CA-SVM: Communication-Avoiding Support Vector Machines on Clusters

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 (yang.you.cs@gmail.com) demmel@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× (7× 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 \( \sqrt{W} \) 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× (7× 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
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 set 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):

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

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

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

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:

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

$\dot{f}_i = f_i + \Delta \alpha_{i, high} y_{high} K_{high,i} + \Delta \alpha_{i, low} y_{low} K_{low,i} \tag{5}$

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

$\Delta \alpha_{high} = -y_{low} y_{high} \Delta \alpha_{low} \tag{7}$

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. \(b_{\text{high}} = -1, t_{\text{high}} = \min \{i : y_i = 1\}\).
4. \(b_{\text{low}} = 1, t_{\text{low}} = \min \{i : y_i = -1\}\).
5. Update \(\alpha_{\text{high}}\) and \(\alpha_{\text{low}}\) according to Equations (6) and (7).
6. Update \(f_i\) according to Equation (5), \(\forall i \in \{1, 2, ..., m\}\).
7. \(I_{\text{high}} = \{i : 0 < \alpha_i < C \land y_i > 0, \alpha_i = 0 \land y_i < 0, \alpha_i = C\}\).
8. \(I_{\text{low}} = \{i : 0 < \alpha_i < C \land y_i > 0, \alpha_i = C \land y_i < 0, \alpha_i = 0\}\).
9. \(t_{\text{high}} = \arg\min \{f_i : i \in I_{\text{high}}\}\).
10. \(t_{\text{low}} = \arg\max \{f_i : i \in I_{\text{low}}\}\).
11. \(b_{\text{high}} = \min \{f_i : i \in I_{\text{high}}\}, b_{\text{low}} = \max \{f_i : i \in I_{\text{low}}\}\).
12. Update \(\alpha_{\text{high}}\) and \(\alpha_{\text{low}}\) according to Equations (6) and (7).
13. If \(b_{\text{low}} > b_{\text{high}}\), 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 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 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 substep for DC-SVM, we review it here.

The objective of K-means clustering is to partition a dataset \(TD\) into \(k \in \mathbb{Z}^+\) 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 center \(CT_1, CT_2, ..., CT_k\) randomly.
3. set \(\delta = 0\).
4. For every \(i\), set \(c_i = \arg\min \|X_i - CT_j\|\).
5. If \(c_i\) has been changed, \(\delta = \delta + 1\).
6. For every \(j\), set \(CT_j = \frac{\sum_{i=1}^{\delta} \delta_j}{\sum_{i=1}^{\delta} 1(s_j = j)} X_i\), \(j \in \{1, 2, ..., k\}\).
7. If \(\delta / m > \text{threshold}\), 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...
partitioned SMO schemes this paper considers, so we leave such comparisons for future work.

III. RE-DESIGNING 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)

$$E = T_1/(pT_p)$$

$E$ is the desired scaling efficiency. Specifically, $T_1 = tcW$ where $tc$ is the time per flop. In this paper, to make it simple, we normalize so that $tc = 1$. In the same way, $ts$ and $tw$ 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 ($P = O(\sqrt{W})$).

$$W = KT_o \text{ with } K = \frac{E}{1 - E}$$ (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.

$$T_p = 14\log P + 2n\log P + 4P^2t_w + \frac{2mn + 4m + 2P + n}{P} + 2P + n$$ (9)

$$T_o = 14P\log P + 2n\log P + 4P^3t_w + 4m + 2P^2 + nP$$ (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 IX to XIV), 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).

