample. For performance reasons, we co-locate these data structures with individual samples. This improves the spatial locality of the proposed solution.

We have also observed that most datasets are sparse in nature (several datasets have less than 20% density). Hence, we use a compressed sparse row (CSR) representation to store the dataset $\mathcal{X}$. For several datasets — especially the large ones — we did not observe a sparsity pattern, which is frequently observed in scientific domains. Hence, we use the basic CSR format, and consider using other formats in the future.

*2) Kernel Cache:* As evident from Eq. (2), gradient updates — the most computationally expensive part of the calculation — can be formulated by using a series of kernel calculations. The individual kernel calculations may be stored in a kernel cache, which itself can be distributed across different processes. However, the space complexity of a complete kernel cache (also known as a kernel-matrix) is $\Theta(\mathcal{N}^2)$, which is prohibitive for large input. Even with a kernel cache, at each step, the entire rows corresponding to indices $i_{up}$ and $i_{low}$ need to be sent to each process. This results in a data movement of $O(\mathcal{N} \cdot \log(p))$ (The rows can be sent using an MPI primitive such as `MPI_Bcast`, which has a time-complexity of $O(\log(p))$).

Besides prohibitive data movement, there are several other reasons to avoid a kernel cache: The primary objective of a kernel cache is to maximize temporal utilization. However, for a fixed kernel cache size, the probability of a cache-hit reduces with increasing size of the dataset. At the same time, even lesser memory is available for kernel cache, since a larger area of memory is occupied by the dataset. Hence, we avoid a kernel completely in our proposed solution.

### B. Parallel Default SVM training Algorithm

Algorithm 2 is a parallel default variant of the sequential algorithm 1. We refer to this algorithm as *Original* for rest of the paper. This algorithm uses the entire dataset for generating the hyperplane. There are two major part of this algorithm — $\alpha$ updates (lines 3 - 8) and $\gamma$ updates (lines 9 - 17).

*1) $\alpha$ update:* To begin with, each process receives two samples — $\boldsymbol{x}_{low}$ and $\boldsymbol{x}_{up}$ — from a default process (process with rank 0) using `MPI_Bcast` primitive. Each process calculates the $\boldsymbol{\alpha}$ corresponding to $i_{up}$ and $i_{low}$ independently. This results in a overall time complexity of $O(l + m \cdot G) \cdot log(p)$ for network communication and three kernel calculations $3 \cdot \lambda$ (ignoring other integer based calculation(s)).

*2) $\gamma$ update:* We observe that $\gamma$ update is required for each sample in the dataset. Hence, this step dominates the entire computation time in the SVM algorithm. Within the *for-loop*, there are several steps: Each gradient update (line 10) requires

**Algorithm 2:** Parallel variant of Algorithm 1; $q$-th CPU perspective.

**Input**: $\mathcal{C}$, $\sigma$, $\mathcal{X}_q \in \Re^{\frac{\mathcal{N}}{p} \times d}$, $y_i \in \{+1, -1\}, \forall i$, $\boldsymbol{\alpha} \in \Re^{\frac{\mathcal{N}}{p} \times 1}$
**Result**: $\zeta$

1 $\forall$i $\in \mathcal{X}_q$, $\gamma_i \leftarrow -y_i, \alpha_i \leftarrow 0$
2 **while** $\beta_{up} + 2 \cdot \epsilon \leq \beta_{low}$ **do**
3    **if** $q == 0$ **then**
4      $\boldsymbol{x}_{i_{low}} \leftarrow \texttt{Recv}(\mathcal{X}_{q_{i_{low}}})$
5      $\boldsymbol{x}_{i_{up}} \leftarrow \texttt{Recv}(\mathcal{X}_{q_{i_{up}}})$
6    **if** $q == q_{i_{low}}$ **then**
7      $\texttt{Send}(\mathcal{X}_{q_{i_{low}}}, 0)$
8    **if** $q == q_{i_{up}}$ **then**
9      $\texttt{Send}(\mathcal{X}_{q_{i_{up}}}, 0)$
10    $\boldsymbol{x}_{i_{low}}, \boldsymbol{x}_{i_{up}} \leftarrow \texttt{Bcast(0)}$ #0
11    $\alpha_{i_{low}} \leftarrow$ using (6) , $\alpha_{i_{up}} \leftarrow$ using (6)
12    **for** $\forall i \in \mathcal{X}_q$ **do**
13      $\gamma_i \leftarrow$ using (2)
14      **if** $i == i_{up}$ *or* $i == i_{low}$ **then**
15        $\alpha_i \leftarrow \alpha_{i_{low}}$ *or* $\alpha_{i_{up}}$
16        $\varsigma_i \leftarrow$ using (4)
17      **if** $\beta_i \leq min(\gamma_j) \mid j < i$ (4) **then**
18        $\beta_{up} \leftarrow \beta_i$
19      **if** $\beta_i \geq max(\gamma_j) \mid j < i$ (4) **then**
20        $\beta_{low} \leftarrow \beta_i$
21    $\beta_{up} \leftarrow \texttt{Allreduce}(\beta_{q_{up}}, p)$
22    $\beta_{low} \leftarrow \texttt{Allreduce}(\beta_{q_{low}}, p)$

several kernel calculations. The other parts are updating the set information ($\varsigma$), however, it is only a comparison operation. In the algorithm, we save the values of local $\beta_{up}$ and $\beta_{low}$. The global values are obtained using `MPI_Allreduce` operation, which has a time complexity of $\Theta(l \cdot \log(p))$ (We ignore the bandwidth term here, since this communication only involves two scalars).

