# Chapter 10
# PSVM: Parallelizing Support Vector Machines on Distributed Computers

**Abstract** Support Vector Machines (SVMs) suffer from a widely recognized scalability problem in both memory use and computational time. To improve scalability, we have developed a parallel SVM algorithm (PSVM), which reduces memory use through performing a row-based, approximate matrix factorization, and which loads only essential data to each machine to perform parallel computation. Let $n$ denote the number of training instances, $p$ the reduced matrix dimension after factorization ($p$ is significantly smaller than $n$) and $m$ the number of machines. PSVM reduces the memory requirement by the Interior Point Method from $\mathcal{O}(n^2)$ to $\mathcal{O}(np/m)$, and improves computation time to $\mathcal{O}(np^2/m)$. Empirical studies show PSVM to be effective. This chapter[†] was first published in NIPS'07 [1] and the open-source code was made available at [2].

**Keywords** Support vector machines · Interior point method · Incomplete Cholesky factorization · MPI · Distributed systems · Matrix factorization

## 10.1 Introduction

Support Vector Machines (SVMs) are a core machine learning technology. They enjoy strong theoretical foundations and excellent empirical successes in many pattern recognition applications. Unfortunately, SVMs do not scale well with respect to the size of training data. Given $n$ training instances, the time to train an SVM model is about $\mathcal{O}(n^2)$ in the average case, and so is the memory required by the interior point method (IPM) to solve the quadratic optimization problem. These excessive costs make SVMs impractical for large-scale applications.

[†] 

E. Y. Chang, *Foundations of Large-Scale Multimedia Information Management and Retrieval*, DOI: 10.1007/978-3-642-20429-6_10,

Let us examine the resource bottlenecks of SVMs in a binary classification setting to explain our proposed solution. Given a set of training data $\mathscr{X} = \{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in \mathbf{R}^d\}_{i=1}^n$, where $\mathbf{x}_i$ is an observation vector, $y_i \in \{-1, 1\}$ is the class label of $\mathbf{x}_i$, and $n$ is the size of $\mathscr{X}$, we apply SVMs on $\mathscr{X}$ to train a binary classifier. SVMs aim to search a hyperplane in the *Reproducing Kernel Hilbert Space* (RKHS) that maximizes the margin between the two classes of data in $\mathscr{X}$ with the smallest training error [3]. This problem can be formulated as the following quadratic optimization problem:

$$\begin{aligned} \min\ & \mathscr{P}(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{n} \xi_i \\ \text{s.t. } & 1 - y_i(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i) + b) \le \xi_i, \quad \xi_i > 0, \end{aligned} \tag{10.1}$$

where $\mathbf{w}$ is a weighting vector, $b$ is a threshold, $C$ a regularization hyperparameter, and $\boldsymbol{\phi}(\cdot)$ a basis function which maps $\mathbf{x}_i$ to an RKHS space. The decision function of SVMs is $f(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + b$, where $\mathbf{w}$ and $b$ are attained by solving $\mathscr{P}$ in (10.1). The optimization problem in (10.1) is called the primal formulation of SVMs with $L_1$ loss. It is hard to solve $\mathscr{P}$ directly, partly because the explicit mapping via $\boldsymbol{\phi}(\cdot)$ can make the problem intractable and partly because the mapping function $\boldsymbol{\phi}(\cdot)$ is often unknown. The method of *Lagrangian multipliers* is thus introduced to transform the primal formulation into the dual one

$$\begin{aligned} \min\ & \mathscr{D}(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q}\boldsymbol{\alpha} - \boldsymbol{\alpha}^T \mathbf{1} \\ \text{s.t. } & \mathbf{0} \le \boldsymbol{\alpha} \le \mathbf{C}, \mathbf{y}^T \boldsymbol{\alpha} = 0, \end{aligned} \tag{10.2}$$

where $[\mathbf{Q}]_{ij} = y_i y_j \boldsymbol{\phi}^T(\mathbf{x}_i)\boldsymbol{\phi}(\mathbf{x}_j)$, and $\boldsymbol{\alpha} \in \mathbf{R}^n$ is the Lagrangian multiplier variable (or dual variable). The weighting vector $\mathbf{w}$ is related with $\boldsymbol{\alpha}$ in $\mathbf{w} = \sum_{i=1}^n \alpha_i \boldsymbol{\phi}(\mathbf{x}_i)$.

The dual formulation $\mathscr{D}(\boldsymbol{\alpha})$ requires an inner product of $\boldsymbol{\phi}(\mathbf{x}_i)$ and $\boldsymbol{\phi}(\mathbf{x}_j)$. SVMs utilize the *kernel trick* by specifying a kernel function to define the inner-product $K(\mathbf{x}_i, \mathbf{x}_j) = \boldsymbol{\phi}^T(\mathbf{x}_i)\boldsymbol{\phi}(\mathbf{x}_j)$. We thus can rewrite $[\mathbf{Q}]_{ij}$ as $y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$. When the given kernel function $K$ is psd (positive semi-definite), the dual problem $\mathscr{D}(\boldsymbol{\alpha})$ is a convex Quadratic Programming (QP) problem with linear constraints, which can be solved via the *Interior-Point method* (IPM) [4]. Both the computational and memory bottlenecks of the SVM training is the IPM solver to the dual formulation of SVMs in (10.2).

Currently, the most effective IPM algorithm is the primal–dual IPM [4]. The principal idea of the primal–dual IPM is to remove inequality constraints using a barrier function and then resort to the iterative Newton's method to solve the KKT linear system related to the Hessian matrix $\mathbf{Q}$ in $\mathscr{D}(\boldsymbol{\alpha})$. The computational cost is $O(n^3)$ and the memory usage $O(n^2)$.

