Design and Implementation of a Communication-Optimal Classifier for Distributed Kernel Support Vector Machines

Yang You, *Member, IEEE,* James Demmel, *Fellow, ACM and IEEE,* Kent Czechowski, *Member, IEEE,* Le Song, *Member, IEEE,* and Rich Vuduc, *Member, IEEE*

Abstract—We consider the problem of how to design and implement communication-efficient versions of parallel kernel 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 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].

Index Terms—distributed memory algorithms; communication-avoidance; statistical machine learning

1 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], [4], [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 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 sparse-matrix-sparse-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 3 versions of a Communication-Efficient SVM or CE-SVM (BKM-SVM, FCFS-SVM, CA-SVM) that achieves load-balance and significantly reduces the amount of inter-node communication. Our contributions are:

(1) We convert a communication-intensive algorithm to an embarrassingly-parallel algorithm by significantly reducing the amount of inter-node communication.

(2) CE-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.

For example, FCFS-SVM achieves 2-13$\times$ (6$\times$ on average) speedups over distributed SMO algorithm with comparable accuracies. The accuracy losses range from none to 1.1% (0.47% 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. FCFS-SVM improves the weak scaling efficiency from 7.9% to 39.4% when we increase the number of processors from 96 to 1536 on NERSC's Edison system [6].

- *The incomplete version of this work (CA-SVM: Communication-Avoiding Support Vector Machines on Distributed Systems) won the* **Best Paper award** *of 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*
- *Yang You and James Demmel are with Computer Science Division, UC Berkeley, CA, USA. {youyang, demmel}@cs.berkeley.edu*
- *Kent Czechowski, Le Song and Rich Vuduc are with College of Computing, Georgia Tech, GA, USA. {kentcz, lsong, richie}@gatech.edu*

Manuscript received Feb 27, 2015.

2 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 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* (§ 2.3) and *Divide-and-Conquer SVM* (§ 2.4). We briefly survey alternative methods in § 2.5. 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.

2.1 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 α_i (called Lagrange multipliers), with each α_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):

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

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

Here, C is a regularization constant that attempts to balance generality and accuracy; 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 1.) 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),

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

TABLE 1
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	$K(X_i, X_j) = \tanh(a{X_i}^\top X_j + r)$

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 α coefficients are *sparse* or mostly zero. The set of samples with non-zero α_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.

It is worth noting that $K(X_i, X_j)$ is (re)computed on demand by all algorithms that use it, as opposed to computing all values once and storing them. The reason is that the kernel matrix needs $O(m^2)$ memory, which is prohibitive for real-world applications because m is usually much larger than the sample dimension. For example, a 357 MB dataset (520,000 $\times$ 90 matrix) [13] would generate a 2000 GB kernel matrix. To clarify the notation, $K(X_i, X_j)$ means the kernel function that computes the kernel value of X_i and X_j. $K_{i,j}$ means the value at i-th row and j-column of kernel matrix K, so we have $K_{i,j} = K(X_i, X_j)$.

2.2 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 [14] and GPUSVM [15]. 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. [16]. 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 samples, 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. This means all the nodes share one model during the training phase. 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 [15]. However, Catanzaro et al target single-node (single-GPU) systems, whereas we focus 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 $b_{high} = -1$, $high = \min\{i : y_i = 1\}$
4 $b_{low} = 1$, $low = \min\{i : y_i = -1\}$.
5 Update α_{high} and α_{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 $high = \arg\min\{f_i : i \in I_{high}\}$
10 $low = \arg\max\{f_i : i \in I_{low}\}$
11 $b_{high} = \min\{f_i : i \in I_{high}\}, b_{low} = \max\{f_i : i \in I_{low}\}$
12 Update α_{high} and α_{low} according to Equations (6) and (7).
13 If $b_{low} > b_{high}$, then go to Step 6.

2.3 Cascade SVM

Cascade SVM is a multi-layer approach designed with distributed systems in mind [10]. As Fig. 1 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. α_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 α_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.

2.4 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.

K-means clustering: since K-means clustering is a critical sub-step 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 [17]. The value

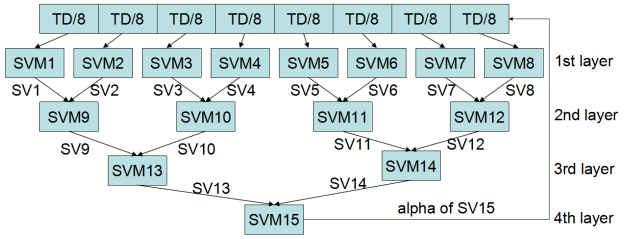


Fig. 1. This figure is an illustration of Cascade SVM [10]. Different layers have to be processed sequentially, i.e. layer $i+1$ can be processed after layer i has been finished. The tasks in the same level can be processed concurrently. If the result at the bottom layer is not good enough, the user 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. One pass is enough in most cases.

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. In this work, k is set to be the number of processors. A naive version of K-means clustering appears in Alg. 2.

Algorithm 2: Naive 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.

2.5 Other methods

There are other potential algorithms for SVMs. One method uses matrix factorization of the kernel matrix K [18]. Another class of methods relies on solving the QP problem using an iteration structure that considers more than two points at a time [19], [20]. Additionally, there are other optimizations for serial approach [9], [21], [22] or parallel approach on shared memory systems [15], [23]. All of these approaches are hard to compare "head-to-head" against the partitioned SMO schemes this paper considers, so we leave such comparisons for future work.

3 Re-Design Divide-and-Conquer Method

3.1 Performance Modeling for Existing Methods

In this section, we will do performance modeling for the three related methods mentioned in Section 2. The related terms are in Table 2 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 2 actually are ratios of communication time to flop time). T_o is the overall overhead, T_o^{comm} is the communication overhead, and T_o^{comp} is the computation

TABLE 2
Terms for Performance Modelling

$m; n; P$	# samples; # features per sample; # nodes or processes
$T_1; T_p$	serial run time; parallel run time
$t_s; t_w$	latency 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 in flops; parallel overhead ($T_o = PT_p - W$)
$s; I; k$	# SVs; # SVM iters; # K-means iters

overhead. 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). 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 ($P = O(\sqrt{W})$).

$$W = \frac{E}{1-E}T_o = \frac{E}{1-E}(T_o^{comm} + T_o^{comp}) \quad (8)$$

3.1.1 Distributed SMO (Dis-SMO)

Our Dis-SMO implementation is based on the idea of Cao's paper, we also include Catanzaro's improvements in the code. The serial runtime (T_1) of a SMO iteration is $2mn$ and its parallel runtime (T_p) per iteration is in Equation (9). Based on the terms in Table 2, the parallel overhead (T_o) can be obtained in Equation (10). The scaling model is in Table 4. This model is based on single-iteration SMO. However, the model 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 3). This will furthermore jeopardize the scalability for large-scale computation.