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

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

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

Table II

<table>
<thead>
<tr>
<th>Terms for Performance Modelling</th>
<th>$m$; $n$; $P$</th>
<th>$T_1$; $T_p$</th>
<th>$t_s$; $t_w$</th>
<th>$V_k$</th>
<th>$L_k$</th>
<th>$P_k$</th>
<th>$W$; $T_o$</th>
<th>$s$; $I$; $k$</th>
<th># SVs; # SVM iters; # K-means iters</th>
</tr>
</thead>
</table>

Table III

| 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 | $W = \Omega(P^3)$ | $W = \Omega(\sum_{k=1}^{logP} nL_kV_{k-1}2^k)$ |
| DC-SVM | $W = \Omega(P^3)$ | $W = \Omega(\sum_{k=1}^{logP} nL_km2^k)$ |
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 IX to XIV). 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 IX to XIV), which is the major reason why we can use it. For example, K-means converges in 7 iterations and costs less than 0.1% of the total runtime for processing the ijcn 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 X), the lower level can be fast and converge within Θ(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.

\[
\sum_{l=1}^{1+\log_P} \frac{(\text{time of layer } l) \times (#\text{nodes of layer } l)}{\sum_{l=1}^{1+\log_P} (\text{time of layer } l)}
\]

(13)

Table V
PROFILE OF 8-NODE & 4-LAYER CASCADE FOR A TOY DATASET

<table>
<thead>
<tr>
<th>level 1\textsuperscript{st}</th>
<th>node rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td></td>
</tr>
<tr>
<td>time: 5.49s</td>
<td>4.87</td>
<td>4.92</td>
<td>4.90</td>
<td>4.68</td>
<td>5.12</td>
<td>5.10</td>
<td>5.49</td>
<td>4.71</td>
<td></td>
</tr>
<tr>
<td>iter: 6168</td>
<td>5648</td>
<td>5712</td>
<td>5666</td>
<td>5415</td>
<td>5936</td>
<td>5904</td>
<td>6168</td>
<td>5453</td>
<td></td>
</tr>
<tr>
<td>SVs: 5532</td>
<td>746</td>
<td>715</td>
<td>717</td>
<td>718</td>
<td>688</td>
<td>707</td>
<td>721</td>
<td>699</td>
<td></td>
</tr>
<tr>
<td>samples</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td>4516</td>
<td></td>
</tr>
<tr>
<td>time: 1.58s</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td>iter: 7485</td>
<td>7485</td>
<td>7211</td>
<td>7111</td>
<td>6713</td>
<td>7035</td>
<td>7035</td>
<td>7035</td>
<td>7035</td>
<td></td>
</tr>
<tr>
<td>SVs: 5050</td>
<td>1292</td>
<td>1263</td>
<td>1256</td>
<td>1239</td>
<td>1239</td>
<td>1239</td>
<td>1239</td>
<td>1239</td>
<td></td>
</tr>
<tr>
<td>samples</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td>7468</td>
<td></td>
</tr>
<tr>
<td>time: 4.87s</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td>4.87</td>
<td></td>
</tr>
<tr>
<td>iter: 9081</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td>8975</td>
<td></td>
</tr>
<tr>
<td>SVs: 4699</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td>2388</td>
<td></td>
</tr>
<tr>
<td>samples</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td>5904</td>
<td></td>
</tr>
<tr>
<td>iter: 14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td>14052</td>
<td></td>
</tr>
<tr>
<td>SVs: 4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td>4475</td>
<td></td>
</tr>
</tbody>
</table>

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 $X$, only the support vectors close to $X$ 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-
If we have $P$ designs, a machine node corresponds to a clustering center. The objective of these three algorithms is to improve K-means. We use them to build the communication-avoiding algorithms. We design three versions of balanced partitioning algorithms and replace K-means with a better partitioning algorithm. We need load balancing 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 the 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. Lines 1-4 of Algorithm 3 are the initiation phase: we randomly pick $P$ samples from the dataset as the initial data centers. Lines 5-14 find the center for each sample. Lines 8-12 find the best under-loaded center for the $i$-th sample. Lines 15-21 recompute 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.

