# CA-SVM: Communication-Avoiding Parallel Support Vector Machines on Distributed Systems

*Yang You*
*James Demmel*
*Kenneth Czechowski*
*Le Song*
*Richard Vuduc*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-9
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-9.html

February 27, 2015

# CA-SVM: Communication-Avoiding Support Vector Machines on Distributed Systems

Yang You[‡†], James Demmel[†], Kenneth Czechowski[*], Le Song[*], Richard Vuduc[*]
[‡]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[†]*Computer Science Division, University of California at Berkeley, Berkeley, CA, USA*
[*]*College of Computing, Georgia Institute of Technology, Atlanta, GA, USA*
*you-y12@mails.tsinghua.edu.cn {youyang, demmel}@eecs.berkeley.edu {kentcz, lsong, richie}@gatech.edu*

***Abstract*—We consider the problem of how to design and implement communication-efficient versions of parallel support vector machines, a widely used classifier in statistical machine learning, for distributed memory clusters and supercomputers. The main computational bottleneck is the training phase, in which a statistical model is built from an input data set. Prior to our study, the parallel isoefficiency of a state-of-the-art implementation scaled as $W = \Omega(P^3)$, where $W$ is the problem size and $P$ the number of processors; this scaling is worse than even a one-dimensional block row dense matrix vector multiplication, which has $W = \Omega(P^2)$.**

**This study considers a series of algorithmic refinements, leading ultimately to a Communication-Avoiding SVM (CA-SVM) method that improves the isoefficiency to nearly $W = \Omega(P)$. We evaluate these methods on 96 to 1536 processors, and show average speedups of $3 - 16\times$ ($7\times$ on average) over Dis-SMO, and a 95% weak-scaling efficiency on six real-world datasets, with only modest losses in overall classification accuracy. The source code can be downloaded at [1].**

***Keywords*-distributed memory algorithms; communication-avoidance; statistical machine learning**

## I. Introduction

This paper concerns the development of communication-efficient algorithms and implementations of kernel support vector machines (SVMs). The kernel SVM is a state-of-the-art algorithm for statistical nonlinear classification problems [2], with numerous practical applications [3]–[5]. However, the method's training phase greatly limits its scalability on large-scale systems. For instance, the most popular kernel SVM training algorithm, Sequential Minimal Optimization (SMO), has very little locality and low arithmetic intensity; we have observed that it might spend as much as 70% of its execution time on network communication on modern high-performance computing (HPC) systems [6].

Intuitively, there are two reasons for SMO's poor scaling behavior [7]. The first reason is that the innermost loop is like a large dense matrix-vector multiply, whose parallel isoefficiency function scales like $W = \Omega(P^2)$. The second reason is that SMO is an iterative algorithm, where the number of iterations scales with the problem size. When combined, these two reasons result in an isoefficiency of $W = \Omega(P^3)$, meaning the method can only effectively use $\sqrt[3]{W}$ processors (refer to Section 5.4.2 of [8] for $W$ and $P$).

In this paper, we first evaluate distributed memory implementations of three state-of-the-art SVM training algorithms: SMO [9], Cascade SVM [10], and Divide-and-Conquer SVM (DC-SVM) [11]. (Our implementations of the latter two are the first-of-their-kind for distributed memory systems, as far as we know.) We then optimize these methods through a series of techniques including: (1) developing a Divide-and-Conquer Filter (DC-Filter) method, which combines Cascade SVM with DC-SVM to balance accuracy and performance; (2) designing a Clustering-Partition SVM (CP-SVM) to improve the parallelism, reduce the communication, and improve accuracy relative to DC-Filter; and (3) designing a novel Communication-Avoiding SVM (CA-SVM) that achieves load-balance and removes nearly all inter-node communication. The relationship among these methods, including how they combine different techniques, is summarized in Fig. 1. Overall, we claim the following specific contributions:

(1) We convert a communication-intensive algorithm to an embarrassingly-parallel algorithm through removing nearly all the inter-node communications. The new algorithm, CA-SVM, is highly parallel and scalable.

(2) CA-SVM achieves significant speedups over the original algorithm with only small losses in accuracy on our test sets. In this way, we manage to balance the speedup and accuracy.

(3) We optimize the state-of-the-art training algorithms step-by-step, which both points out the problems of the existing approaches and suggests possible solutions.

In short, CA-SVM achieves 3-16$\times$ ($7\times$ on average) speedups over distributed SMO algorithm with comparable accuracies. The accuracy losses range from none to 3.6% (1.3% on average). According to previous work by others, such accuracy losses may be regarded as small and are likely to be tolerable in practical applications. CA-SVM also achieves 95.3% weak scaling efficiency when we increase the number of processors from 96 to 1536 on NERSC's Edison system [6]. We believe the approaches in this paper could be applied to other statistical learning methods, such as neural networks and regression analysis.

## II. Background and Related Work

SVMs have two major phases: training and prediction. The training phase builds the model from a labeled input data set, which the prediction phase uses to classify new

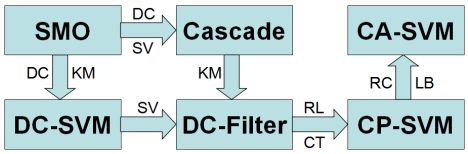



Figure 1. General Flow of the algorithmic improvements in this paper. DC: divide-and-conquer; SV: only pass support vectors layer-by-layer; KM: use K-means to partition the dataset; RL: remove the lower layers; CT: build data center on each node and cluster the test dataset; RC: remove communication; LB: load balance.

data. The training phase is the main limiter to scaling, both with respect to increasing the training set size and increasing the number of processors. By contrast, prediction is embarrassingly parallel and fairly "cheap" per data point. Therefore, this paper focuses on training, just like prior papers on SVM-acceleration [9], [10], [12].

In terms of potential training algorithms, there are many options. In this paper, we focus on a class of algorithms we will call *partitioned SMO algorithms*. These algorithms work essentially by partitioning the data set, building kernel SVM models for each partition using SMO as a building block, and then combining the models to derive a single final model. In addition, they estimate model parameters using iterative methods. We focus on two exemplars of this class, *Cascade SVM* (§ II-C) and *Divide-and-Conquer SVM* (§ II-D). We briefly survey alternative methods in § II-E. Our primary reason for excluding them in this study is that they use very different approaches that are both complex to reproduce and that do not permit the same kind of head-to-head comparisons as we wish to consider here.

### A. SVM Training and Prediction

We focus on two-class (binary-class) kernel SVMs, where each data point has a binary label that we wish to predict. Multi-class (3 or more classes) SVMs may be implemented as several independent binary-class SVMs; a multi-class SVM can be easily processed in parallel once its constituent binary-class SVMs are available. The training data in an SVM consists of $m$ samples, where each sample is a pair $(X_i, y_i)$ and $i \in \{1, 2, ..., m\}$. Each $X_i$ is the $i$-th training sample, represented as a vector of features. Each $y_i$ is the $i$-th sample's label; in the binary case, each $y_i$ has one of two possible values, $\{-1, 1\}$. Mathematically, the kernel SVM training is typically carried out in its dual formulation where a set of coefficients $\alpha_i$ (called Lagrange multipliers), with each $\alpha_i$ associated with a sample $(X_i, y_i)$, are found by solving the following linearly-constrained convex Quadratic Programming (QP) problem, eqns. (1–2):

$$\text{Maximize: } F(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K_{i,j} \quad (1)$$

Table I
STANDARD KERNEL FUNCTIONS

| | |
|---|---|
| Linear | $K(X_i, X_j) = {X_i}^\top X_j$ |
| Polynomial | $K(X_i, X_j) = (a{X_i}^\top X_j + r)^d$ |
| Gaussian | $K(X_i, X_j) = \exp(-\gamma \|\|X_i - X_j\|\|^2)$ |
| Sigmoid | $K(X_i, X_j) = \tanh(a{X_i}^\top X_j + r)$ |

$$\text{Subject to: } \sum_{i=1}^{m} \alpha_i y_i = 0 \text{ and } 0 \le \alpha_i \le C, \forall i \in \{1, 2, ..., m\}. \quad (2)$$

Here, $C$ is a regularization constant that attempts to balance generality and accuracy; each computed $\alpha_i$ is a Lagrange multiplier; and $K_{i,j}$ denotes the value of a kernel function evaluated at a pair of samples, $X_i$ and $X_j$. (Typical kernels appear in Table I.) The value $C$ is chosen by the user.

The training produces the vector of Lagrange multipliers, $[\alpha_1, \alpha_2, ..., \alpha_m]$. The predicted label for a new sample, $\hat{X}$, is computed by evaluating eqn. (3),

$$\hat{y} = \sum_{i=1}^{m} \alpha_i y_i K(\hat{X}, X_i) \quad (3)$$

In effect, eqn. (3) is the model learned during training. One goal of SVM training is to produce a compact model, that is, one whose $\alpha$ coefficients are *sparse* or mostly zero. The set of samples with non-zero $\alpha_i$ are called the *support vectors*. Observe that only the samples with non-zero Lagrange multipliers ($\alpha_i \neq 0$) can have an effect on the prediction result.

### B. Sequential Minimal Optimization (SMO)

The most widely used kernel SVM training algorithm is Platt's *Sequential Minimal Optimization* (*SMO*) algorithm [9]. It is the basis for popular SVM libraries and tools, including LIBSVM [13] and GPUSVM [14]. The overall structure of the SMO algorithm appears in Alg. 1. In essence, it iteratively evaluates the following formulae:

$$f_i = \sum_{j=1}^{m} \alpha_j y_j K(X_i, X_j) - y_i \quad (4)$$

$$\hat{f}_i = f_i + \Delta\alpha_{high} y_{high} K_{high,i} + \Delta\alpha_{low} y_{low} K_{low,i} \quad (5)$$

$$\Delta\alpha_{low} = \frac{y_{low}(b_{high} - b_{low})}{K_{high,high} + K_{low,low} - 2K_{high,low}} \quad (6)$$

$$\Delta\alpha_{high} = -y_{low} y_{high} \Delta\alpha_{low} \quad (7)$$

For a detailed performance bottleneck analysis of SMO, see You et al. [15]. The most salient observations we can make are that (a) the dominant update rule is eqn. (4), which is a matrix-vector multiply (with kernel); and (b) the number of iterations necessary for convergence will tend to scale with the number of input points, $m$.