TABLE 3
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

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

$$T_o = 14PlogPt_s + [2nPlogP + 4P^3]t_w + 4m + 2P^2 + nP \quad (10)$$

TABLE 4
Scaling Comparison for Iso-efficiency Function

Method	Communication	Computation
1D Mat-Vec-Mul	$W = \Omega(P^2)$	$W = \Theta(P^2)$
2D Mat-Vec-Mul	$W = \Omega(P)$	$W = \Theta(P)$
Distributed-SMO	$W = \Omega(P^3)$	$W = \Omega(P^2)$
Cascade	$W = \Omega(P^3)$	$W = O(\sum_{k=1}^{logP} nL_kV_{k-1}2^k)$
DC-SVM	$W = \Omega(P^3)$	$W = O(\sum_{k=1}^{logP} nL_km2^k)$

3.1.2 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 4). For DC-SVM, since the K-means time is significantly less than the SVM time (Tables 9 to 14), 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 4).

$$W^{cascade,comm} = \Theta((\sum_{k=2}^{logP} n2^kV_k) + P^2V_{1+logP}) \quad (11)$$

$$W^{cascade,comp} = \Theta(n(\sum_{k=2}^{1+logP} L_kV_{k-1}2^k - 2Im)) \quad (12)$$

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 5 are in line with our analysis.

3.2 DC-Filter: Combination of Cascade and DC-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 9 to 14). 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

itself does not significantly increase the computation and communication overhead (Tables 9 to 14). For example, K-means converges in 7 loops and only costs less than 0.1% of the total runtime for processing the ijcnn dataset. However, we need to redistribute the data after K-means, which may increase the communication overhead. 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.

4 Communication-Efficient Design

4.1 CP-SVM: Clustering-Partition SVM

The node management for Cascade, DC-SVM, and DC-Filter are actually similar to each other (i.e. Fig. 1). Table 5 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 5. 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 10), the lower level can be fast and converge within $\Theta(1)$ iterations. For other datasets (e.g. Table 5), 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 use of all the computing nodes.

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

The analysis in this section is based on the Gaussian kernel with $\gamma > 0$ because it is the most widely used case [15]. 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 process these P sub-datasets, which is like the top layer of the DC-Filter algorithm.

TABLE 5
Profile of 8-node & 4-layer Cascade for a subset of ijcnn 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							

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 (in Euclidean distance) 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. 2 is the general flow of CP-SVM. CP-SVM is highly parallel because all the sub-problems are independent of each other. The communication overhead of CP-SVM is from K-means clustering and data distribution. CP-SVM generally is faster than the previous algorithms and its accuracy is closer to the SMO algorithm (Tables 9 to 14). However, in terms of scalability and speed, it is still not good enough. It is worth noting that K-means itself does not significantly increase the computation and communication overhead. However, we need to redistribute the data after K-means, which may increase the communication overhead.

4.2 Communication-Efficient SVM

Based on the profiling result in Fig. 6, 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. 6 has to handle 35,137 samples while processor 7 only needs to process 9,685 samples. Therefore, we need to replace K-means with a better partitioning algorithm that balance the load while maintaining accuracy. We design three versions of balanced partitioning algorithms and use them to build the communication-efficient algorithms.

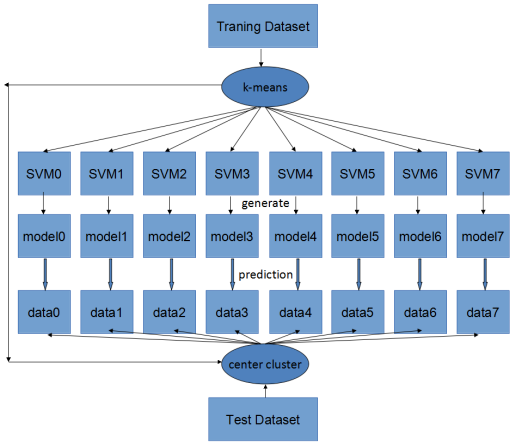


Fig. 2. General Flow for CP-SVM. In the training part, different SVMs process its own dataset independently. In the classification part, different models can make the prediction independently.

4.2.1 First Come First Served (FCFS) SVM

The goal of FCFS is to assign an equal number of samples m/P to each processor, where each sample is assigned to the processor with the closest center that has not already been assigned m/P samples. Centers are the locations of the first particles randomly chosen and assigned to each processor. (Other choices of centers are imaginable, such as doing K-means; this is the BKM algorithm below.) The detailed FCFS partitioning method is in Algorithm 3. Lines 1-4 of Algorithm 3 is the initiation phase: we randomly pick P samples from the dataset as the initial data centers. Lines 5-15 find the center for each sample. Lines 7-13 find the best under-loaded center for the i-th sample. Lines 16-22 recompute the data center by averaging all the samples assigned to each center. Recomputing the centers by averaging is optional because it will not necessarily make the results better. Fig. 3 is an example of Algorithm 3. From Fig. 4 we can observe that FCFS can partition the dataset in a balanced way. After FCFS partitioning, all the nodes have the same number of samples. Then the algorithm framework is the same as CP-SVM.

4.2.2 Balanced K-means (BKM) SVM

As mentioned above, the objective of BKM 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 Euclidean distance. The basic idea of this algorithm is to slightly rearrange the results of the original K-means algorithm. We will keep moving samples from the over-loaded centers to under-loaded centers till they are balanced. The balanced K-means partitioning method is detailed in Algorithm 4. Lines 1-4 of Algorithm 4 compute the K-means clustering of all the inputs. In lines 6-8, we calculate the Euclidean distance distance between every sample and every center: $dist[i][j]$ is the Euclidean distance between i-th sample and j-th center. The variable $balanced$ is the number of samples every center should have in the load-balanced situation. After the K-means clustering, some centers will have more than $balanced$ samples. In lines 9-26, 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 (lines 13-16). The id of the farthest sample is $maxind$. In lines 17-23, we find the closest under-loaded center to sample $maxind$. In lines 24-26, we move sample $maxind$ from its over-loaded center to the best under-loaded center. In lines 27-33, we recompute the data center by averaging the all the samples in a certain center. Recomputing the centers by averaging is optional. Fig. 5 is an example of Algorithm 4. After the BKM algorithm is finished and the load-balance is achieved, the algorithm framework is the same as CP-SVM.