**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 ($RS[1:P]$)
2. for $i \in 1 : P$ do
3.     $CT[i] = RS[i]$
4.     $balanced = m/P$
5. for $i \in 1 : m$ do
6.     $mindis = \infty$
7.     $minind = 0$
8. for $j \in 1 : P$ do
9.     $dist = \text{Distance}(SA[i], CT[j])$
10. if $dist < mindis$ and $CS[j] < \text{balanced}$ then
11.     $mindis = dist$
12.     $minind = j$
13. $CS[minind]++$
14. $MB[i] = minind$
15. for $i \in 1 : P$ do
16.     $CT[i] = 0$
17. for $i \in 1 : m$ do
18.     $j = MB[i]$
19.     $CT[j] += SA[i]$
20. for $i \in 1 : P$ do
21.     $CT[i] = CT[i] / CS[i]$

To do FCFS algorithm in parallel, we use the divide-and-conquer approach. The basic idea is to convert one $m \rightarrow P \times$
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 \((RS[1:P])\)
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. /* each process operates on its own subset of \(SA[]\) */
   9. for \(i \in 1 : pm\) do
      10. \(mindis = \infty\)
      11. for \(j \in 1 : P\) do
          12. \(dist = Distance(SA[i], CT[j])\)
          13. if \(dist < mindis \text{ and } CS[j] < balanced\) 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

The parallel FCFS partitioning approach is detailed in Algorithm 4. Specifically, lines 4-7 of Algorithm 4 are the dividing phase. We distribute all the samples evenly to all the nodes. All the nodes will have a copy of data centers. Lines 8-22 of Algorithm 4 are the parallel phase, during which all the nodes do the same thing independently. Each node will finish its own FCFS algorithm. Lines 23-27 of Algorithm 4 are the conquer phase. In lines 18-22, we need to use all the samples on all the nodes to recompute the data centers.

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 Euclidean 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 algorithm. We will keep moving samples from the over-loaded center to an under-loaded center till they are balanced. The balanced K-means partitioning method is detailed in Algorithm 5. Lines 1-4 of Algorithm 5 is the K-means clustering. In lines 6-8, we calculate the Euclidean 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 9-27 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 (lines 14-17 of Algorithm 5). The id of the farthest sample is \(maxind\). In lines 20-24 of Algorithm 5, we find the closest under-loaded center to sample \(maxind\). In lines 25-27, we move sample \(maxind\) from its over-loaded center to the best under-loaded center. In lines 15-21, we recompute the data center by averaging the all the samples in a certain center. Recomputing the centers by averaging is optional. Like FCFS partitioning algorithm, we 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
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

1. Randomly pick $P$ samples from $m$ samples ($RS[1:P]$)
2. for $i \in 1 : P$ do
3.   $CT[i] = RS[i]$
4. do kmeans clustering
5.   balanced = $m/P$
6.   for $i \in 1 : m$ do
7.     for $j \in 1 : P$ do
8.       $dist[i][j] = Distance(SA[i], CT[j])$
9.   for $j \in 1 : P$ do
10.   while $CS[j] > balanced$ do
11.     maxdist = 0
12.     maxind = 0
13.     for $i \in 1 : m$ do
14.       if $dist[i][j] > maxdist$ and $MB[i] = j$ then
15.         maxdist = $dist[i][j]$
16.         maxind = $i$
17.     mindist = inf
18.     minind = $j$
19.     for $k \in 1 : P$ do
20.       if $dist[maxind][k] < mindist$ then
21.         if $CS[k] < balanced$ then
22.           mindist = $dist[maxind][k]$
23.           minind = $k$
24.       $MB[maxind] = minind$
25.       $CS[j] = CS[j] - 1$
26.       $CS[minind] = CS[minind] + 1$
27.   end while
28.   $CT[i] = 0$
29. for $i \in 1 : m$ do
30.   $j = MB[i]$
31.   $CT[j] = SA[i]$
32. for $i \in 1 : P$ do
33.   $CT[i] = CT[i] / CS[i]$

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 VI. The experimental results in the table are based on the 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 ($X$), if its closest data center is $CT_i$, we will only use $MF_i$ to make prediction for $X$. The communication overhead of CP-SVM, FCFS-CA and BKM-CA are from the data transfer and distribution in K-means like partitioning algorithm. 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. 7). All the results of CA-SVM in Section V are based on casvm2. To make it clear, the framework of CA-SVM (FCFS-CA, BKM-CA, and RA-CA) is summarized in Algorithm 6.
6.1 Dis-SMO: 34MB
6.2 Cascade: 8MB
6.3 DC-SVM: 29MB
6.4 DC-Filter: 18MB
6.5 CP-SVM: 17MB
6.6 CA-SVM: 0MB