## IV. Proposed SVM Algorithm and Heuristics

In the previous section, we presented the default SVM algorithm, which uses the entire dataset for finding the maximal margin hyperplane. As we observed in Figure 1, only a small fraction of samples are actual support vectors. Specifically, we observe the following property for non-shrinkable samples:

$$\mathcal{A} = \{\, \mathbf{x}_k \mid \gamma_k < \beta_{low} \text{ or } \gamma_k > \beta_{up}, 0 < \alpha_k \,\} \quad \text{and} \quad |\mathcal{A}| \ll |\mathcal{X}| \tag{8}$$

The premise of the proposed algorithm is to eliminate non-contributing samples from consideration — at the earliest possible time. A sample can be eliminated from consideration, if its $\alpha$, $\gamma$, $I_1, I_2, I_3$ and $I_4$ (defined in (4)) and $y$ satisfy the following conditions:

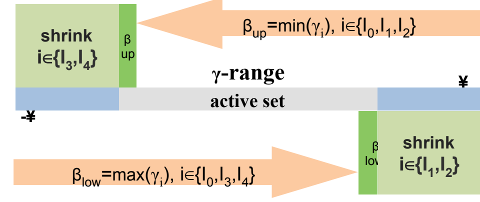


Fig. 2: Condition(s) for eliminating samples during computation. Samples with $\gamma$ between $\beta_{up}$ and $\beta_{low}$ are active, and others are *shrunk*

$$i \in \{I_3 \cup I_4\} \quad and \quad \gamma_i < \beta_{up} \quad or \quad i \in \{I_1 \cup I_2\} \quad and \quad \gamma_i > \beta_{low} \tag{9}$$

This is explained further in the Figure 2. We refer to the property of eliminating samples adaptively as *shrinking* for rest of the paper. Shrinking reduces the number of gradient calculations, which need to be executed by each process. Naturally, this leads to faster convergence in comparison to the default algorithm.

It is worthwhile observing, that an accurate condition for shrinking is unknown [15], [17]— before the final solution is achieved. Hence, the proposed conditions are heuristics, which makes it possible for support vectors to be eliminated prematurely. A possible design choice is to eliminate the sample permanently, as soon as these conditions hold true. However, the algorithm may lose accuracy — an approach recently considered by Communication-Avoiding SVM [27]. However, we consider only accurate solutions in this paper, since it is un-attractive for several science domains to lose accuracy.

There are two major design elements of our proposed algorithm: *When to shrink a sample?* and *How to maintain the accuracy of the proposed SVM algorithm?* We address each of these questions in the following sections.

### A. Shrinking Heuristics

From section II, we also observe that $\gamma$, $\alpha$ and other data structures are updated iteratively. Hence, there are several instances at which a sample may be shrunk. An early elimination of sample (an aggressive technique) can reduce the size of the working set, although it increases the probability of a false elimination. On the other hand, a conservative sample elimination reduces the improvement in time-complexity. As the iterative procedure continues, we expect the $\gamma$ and $\alpha$ of each sample to stabilize. A sample, which can be a potential support vector satisfies the following condition:

$$\grave{\mathcal{A}} = \{\, \mathbf{x}_k \mid \beta_{low} \geq \gamma_k \geq \beta_{up} \,\} \tag{10}$$

As a result, one or more samples from the set $\mathcal{X} - \grave{\mathcal{A}}$ can be eliminated without changing the current solution.

We consider two design elements in selecting a shrinking threshold. The following section shows the heuristics for

TABLE II: Heuristics. Description and classification. $\star$: Aggressive shrinking class, $\bullet$: Conservative, $\diamond$: Average

| # | Shrinking Type | $\gamma$-Recon. | Name | Class |
|---|---|---|---|---|
| 1) | None | N/A | Original | N/A |
| 2) | random: 2 | Single | Single2 | $\star$ |
| 3) | random: 500 | Single | Single500 | $\star$ |
| 4) | random: 1000 | Single | Single1000 | $\diamond$ |
| 5) | numsamples: 5% | Single | Single5pc | $\star$ |
| 6) | numsamples: 10% | Single | Single10pc | $\diamond$ |
| 7) | numsamples: 50% | Single | Single50pc | $\bullet$ |
| 8) | random: 2 | Multi | Multi2 | $\star$ |
| 9) | random: 500 | Multi | Multi500 | $\star$ |
| 10) | random: 1000 | Multi | Multi1000 | $\diamond$ |
| 11) | numsamples: 5% | Multi | Multi5pc | $\star$ |
| 12) | numsamples: 10% | Multi | Multi10pc | $\diamond$ |
| 13) | numsamples: 50% | Multi | Multi50pc | $\bullet$ |

selecting an initial shrinking threshold. This is followed by a discussion on selecting subsequent shrinking threshold.