Algorithm 3: First Come First Served Partitioning

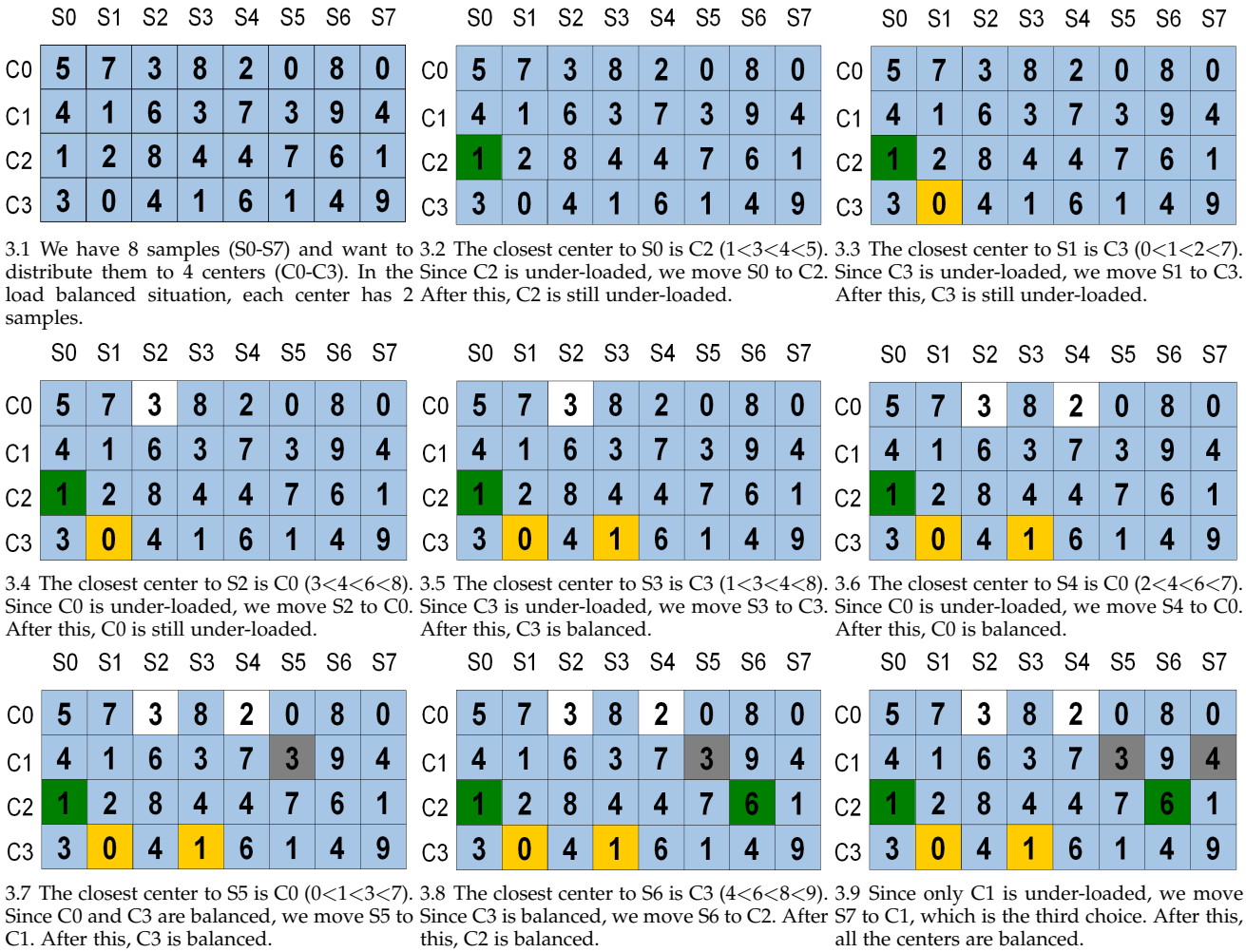
Input:
SA[i] is the i-th sample
m is the number of samples
P is the number of clusters (processes)

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

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

4.2.3 Communication-Avoiding SVM (CA-SVM)

For CA-SVM, 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. 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



3.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

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

3.8 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.

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

Fig. 3. This is an example of First Come First Served (FCFS) 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 than the original one, then it means that j-th sample belongs to i-th center.

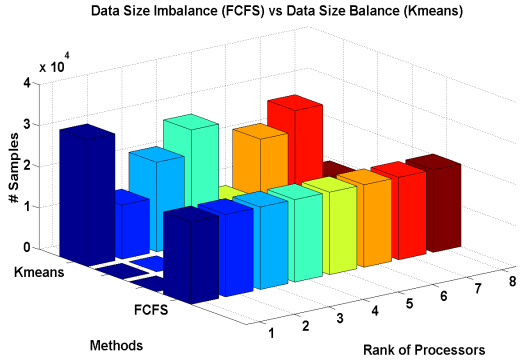
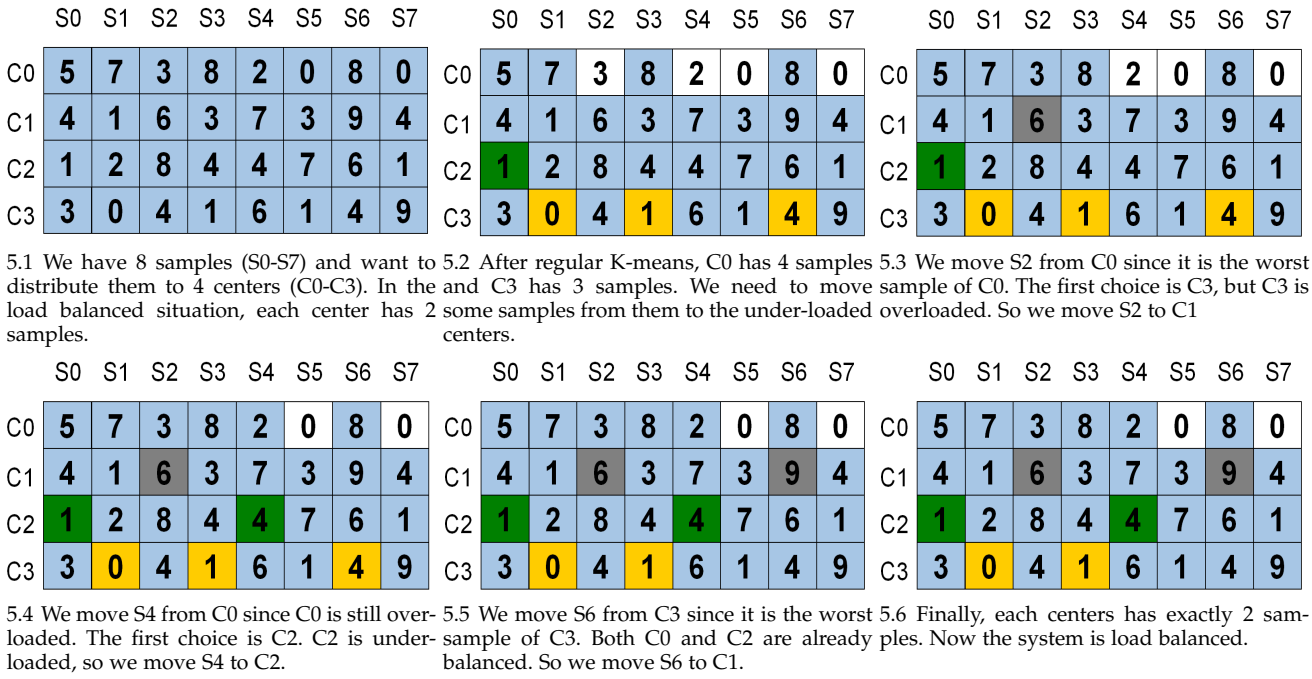