All of the algorithmic improvements in this paper start essentially from SMO. In particular, we adopt the approach

of Cao et al. [12], who designed a parallel SMO implementation for distributed memory systems. As far as we know, it is the best distributed SMO implementation so far. The basic idea is to partition the data among nodes and launch a big distributed SVM across those nodes. Their implementation fits within a map-reduce framework. The two-level ("local" and "global") map-reduce strategy of Catanzaro et al. can significantly reduce the amount of communication [14]. However, the basic algorithm and ideas of Catanzaro et al target single-node (single-GPU) systems, whereas our focus in this paper is on distributed memory scaling.

**Algorithm 1:** Sequential Minimal Optimization (SMO)

1 Input the samples $X_i$ and labels $y_i$, $\forall i \in \{1, 2, ..., m\}$.
2 $\alpha_i = 0$, $f_i = -y_i$, $\forall i \in \{1, 2, ..., m\}$.
3 $b_{high} = -1$, $i_{high} = \min\{i : y_i = 1\}$
4 $b_{low} = 1$, $i_{low} = \min\{i : y_i = -1\}$.
5 Update $\alpha_{high}$ and $\alpha_{low}$ according to Equations (6) and (7).
6 Update $f_i$ according to Equation (5), $\forall i \in \{1, 2, ..., m\}$
7 $I_{high} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = 0 \vee y_i < 0, \alpha_i = C\}$
8 $I_{low} = \{i : 0 < \alpha_i < C \vee y_i > 0, \alpha_i = C \vee y_i < 0, \alpha_i = 0\}$
9 $i_{high} = \arg\min\{f_i : i \in I_{high}\}$
10 $i_{low} = \arg\max\{f_i : i \in I_{low}\}$
11 $b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\}$
12 Update $\alpha_{high}$ and $\alpha_{low}$ according to Equations (6) and (7).
13 If $b_{low} > b_{high}$, then go to Step 6.

### C. Cascade SVM

Cascade SVM is a multi-layer approach designed with distributed systems in mind [10]. As Fig. 2 illustrates, its basic idea is to divide the SVM problem into $P$ smaller SVM sub-problems, and then use a kind of "reduction tree" to re-combine these smaller SVM models into a single result. The subproblems and combining steps could in principle use any SVM training method, though in this paper we consider those that use SMO. A Cascade SVM system with $P$ computing nodes has $\log(P)+1$ layers. In the same way, the whole training dataset ($TD$) is divided into $P$ smaller parts ($TD_1, TD_2, ..., TD_P$), each of which is processed by one sub-SVM. The training process selects certain samples (with non-zero Lagrange multiplier, i.e. $\alpha_i$) out of all the samples. The set of support vectors, $SV$, is a subset of the training dataset ($SV_i \subseteq TD_i, i \in \{1, 2, ..., P\}$). Each sub-SVM can generate its own $SV$. For Cascade, only the $SV$ will be passed from the current layer to next layer. The $\alpha_i$ of each support vector will also be passed to the next layer to provide a good initialization for the next layer, which can significantly reduce the iterations for convergence. On the next layer, any two consecutive SV sets ($SV_i$ and $SV_{i+1}$) will be combined into a new sub-training dataset. In this way, there is only one sub-SVM on the $(\log(P) + 1)$-st layer.

### D. Divide-and-Conquer SVM (DC-SVM)

DC-SVM is similar to Cascade SVM [11]. However, it differs in two ways: (1) Cascade SVM partitions the training dataset evenly on the first layer, while DC-SVM uses K-means clustering to partition the dataset; and (2) Cascade SVM only passes the set of support vectors from one layer to the next, whereas DC-SVM passes *all* of the training dataset from layer to layer. At the last layer of DC-SVM, a single SVM operates on the whole training dataset.

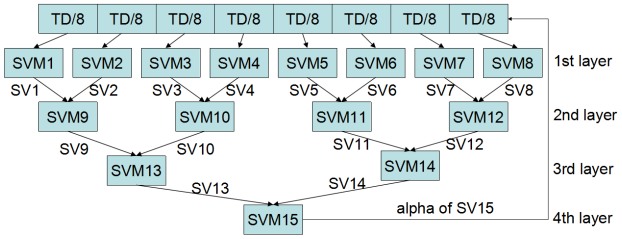



Figure 2. This figure is an illustration of Cascade SVM [10]. Different layers have to be processed sequentially, i.e. layer $i+1$ can be processed after layer $i$ has been finished. The tasks in the same level can be processed concurrently. If the result at the bottom layer is not good enough, the user can distribute all the support vectors (SV15 in the figure) to all the nodes and re-do the whole pass from the top layer and to the bottom layer. However, for most applications, the result will not become better after another Cascade pass. Thus, one pass is enough in most of the cases.

*K-means clustering:* Since K-means clustering is a critical substep for DC-SVM, we review it here.

The objective of K-means clustering is to partition a dataset $TD$ into $k \in Z^+$ sub-datasets ($TD_1, TD_2, ..., TD_k$), using a notion of proximity based on Euclidean distance [16]. The value of $k$ is chosen by the user. Each sub-dataset has a center ($CT_1, CT_2, ..., CT_k$). The center has the same structure as a sample (i.e. $n$-dimensional vector). Sample $X$ will belong to $TD_i$ if $CT_i$ is the closest data center to $X$. A naïve version of K-means clustering appears in Alg. 2.

**Algorithm 2:** Naïve K-means Clustering

1 Input the training samples $X_i$, $i \in \{1, 2, ..., m\}$.
2 Initialize data center $CT_1, CT_2, ..., CT_k$ randomly.
3 set $\delta = 0$
4 For every $i$, set $c^i = argmin_j ||X_i - CT_j||$.
5 If $c^i$ has been changed, $\delta = \delta + 1$
6 For every $j$, set $CT_j = \frac{\sum_{i=1}^{m} 1\{c^i=j\} X_i}{\sum_{i=1}^{m} 1\{c^i=j\}}$, $j \in \{1, 2, ..., k\}$.
7 If $\delta/m >$ threshold, then go to Step 3.

### E. Other methods

There are other potential algorithms for SVMs. One method uses matrix factorization of the kernel matrix $K$ [17]. Another class of methods relies on solving the QP problem using an iteration structure that considers more than two points at a time [18], [19]. Additionally, there are other optimizations for serial approach [9], [20], [21] or parallel approach on shared memory systems [14], [22]. All of these approaches are hard to compare "head-to-head" against the

Table II
TERMS FOR PERFORMANCE MODELLING

| | |
|---|---|
| $m$; $n$; $P$ | # samples; # features per sample; # nodes or processes |
| $T_1$; $T_p$ | serial run time; parallel run time |
| $t_s$; $t_w$ | startup time for communication; per-word transfer time |
| $V_k$ | # SVs in $k_{th}$ Cascade layer, $V_1 = m$ |
| $L_k$ | maximal # iters of all nodes in $k_{th}$ Cascade layer |
| $P_k$ | # processes in $k_{th}$ Cascade layer |
| $W$; $T_o$ | problem size; parallel overhead ($T_o = PT_p - W$) |
| $s$; $I$; $k$ | # SVs; # SVM iters; # K-means iters |

partitioned SMO schemes this paper considers, so we leave such comparisons for future work.

## III. RE-DESIGN DIVIDE-AND-CONQUER METHOD

### A. Performance Modeling for Existing Methods

In this section, we will do performance modeling for the three related methods mentioned in Section II. The related terms are in Table II and the proofs can be found in [7]. To evaluate the scalability, we refer to Iso-efficiency function (Section 5.4.2 of [8]), shown in Equation (8) where $E$ ($E = T_1/(pT_p)$) is the desired scaling efficiency (Specifically, $T_1 = t_c W$ where $t_c$ is the time per flop. In this paper, to make it simple, we normalize so that $t_c = 1$. In the same way, $t_s$ and $t_w$ in Table II actually are ratios of communication time to flop time). The minimum problem size $W$ can usually be obtained as a function of $P$ by algebraic manipulations. This function dictates the growth rate of $W$ required to keep the efficiency fixed as $P$ increases. For example, the Iso-efficiency function of 1D Mat-Vec-Mul is $W = \Omega(P^2)$, and it is $W = \Omega(P)$ for 2D Mat-Vec-Mul (Section 8.1 of [8], $W = n^2$ where $n$ is the matrix dimension for Mat-Vec-Mul). The Mat-Vec-Mul is more scalable with 2-D partitioning because it can deliver the same efficiency on more processors with 2-D partitioning ($P = O(W)$) than with 1-D partitioning ($P = O(\sqrt{W})$).

$$W = KT_o \;\; with \;\; K = \frac{E}{1-E} \tag{8}$$

*1) Distributed SMO (Dis-SMO):* The serial runtime ($T_1$) of a SMO iteration is $\Theta(2mn)$ and its parallel runtime ($T_p$) per iteration is in Equation (9). Its problem size ($W$) is also $\Theta(mn)$. Based on the terms in Table II, the parallel overhead ($T_o$) can be obtained in Equation (10). The scaling modeling results are in Table IV. This modeling result is based on single-iteration SMO. However, the modeling result of the completely converged SMO algorithm will be worse (i.e. the lower bound will be larger) because the number of iterations is proportional to the number of samples (Table III). This will furthermore jeopardize the scalability for large-scale computation.