*1) Initial Shrinking Threshold Calculation:* We design our heuristics by making an underlying assumption, that the number of support vectors is much smaller than the number of samples ($\zeta \ll \mathcal{N}$). By using this intuition, we propose heuristics for calculating initial shrinking threshold, which are based on the $\mathcal{N}$. This method is referred to as *numsamples* based approach in Table II. Using this threshold, the first iteration at which a sample can be shrunk is at $n \cdot \mathcal{N}$, where $n < 1$. A *conservative* shrinking heuristic uses a larger value of $n$. An alternative method is to assign initial shrinking threshold to a *random* value, similar to the approach proposed by Lin *et al* [15]. The default case is achieved by no-shrinking ($n = \infty$).

*2) Subsequent Shrinking Threshold Calculation:* In our proposed algorithm, we continue to shrink the samples adaptively, as we find samples which can be eliminated. Hence, it is important to calculate a subsequent shrinking threshold at which samples can be considered for shrinking again. The default approach is to use the initial shrinking threshold as the subsequent threshold.

However, we consider an alternative heuristic. We use the size of the active working set as the subsequent shrinking threshold. The intuition for this heuristic is that the size of the working set gives sufficient *opportunities* for samples to be considered at least once, such that their $\alpha$ and $\gamma$ can be stabilized. (Recall that SMO uses precisely two samples at each iteration). The new subsequent shrinking threshold is calculated at each shrinking step by using an `MPI_Allreduce`. Naturally, the intuition is that after several shrinking thresholds, the algorithm will converge to the samples which mostly constitutes the support vectors.

### B. Maintaining Accuracy of SVM

Recall, that the heuristics can prematurely eliminate the samples — especially when their $\alpha$ and $\gamma$ have not stabilized. In many cases, especially with large datasets and with high dimensions, it is possible that many samples are eliminated prematurely. Specifically, in shrinking, we assume that $\zeta \subseteq \grave{\mathcal{A}}$, however, it is possible that there are several support vectors in $\mathcal{X} - \grave{\mathcal{A}}$. Hence, we need to consider this set as well for maintaining accuracy.

When a sample is shrunk, its $\alpha$ and $\gamma$ are not updated. To maintain the accuracy, we need to synchronize these data structures. Recall that $\alpha$ is updated for the samples $\boldsymbol{x_{up}}$ and $\boldsymbol{x_{low}}$. For sequential algorithm, Joachims *et al.* have proposed to update the $\gamma$ for each sample by using the $\alpha$ from each other sample. In this paper, we propose a distributed memory algorithm for this purpose. The main steps of the algorithm are shown in in algorithm 3.

**Algorithm 3:** Gradient Reconstruction; $q$-th CPU perspective.

**Input**: $\sigma$, $\mathcal{X}_q \in \Re^{\frac{\mathcal{N}}{p} \times d}$, $y_i \in \{+1, -1\}, \forall i$

1. $\omega_q = \mathcal{X}_q - \pi_q$ // set of previously eliminated samples;
2. **for** $\forall i \in |\omega_q|$ **do**
3. $\quad \gamma_{temp} \leftarrow 0$;
4. $\quad$ **for** $\forall j \in |\mathcal{X}| \mid \alpha_j > 0$ **do**
5. $\quad\quad \gamma_{temp} += \alpha_j \cdot y_j \cdot (\Phi(\mathbf{x}_j, \mathbf{x}_i))$;
6. $\quad \gamma_i \leftarrow \gamma_{temp} - y_i$;
7. $\quad$ **if** $\beta_i \leq min(\gamma_j) \mid j < i$ (4) **then**
8. $\quad\quad \beta_{q_{up}} \leftarrow \beta_i$;
9. $\quad$ **if** $\beta_i \geq max(\gamma_j) \mid j < i$ (4) **then**
10. $\quad\quad \beta_{q_{low}} \leftarrow \beta_i$;
11. $\beta_{up} \leftarrow \texttt{Allreduce}(\beta_{q_{up}}, p)$;
12. $\beta_{low} \leftarrow \texttt{Allreduce}(\beta_{q_{low}}, p)$;

For brevity, we have abstracted the communication steps in the algorithm (line 1). In practice, this is implemented using a ring algorithm with `MPI_Isend`, `MPI_Irecv` and `MPI_Waitall` of CSR data. For each eliminated sample, we calculate the gradient using the samples received in the previous step from the neighbor (By the property of the ring algorithm, at the end of $p$ steps, we have updated the gradient using the entire dataset). With the newly calculated gradient, the iterative procedure eventually converges to the accurate solution, since the proposed algorithm ensures that each sample is considered.

*1) Computation Time Complexity Analysis of Gradient Reconstruction:* Let us consider a less-noisy dataset, such that $\zeta \ll \mathcal{N}$ and on an average, $\pi_q = \frac{\zeta}{p}$. Then, the expected time complexity of algorithm 3 is $\left|\frac{\mathcal{X}-\zeta}{p}\right| \cdot \zeta \cdot \lambda$. By differentiating this function with $\zeta$, we see that the maxima occurs at $\zeta = \frac{|\mathcal{X}|}{2}$. As a result, the expected computational complexity is $O(\frac{|\mathcal{X}|^2}{p})$ for each process. The expected time complexity for the default SMO algorithm is $\Theta(\frac{|\mathcal{X}|^3}{p})$. Hence, it is important to perform gradient reconstruction sparingly.