Fig. 4. The figure shows that the partitioning by K-means is imbalanced while the partitioning by FCFS is balanced. Specifically, each node has exactly 20,000 samples after FCFS partitioning. The test dataset is $face$ with 160,000 samples (361 features per sample). 8 nodes are used in this test.

only use MF_i to make prediction for $\hat{X}$. The communication overheads of CP-SVM and BKM-SVM are from the data transfer and distribution in K-means like partitioning algorithm. The communication overhead of FCFS-SVM is from the FCFS clustering method. In this new method, we replace the K-means variants or FCFS with a no-communication partition. Thus, we can also directly refer it 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. 8). All the results of CA-SVM in Section 5 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 9 to 14 show that it achieves significant speedup with comparable results.

The framework of CA-SVM is shown in Algorithm 5. The prediction process may need a little communication. However, both the data centers and test samples are pretty small compared with the training samples. Also, the overhead of single variable reduce operation is very low. This communication will not bring about significant overhead. On the other hand, the majority of SVM time is spent on the



5.1 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.

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

5.3 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

5.4 We move S4 from C0 since C0 is still overloaded. The first choice is C2. C2 is underloaded, so we move S4 to C2.

5.5 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.

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

Fig. 5. This is an example of Balanced K-means 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 than the original one, then it means that j-th sample belongs to i-th center.

training process. Like previous work (e.g. SMO, Cascade, DC-SVM), the focus of this paper is on optimizing the training process.

4.3 Initial Data Distribution

The major communication overhead of CP-SVM or BKM-SVM are from three parts: (1) The distributed K-means-like clustering algorithm. (2) Before K-means, if we do not use parallel IO, we read the data from root node, then distribute the data to all the nodes; if we use the parallel IO, each node reads m/p samples. Then the algorithm does a gather operation to make the root node have all the data. (3) After the clustering part, the root node gets the redistribution information and distributes the data to all the nodes.

For the parallel IO version of part (2), the reason why we have to gather all the data to the root node is that we use the CSR (Compressed Row Storage) format to store the data to reduce the redundant memory requirement. If we do not use the CSR data format, we can not process high-dimensional data sets like webspam [24] in Table 8. Because of the CSR format implementation, we must get the global row index and data index on a single node.

For CA-SVM (casvm2 implementation) we use both parallel IO and CSR input format by assuming the sparse input matrix has been prepartitioned into P disjoint row blocks, each in CSR format. The sub-problems of CA-SVM are independent of each other. Each sub-problem generates its data center (CT_1, CT_2, CT_3, ..., CT_p) and its own model ($model_1$, $model_2$, $model_3$, ..., $model_p$). For an unknown test sample $\hat{x}$, each node will get a copy of $\hat{x}$. Each node computes the distance between $\hat{x}$ and its data center. Let us use $dist_1$, $dist_2$, $dist_3$, ..., $dist_p$ to represent the distances. If $dist_i$ is the smallest one, then we will use $model_i$ to make prediction for

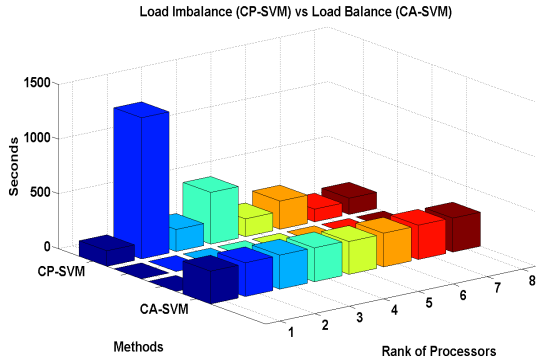


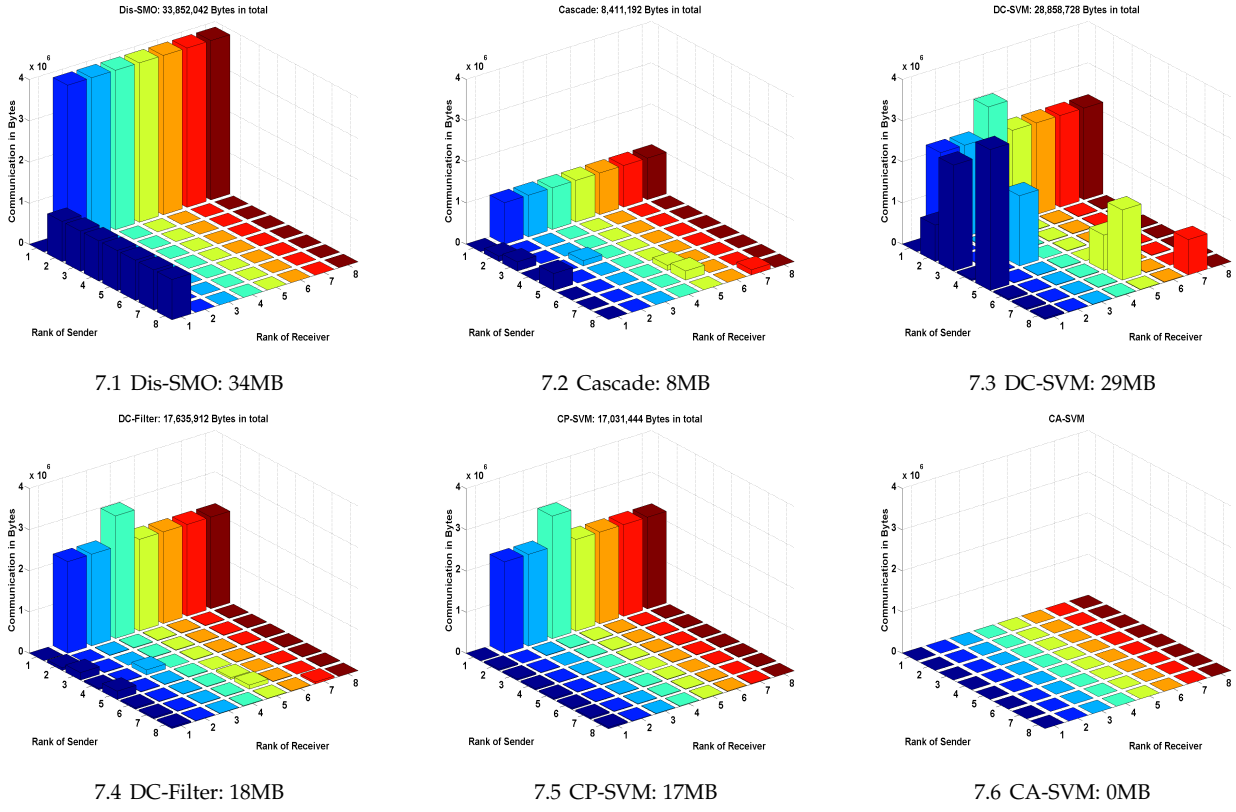
Fig. 6. The figure shows that CP-SVM is load imbalanced while CA-SVM is load-balanced. The test dataset is $epsilon$ with 128,000 samples (2,000 nnz per sample). 8 nodes are used in this test.