Figure 6. 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: Framework of SMO and CA-SVM

Original SMO:
1 training dataset: trd
2 test dataset: ted
3 1: model = svm-training(trd)
4 2: solution = svm-solver(ted, model)
CA-SVM:
5 training dataset: trd
6 test dataset: ted
7 i-th subset of trd: trdi
8 data center of trdi: ceni
9 1: ([trd1, trd2, ..., trdp], [cen1, cen2, ..., cenp]) = partition(trd)
10 2: [ted1, ted2, ..., tedp] = partition(ted, [cen1, cen2, ..., cenp])
11 3: for (i=1;i≤p;i++) modeli = svm-training(trdi)
12 4: for (i=1;i≤p;i++) solutioni = svm-solver(tedi, modeli)
13 5: solution = [solution1, solution2, ..., solutionp]

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 (i.e. ijcnn in Table VIII), m is 48,000, n is 13, and s is 4474. We can predict the communication volume of Cascade is about 3 × (48000 × 13 + 48000 + 4474 × 13) × 4B = 8.4MB. Our experimental result is 8.41MB, which means the prediction for Cascade is very close to the actual volume.

Table VI

<table>
<thead>
<tr>
<th>Method</th>
<th>Formula</th>
<th>Prediction</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dis-SMO</td>
<td>Θ(26Ip + 2pm + 4mn)</td>
<td>36MB</td>
<td>34MB</td>
</tr>
<tr>
<td>Cascade</td>
<td>O(3mn + 3m + 3n)</td>
<td>8.4MB</td>
<td>8.4MB</td>
</tr>
<tr>
<td>DC-SVM</td>
<td>Θ(9mn + 12m + 2kpn)</td>
<td>24MB</td>
<td>29MB</td>
</tr>
<tr>
<td>DC-Filter</td>
<td>O(6mn + 7m + 3n + 2kpn)</td>
<td>16.2MB</td>
<td>18MB</td>
</tr>
<tr>
<td>CP-SVM</td>
<td>Θ(6mn + 7m + 2kpn)</td>
<td>15.6MB</td>
<td>17MB</td>
</tr>
<tr>
<td>FCFS-CA</td>
<td>Θ(6mn + 6m + 2pn)</td>
<td>15.4MB</td>
<td>16MB</td>
</tr>
<tr>
<td>RA-CA</td>
<td>0</td>
<td>0MB</td>
<td>0MB</td>
</tr>
</tbody>
</table>

2) Point-to-Point profiling: Fig. 6 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 VII we can observe that CP-SVM is more efficient than Cascade since the volume of communication
Figure 7. 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.

3) Ratio of Communication to Computation: Fig. 7 shows the ratio of communication time to computation time for different methods. From Fig. 7 we can observe that our 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 VI shows that the communication volumes of DC-Filter and CP-SVM are similar. However, Fig. 7 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 VII

<table>
<thead>
<tr>
<th>Method</th>
<th>Amount</th>
<th>Comm Operations</th>
<th>Amount/Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dis-SMO</td>
<td>34MB</td>
<td>335,186</td>
<td>101B</td>
</tr>
<tr>
<td>Cascade</td>
<td>8MB</td>
<td>56</td>
<td>150,200B</td>
</tr>
<tr>
<td>DC-SVM</td>
<td>29MB</td>
<td>80</td>
<td>360,734B</td>
</tr>
<tr>
<td>DC-Filter</td>
<td>18MB</td>
<td>80</td>
<td>220,449B</td>
</tr>
<tr>
<td>CP-SVM</td>
<td>17MB</td>
<td>24</td>
<td>709,644B</td>
</tr>
<tr>
<td>FCFS-CA</td>
<td>16MB</td>
<td>13</td>
<td>1,240,625B</td>
</tr>
<tr>
<td>RA-CA</td>
<td>0MB</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

V. EXPERIMENTAL RESULTS AND ANALYSIS

The test datasets in our experiments are shown in Table VIII, 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 VIII