We consider two possibilities for gradient reconstruction. In one method, we reconstruct the gradient precisely once during the entire algorithm. The steps are shown in algorithm 4. Once the gradient is reconstructed we do not shrink samples again, and hence the final solution is precise. While this is

**Algorithm 4:** Shrinking with Single Gradient Reconstruction Algorithm; $q$-th CPU perspective.

**Input**: $\mathcal{C}$, $\sigma$, $\mathcal{X}_q \in \Re^{\frac{\mathcal{N}}{p} \times d}$, $y_i \in \{+1, -1\}, \forall i$, $\boldsymbol{\alpha} \in \Re^{\frac{\mathcal{N}}{p} \times 1}$
**Result**: $\zeta$

```
1  ∀i ∈ X_q, γ_i ← −y_i, α_i ← 0
2  δ_c ← δ
3  while β_up + 2·ε ≤ β_low do
4      if q == 0 then
5          x_{i_low} ← Recv(X_{p_{i_low}})
6          x_{i_up} ← Recv(X_{p_{i_up}})
7      x_{i_low}, x_{i_up} ← Bcast(0) #0
8      α_{i_low} ← using (6) , α_{i_up} ← using (6)
9      δ_c ← δ_c − 1
10     if δ_c = 0 then
11         δ_new ← 0 // counter for subsequent shrinking
12     for ∀i ∈ X_q do
13         γ_i ← using (2)
14         if δ_c = 0& 9 then
15             ω_q ← ω_q ∪ x_i // shrink sample
16         else
17             δ_new ← δ_new + 1
18         if i == i_up or i == i_low then
19             α_i ← α_{i_low} or α_{i_up}
20             ς_i ← using (4)
21         if β_i ≤ min(γ_j) | j < i (4) then
22             β_{q_up} ← β_i
23         if β_i ≥ max(γ_j) | j < i (4) then
24             β_{q_low} ← β_i
25     β_up ← Allreduce(β_{q_up}, p)
26     β_low ← Allreduce(β_{q_low}, p)
27     if δ_c = 0 then
28         δ ← Allreduce(δ_new, p) // Allreduce sum
29         δ_c ← δ
30     if β_up + 2·ε < β_low then
31         call Algorithm 3 to update β_up, β_low
32         δ_c ← ∞ // single gradient reconstruction, should
           not shrink samples again
```

**Algorithm 5:** Shrinking with Multiple Gradient Reconstruction Algorithm; $q$-th CPU perspective.

**Input**: $\mathcal{C}$, $\sigma$, $\mathcal{X}_q \in \Re^{\frac{\mathcal{N}}{p} \times d}$, $y_i \in \{+1, -1\}, \forall i$, $\boldsymbol{\alpha} \in \Re^{\frac{\mathcal{N}}{p} \times 1}$
**Result**: $\zeta$

```
1  ∀i ∈ X_q, γ_i ← −y_i, α_i ← 0
2  δ_c ← δ
3  while β_up + 20·ε ≤ β_low do
4      call lines 4-30 of 4
5  call 3 to update β_up, β_low
6  while β_up + 2·ε ≤ β_low do
7      call lines 4-30 of 4
8      if β_up + 2·ε < β_low then
9          call Algorithm 3 to update β_up, β_low
10     // do not update δ_c to allow as many gradient
       reconstructions as required
```

an attractive approach to minimize the overhead of gradient reconstruction, it is not able to reduce the time to convergence after the gradient reconstruction step.

The other possibility is to use multiple gradient reconstruction. This is shown in algorithm 5. We first synchronize the gradient at $20 \cdot \epsilon$, which allows us to re-introduce previously eliminated samples, when we are *close enough* to the final solution ($2 \cdot \epsilon$). The other possibility is to let the solution converge to $2 \cdot \epsilon$ and reconstruct the gradient. We choose the earlier approach, since it allows us to reconstruct gradient at an intermediate step, while not necessarily waiting to the termination condition.

*2) Communication Time Complexity Analysis of Gradient Reconstruction:* In algorithm 3, we update the $\gamma$ values for samples, which have been shrunk previously. To achieve this, each process needs $\mathcal{X} - \grave{A}$, which is obtained in a ring communication. This results in the communication of samples owned by each process. Due to the limitations of MPI collectives, we cannot use `MPI_Allgather(v)`, since the collectives require a buffer large enough to hold the entire dataset!

The time complexity of communication at each step is $l + \frac{|\mathcal{X} - \grave{A}|}{p} \cdot G \approx \frac{|\mathcal{X} - \grave{A}|}{p} \cdot G$, since dataset movement is bandwidth bound. For the entire gradient reconstruction — by using $p$ steps — the time complexity is $\Theta(|\mathcal{X} - \grave{A}| \cdot G)$

## V. Performance Evaluation

In this section, we present a performance evaluation of our proposed SVM algorithm with the no-shrinking (default) algorithm and OpenMP enhanced-`libsvm`. We use 10 datasets, using up to 2.6 million samples. We consider all heuristics shown in section III including shrinking threshold and single/multiple gradient reconstruction. We use up to 4096 processes for performance evaluation.