$\hat{x}$. It is only necessary to do a reduction operation. In a large datacenter, we expect the user's data to be distributed across different nodes. Considering the load balance issue, we also assume the data should be distributed in a nearly balanced way. Note that even if it is not balanced, CA-SVM can still work (in a slightly inefficient way). On the other hand, from Fig. 8, we can observe that the performance of casvm1 (serial IO) is close to the performance of casvm2 (parallel IO).

4.4 Communication Pattern

4.4.1 *Communication Modeling*

We only give the results because the space is limited, the detailed proofs are in [7]. The formulas for communication volume are in Table 6. The experimental results in the table are based on the dense ijcnn dataset on 8 Hopper nodes [6]. The terms used in the formulas are in Table 2. We can use the



7.1 Dis-SMO: 34MB 7.2 Cascade: 8MB 7.3 DC-SVM: 29MB

7.4 DC-Filter: 18MB 7.5 CP-SVM: 17MB 7.6 CA-SVM: 0MB

Fig. 7. Communication Patterns of different approaches. The data is from running the 6 approaches on 8 nodes with the same 5MB real-world dataset (subset of ijcnn 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. The communication pattern of BKM-SVM is similar to that of CP-SVM. The communication pattern of FCFS-SVM is similar to that of cascade without point-to-point communication.

formulas to predict the communication volume for a given method. For example, for ijcnn dataset, m is 48,000, n is 13, and s is 4474. We can predict the communication volume of Cascade is about $3 \times (48000 \times 13 + 48000 + 4474 \times 13) \times 4\text{B} = 8.4\text{MB}$. Our experimental result is 8.41MB, which means the prediction for Cascade is very close to the actual volume.

4.4.2 Point-to-Point profiling

Fig. 7 shows the communication patterns of these six approaches for a subset of ijcnn. 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 7 we can observe that CP-SVM is more efficient than Cascade since the volume of communication per operation is higher.

4.4.3 Ratio of Communication to Computation

Fig. 8 shows the communication and computation time for different methods applied to a subset of ijcnn. From Fig. 8 we can observe that our algorithms significantly reduce the volume of communication and the ratio of communication to computation. This is important since the existing supercomputers [26] are generally more suitable for computation-intensive than communication-intensive applications. Besides, less communication can greatly reduce the power consumption [27]. Table 6 shows that the communication

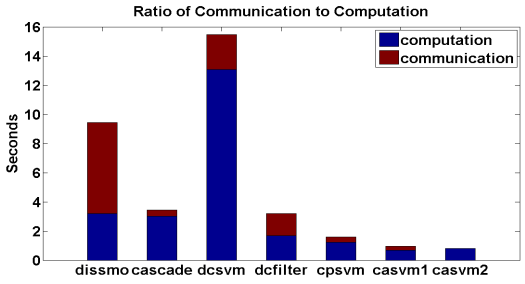


Fig. 8. The ratio of computation to communication. The experiment is based on a subset of ijcnn dataset. To give a fair comparison, we implemented two versions of CA-SVM. **casvm1** means that we put the initial dataset on the same node, which needs communication to distribute the dataset to different nodes. **casvm2** means that we put the initial dataset on different nodes, which needs no communication.

volumes of DC-Filter and CP-SVM are similar. However, Fig. 8 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 by collective operations (e.g. Scatter) but DC-Filter has some point-to-point communications (e.g. Send/Recv) on the lower levels (Fig. 1).

5 Experimental Results and Analysis

The test datasets in our experiments are shown in Table 8, and they are from real-world applications. Some of the

Algorithm 4: Balanced K-means Partitioning

Input:
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
CS[i] is the size of i-th cluster

Randomly pick P samples from m samples (RS[1:P])
for $i \in 1:P$ **do**
  $CT[i] = RS[i]$
do kmeans clustering on all input data
$balanced = m/P$
for $i \in 1:m$ **do**
  for $j \in 1:P$ **do**
    $dist[i][j]$ = EuclideanDistance($SA[i]$, $CT[j]$)
for $j \in 1:P$ **do**
  while $CS[j] >$ *balanced* **do**
    $maxdist = 0$
    $maxind = 0$
    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**
  $CT[i] = CT[i] \,/\, CS[i]$

Algorithm 5: CA-SVM (casvm2 in Fig. 8)

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: On j-th node, d_j = dist($\hat{X}$, CT_j)
2: Global reduce: $id = argmin_j(d_j)$
3 If rank $==$ id, then use MF_j to make prediction for $\hat{X}$

datasets are sparse, we use CSR format in our implementation for all the datasets. 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 [16]. The K-means partitioning in DC-SVM, DC-Filter, CP-SVM, and BKM are distributed versions, which achieved the same partitioning result and comparable performance with Liao's implementation [28]. Our experiments are conducted on NERSC Hopper and Edison systems [6].