Table III
THE NUMBER OF ITERATIONS WITH DIFFERENT NUMBER OF SAMPLES, EPSILON AND FOREST ARE THE TEST DATASETS

| **Samples** | 10k | 20k | 40k | 80k | 160k | 320k |
|---|---|---|---|---|---|---|
| **Iters (epsilon)** | 4682 | 8488 | 15065 | 26598 | 49048 | 90320 |
| **Iters (forest)** | 3057 | 6172 | 11495 | 22001 | 47892 | 103404 |

Table IV
SCALING COMPARISON FOR ISO-EFFICIENCY FUNCTION

| Method | Communication | Computation |
|---|---|---|
| 1D Mat-Vec-Mul | $W = \Omega(P^2)$ | $W = \Theta(1)$ |
| 2D Mat-Vec-Mul | $W = \Omega(P)$ | $W = \Theta(1)$ |
| Distributed-SMO | $W = \Omega(P^3)$ | $W = \Omega(P^2)$ |
| Cascade | $W = \Omega(P^3)$ | $W = O(\sum_{k=1}^{logP} nL_k V_{k-1} 2^k)$ |
| DC-SVM | $W = \Omega(P^3)$ | $W = O(\sum_{k=1}^{logP} nL_k m 2^k)$ |

$$T_p = 14logPt_s + [2nlogP + 4P^2]t_w + \frac{2mn+4m}{P} + 2P + n \tag{9}$$

$$T_o = 14PlogPt_s + [2nPlogP + 4P^3]t_w + 4m + 2P^2 + nP \tag{10}$$

*2) Cascade and DC-SVM:* The communication and computation Iso-efficiency functions of Cascade are in Equation (11) and Equation (12) respectively. Since $V_{1+logP}$ is the number of support vectors of the whole system, we can get that $V_{1+logP} = \Theta(m)$. On the other hand, the number of training samples can not be less than the number of nodes (i.e. $m = \Omega(P)$), because we can not keep all $P$ nodes busy. That is $V_{1+logP} = \Omega(P)$. Therefore, after substituting $V_{1+logP}$ by $\Omega(P)$ in Equation (11), we obtain that the lower bound of communication Iso-efficiency function $W = \Omega(P^3)$. Because we can not predict the number of support vectors and the number of iterations on each level (i.e. $V_{k-1}$ and $L_k$ in Equation (12)) beforehand, we can only get the upper bound for the computation Iso-efficiency function (Table IV). For DC-SVM, since the K-means time is significantly less than the SVM time (Tables XIII to XVIII), we ignore the effect of K-means on the whole system performance. Therefore, we get the Iso-efficiency function of DC-SVM by replacing $V_k$ of Cascade with $m$ (Table IV).

$$W_{cascade,comm} = \Theta((\sum_{k=2}^{logP} n2^k V_k) + P^2 V_{1+logP}) \tag{11}$$

$$W_{cascade,comp} = \Theta(n(\sum_{k=2}^{1+logP} L_k V_{k-1} 2^k - 2Im)) \tag{12}$$

We compare with Mat-Vec-Mul, which is a typical communication-intensive kernel. Actually, the scalability of

these three methods are even worse than 1D Mat-Vec-Mul, which means we need to design a new algorithm to scale up SVM on future exascale computing systems. Our scaling results in Section V are in line with our analysis.

### B. DC-Filter: Combination of Cascade and SVM

From our experimental results, we observe that Cascade is faster than Dis-SMO. However, the classification accuracy of Cascade is worse. DC-SVM can obtain a higher classification accuracy. Nevertheless, the algorithm becomes extremely slow (Tables XIII to XVIII). The reason is that DC-SVM has to pass all the samples layer-by-layer, and this significantly increases the communication overhead. In addition, more data on each node means the processors have to do more on-chip communication and computation. Therefore, our first design is to combine Cascade with DC-SVM. We refer to this approach as Divide-and-Conquer Filter (DC-Filter).

Like DC-SVM, we apply K-means in DC-Filter to get a better data partition, which can help to get a good classification accuracy [11]. It is worth noting that K-means does not significantly increase the computation and communication overhead (Tables XIII to XVIII), which is the major reason why we can use it. For example, K-means converges in 7 loops and only costs less than 0.1% of the total runtime for processing the ijcnn dataset. On the other hand, we apply the filter function of Cascade in the combined approach. On each layer, only the support vectors rather than all the training samples will be sent to next layer, which is like a filter since SV is a subset of the original training dataset. The Lagrange multiplier of each support vector will be sent with it to give a good initialization for next layer, which can reduce the number of iterations for convergence [10]. In our experiments, the speed and accuracy of DC- Filter fall in between Cascade and DC-SVM, or perform better than both of them. DC-Filter is a compromise between these two existing approaches, which is our first attempt to balance the accuracy and the speedup.

## IV. Communication-Avoiding Design

### A. CP-SVM: Clustering-Partition SVM

The node management for Cascade, DC-SVM, and DC-Filter are actually similar to each other (i.e. Fig. 2). Table V provides the detailed profiling result of a toy Cascade example to show how they work. We can observe that only 27% (5.49/20.1) of the total time is spent on the top layer, which makes full use of all the nodes. In fact, almost half (9.69/20.1) of the total time is spent on the bottom layer, which only uses one node. In this situation, the Cascade-like approach does not perform well because the parallelism in most of the algorithm is extremely low. The weighted average number of nodes used is only 3.3 (obtained by Equation (13)) for the example in Table V. However, the system actually occupies 8 nodes for the whole runtime. Specifically, the parallelism is decreasing by a factor of 2 layer-by-layer. For some datasets (e.g. Table XIV), the lower level can be fast and converge within $\Theta(1)$ iterations. For other datasets (e.g. Table V), the lower level is extremely slow and becomes the bottleneck of the runtime performance. Therefore, we need to redesign the algorithm again to make it highly parallel and make full of all the computing nodes.

$$\frac{\sum_{l=1}^{1+logP}((time\_of\_layer\_l) \times (\#nodes\_of\_layer\_l))}{\sum_{l=1}^{1+logP}(time\_of\_layer\_l)} \tag{13}$$

Table V
Profile of 8-node & 4-layer Cascade for a Toy Dataset

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **level $1^{st}$** | | | | | | | | |
| **node rank** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **samples** | 6000 | 6000 | 6000 | 6000 | 6000 | 6000 | 6000 | 6000 |
| **time: 5.49s** | 4.87 | 4.92 | 4.90 | 4.68 | 5.12 | 5.10 | 5.49 | 4.71 |
| **iter: 6168** | 5648 | 5712 | 5666 | 5415 | 5936 | 5904 | 6168 | 5453 |
| **SVs: 5532** | 746 | 715 | 717 | 718 | 686 | 707 | 721 | 699 |
| **level $2^{nd}$** | | | | | | | | |
| **node rank** | 1 | | 3 | | 5 | | 7 | |
| **samples** | 1461 | | 1435 | | 1393 | | 1420 | |
| **time: 1.58s** | 1.58 | | 1.50 | | 1.35 | | 1.45 | |
| **iter: 7485** | 7485 | | 7211 | | 6713 | | 7035 | |
| **SVs: 5050** | 1292 | | 1263 | | 1256 | | 1239 | |
| **level $3^{rd}$** | | | | | | | | |
| **node rank** | 1 | | | | 5 | | | |
| **samples** | 2555 | | | | 2495 | | | |
| **time: 3.34s** | 3.34 | | | | 3.30 | | | |
| **iter: 9081** | 8975 | | | | 9081 | | | |
| **SVs: 4699** | 2388 | | | | 2311 | | | |
| **level $4^{th}$** | | | | | | | | |
| **node rank** | 1 | | | | | | | |
| **samples** | 4699 | | | | | | | |
| **time: 9.69s** | 9.69 | | | | | | | |
| **iter: 14052** | 14052 | | | | | | | |
| **SVs: 4475** | 4475 | | | | | | | |

The analysis in this section is based on the Gaussian kernel with $\gamma > 0$ because it is the most widely used case [14]. Other cases can work in the same way with different implementations. For any two training samples, their kernel function value is close to zero ($exp\{-\gamma||X_i - X_j||^2\} \to 0$) when they are far away from each other in Euclidean distance ($||X_i - X_j||^2 \to \infty$). Therefore, for a given sample $\hat{X}$, only the support vectors close to $\hat{X}$ can have an effect on the prediction result (Equation (3)) in the classification process. Based on this idea, we can divide the training dataset into $P$ parts ($TD_1, TD_2, ..., TD_P$). We use K-means to divide the initial dataset since K-means clustering is based on Euclidean distance. After K-means clustering, each sub-dataset will get its data center ($CT_1, CT_2, ..., CT_P$). Then we launch $P$ independent support vector machines ($SVM_1, SVM_2, ..., SVM_P$) to pro-

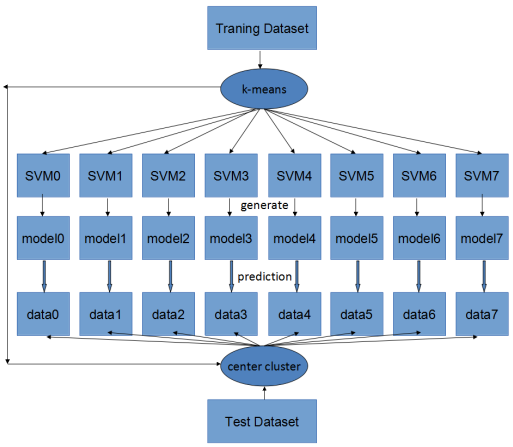



Figure 3. General Flow for CP-SVM. In the training part, different SVMs process its own dataset independently. In the classification part, different models can make the prediction independently.

cess these $P$ sub-datasets, which is like the top layer of the DC-Filter algorithm.

After the training process, each sub-SVM will generate its own model file ($MF_1, MF_2, ..., MF_P$). We can use these model files independently for classification. For a given sample $\hat{X}$, if its closest data center is $CT_i$, we will only use $MF_i$ to make a prediction for $\hat{X}$ because the support vectors in other model files have little impact on the classification result. Fig. 3 is the general flow of CP-SVM. CP-SVM is highly parallel because all the sub-problems are independent of each other. CP-SVM has little communication overhead from the operations in K-means clustering. CP-SVM generally is faster than the previous algorithms and its accuracy is closer to the SMO algorithm (Tables XIII to XVIII). However, in terms of scalability and speed, it is still not good enough.

### *B. CA-SVM: Communication-Avoiding SVM*

Based on the profiling result in Fig. 7, we can observe that CP-SVM is not well load-balanced. The reason is that the partitioning by K-means is irregular and imbalanced. For example, processor 2 in Fig. 7 has to handle 35,137 samples while processor 7 only needs to process 9,685 samples. On the other hand, the partitioning by K-means is data-dependent, which means we can not predict the partitioning pattern beforehand. This makes it unreliable to be used in practice. Therefore, we need to replace K-means with a better partitioning algorithm. We design three versions of balanced partitioning algorithms and use them to build the communication-avoiding algorithms. The objective of these three algorithms is to improve K-means.