### A. `libsvm` Enhancements

`libsvm` is the default sequential SVM library. We enhance `libsvm` to use OpenMP, so that it can use the multi-core systems effectively. We use 3.18 version of the software. While our implementation does not use any kernel cache, we allow `libsvm` to use a compute node's entire memory as a kernel cache. This presents the best execution scenario for `libsvm`. We setup the baseline for `libsvm` by using all available cores. We intend to contribute this enhancement, such that it can help the broader community.

*1) Comparison with Other Parallel SVM Software:* We explored performance comparisons with `PSVM`, and `MLLib`, in addition to `libsvm`. `MLLib` has two problems: It does not support non-linear SVM (Gaussian kernel) — which we use for performance evaluation — and it does not use high

TABLE III: Dataset Characteristics and hyper-parameter settings

| Name | Training Set Size | Testing Set Size | C | $\sigma^2$ |
|---|---|---|---|---|
| Higgs Boson | 2600000 | N/A | 32 | 64 |
| Offending URL | 2300000 | N/A | 10 | 4 |
| Forest | 581012 | N/A | 10 | 4 |
| real-sim | 72309 | N/A | 10 | 4 |
| MNIST | 60000 | 10000 | 10 | 25 |
| cod-rna | 59535 | 271617 | 32 | 64 |
| Adult-9 (a9a) | 32561 | 16281 | 32 | 64 |
| Web (w7a) | 24692 | 25057 | 32 | 64 |

performance interconnects in the native mode (such as by using RDMA). The latest version is only able to use TCP/IP, which gives it a significant disadvantage in comparison to our proposed solution. While we were able to run PSVM, it did not scale well even on a small problem size. This can be attributed to its scalability limitations of incomplete Cholesky factorization (ICF) [12], besides its prohibitive space complexity problem.

### B. Experimental TestBed

We use the PNNL Cascade Supercomputer [28], which is equipped with Intel Sandybridge CPU and InfiniBand FDR interconnect. The performance evaluation uses up to 4096 cores (256 compute nodes). We use MVAPICH2-2.0.1 for performance evaluation. Since datasets vary in size we use between 1 and 256 compute nodes, depending on the size and properties of the dataset. As a result, the proposed algorithm can be used on multi-core machines such as a desktop, supercomputers or cloud computing systems. The datasets are downloaded from libsvm web-page [15]. Please refer to the page for datasets' description.

### C. Hyperparameter Settings

We conducted a ten-fold cross validation for selecting hyper-parameter settings for the datasets by using libsvm [15]. These are shown in Table III. The hyper-parameter $C$ is described in section III and $\sigma^2$ is the kernel width in the Gaussian kernel: $\Phi(\mathbf{x}, \mathbf{y}) = e^{-\gamma \cdot ||\mathbf{x}-\mathbf{y}||^2}$. The proposed infrastructure allows us to plugin other kernels (such as linear, polynomial). For larger datasets — where the cost of ten-cross validation is too expensive — we selected values after conducting in-depth literature study. Table II shows the heuristics which we used for performance evaluation of these datasets.

### D. Results and Analysis

In each of the following charts (Figures 4 - Figure 7) each bar shows a relative speedup to libsvm-enhanced which uses OpenMP and 16 cores. Each chart has three bars: default parallel algorithm (no shrinking), shrinking (best heuristic) and shrinking (worst heuristic). For brevity, we only present the results of the best and worst heuristics. For UCI HIGGS dataset, the time to solution is more than 2 days for libsvm-enhanced (the maximum time allowed on PNNL Cascade for a job). Hence, we only compare the execution times of Default, Shrinking (Worst) and Shrinking (Best) for UCI HIGGS dataset. Due to time constraints on our supercomputer, we use 2.6M samples out of 11M samples of the entire UCI HIGGS dataset.

*1) Large Datasets: UCI Higgs Dataset and URL Dataset:* Figure 3 shows the performance of UCI HIGGS dataset using up to 4096 cores (256 compute nodes). On 4096 cores, the proposed algorithm provides **1.56x** speedup in comparison to the Default algorithm, while on 1024 cores the speedup is **2.27x**. The Shrinking (Best) heuristic is Multi5pc (Multiple gradient synchronization with 5% samples as the initial shrinking threshold) and Shrinking (Worst) is Single50pc (Single Gradient Synchronization with 50% samples as initial shrinking threshold). The overall calculation takes 34M iterations. With increasing number of iterations, the number of active samples decreases, which increases the relative time of communication to computation. An important part of the overall computation is gradient synchronization, which keeps the accuracy of the proposed solution intact. We observed that for 4096 processes, gradient synchronization takes less than 10% of the overall time. Hence, the proposed approach is effective for large datasets, since the cost of gradient synchronization is amortized by the benefits of shrinking.

Figure 4 shows the performance of Offending URL dataset using up to 256 nodes (4096 processes). libsvm-enhanced is able to complete the training in 39 hours using 1 node and 16 threads. Using 256 nodes, we observe that we achieve $\approx$ **250x** speedup for the Shrinking (Best), while Shrinking (Worst) and Default algorithms do not perform that well. We also observe that the best shrinking heuristic is Multi5pc (multiple gradient reconstruction with 5% samples as the initial shrinking threshold). The worst shrinking heuristic is single50pc, which implies that for a large dataset such as this, we are able to amortize the cost of multiple gradient reconstruction by shrinking repeatedly.