TABLE 6
Modeling of Communication Volume based on a subset of ijcnn [25]

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

TABLE 7
Efficiency of Communication based on a subset of IJCNN [25]

Method	Volume	Comm Operations	Volume/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

TABLE 8
The Test Datasets

Dataset	Application Field	#samples	#features
adult [9]	Economy	32,561	123
epsilon [29]	Character Recognition	400,000	2,000
face [30]	Face Detection	489,410	361
gisette [31]	Computer Vision	6,000	5,000
ijcnn [25]	Text Decoding	49,990	22
usps [32]	Transportation	266,079	675
webspam [24]	Management	350,000	16,609,143

5.1 Speedup and Accuracy

From Tables 9 to 14, we observe that CA-SVM can achieve 3× - 16× (7× on average) speedups over distributed SMO algorithm with comparable accuracies. The Init time includes the partition time like K-means, and the Training time includes the redistribution and the SVM training processes. For Cascade, DC-SVM, and DC-Filter, the training process includes the level-by-level (point-to-point) communications. The accuracy loss ranges from none to 3.6% (1.3% on average). According to previous work [18], the accuracy loss in this paper is small and tolerable for practical applications. Additionally, we can observe that CA-SVM reduces the number of iterations, which means it is intrinsically more efficient than other algorithms. For DC-SVM, DC-Filter, CP-SVM, and BKM the majority of the Init time is spent on K-means clustering. K-means itself does not significantly increase the computation or communication cost. However, we need to redistribute the data after K-means, which increases the communication cost.

5.2 Strong Scaling and Weak Scaling

Tables 15 and 16 show the results of strong scaling time and efficiency. We observe that the strong scaling efficiency of CA-SVM is increasing with the number of processors. The

TABLE 9
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-SVM	83.3%	1,482	1.61s (0.057, 1.54)
FCFS-SVM	83.6%	1,621	1.21s (0.005, 1.19)
CA-SVM	83.1%	1,160	0.96s (4e-4, 0.95)

TABLE 10
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-SVM	98.0%	2,209	88.9s (17.8, 71.0)
FCFS-SVM	98.0%	2,194	65.3s (0.43, 64.9)
CA-SVM	98.0%	2,268	66.4s (0.08, 66.4)

TABLE 11
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-SVM	95.8%	452	4.75s (2.29, 2.46)
FCFS-SVM	96.5%	441	2.48s (0.07, 2.41)
CA-SVM	94.0%	487	2.9s (0.014, 2.87)

TABLE 12
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-SVM	98.3%	5,004	3.0s (0.08, 2.87)
FCFS-SVM	98.5%	7,450	3.6s (0.005, 3.55)
CA-SVM	98.0%	6,110	3.4s (3e-4, 3.4)

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. For the weak scaling results in Tables 17 and 18, we 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 constant 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.

5.3 Efficiency of CA-SVM

Here, we use m for simplicity to refer to the problem size and P to refer to the number of nodes. To be more precise, let $t(m, P)$ be the per-iteration time, which is a function of m and P; and let $i(m, P)$ be the number of iterations, a function of m and P. For Dis-SMO, we observe $i(m, P) = \Theta(m)$, that is, there is no actual dependence on P. Then, the total time should really be

$$T(m, P) = i(m, P) \times t(m, P)$$

Thus, the efficiency becomes

$$E(m, P) = \frac{i(m, 1) \times t(m, 1)}{P \times i(m, P) \times t(m, P)}$$

For Dis-SMO, $i(m, 1) = i(m, P)$, which means

$$E(m, P) = \frac{t(m, 1)}{P \times t(m, P)}$$

If the per-iteration time scales perfectly — meaning $t(m, P) = t(m, 1)/P$ — the efficiency of SMO should be $E(m, P) = 1$ in theory. For CA-SVM, each node is actually an independent SVM. Thus we expect that $i(m, P) = \Theta(m/P)$ because each node only trains m/P samples. In other words, each node is a SMO problem with m/P samples. Therefore, $i(m, 1)$ is close to $P \times i(m, P)$, which means

$$E(m, P) = \frac{t(m, 1)}{t(m, P)}$$

On the other hand $t(m, 1)$ is close to $P \times t(m, P)$ because each node only has m/P samples. In this way, we get

$$E(m, P) = \frac{P \times t(m, P)}{t(m, P)} = P$$

This means the efficiency of CA-SVM is close to P in theory. Usually, we expect efficiency to lie between 0 and 1. The way we set this up is perhaps not quite right – the sequential baseline should be the best sequential baseline, not the naive (plain SMO) one. If we execute CA-SVM sequentially by simulating P nodes with only 1 node, then the sequential time would be

$$P \times (i(m, 1)/P \times t(m, 1)/P) = i(m, 1) \times t(m, 1)/P$$

So then $E(m, P)$ would approach 1 rather than P. Put another way, CA-SVM is better than SMO, even in the sequential case. That is, we can beat SMO by running CA-SVM to simulate P nodes using only 1 node.

TABLE 13
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-SVM	98.9%	6,122	30.4s (2.02, 28.4)
FCFS-SVM	99.0%	6,513	30.1s (0.04, 29.7)
CA-SVM	98.9%	6,435	24.5s (0.0018, 24.5)

TABLE 14
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-SVM	98.5%	14,208	24.3s (1.12, 23.0)
FCFS-SVM	98.3%	12,369	21.2s (0.03, 21.0)
CA-SVM	96.9%	10,430	17.3s (0.003, 17.3)

TABLE 15
Strong Scaling Time for epsilon dataset on Hopper: 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
BKM-SVM	1031s	355s	137s	88.6s	48.4s
FCFS-SVM	1064s	303s	85.8s	25.4s	15.6s
CA-SVM	1095s	313s	86s	23s	6s

TABLE 16
Strong Scaling Efficiency for epsilon dataset on Hopper: 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%
BKM-SVM	100%	145.2%	188.1%	145.5%	133.1%
FCFS-SVM	100%	175.6%	310.0%	523.6%	426.3%
CA-SVM	100%	175.0%	319.5%	603.0%	1068.7%

5.4 The Approximation Accuracy

5.4.1 The Mathematical Derivation

The intuition behind the divide-and-conquer heuristic is this: Suppose we can partition (say by K-means) the m training samples into p disjoint clusters $D_1 \cup D_2 \cup \cdots \cup D_p$, where the samples in each D_i are close together, and far from other D_j. Then classifying a new sample $\hat{X}$ may be done by (1) finding the cluster D_i to which $\hat{X}$ is closest, and (2) using the samples inside D_i to classify $\hat{X}$ (say by an SVM using only D_i as training data). Since we use nearby data to classify $\hat{X}$, we expect this to work well in many situations.