*1) First Come First Served CASVM (FCFS-CA):* In our design, a machine node corresponds to a clustering center. If we have $P$ machine nodes, then we will partition the dataset into $P$ parts. As mentioned above, the objective of FCFS-CA partitioning algorithm is to make the number of samples on each node close to $m/P$. If a data center has $m/P$ samples, then it is balanced. The basic idea of this algorithm is to find the closest center for each sample. If a given center has been balanced, then no additional sample will be added to this center. The detailed FCFS-CA partitioning method is in Algorithm 3. Line 1-4 of Algorithm 3 is the initiation phase: we randomly pick $P$ samples from the dataset as the initial data centers. Line 5-14 is finding the center for each sample. Line 8-12 is finding the best under-loaded center for the $i$-th sample. Line 15-21 is recomputing the data center by averaging the all the samples in a certain center. Recomputing the centers by averaging is optional because it will not necessarily make the results better. Fig. 4 is an example of Algorithm 3.

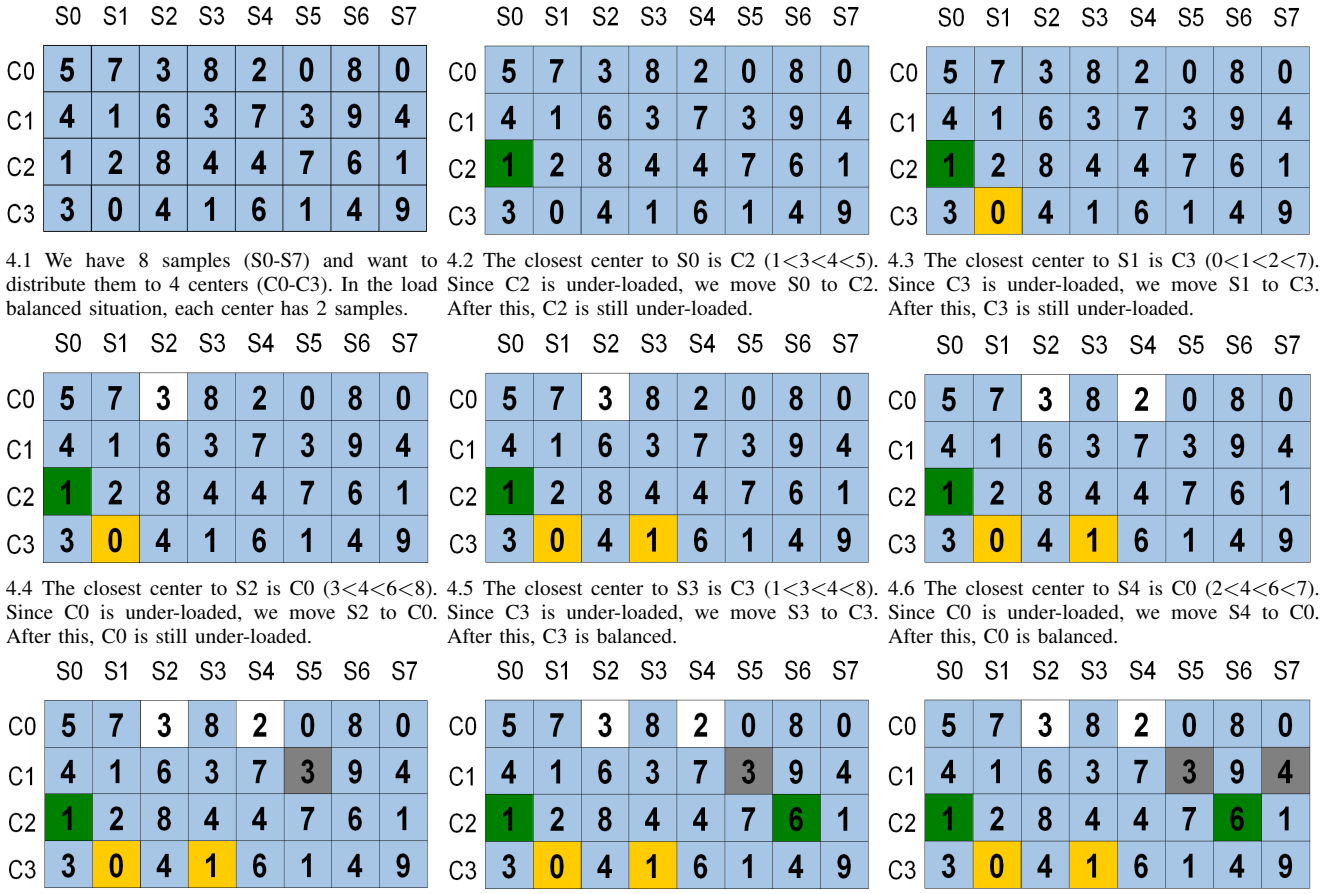
**Algorithm 3:** First Come First Served Partitioning

**Input**:
$CT$[i] is the center of i-$th$ cluster
$CS$[i] is the size of i-$th$ cluster
$SA$[i] is the i-$th$ sample
$m$ is the number of samples
$P$ is the number of clusters (processes)

**Output**:
$MB$[i] is the closest center to i-$th$ sample
$CT$[i] is the center of i-$th$ cluster

```
1  Randomly pick P samples from m samples (RS[0:P])
2  for i ∈ 1 : P do
3      CT[i] = RS[i]
4  balanced = m/P
5  for i ∈ 1 : m do
6      mindis = inf
7      minind = 0
8      for j ∈ 1 : P do
9          dist = euclidDistance(SA[i], CT[j])
10         if dist < mindis and CS[j] < balanced then
11             mindis = dist
12             minind = j
13     CS[minind]++
14     MB[i] = minind
15 for i ∈ 1 : P do
16     CT[i] = 0
17 for i ∈ 1 : m do
18     j = MB[i]
19     CT[j] += SA[i]
20 for i ∈ 1 : P do
21     CT[i] = CT[i] / CS[i]
```

From Fig. 5 we can observe that FCFS-CA can partition the dataset in a balanced way. After FCFS partitioning, all the nodes have the same number of samples. However, Table




4.1 We have 8 samples (S0-S7) and want to distribute them to 4 centers (C0-C3). In the load balanced situation, each center has 2 samples.

4.2 The closest center to S0 is C2 (1<3<4<5). Since C2 is under-loaded, we move S0 to C2. After this, C2 is still under-loaded.

4.3 The closest center to S1 is C3 (0<1<2<7). Since C3 is under-loaded, we move S1 to C3. After this, C3 is still under-loaded.

4.4 The closest center to S2 is C0 (3<4<6<8). Since C0 is under-loaded, we move S2 to C0. After this, C0 is still under-loaded.

4.5 The closest center to S3 is C3 (1<3<4<8). Since C3 is under-loaded, we move S3 to C3. After this, C3 is balanced.

4.6 The closest center to S4 is C0 (2<4<6<7). Since C0 is under-loaded, we move S4 to C0. After this, C0 is balanced.

4.7 The closest center to S5 is C0 (0<1<3<7). Since C0 and C3 are balanced, we move S5 to C1. After this, C1 is balanced.

4.8 The closest center to S6 is C3 (4<6<8<9). Since C3 is balanced, we move S6 to C2. After this, C2 is balanced.

4.9 Since only C1 is under-loaded, we move S7 to C1, which is the third choice. After this, all the centers are balanced.

Figure 4. This is an example of First Come First Served partitioning algorithm. Each figure is a distance matrix, which is referred as $dist$. For example, $dist$[i][j] is the distance between i-th center and j-th sample. The color of the matrix in the first figure is the original color. If $dist$[i][j] has a different color with the original one, then it means that j-th sample belongs to i-th center.

VI shows that the system is still loaded imbalanced. From Table VI we can see that Rank 6 has to do 1,552 iterations while Rank 5 only need to do 77 iterations. This means balanced data load does not guarantee balanced task load. This reasons are that: (1) From Table VII we can observe that the ratios of positive samples and negative samples on different nodes are different from each other. For example, the ratio on node 1 (0.0841) is 22 times of the ratio on node 5 (0.0038). (2) From Table VII we can also observe that the ratios of positive SVs and negative SVs on different nodes are close to each other (≈1). For this dataset, SVM wants the ratio of positive SVs and negative SVs to be 1. Therefore, more positive samples means there will be more SVs (because the number of negative samples is much larger than the number of positive samples). More SVs means it will need more computations. Node 1 has the largest number of positive samples (Table VII), thus it is the slowest one (Table VI). Node 5 has the smallest number of positive samples (Table VII), thus it is the fastest one (Table VI).

Therefore, in order to achieve load balance, we not only need to distribute the same volume of data to each node. We also need to make sure all the nodes have the same ratio of positive samples and negative samples. The idea is similar to Algorithm 3. To achieve this, we need to use positive $CS$[i] and negative $CS$[i] to replace $CS$[i] in Algorithm 3. We need to use positive $SA$[i] and negative $SA$[i] to replace $SA$[i] in Algorithm 3. We also need to use the number of positive samples and the number of negative sample to replace $m$ in Algorithm 3. After we can have made all the nodes have the same volume of data and the same pos-neg ratio (Table VIII), we can observe that we achieve load balancing (Table IX). For Table VI and Table IX, although in both situation Rank 5 is faster than Rank 1, we have significantly reduced the speedup of the fastest over the slowest from 20× (13.8/0.69) to 1× (6.50/6.21). To do FCFS partitioning algorithm in parallel, we use the

**Algorithm 4:** Parallel FCFS Partitioning

**Input**:
$CT$[i] is the center of i-$th$ cluster
$CS$[i] is the size of i-$th$ cluster
$SA$[i] is the i-$th$ sample
$m$ is the number of samples
$P$ is the number of clusters (processes)

**Output**:
$MB$[i] is the closest center to i-$th$ sample
$CT$[i] is the center of i-$th$ cluster

1 Randomly pick $P$ samples from $m$ samples ($RS$[0:P])
2 **for** $i \in 1 : P$ **do**
3  $CT[i] = RS[i]$
4 Broadcast $CT[1 : P]$ to all the nodes
5 Distribute $SA[1 : m]$ to all the nodes
6 $pm = m/P$
7 $balanced = pm/P$
8 **for** $i \in 1 : pm$ **do**
9  $mindis$ = inf
10  $minind$ = 0
11  **for** $j \in 1 : P$ **do**
12  $dist$ = euclidDistance($SA[i]$, $CT[j]$)
13  **if** $dist < mindis$ **and** $CS[j] < balanced$ **then**
14  $mindis = dist$
15  $minind = j$
16  $CS[minind]$++
17  $MB[i] = minind$
18 **for** $i \in 1 : P$ **do**
19  $CT[i]$ = 0
20 **for** $i \in 1 : pm$ **do**
21  $j = MB[i]$
22  $CT[j]$ += $SA[i]$
23 Recompute $CS$ by All-Reduce-Sum
24 Recompute $CT$ by All-Reduce-Sum
25 **for** $i \in 1 : P$ **do**
26  $CT[i] = CT[i] / CS[i]$
27 Gather $MB$ to node 0

divide-and-conquer approach. The basic idea is to convert one $m \rightarrow P \times m/P$ problem to several $m/P \rightarrow P \times m/P^2$ problems. The parallel FCFS partitioning approach is detailed in Algorithm 4. Specifically, line 4-7 of Algorithm 4 is the dividing phase. We distribute all the samples evenly to all the nodes. All the nodes will have a copy of data centers. Line 8-22 of Algorithm 4 is the parallel phase, during which all the nodes do the same thing independently. Each node will finish its own FCFS algorithm. Line 23-27 of Algorithm 4 is the conquer phase. In line 18-22, we need to use all the samples on all the nodes to recompute the data centers.