*2) Lessons Learned:* There are a few observations from the two largest datasets used for performance evaluation: 1) Shrinking provides major speedup in comparison to both the Default algorithm and libsvm-enhanced 2) The cost of gradient synchronization is easily amortized by the benefits of reduction in working set — and hence multiple gradient synchronization based heuristics have shown promise 3) Multi5pc is an effective shrinking heuristic, which can be used for other datasets as well.

*3) Forest:* Figure 5 shows the performance of forest cover dataset, using up to 64 nodes (1024 processes). We observe that in comparison to libsvm-enhanced, the proposed Shrinking (Best) heuristic achieves **19.8x** speedup. It takes 2.07 million iterations to complete the training. However, unlike URL, the shrinking of samples is more gradual for forest cover dataset. We observed that shrinking continues to occur almost to the convergence, which provides significant benefits in comparison to the default case. After the first gradient-reconstruction ($20 \cdot \epsilon$) the training recovers the *false-positives* very quickly. Just like the URL dataset, Multi5pc is the best shrinking heuristic, and Single50pc is the worst performing

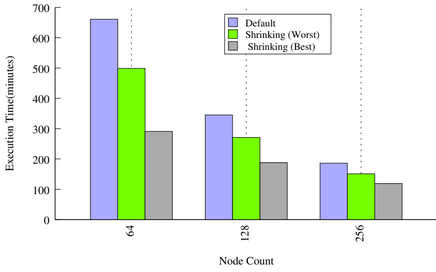


Fig. 3: UCI HIGGS Dataset Performance.

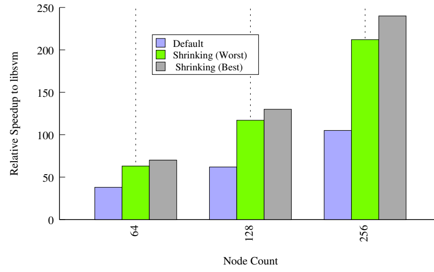


Fig. 4: URL Dataset Performance.

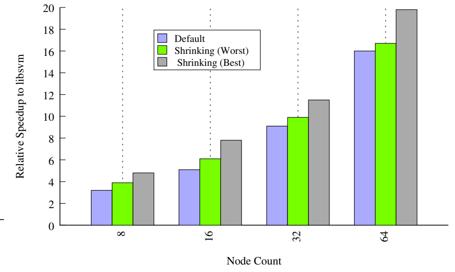


Fig. 5: Forest Dataset Performance.

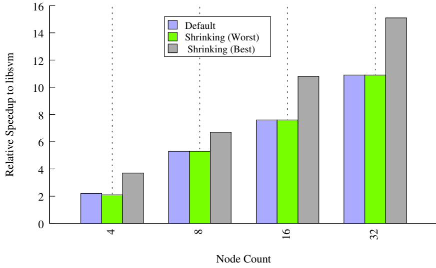


Fig. 6: MNIST Dataset Performance.

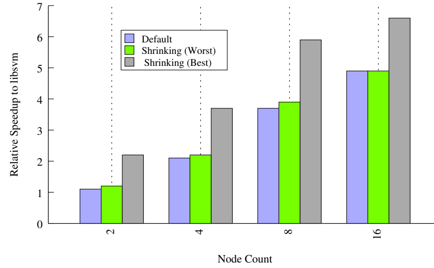


Fig. 7: Real-Sim Dataset Performance.

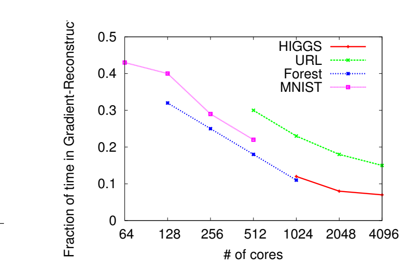


Fig. 8: Fraction of Overall Time Spent in Gradient Reconstruction with Best Heuristic (Multi5pc)

heuristic. With shrinking, since the number of active samples reduce over iterations, the overall efficiency also reduces with scale.

*4) MNIST:* Figure 6 shows the performance of MNIST dataset using up to 32 nodes (512 processes). In comparison to `libsvm`-enhanced, we observe a speedup of **15x** with Shrinking (Best) heuristic. We observed that for 75% of the iterations, the active set is a fraction of the overall number of samples (20%). This validates our premise that a small fraction of samples actually contribute to the definition of a classifier.

We also observe that the Shrinking (Worst) case performs equivalent to the Default case. The primary reason is that MNIST takes 21K iterations to converge. Since the MNIST dataset has 60K samples, the initial shrinking threshold is 30K iterations. However, the algorithm converges before the initial shrinking threshold — making the Default and Shrinking (Worst) case equivalent. The best shrinking heuristic is Multi5pc for MNIST.

*5) Real-Sim:* Figure 7 shows the performance of real-sim dataset using up to 16 nodes (256 processes). We observe a speedup of **6.6x** using 16 nodes. The iterative calculation takes 47K iterations, and the primary advantage is observed after first gradient-reconstruction, since it is able to eliminate the *false positives* completely. After the first gradient-reconstruction, we observed that less than 10% of the samples are actually active. However, this also results in reduced efficiency with scale, due to increased ratio of communication to computation. While Multi5pc is the best performing heuristic, Single50pc performs the worst. Since the dataset has $\approx$ 72K samples, the first shrinking occurs at 36K iterations. As a result, much of the benefits from shrinking are lost, since most of the samples are active for 75% of the execution time.