In this section we quantify this observation as follows: Let K be the m-by-m kernel matrix, with $K_{i,j} = K(X_i, X_j)$, permuted so that the first $|D_1|$ indices are in D_1, the next $|D_2|$ indices are in D_2, etc. Let K_1 be the leading $|D_1|$-by-$|D_1|$ diagonal submatrix of K, K_2 the next $|D_2|$-by-$|D_2|$ diagonal submatrix, etc. Let $\widetilde{K} = \mathrm{diag}(K_1, K_2, ..., K_p)$ be the submatrix of K consisting just of these diagonal blocks. Then we may ask how well $\widetilde{K}$ "approximates" K. In the extreme case, when $\widetilde{K} = K$ and all samples in each cluster have the same classification, it is natural to assign $\hat{X}$ the same classification as its closest cluster. As we will see, depending both on the kernel function $K()$ and our metric for how we measure how well $\widetilde{K}$ approximates K, our algorithm for finding clusters D_i will naturally improve the approximation. One can also view choosing D_i as a graph partitioning problem, where $K_{i,j}$ is the weight of edge (i, j) [33] [34] [35].

In the simple case of a linear kernel $K(X_i, X_j) = X_i^T \cdot X_j$, a natural metric to try to maximize (inspired by [33]) is

$$J_1 = \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} K_{i,j}$$

Letting $X = [X_1, ..., X_m]$, it is straightforward to show that

$$\|X\|_F^2 - J_1 = \sum_{k=1}^{p} \sum_{i \in D_k} \|X_i - \mu_k\|_2^2 \equiv J^{kmeans}$$

where $\mu_k = |D_k|^{-1} \sum_{i \in D_k} X_i$ is the mean of cluster D_k. It is also known that the goal of K-means is to choose clusters to minimize the objective function J^{kmeans}, i.e. to maximize J_1. Since the polynomial and sigmoid kernels are also increasing functions of $X_i^T \cdot X_j$, we also expect K-means to choose a good block diagonal approximation for them.

Now we consider the Gaussian kernel, or more generally kernels for which $K(X_i, X_j) = f(\|X_i - X_j\|_2)$ for some function $f()$. (The argument below may also be generalized to shift-invariant kernels $K(X_i, X_j) = f(X_i - X_j)$.) Now we use the metric

$$J_2 = \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} K_{i,j}^2$$

to measure how well $\widetilde{K}$ approximates K, and again relate minimizing J^{kmeans} to maximizing J_2.

The mean value theorem tells us that $K(X_i, X_j) = f(\|X_i - X_j\|_2) = f(0) + f'(s)\|X_i - X_j\|_2$ for some $s \in [0, \|X_i - X_j\|_2]$. For the Gaussian Kernel $f(s) = e^{-\gamma s^2}$ so $f'(s) = -2\gamma s e^{-\gamma s^2}$, which lies in the range $0 > f'(s) \geq -\sqrt{2\gamma} e^{-1/2} \equiv R$. Thus $1 \geq K(X_i, X_j) \geq 1 + R\|X_i - X_j\|_2$,

TABLE 17
Weak Scaling Time for epsilon dataset on Hopper: 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
BKM-SVM	6.72s	9.14s	16.6s	31.2s	48.4s
FCFS-SVM	6.14s	6.71s	6.88s	10.2s	15.6s
CA-SVM	6.1s	6.2s	6.2s	6.4s	6.4s

TABLE 18
Weak Scaling Efficiency for epsilon dataset on Hopper: 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%
BKM-SVM	100%	73.5%	40.5%	21.5%	13.9%
FCFS-SVM	100%	91.5%	89.2%	60.2%	39.4%
CA-SVM	100%	98.9%	97.8%	96.0%	95.3%

which in turn implies $1 \leq (K(X_i, X_j) - R\|X_i - X_j\|^2)^2 \leq 2K^2(X_i, X_j) + 2R^2\|X_i - X_j\|_2^2$ or $K^2(X_i, X_j) \geq \frac{1}{2} - R^2\|X_i - X_j\|_2^2$. Substituting this into the above expression for J_2 and simplifying we get

$$\begin{aligned} J_2 &\geq \frac{m}{2} - R^2 \sum_{k=1}^{p} |D_k|^{-1} \sum_{i,j \in D_k} \|X_i - X_j\|_2^2 \\ &= \frac{m}{2} - 2R^2 J^{kmeans} \end{aligned}$$

So again minimizing J^{kmeans} means maximizing (a lower bound for) J_2.

5.4.2 The Block Diagonal Matrix

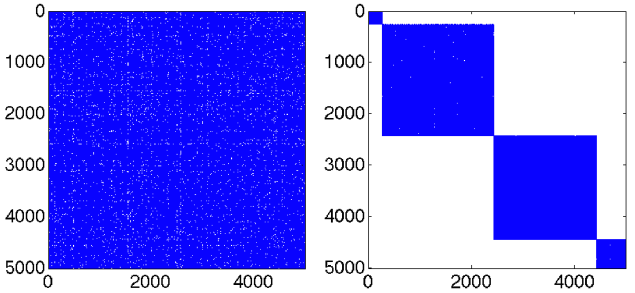
For the experiment, we use 5,000 samples from the UCI covtype dataset [36]. The kernel matrix is 5,000-by-5,000 with 458,222 nonzeroes. In Fig. 9, the first part is the original kernel matrix, the second part is the kernel matrix after clustering. From these figures we can observe that the kernel matrix is block-diagonal after clustering. Let us use Fn to represent the Frobenius norm (F-norm) and $\widehat{Fn}$ means the F-norm of the original kernel matrix. The definition of Error (kernel approximation error) is given by

$$Error = \frac{|\widehat{Fn} - Fn|}{\widehat{Fn}} \tag{14}$$

The γ in Table 19 is defined in the Gaussian kernel of Table 1. From Table 19 we can observe that when γ is small, the approximation error of Random method is much larger than the approximation error of Clustering method. Based on F-norm, the approximation matrix by clustering method is almost the same with the original matrix. The approximation error of Random partition is decreasing when the γ is increasing. There, if we can use large γ parameter in the real-world applications, the approximation error of Random partition (i.e. CA-SVM) can be very low.

TABLE 19
The error of different kernel approximations. The definition of Error is in Equation (14).