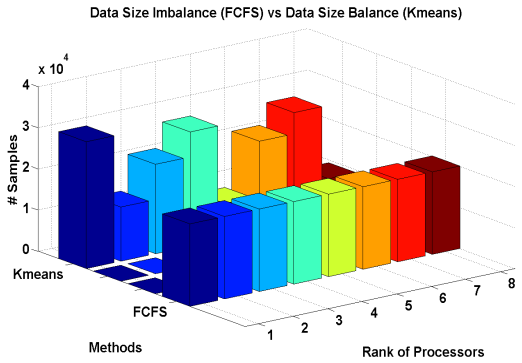



Figure 5. The figure shows that the partitioning by K-means is imbalanced while the partitioning by FCFS is balanced. Specifically, each node has exactly 20,000 samples after FCFS partitioning. The test dataset is $face$ with 160,000 samples (361 features per sample). 8 nodes are used in this test.

Table VI
BALANCED DATA $\neq$ BALANCED LOAD

| **Rank** | 5 | 7 | 0 | 2 | 3 | 4 | 6 | 1 |
|---|---|---|---|---|---|---|---|---|
| **Samples** | 20k | 20k | 20k | 20k | 20k | 20k | 20k | 20k |
| **Size (MB)** | 83 | 83 | 83 | 83 | 83 | 83 | 83 | 83 |
| **Iter** | 77 | 307 | 630 | 644 | 685 | 927 | 934 | 1552 |
| **Time (s)** | 0.69 | 2.74 | 5.66 | 5.78 | 6.14 | 8.26 | 8.33 | 13.8 |

*2) Balanced K-means CASVM (BKM-CA):* As mentioned above, the objective of BKM-CA partitioning algorithm is to make the number of samples on each node close to $m/P$ (a machine node corresponds to a data center) based on Euclid distance. If a data center has $m/P$ samples, then it is balanced. The basic idea of this algorithm is to slightly rearrange the results of the original K-means algorithms. We will keep moving samples from the over-loaded center to under-loaded center till they are balanced. The balanced K-means partitioning method is detailed in Algorithm 5. Line 1-3 of Algorithm 5 is the K-means clustering. In line 6-8, we calculate the Euclid distance distance between every sample and every center. For example, $dist[i][j]$ is the Euclid distance between $i$-th sample and $j$-th center. The variable $balanced$ is the number of samples every center should have in the load-balance situation. After the K-means clustering, some centers will have more than $balanced$ samples. In line 9-27 of Algorithm 5, the algorithm will move some samples from the over-loaded centers to the under-loaded centers. For a given over-loaded center, we will find the farthest sample (line 14-17 of Algorithm 5). The id of the farthest sample is $maxind$. In line 20-24 of Algorithm 5, we find the closest under-loaded center to sample $maxind$. In line 25-27, we move sample $maxind$ from its over-loaded center to the best under-loaded center. In line 15-21, we recompute the data center by averaging the all the samples in a certain center. Recomputing the centers by averaging is optional.

Table VII
THE NUMBER OF SAMPLES, POSITIVE SAMPLES, NEGATIVE SAMPLES; THE RATIO OF POSITIVE SAMPLES AND NEGATIVE SAMPLES; THE NUMBER OF SVS, POSITIVE SVS, NEGATIVE SVS; THE RATIO OF POSITIVE SVS AND NEGATIVE SVS;

| Rank | Samples | SVs |
|---|---|---|
| | # \| # (+) \| # (-) \| (+)/(-) | # \| # (+) \| # (-) \| (+)/(-) |
| 0 | 20k \| 629 \| 19371 \| 0.0325 | 1259 \| 629 \| 630 \| 0.9984 |
| 1 | 20k \| 1551 \| 18449 \| 0.0841 | 3103 \| 1551 \| 1552 \| 0.9994 |
| 2 | 20k \| 642 \| 19358 \| 0.0332 | 1285 \| 642 \| 643 \| 0.9984 |
| 3 | 20k \| 684 \| 19316 \| 0.0354 | 1369 \| 684 \| 685 \| 0.9985 |
| 4 | 20k \| 926 \| 19074 \| 0.0485 | 1853 \| 926 \| 927 \| 0.9989 |
| 5 | 20k \| 76 \| 19924 \| 0.0038 | 153 \| 76 \| 77 \| 0.9870 |
| 6 | 20k \| 932 \| 19068 \| 0.0489 | 1865 \| 932 \| 933 \| 0.9989 |
| 7 | 20k \| 306 \| 19694 \| 0.0155 | 613 \| 306 \| 307 \| 0.9967 |

Table VIII
THE NUMBER OF SAMPLES, POSITIVE SAMPLES, NEGATIVE SAMPLES; THE RATIO OF POSITIVE SAMPLES AND NEGATIVE SAMPLES; THE NUMBER OF SVS, POSITIVE SVS, NEGATIVE SVS; THE RATIO OF POSITIVE SVS AND NEGATIVE SVS;

| Rank | Samples | SVs |
|---|---|---|
| | # \| # (+) \| # (-) \| (+)/(-) | # \| # (+) \| # (-) \| (+)/(-) |
| 0 | 19967 \| 722 \| 19245 \| 0.0375 | 1445 \| 722 \| 723 \| 0.9986 |
| 1 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |
| 2 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |
| 3 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |
| 4 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |
| 5 | 19979 \| 692 \| 19287 \| 0.0359 | 1385 \| 692 \| 693 \| 0.9986 |
| 6 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |
| 7 | 20009 \| 722 \| 19287 \| 0.0374 | 1445 \| 722 \| 723 \| 0.9986 |

Fig. 6 is an example of Algorithm 5. Like FCFS partitioning algorithm, we need to add ratio-balancing technique to the above data-balancing method to achieve load-balancing. We also need to use the divide-and-conquer technique used in parallel FCFS (Algorithm 4) to parallelize the balanced K-means partitioning algorithm.

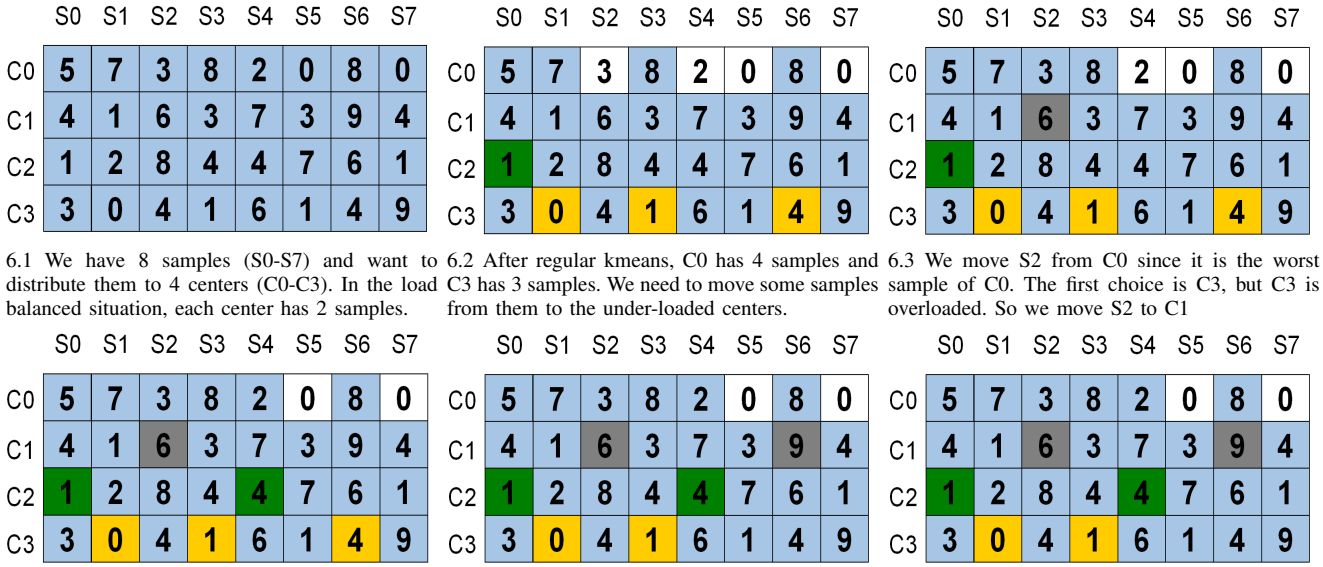
*3) Randomly-Averaging CASVM (RA-CA):* The basic idea is to randomly divide the original training dataset into $P$ parts ($TD_1, TD_2, ..., TD_P$) evenly. After partitioning, each sub-dataset will generate its own data center ($CT_1, CT_2, ..., CT_P$). For $TD_i$ ($i \in \{1, 2, ..., P\}$), its data center (i.e. $CT_i$) is the average of all the samples on node $i$ (Equation 14). Then we launch $P$ independent support vector machines ($SVM_1, SVM_2, ..., SVM_P$) to process these $P$ sub-datasets in parallel. After the training process, each sub-SVM will generate its own model file ($MF_1, MF_2, ..., MF_P$). Like CP-SVM, we can use these model files independently for classification. For any unknown sample ($\hat{X}$), if its closest data center is $CT_i$, we will only use $MF_i$ to make prediction for $\hat{X}$. The communication overhead of CP-SVM, FCFS-CA and BKM-CA are from the data transfer and distribution in K-means like partitioning