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Application Field</th>
<th>#samples</th>
<th>#features</th>
</tr>
</thead>
<tbody>
<tr>
<td>adult</td>
<td>Economy</td>
<td>32,561</td>
<td>123</td>
</tr>
<tr>
<td>epsilon</td>
<td>Character Recognition</td>
<td>400,000</td>
<td>2,000</td>
</tr>
<tr>
<td>face</td>
<td>Face Detection</td>
<td>489,410</td>
<td>361</td>
</tr>
<tr>
<td>gisette</td>
<td>Computer Vision</td>
<td>6,000</td>
<td>5,000</td>
</tr>
<tr>
<td>ijcnn</td>
<td>Text Decoding</td>
<td>49,990</td>
<td>22</td>
</tr>
<tr>
<td>usps</td>
<td>Transportation</td>
<td>266,079</td>
<td>675</td>
</tr>
<tr>
<td>webspam</td>
<td>Management</td>
<td>350,000</td>
<td>16,609,143</td>
</tr>
</tbody>
</table>

Table IX

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Iterations</th>
<th>Time (Init, Training)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dis-SMO</td>
<td>84.3%</td>
<td>8,054</td>
<td>5.64s (0.006, 5.64)</td>
</tr>
<tr>
<td>Cascade</td>
<td>83.6%</td>
<td>1,323</td>
<td>1.05s (0.007, 1.04)</td>
</tr>
<tr>
<td>DC-SVM</td>
<td>83.7%</td>
<td>8,099</td>
<td>17.1s (0.042, 17.1)</td>
</tr>
<tr>
<td>DC-Filter</td>
<td>84.4%</td>
<td>3,317</td>
<td>2.23s (0.042, 2.18)</td>
</tr>
<tr>
<td>CP-SVM</td>
<td>83.0%</td>
<td>2,497</td>
<td>1.66s (0.041, 1.59)</td>
</tr>
<tr>
<td>BKM-CA</td>
<td>83.3%</td>
<td>1,482</td>
<td>1.61s (0.057, 1.54)</td>
</tr>
<tr>
<td>FCFS-CA</td>
<td>83.6%</td>
<td>1,621</td>
<td>1.21s (0.005, 1.19)</td>
</tr>
<tr>
<td>RA-CA</td>
<td>83.1%</td>
<td>1,160</td>
<td>0.96s (4e-4, 0.95)</td>
</tr>
</tbody>
</table>

Table X

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

A. Speedup and Accuracy

From Table IX to Table XIV, we can observe that CA-SVM (i.e. RA-CA) can achieve $3 \times - 16 \times (7 \times$ 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 compared to Dis-SMO 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 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 XV and XVI show the results of strong scaling time and efficiency. We can observe that the strong 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 are all versions of CA-SVM. RA-CA manages to totally avoid inter-node communication, obtain a perfect load-balancing, and achieve 7× average speedup with only 1.3% average loss in accuracy compared to Dis-SMO for six
real-world application datasets. Because of faster iteration and reduced number of iterations, RA-CA can achieve 1068.7% strong scaling when we increase the number of processors from 96 to 1536. Thanks to the removal of communication overhead, RA-CA 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. Even though random-partition CA-SVM (RA-CA) is not reliable in general, it provides a lower bound on performance of the other algorithms, to which to compare. Compared to Dis-SMO, FCFS-CA achieves 2.2 – 13× (6× on average) speedups. The accuracy losses from Dis-SMO to FCFS-CA range from none to 1.1% (0.46% on average).

**ACKNOWLEDGMENT**

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), Computer Science Study Group, and the U.S. National Science Foundation (NSF) under Award Number 1339749 and CAREER Award Number 0953100. This research also used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA, DOE, or NSF. In addition, the research is funded by DARPA Award HR0011-12-0-0016, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, MathWorks and Samsung. Also funded by U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, DE-SC0008699, DE-SC0010200 and DOE AC02-05CH11231. Dr. Le Song was supported in part by NSF/NIH BIGDATA 1R01GM108341, NSF IIS-1116886, NSF CAREER IIS-1350983 and a Raytheon Faculty Fellowship. Yang You is a Siebel Scholar.

**REFERENCES**