γ	Original F-norm / Error	Random F-norm / Error	Clustering F-norm / Error
20	154.239899 / 0.0%	98.661888 / 36.0%	154.279709 / 0.0%
30	117.745422 / 0.0%	85.005592 / 27.8%	117.765900 / 0.0%
40	100.739906 / 0.0%	79.307716 / 21.3%	100.755119 / 0.0%
50	91.576241 / 0.0%	76.469208 / 16.5%	91.585464 / 0.0%
60	86.123299 / 0.0%	74.869514 / 13.1%	86.129494/ 0.0%
70	82.630066 / 0.0%	73.882416 / 10.6%	82.635559/ 0.0%



9.1 Original Kernel Matrix 9.2 Kernel Matrix After Clustering

Fig. 9. We use the 5,000 samples from the UCI covtype dataset [36] for this experiment. The kernel matrix is 5,000-by-5,000 with 458,222 nonzeroes. The first figure is the original kernel matrix, the second figure is the kernel matrix after clustering. From these figures we can observe that the kernel matrix is block-diagonal after the clustering algorithm.

5.5 Tradeoffs

CA-SVM is the only algorithm presented that can achieve nearly zero communication. Thus, CA-SVM should be the fastest one in general. However, CA-SVM also suffers the most loss in accuracy, if surprisingly little. Basically, our methods are for applications that most need to be accelerated or scaled up, and do not require the highest accuracy. For applications that require both accuracy and speed, using FCFS or BKM is a better choice. So there is a trade-off between time (communication) and accuracy.

5.6 Accuracy of CA-SVM

We conduct the following two experiments: (1) **Random-Assign**: assign the test sample not to the processor with the closest data center, but to a random processor. (2) **Sub-Sampling**: pick random subset of 1/p-th of all the data, and just use it to build an SVM for all the test samples. Let use the ijcnn dataset (Table 8) as an example. We use 8 nodes and divide the dataset into 8 parts for CA-SVM. After the experiment, the accuracy of **Random-Assign** is 85% (77987/91701), the accuracy of **Sub-Sampling** is 68% (62340/91701), and the accuracy of **CA-SVM** is 98%

(89852/91701). **Sub-Sampling** has the lowest accuracy because it uses a much smaller dataset (6k training samples) and thus can only build an inferior model. The difference between **Random-Assign** and **CA-SVM** is that each model of **Random-Assign** roughly receives the same number of test samples because it used the random assignment method. However, a test sample of **CA-SVM** will be sent to its closest cluster rather than a random cluster. This makes different nodes of **CA-SVM** have different numbers of test samples. For example, the 0-th node of **Random-Assign** receives 11,518 test samples while the 0-th node of **CA-SVM** only receives 5,037 test samples.

6 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, FCFS, and CA-SVM) through step-by-step optimizations. BKM, FCFS, and CA-SVM all reduce communication significantly compared to previous methods, with CA-SVM avoiding all communication. We manage to 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.

Acknowledgment

We would like to thank Prof. Inderjit S. Dhillon, Dr. Cho-Jui Hsieh, and Dr. Si Si at UT Austin for their helpful discussions in machine learning with us. This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0010200; by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008700 and AC02-05CH11231; by DARPA Award Number HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle and Samsung. Other industrial sponsors include Mathworks and Cray. L.S. was supported in part by NSF/NIH BIGDATA 1R01GM108341, ONR N00014-15-1- 2340, NSF IIS-1218749, NSF CAREER IIS-1350983, Intel and NVIDIA. The funding information in [37] maybe also relevant.

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.

[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] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference, October 24-28, 2011, Miami, Florida*. University of Miami, 2011, pp. 591–596.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

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

[27] 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.
[28] W.-K. Liao. (2013) Parallel k-means. [Online]. Available: http://users.eecs.northwestern.edu/ wkliao/Kmeans/
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 8, pp. 888–905, 2000.
[34] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.
[35] S. Si, C.-J. Hsieh, and I. Dhillon, "Memory efficient kernel approximation," in *Proceedings of The 31st International Conference on Machine Learning*, 2014, pp. 701–709.
[36] U. I. M. L. Repository. Covertype data set. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Covertype
[37] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "Ca-svm: Communication-avoiding support vector machines on distributed systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 847–859.

Biography Yang You is a PhD student at UC Berkeley Computer Science Division. His advisor is Prof. James Demmel, who is leading the BeBOP group. Yang You went to college at September of 2009, and he received his Master Degree in Computer Science from Tsinghua University at July of 2015. His research interests include Parallel Computing, Distributed Systems, and Machine Learning. He is a winner of IPDPS 2015 Best Paper award. Yang You is a Siebel Scholar.

James Demmel is Computer Science Division Chair, and EECS Associate Chair of UC Berkeley. He is also the Dr. Richard Carl Dehmel Distinguished Professor of Computer Science and Mathematics at the UC Berkeley. His personal research interests are in numerical linear algebra, high performance computing, and communication avoiding algorithms in general. He is known for his work on the LAPACK and ScaLAPACK linear algebra libraries. He is a member of the National Academy of Sciences, National Academy of Engineering, a Fellow of the ACM, IEEE and SIAM, and winner of the IEEE Computer Society Sidney Fernbach Award, the SIAM J. H. Wilkinson Prize in Numerical Analysis and Scientific Computing, IPDPS Charles Babbage Award, and numerous best paper prizes, including being the only 3-time winner of the SIAM Linear Algebra (SIAG/LA) Prize. He was an invited speaker at the 2002 International Congress of Mathematicians and the 2003 International Congress on Industrial and Applied Mathematics.

Kent Czechowski is a PhD student at Georgia Tech. His area of research is High Performance Computing (HPC). In particular, he focus on Algorithm-Architecture co-design, performance modeling for GPU/many-core architectures, and Distributed Algorithms. He is a student member of IEEE and ACM.

Le Song is an Associate Professor at Georgia Tech in the School of CSE. He received his Doctoral degree in computer science at the University of Sydney, Australia. He was also a PhD. student with the Statistical Machine Learning Program at NICTA. Since Summer 2008, He was a Lane postdoc fellow at Carnegie Mellon Univeristy, working on machine learning and computational biology projects with Eric Xing, Carlos Guestrin, Geoff Gordon and Jeff Schneider. Right before he came to Georgia Tech, he spent some time as a research scientist in Fernando Pereira's group at Google Research.

Rich Vuduc is an Associate Professor at Georgia Tech in the School of CSE. His research lab, the HPC Garage (hpcgarage.org), is interested in all-things-high-performance-computing, with an emphasis on parallel algorithms, performance analysis, and performance tuning. He is a member of the DARPA Computer Science Study Panel, recipient of the NSF CAREER Award, and co-recipient of the Gordon Bell Prize (2010). His lab's work has received a number of best paper nominations and awards, including most recently the 2012 Best Paper Award from the SIAM Conference on Data Mining.