**Algorithm 5:** Balanced Kmeans Partitioning

**Input**:
$CT$[i] is the center of i-$th$ cluster
$CS$[i] is the size of i-$th$ cluster
$SA$[i] is the i-$th$ sample
$m$ is the number of samples
$P$ is the number of clusters (processes)

**Output**:
$MB$[i] is the closest center to i-$th$ sample
$CT$[i] is the center of i-$th$ cluster

Randomly pick $P$ samples from $m$ samples ($RS$[0:P])
**for** $i \in 1 : P$ **do**
  $CT[i]$ = $RS[i]$
**do kmeans clustering**
$balanced = m/P$
**for** $i \in 1 : m$ **do**
  **for** $j \in 1 : P$ **do**
    $dist$[i][j] = euclidDistance($SA[i]$, $CT[j]$)
**for** $j \in 1 : P$ **do**
  **while** $CS[j] > balanced$ **do**
    $CS[j] < balanced$
    $maxdist$ = 0
    $maxind$ = 0
    **for** $i \in 1 : m$ **do**
      **if** *dist[i][j]>maxdist* **and** *MB[i]==j* **then**
        $maxdist$ = $dist$[i][j]
        $maxind$ = $i$
    $mindist$ = inf
    $minind$ = $j$
    **for** $k \in 1 : P$ **do**
      **if** $dist[maxind][k] < mindist$ **then**
        **if** $CS[k] < balanced$ **then**
          $mindist$ = $dist[maxind][k]$
          $minind$ = $k$
    $MB[maxind]$ = $minind$
    $CS[j]$=$CS[j]$-1
    $CS[minind]$=$CS[minind]$+1
**for** $i \in 1 : P$ **do**
  $CT[i]$ = 0
**for** $i \in 1 : m$ **do**
  $j$ = $MB[i]$
  $CT[j]$ += $SA[i]$
**for** $i \in 1 : P$ **do**
  $CT[i]$ = $CT[i]$ / $CS[i]$




6.1 We have 8 samples (S0-S7) and want to distribute them to 4 centers (C0-C3). In the load balanced situation, each center has 2 samples.

6.2 After regular kmeans, C0 has 4 samples and C3 has 3 samples. We need to move some samples from them to the under-loaded centers.

6.3 We move S2 from C0 since it is the worst sample of C0. The first choice is C3, but C3 is overloaded. So we move S2 to C1

6.4 We move S4 from C0 since C0 is still over-loaded. The first choice is C2. C2 is under-loaded, so we move S4 to C2.

6.5 We move S6 from C3 since it is the worst sample of C3. Both C0 and C2 are already balanced. So we move S6 to C1.

6.6 Finally, each centers has exactly 2 samples. Now the system is load balanced.

Figure 6. This is an example of First Come First Served partitioning algorithm. Each figure is a distance matrix, which is referred as $dist$. For example, $dist$[i][j] is the distance between i-th center and j-th sample. The color in the first figure is the original color. If $dist$[i][j] has a different color with the original one, then it means that j-th sample belongs to i-th center.

Table IX
BALANCED DATA + BALANCED RATIO = BALANCED LOAD

| **Rank** | 5 | 7 | 3 | 0 | 6 | 4 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|
| **Samples** | 20k | 20k | 20k | 20k | 20k | 20k | 20k | 20k |
| **Size (MB)** | 83 | 83 | 83 | 83 | 83 | 83 | 83 | 83 |
| **Iter** | 693 | 723 | 723 | 723 | 723 | 723 | 723 | 724 |
| **Time (s)** | 6.21 | 6.48 | 6.48 | 6.49 | 6.49 | 6.49 | 6.49 | 6.50 |

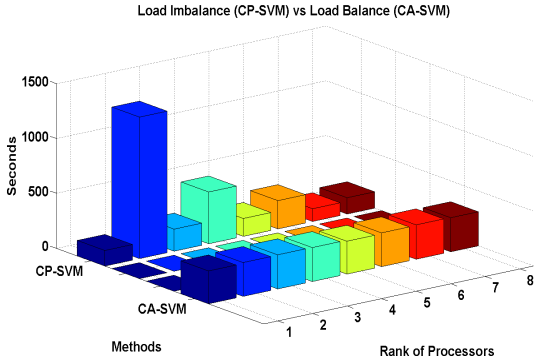



Figure 7. The figure shows that CP-SVM is load imbalanced while CA-SVM is load-balanced. The test dataset is $epsilon$ with 128,000 samples (2,000 nnz per sample). 8 nodes are used in this test.

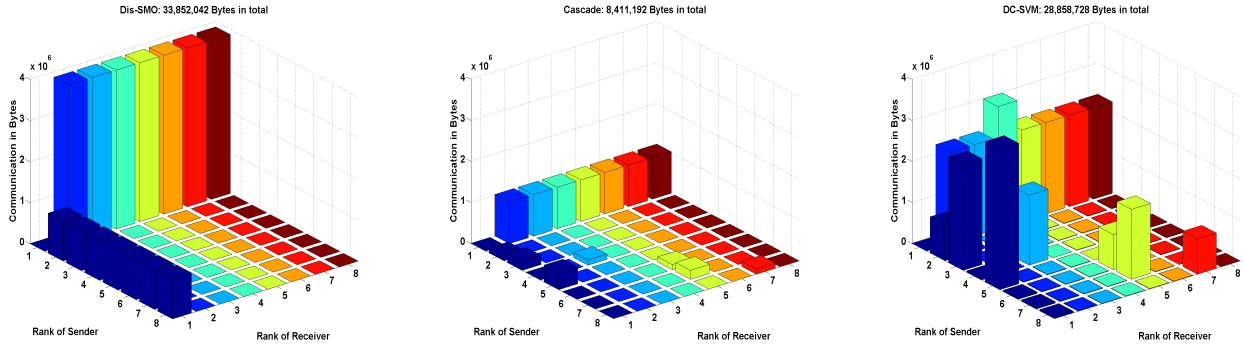
algorithm. In this new method, we replace the K-means variants with a no- communication partition. Thus, we can also directly refer RA-CA as CA-SVM (Communication-Avoiding SVM). However, this assumes that originally the dataset is distributed to all the nodes. To give a fair comparison, we implement two versions of CA-SVM. **casvm1** means that we put the initial dataset on just one node, which needs communication to distribute the dataset to different nodes. **casvm2** means that we put the initial dataset on different nodes, which needs no communication (Fig. 9). All the results of CA-SVM in Section V are based on casvm2. CA-SVM may lose accuracy because evenly-randomly dividing does not get the best partitioning in terms of Euclidean distance. However, the results in Tables XIII to XVIII show that it achieves significant speedup with comparable results.

The framework of CA-SVM is shown in Algorithm 6. The prediction process may need a little communication. However, both the data centers and test samples are pretty small compared with the training samples. This communication will not bring about significant overheads. On the other hand, the majority of SVM time is spent on the training process. Like previous work (e.g. SMO, Cascade, DC-SVM), the focus of this paper is on optimizing the training process.

$$CT_i = \frac{\sum_{j=1}^{\#samples}\{X_j | X_j \in TD_i\}}{\sum_{j=1}^{\#samples}\{1 | X_j \in TD_i\}} \tag{14}$$

### C. Communication Pattern

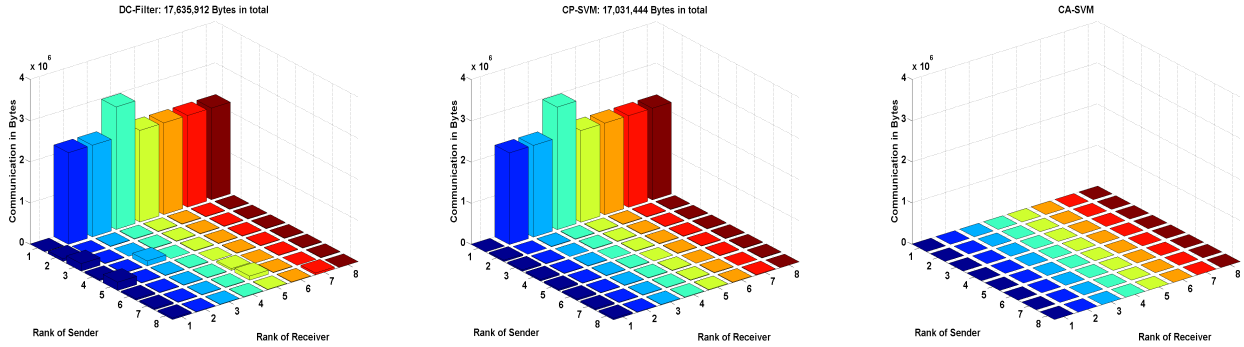
*1) Communication Modeling:* We only give the results because the space is limited, the detailed proofs are in [7]. The formulas of communication volume are in Table




8.1 Dis-SMO: 34MB

8.2 Cascade: 8MB

8.3 DC-SVM: 29MB




8.4 DC-Filter: 18MB

8.5 CP-SVM: 17MB

8.6 CA-SVM: 0MB

Figure 8. Communication Patterns of different approaches. The data is from running the 6 approaches on 8 nodes with the same 5MB Toy Dataset. x-axis is the rank of sending processors, y-axis is the rank of receiving processors, and z-axis is the volume of communication in bytes. The vertical ranges (z-axis) of these 6 sub-figures are the same.