*6) Analysis of Gradient Reconstruction Time:* An important part of the proposed solution is gradient synchronization. Figure 8 shows the ratio of time spent in gradient synchronization to the overall time for the four large datasets considered in this paper. The expected time-complexity of SMO is $O(\frac{\mathcal{N}^3}{p})$. We also observed in section III that the time-complexity of gradient-reconstruction is $O(\frac{\mathcal{N}^2}{p})$. Hence, we expect that the ratio of gradient-reconstruction time to overall time should remain constant for a dataset.

However, we observe that the ratio decreases with the increasing scale. We observe that as the scale increases, the efficiency of the iterative calculation decreases (as presented in the previous section). We expect that on an average, each process spends $\approx \Theta(N - A) \cdot \frac{A}{p}$ time in gradient-reconstruction for $A$ number of active samples. We also observe that $A \ll N$, hence the gradient- reconstruction is primarily dominated by $\Theta(\frac{N}{p})$, which results in reduced time for gradient-reconstruction with increasing scale.

*7) Results on Smaller Size Datasets:* Table IV shows the results of proposed heuristics on smaller sized datasets. RCV1, Adult and w7a perform well in comparison to `libsvm`, however datasets like USPS and Mushrooms do not scale very well, since they only have a few thousand samples.

*8) Testing Accuracy Results:* Table V shows the accuracy of testing on the datasets used in this paper, which have testing set available. As reflected in the table, the testing accuracy of the proposed heuristics matches with the testing accuracy of `libsvm`.

TABLE IV: Relative Speedup to libsvm-sequential with smaller datasets

| Name | Default | Shrinking (Worst) | Shrinking (Best) | Proc |
|---|---|---|---|---|
| Adult-9 | 1.5 | 3.1 | 3.2 | 16 |
| RCV1 | 27 | 31 | 39 | 64 |
| USPS | 0.5 | 0.7 | 1.3 | 4 |
| Mushrooms | 0.4 | 1.09 | 1.9 | 4 |
| Web (w7a) | 1.7 | 2.4 | 3.1 | 16 |

TABLE V: Testing Accuracy

| Name | Test Acc. - Ours(%) | Test Acc.-libsvm(%) |
|---|---|---|
| Adult-9 | 85.18 | 83.12 |
| USPS | 97.6 | 97.75 |
| MNIST | 98.9 | 98.62 |
| Cod-RNA | 92.33 | 92.1 |
| Web(w7a) | 98.82 | 98.9 |

## VI. Related Work

With the advent of multi-core systems and cluster computing, several parallel and distributed algorithms have been proposed in literature. This section provides a brief overview of these algorithms: Architecture specific solutions using GPUs [9], [10] have been proposed, and other approaches which may require a special cluster setup have been proposed as well [29]. Graf *et al.* have proposed Cascade SVM [29], which provides a parallel solution to the dual optimization problem. In Cascade SVM, the basic idea is to divide the original problem into completely independent sub-problems, and recursively combine the independent solutions to obtain the final set of support vectors. However, Cascade SVM suffers from load imbalance, since many processes finish their individual sub-problem before others. As a result, this approach does not scale well for very large scale of processes. We address this limitation by providing a shrinking based solution to the problem. The advent of SIMD architectures such as GPUs has resulted in research conducted for Support Vector Machines on GPUs [9]. Under their approach, a thread is created for each data point in the training set and the MapReduce paradigm is used for compute-intensive steps. The primary approach proposed in this paper is suitable for large scale systems, and not restricted to GPUs. Similarly, MIC-SVM [11] is a single node multi-core solution for Intel Xeon Phi, which provides excellent speedup, but with restrictions to a single-node. Yu *et al.* considered a small discussion on shrinking (with no proposed algorithm/implementation), and concluded that the engineering difficulty of shrinking is tedious. However, in this paper, we demonstrate the efficacy of shrinking on several large datasets by designing scalable SVM using smart heuristics for shrinking and gradient-reconstruction.

Several other researchers have proposed alternative mechanisms for solving QP problems. An example of variable projection method is proposed by Zanghirati and Zanni [30]. They use an iterative solver for QP problems leveraging the decomposition strategy of SVM$^{light}$ [17]. Chang *et al.* [12] have also considered a working-set size $> 2$ and solve the problem using Incomplete Cholesky Factorization and Interior Point method (IPM). Woodsend *et al.* [31] have proposed parallelization of linear SVM using IPM and a combination of MPI and OpenMP. However, their approach is not a working-set method, as it does not decompose a large problem into smaller ones. A more recent work [32] exploits modern multi-core architectures and intelligent caching to improve the speed of training linear SVMs.

## VII. Acknowledgement

The research described in this paper is part of the Analysis in Motion Initiative at Pacific Northwest National Laboratory. It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy.