**Algorithm 6:** CA-SVM (casvm2 in Fig. 9)

**Training Process (no communication):**
**0**: $i \in \{1, 2, ..., m/P\}, j \in \{1, 2, ..., P\}$
**1**: For node $N_j$, input the samples $X_i$ and labels $y_i$.
**2**: For node $N_j$, get its data center $CT_j$.
**3**: For node $N_j$, launch a SVM training process $SVM_j$.
**4**: For node $N_j$, save the model file of converged $SVM_j$ as $MF_j$.
**Prediction Process (little communication):**
**1**: Send all the data centers $CT_j, j \in \{1, 2, ..., P\}$ to root node
**2**: If $CT_j$ is closer to $\hat{X}$ than other centers, then send $\hat{X}$ to $N_j$
**3** Use $MF_j$ to make prediction for $\hat{X}$

X. The experimental results in the table are based on the ijcnn dataset on 8 Hopper nodes [6]. The terms used in the formulas are in Table II. We can use the formulas to predict the communication volume for a given method. For example, for the test dataset used in this experiment, $m$ is 48,000, $n$ is 13, and $s$ is 4474. We can predict the communication volume of Cascade is about $3\times(48000\times13+48000+4474\times13)\times4$B = 8.4MB. Our experimental result is 8.41MB, which means the prediction for Cascade is very close to the actual volume.

Table X
MODELING OF COMMUNICATION VOLUME

| **Method** | Formula | Prediction | Test |
|---|---|---|---|
| **Dis-SMO** | $\Theta(26Ip + 2pm + 4mn)$ | 36MB | 34MB |
| **Cascade** | $O(3mn + 3m + 3sn)$ | 8.4MB | 8.4MB |
| **DC-SVM** | $\Theta(9mn + 12m + 2kpn)$ | 24MB | 29MB |
| **DC-Filter** | $O(6mn + 7m + 3sn + 2kpn)$ | 16.2MB | 18MB |
| **CP-SVM** | $\Theta(6mn + 7m + 2kpn)$ | 15.6MB | 17MB |
| **CA-SVM** | 0 | 0MB | 0MB |

*2) Point-to-Point profiling:* Fig. 8 shows the communication patterns of these six approaches. To improve the efficiency of communication, we use as many collective communications as possible because a single collective operation is more efficient than multiple send/receive operations. Due to the communications of K-means, DC-Filter and CP-SVM have to transfer more data than Cascade. However, from Table XI we can observe that CP-SVM is more efficient than Cascade since the volume of communication per operation is higher.

*3) Ratio of Communication to Computation:* Fig. 9 shows the ratio of communication time to computation time for different methods. From Fig. 9 we can observe that our

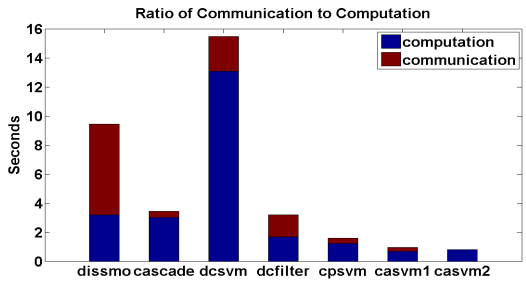



Figure 9. The ratio of computation to communication. The experiment is based on a toy dataset. To give a fair comparison, we implemented two versions of CA-SVM. **casvm1** means that we put the initial dataset on the same node, which needs communication to distribute the dataset to different nodes. **casvm2** means that we put the initial dataset on different nodes, which needs no communication.

algorithms significantly reduce the volume of communication and the ratio of communication to computation. This is highly important since the existing supercomputers [23] are generally suitable for computation-intensive applications rather than communication-intensive applications. Besides, less communication can greatly reduce the power consumption [24]. Table X shows that the communication volumes of DC-Filter and CP-SVM are similar. However, Fig. 9 shows that there is a big difference between DC-Filter communication time and CP-SVM time. The reason is that the communication of CP-SVM can be done only by collective operations (e.g. Scatter) but DC-Filter has to conduct some point-to-point communications (e.g. Send/Recv) on the lower levels ( Fig. 2).

Table XI
EFFICIENCY OF COMMUNICATION

| **Method** | Amount | Comm Operations | Amount/Operation |
|---|---|---|---|
| **Dis-SMO** | 34MB | 335,186 | 101B |
| **Cascade** | 8MB | 56 | 150,200B |
| **DC-SVM** | 29MB | 80 | 360,734B |
| **DC-Filter** | 18MB | 80 | 220,449B |
| **CP-SVM** | 17MB | 24 | 709,644B |
| **CA-SVM** | 0MB | 0 | N/A |

## V. EXPERIMENTAL RESULTS AND ANALYSIS

The test datasets in our experiments are shown in Table XII, and they are from real-world applications. We use MPI for distributed processing, OpenMP for multi-threading, and Intel Intrinsics for SIMD parallelism. To give a fair comparison, all the methods in this paper are based on the same shared-memory SMO implementation (Our previous work in [15] showed that it is faster than the state-of-the-art implementation on a shared memory system). The K-means partitioning in DC-SVM, DC-Filter, CP-SVM, and BKM-CA are distributed versions, which achieved the same partitioning result and comparable performance with Liao's implementation [25]. Our experiments are conducted on NERSC Hopper and Edison systems [6].

Table XII
THE TEST DATASETS

| Dataset | Application Field | #samples | #features |
|---|---|---|---|
| adult [9] | Economy | 32,561 | 123 |
| epsilon [26] | Character Recognition | 400,000 | 2,000 |
| face [27] | Face Detection | 489,410 | 361 |
| gisette [28] | Computer Vision | 6,000 | 5,000 |
| ijcnn [29] | Text Decoding | 49,990 | 22 |
| usps [30] | Transportation | 266,079 | 675 |
| webspam [31] | Management | 350,000 | 16,609,143 |

Table XIII
ADULT DATASET ON HOPPER (K-MEANS CONVERGED IN 8 LOOPS)

| **Method** | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 84.3% | 8,054 | 5.64s (0.006, 5.64) |
| **Cascade** | 83.6% | 1,323 | 1.05s (0.007, 1.04) |
| **DC-SVM** | 83.7% | 8,699 | 17.1s (0.042, 17.1) |
| **DC-Filter** | 84.4% | 3,317 | 2.23s (0.042, 2.18) |
| **CP-SVM** | 83.0% | 2,497 | 1.66s (0.041, 1.59) |
| **BKM-CA** | 83.3% | 1,482 | 1.61s (0.057, 1.54) |
| **FCFS-CA** | 83.6% | 1,621 | 1.21s (0.005, 1.19) |
| **RA-CA** | 83.1% | 1,160 | 0.96s (4e-4, 0.95) |

Table XIV
FACE DATASET ON HOPPER (K-MEANS CONVERGED IN 29 LOOPS)

| **Method** | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 98.0% | 17,501 | 358s (2e-4, 358) |
| **Cascade** | 98.0% | 2,274 | 67.0s (0.10, 66.9) |
| **DC-SVM** | 98.0% | 20,331 | 445s (13.6, 431) |
| **DC-Filter** | 98.0% | 13,999 | 314s (13.6, 297) |
| **CP-SVM** | 98.0% | 13,993 | 311s (13.6, 295) |
| **BKM-CA** | 98.0% | 2,209 | 88.9s (17.8, 71.0) |
| **FCFS-CA** | 98.0% | 2,194 | 65.3s (0.43, 64.9) |
| **RA-CA** | 98.0% | 2,268 | 66.4s (0.08, 66.4) |

### A. *Speedup and Accuracy*

From Table XIII to Table XVIII, we can observe that CA-SVM (i.e. RA-CA) can achieve 3× - 16× (7× on average) speedups over distributed SMO algorithm with comparable accuracies. The Init time in these tables include the partition time like K-means, and the Training time denotes the SVM training process. The accuracy loss ranges from none to 3.6% (1.3% on average). According to previous work [17], the accuracy loss in this paper is small and tolerable for practical applications. Additionally, we can observe that CA-SVM can reduce the number of iterations, which means it is intrinsically more efficient than other algorithms. For DC-SVM, DC-Filter, CP-SVM, and BKM-CA the majority of

Table XV
GISETTE DATASET ON HOPPER (K-MEANS CONVERGED IN 31 LOOPS)

| Method | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 97.6% | 1,959 | 8.1s (0.26, 7.86) |
| **Cascade** | 88.3% | 1,520 | 15.9s (0.20, 15.7) |
| **DC-SVM** | 90.9% | 4,689 | 130.7s (2.35, 127.9) |
| **DC-Filter** | 85.7% | 1,814 | 20.1s (2.39, 17.2) |
| **CP-SVM** | 95.8% | 521 | 8.30s (2.30, 5.4) |
| **BKM-CA** | 95.8% | 452 | 4.75s (2.29, 2.46) |
| **FCFS-CA** | 96.5% | 441 | 2.48s (0.07, 2.41) |
| **RA-CA** | 94.0% | 487 | 2.9s (0.014, 2.87) |

Table XVI
IJCNN DATASET ON HOPPER (K-MEANS CONVERGED IN 7 LOOPS)

| Method | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 98.7% | 30,297 | 23.8s (0.008, 23.8) |
| **Cascade** | 95.5% | 37,789 | 13.5s (0.007, 13.5) |
| **DC-SVM** | 98.3% | 31,238 | 59.8s (0.04, 59.7) |
| **DC-Filter** | 95.8% | 17,339 | 8.4s (0.04, 8.3) |
| **CP-SVM** | 98.7% | 7,915 | 6.5s (0.04, 6.4) |
| **BKM-CA** | 98.3% | 5,004 | 3.0s (0.08, 2.87) |
| **FCFS-CA** | 98.5% | 7,450 | 3.6s (0.005, 3.55) |
| **RA-CA** | 98.0% | 6,110 | 3.4s (3e-4, 3.4) |

Table XVII
USPS DATASET ON EDISON (K-MEANS CONVERGED IN 28 LOOPS)

| Method | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 99.2% | 47,214 | 65.9s (2e-4, 65.9) |
| **Cascade** | 98.7% | 132,503 | 969s (0.008, 969) |
| **DC-SVM** | 98.7% | 83,023 | 1889s (1.5, 1887) |
| **DC-Filter** | 99.6% | 67,880 | 242s (1.5, 240) |
| **CP-SVM** | 98.9% | 7,247 | 35.7s (1.5, 33.9) |
| **BKM-CA** | 98.9% | 6,122 | 30.4s (2.02, 28.4) |
| **FCFS-CA** | 99.0% | 6,513 | 30.1s (0.04, 29.7) |
| **RA-CA** | 98.9% | 6,435 | 24.5s (0.0018, 24.5) |

Table XVIII
WEBSPAM DATASET ON HOPPER (K-MEANS CONVERGED IN 38 LOOPS)

| Method | Accuracy | Iterations | Time (Init, Training) |
|---|---|---|---|
| **Dis-SMO** | 98.9% | 164,465 | 269.1s (0.05, 269.0) |
| **Cascade** | 96.3% | 655,808 | 2944s (0.003, 2944) |
| **DC-SVM** | 97.6% | 229,905 | 3093s (0.95, 3092) |
| **DC-Filter** | 97.2% | 108,980 | 345s (1.0, 345) |
| **CP-SVM** | 98.7% | 14,744 | 41.8s (1.0, 40.7) |
| **BKM-CA** | 98.5% | 14,208 | 24.3s (1.12, 23.0) |
| **FCFS-CA** | 98.3% | 12,369 | 21.2s (0.03, 21.0) |
| **RA-CA** | 96.9% | 10,430 | 17.3s (0.003, 17.3) |

the initial time is spent on K-means clustering. However, we can observe that K-means actually is extremely fast in most of the situations.

### *B. Strong Scaling and Weak Scaling*

Since BKM-CA, FCFS-CA, and RA-CA have the same kind of scaling pattern, we use RA-CA to represent CA-SVM. Tables XIX and XX show the results of strong scaling time and efficiency. We can observe that the strong scaling efficiency of CA-SVM is increasing with the number of processors. The reason is that the number of iterations is decreasing since the number of samples ($m/P$) on each node is decreasing. The single iteration time is also reduced with fewer samples on each node. More information about the efficiency of CA-SVM can be found in [7]. For the weak scaling results in Tables XXI and XXII, we can observe that all the efficiencies of these six algorithms are decreasing with the increasing number of processors. In theory, the work load of CA-SVM on each node is the same with the increasing number processors. However, in practice, the system overhead is higher with more processors. The weak scaling efficiency of CA-SVM only decreases 4.7% with a 16× increase in the number of processors. Therefore, compared with these five other algorithms, CA-SVM is intrinsically efficient.

## VI. CONCLUSION

Existing distributed SVM approaches like Dis-SMO, Cascade, and DC-SVM suffer from intensive communication, computation inefficiency and bad scaling. In this paper, we design and implement five efficient approaches (i.e. DC-Filter, CP-SVM, BKM-CA, FCFS-CA, and RA-CA) through step-by-step optimizations. BKM-CA, FCFS-CA, and RA-CA belong to CA-SVM. We manage to totally avoid inter-node communication, obtain a perfect load-balancing, and achieve 7× average speedup with only 1.3% average loss in accuracy for six real-world application datasets. Because of faster iteration and reduced number of iterations, CA-SVM can achieve 1068.7% strong scaling when we increase the number of processors from 96 to 1536. Thanks to the removal of communication overhead, CA-SVM attains a 95.3% weak scaling from 96 to 1536 processors. The results justify that the approaches proposed in this paper can be used in large-scale applications.

## REFERENCES


[1] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc. (2015) Source code of casvm. [Online]. Available: https://github.com/fastalgo/casvm

[2] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[3] T. Joachims, *Text categorization with support vector machines: Learning with many relevant features*. Springer, 1998.

[4] F. E. Tay and L. Cao, "Application of support vector machines in financial time series forecasting," *Omega*, vol. 29, no. 4, pp. 309–317, 2001.

Table XIX
STRONG SCALING TIME FOR EPSILON DATASET: 128K SAMPLES, 2K NNZ PER SAMPLE

| **Processors** | 96 | 192 | 384 | 768 | 1536 |
|---|---|---|---|---|---|
| **Dis-SMO** | 2067s | 1135s | 777s | 326s | 183s |
| **Cascade** | 1207s | 376s | 154s | 76.1s | 165s |
| **DC-SVM** | 11841s | 8515s | 4461s | 3909s | 3547s |
| **DC-Filter** | 2473s | 1517s | 1100s | 1519s | 1879s |
| **CP-SVM** | 2248s | 1332s | 877s | 546s | 202s |
| **CA-SVM** | 1095s | 313s | 86s | 23s | 6s |

Table XX
STRONG SCALING EFFICIENCY FOR EPSILON DATASET: 128K SAMPLES, 2K NNZ PER SAMPLE

| **Processors** | 96 | 192 | 384 | 768 | 1536 |
|---|---|---|---|---|---|
| **Dis-SMO** | 100% | 91.1% | 66.5% | 79.2% | 70.4% |
| **Cascade** | 100% | 160.5% | 195.4% | 198.4% | 45.7% |
| **DC-SVM** | 100% | 69.5% | 66.4% | 37.9% | 20.9% |
| **DC-Filter** | 100% | 81.5% | 56.2% | 20.3% | 8.2% |
| **CP-SVM** | 100% | 84.4% | 64.1% | 51.4% | 69.7% |
| **CA-SVM** | 100% | 175.0% | 319.5% | 603.0% | 1068.7% |

Table XXI
WEAK SCALING TIME FOR EPSILON DATASET: 2K SAMPLES PER NODE, 2K NNZ PER SAMPLE

| **Processors** | 96 | 192 | 384 | 768 | 1536 |
|---|---|---|---|---|---|
| **Dis-SMO** | 14.4s | 27.9s | 51.3s | 94.8s | 183s |
| **Cascade** | 7.9s | 8.5s | 11.9s | 52.9s | 165s |
| **DC-SVM** | 17.8s | 67.9s | 247s | 1002s | 3547s |
| **DC-Filter** | 16.8s | 51.2s | 181s | 593s | 1879s |
| **CP-SVM** | 13.8s | 36.1s | 86.8s | 165s | 202s |
| **CA-SVM** | 6.1s | 6.2s | 6.2s | 6.4s | 6.4s |

Table XXII
WEAK SCALING EFFICIENCY FOR EPSILON DATASET: 2K SAMPLES PER NODE, 2K NNZ PER SAMPLE

| **Processors** | 96 | 192 | 384 | 768 | 1536 |
|---|---|---|---|---|---|
| **Dis-SMO** | 100% | 51.7% | 28.2% | 15.2% | 7.9% |
| **Cascade** | 100% | 93.2% | 66.2% | 14.9% | 4.8% |
| **DC-SVM** | 100% | 26.3% | 7.2% | 1.8% | 0.5% |
| **DC-Filter** | 100% | 32.8% | 9.3% | 2.8% | 0.9% |
| **CP-SVM** | 100% | 38.2% | 15.9% | 8.3% | 6.8% |
| **CA-SVM** | 100% | 98.9% | 97.8% | 96.0% | 95.3% |


[5] C. Leslie, E. Eskin, and W. S. Noble, "The spectrum kernel: A string kernel for SVM protein classification," in *Proceedings of the Pacific symposium on biocomputing*, vol. 7. Hawaii, USA., 2002, pp. 566–575.

[6] NERSC. (2014) NERSC Systems. [Online]. Available: https://www.nersc.gov/users/computational-systems/

[7] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc. Appendix of casvm. [Online]. Available: https://sites.google.com/site/yangyouresearch/files/appendix.pdf

[8] A. Grama, *Introduction to parallel computing*. Pearson Education, 2003.

[9] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods Support Vector Learning*. MIT Press, 1999, pp. 185–208.

[10] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The Cascade SVM," *Advances in neural information processing systems*, vol. 17, pp. 521–528, 2004.

[11] C.-J. Hsieh, S. Si, and I. S. Dhillon, "A divide-and-conquer solver for kernel support vector machines," *arXiv preprint arXiv:1311.0914*, 2013.

[12] L. J. Cao and S. Keerthi, "Parallel sequential minimal optimization for the training of support vector machines," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1039–1049, 2006.

[13] C.-C. Chang and C.-J. Lin, "LIBSVM: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[14] 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*. ACM, 2008, pp. 104–111.

[15] Y. You, S. Song, H. Fu, A. Marquez, M. Dehnavi, K. Barker, K. W. 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 *2014 International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE.

[16] E. W. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[17] E. Y. Chang, "PSVM: Parallelizing support vector machines on distributed computers," in *Foundations of Large-Scale Multimedia Information Management and Retrieval*. Springer, 2011, pp. 213–230.

[18] G. Wu, E. Chang, Y. K. Chen, and C. Hughes, "Incremental approximate matrix factorization for speeding up support vector machines," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 760–766.

[19] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel software for training large scale support vector machines on multiprocessor systems," *The Journal of Machine Learning Research*, vol. 7, pp. 1467–1492, 2006.

[20] T. Joachims, "Making large scale SVM learning practical," in *Advances in Kernel Methods Support Vector Learning*. MIT Press, 1999, pp. 169–184.

[21] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *The Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[22] T.-K. Lin and S.-Y. Chien, "Support vector machines on gpu with sparse matrix format," in *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*. IEEE, 2010, pp. 313–318.

[23] J. Dongarra. (2014) Top500 june 2014 list. [Online]. Available: http://www.top500.org/lists/2014/06/

[24] J. Demmel, A. Gearhart, B. Lipshitz, and O. Schwartz, "Perfect strong scaling using no additional energy," in *2013 International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE.

[25] W.-K. Liao. (2013) Parallel k-means. [Online]. Available: http://users.eecs.northwestern.edu/ wkliao/Kmeans/

[26] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag, "Pascal large scale learning challenge," in *25th International Conference on Machine Learning (ICML2008) Workshop. http://largescale. first. fraunhofer. de. J. Mach. Learn. Res*, vol. 10, 2008, pp. 1937–1953.

[27] I. W. Tsang, J.-Y. Kwok, and J. M. Zurada, "Generalized core vector machines," *Neural Networks, IEEE Transactions on*, vol. 17, no. 5, pp. 1126–1140, 2006.

[28] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, "Result analysis of the NIPS 2003 feature selection challenge," in *Advances in Neural Information Processing Systems*, 2004, pp. 545–552.

[29] D. Prokhorov, "Ijcnn 2001 neural network competition," *Slide presentation in IJCNN*, vol. 1, 2001.

[30] J. J. Hull, "A database for handwritten text recognition research," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 5, pp. 550–554, 1994.

[31] S. Webb, J. Caverlee, and C. Pu, "Introducing the webb spam corpus: Using email spam to identify web spam automatically." in *CEAS*, 2006.