## VIII. Conclusions

In this paper, we have undertaken the challenge in designing a scalable Support Vector Machine algorithm on distributed memory systems. We have presented the case that in many classification datasets, only a small fraction of samples contribute to the definition of the classifier. These samples are also known as *support vectors*. Our novel algorithm adaptively eliminates (*shrinks*) the samples which are *unlikely* to contribute to the classifier definition. We have also observed that eliminating a sample prematurely can lead to an inaccurate solution. For maintaining the accuracy, we have proposed a method to synchronize the data structures (gradient synchronization) at regular intervals by using detailed time-space complexity analysis. We have considered a broad set of heuristics for shrinking and gradient synchronization.

We have implemented the proposed algorithm using MPI and evaluated it with ten datasets using up to 4096 cores. We have enhanced libsvm — *de facto* sequential SVM software — to use OpenMP for exploiting multi-core parallelism. This provides a fairer comparison of our approach with libsvm. Our performance evaluation includes UCI HIGGS Boson dataset and Offending URL dataset, among a total of 10 datasets. On a 2.3M samples dataset, our proposed algorithm with shrinking takes 8 minutes for training, while the default non-shrinking algorithm takes 13 minutes. The enhanced libsvm takes 39 hours on a 16-core sandybridge machine. We plan to release our code with Machine Learning Toolkit for Extreme Scale (MaTEx) [16]. We are planning to contribute the OpenMP enhanced version to libsvm.

The implications of the proposed novel algorithms are such that even larger datasets than considered in this paper can now be used for classification and regression, without any accuracy loss. We expect that the proposed algorithms will be useful for several other science domains not considered in this paper.

## References

[1] Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, “Scientific Discovery at the Exascale,” 2011.

[2] DOE ASCAC Subcommittee, “Synergistic Challenges in Data-Intensive Science and Exascale Computing,” 2013.

[3] A. L. Tarca, V. J. Carey, X.-w. Chen, R. Romero, and S. Drghici, "Machine learning and its applications to biology," *PLoS Comput Biol*, vol. 3, no. 6, p. e116, 06 2007.

[4] A. Vossen, "Support vector machines in high-energy physics," 2008.

[5] P. Balaprakash, Y. Alexeev, S. A. Mickelson, S. Leyffer, R. L. Jacob, and A. P. Craig, "Machine learning based load-balancing for the cesm climate modeling package," 2013.

[6] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10115-007-0114-2

[7] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: Primal estimated sub-gradient solver for svm," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 807–814. [Online]. Available: http://doi.acm.org/10.1145/1273496.1273598

[8] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear svm," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 408–415. [Online]. Available: http://doi.acm.org/10.1145/1390156.1390208

[9] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine Learning*, ser. ICML '08. ACM, 2008, pp. 104–111.

[10] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored approach for training kernelized SVMs," in *Proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining*, ser. KDD '11, 2011, pp. 805–813.

[11] Y. You, S. Song, H. Fu, A. Marquez, M. Dehnavi, K. Barker, K. Cameron, A. Randles, and G. Yang, "Mic-svm: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 809–818.

[12] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "Psvm: Parallelizing support vector machines on distributed computers," in *NIPS*, 2007, software available at http://code.google.com/p/psvm.

[13] MLlib, "Machine Learning Library."

[14] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in kernel methods: support vector learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 185–208.

[15] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[16] Machine Learning Toolkit for Extreme Scale, "MaTEx," http://hpc.pnl.gov/matex.

[17] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in kernel methods*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. MIT Press, 1999, pp. 169–184.

[18] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K.R.K.Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural Computation*, vol. 13, no. 3, pp. 637–649, 2001.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[20] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135.

[21] A. Vishnu and M. Krishnan, "Efficient On-demand Connection Management Protocols with PGAS Models over InfiniBand," in *CCGrid*, 2010.

[22] A. Vishnu and D. K. P. M. K. Krishnan, "A Hardware-Software Approach to Network Fault Tolerance wwith InfiniBand Cluster," in *International Conference on Cluster Computing*, 2009, pp. 479–486.

[23] A. Vishnu, B. Benton, and D. K. Panda, "High Performance MPI on IBM 12x InfiniBand Architecture," in *International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS '07 (HIPS'07)*, 2007.

[24] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *SMTPS*, March 2007.

[25] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu, "Noncollective communicator creation in mpi," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 282–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042508

[26] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman, "Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems," *Comput. Sci.*, vol. 26, no. 3-4, pp. 247–256, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0168-y

[27] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc , "Ca-svm: Communication-avoiding support vector machines on clusters," in *IPDPS*, 2015.

[28] Cascade Supeercomputer, "Pacific Northwest National Laboratory," http://www.emsl.pnl.gov/emslweb/instruments/cascade-supercomputer.

[29] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *Advances in Neural Information Processing Systems*. MIT Press, 2005, pp. 521–528.

[30] G. Zanghirati and L. Zanni, "A parallel solver for large quadratic programs in training support vector machines," *Parallel Computing*, vol. 29, no. 4, pp. 535–551, 2003.

[31] K. Woodsend and J. Gondzio, "Hybrid mpi/openmp parallel linear support vector machine training," *J. Mach. Learn. Res.*, vol. 10, pp. 1937–1953, Dec. 2009.

[32] S. Matsushima, S. Vishwanathan, and A. J. Smola, "Linear support vector machines via dual cached loops," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 177–185. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339559