In this work, we propose a parallel SVM algorithm (PSVM) to reduce memory use and to parallelize both data loading and computation. Given $n$ training instances each with $d$ dimensions, PSVM first loads the training data in a round-robin fashion onto $m$ machines. The memory requirement per machine is $\mathscr{O}(nd/m)$. Next,

PSVM performs a parallel row-based Incomplete Cholesky Factorization (ICF) on the loaded data. At the end of parallel ICF, each machine stores only a fraction of the factorized matrix, which takes up space of $\mathscr{O}(np/m)$, where $p$ is the column dimension of the factorized matrix. (Typically, $p$ can be set to be about $\sqrt{n}$ without noticeably degrading training accuracy.) PSVM reduces memory use of IPM from $O(n^2)$ to $O(np/m)$, where $p/m$ is much smaller than $n$. PSVM then performs parallel IPM to solve the quadratic optimization problem in (10.2). The computation time is improved from about $\mathscr{O}(n^2)$ of a decomposition-based algorithm (e.g., SVMLight [5], LIBSVM [6], SMO [7], and SimpleSVM [8]) to $\mathscr{O}(np^2/m)$. This work's main contributions are: (1) PSVM achieves memory reduction and computation speedup via a parallel ICF algorithm and parallel IPM. (2) PSVM handles kernels (in contrast to other algorithmic approaches [9, 10]). (3) We have implemented PSVM on our parallel computing infrastructures. PSVM effectively speeds up training time for large-scale tasks while maintaining high training accuracy.

PSVM is a practical, parallel approximate implementation to speed up SVM training on today's distributed computing infrastructures for dealing with Web-scale problems. What we do *not* claim are as follows: (1) We make no claim that PSVM is the sole solution to speed up SVMs. Algorithmic approaches such as [9–12] can be more effective when memory is not a constraint or kernels are not used. (2) We do not claim that the algorithmic approach is the only avenue to speed up SVM training. Data-processing approaches such as [13] can divide a serial algorithm (e.g., LIBSVM) into subtasks on subsets of training data to achieve good speedup. (Data-processing and algorithmic approaches complement each other, and can be used together to handle large-scale training.)

## 10.2 Interior Point Method With Incomplete Cholesky Factorization

Interior Point Method (IPM) is one of the state-of-the-art algorithms to solve convex optimization problem with inequality constraints and the primal–dual IPM is one of the most efficient IPM methods. Whereas the detailed derivation could be found in [4, 14], this section briefly reviews primal–dual IPM.

First, we take (10.2) as a primal problem (it is the dual form of SVMs, however, it is treated as primal optimization problem here) and its dual form can be written as

$$\begin{aligned} \max_{\nu,\boldsymbol{\lambda},\boldsymbol{\xi}} \ & \mathscr{D}'(\boldsymbol{\alpha},\boldsymbol{\lambda}) = -\frac{1}{2}\boldsymbol{\alpha}^T\mathbf{Q}\boldsymbol{\alpha} - C\sum_{i=1}^{n}\lambda_i \\ \text{s.t.} \ & -\mathbf{Q}\boldsymbol{\alpha} - \nu\mathbf{y} + \boldsymbol{\xi} - \boldsymbol{\lambda} = -\mathbf{1} \\ & \boldsymbol{\xi} \geq 0, \quad \boldsymbol{\lambda} \geq 0, \end{aligned} \tag{10.3}$$

where $\boldsymbol{\lambda}$, $\boldsymbol{\xi}$ and $\nu$ are the dual variables in SVMs for constraints $\boldsymbol{\alpha} \leq \mathbf{C}$, $\boldsymbol{\alpha} \geq \mathbf{0}$ and $\mathbf{y}^T, \boldsymbol{\alpha} = 0$, respectively.

$\alpha = 0$, $\nu = 0$, $\lambda \geq \mathbf{0}$, $\xi \geq \mathbf{0}$
**repeat**
Determine $t = 2n\mu/\hat{\eta}$
Compute $\triangle\mathbf{x}$, $\triangle\lambda$, $\triangle\xi$, and $\triangle\nu$ according to Eq.(10.5).
Determine step length $s > 0$ through backtracking line search and update $\alpha = \alpha + s\triangle\mathbf{x}$, $\lambda = \lambda + s\triangle\lambda$, $\xi = \xi + s\triangle\xi$, $\nu = \nu + s\triangle\nu$ .
**until** $\alpha$ is primal feasible and $\lambda$, $\xi$, $\nu$ is dual feasible and the surrogate gap $\hat{\eta}$ is smaller than a threshold

**Fig. 10.1** Interior Point Method

The basic idea of the primal–dual IPM is to optimize variables $\alpha$, $\lambda$, $\xi$, and $\nu$ concurrently. The algorithm applies Newton's method on each variable iteratively to gradually reach the optimal solution. The basic flow is depicted in Fig. 10.1, where $\mu$ is a tuning parameter and the *surrogate gap*

$$\hat{\eta} = C\sum_{i=1}^{n}\lambda_i - \boldsymbol{\alpha}^T\boldsymbol{\lambda} + \boldsymbol{\alpha}^T\boldsymbol{\xi} \tag{10.4}$$

is used to compute $t$ and check convergence. We omit how to compute $s$ here as all the details could be found in [14].

Newton update, the core step of IPM, could be written as solving the following equation

$$\begin{aligned}
&\begin{pmatrix} \mathbf{Q}_{nn} & \mathbf{I}_{nn} & -\mathbf{I}_{nn} & \mathbf{y}_n \\ -\text{diag}(\boldsymbol{\lambda})_{nn} & \text{diag}(\mathbf{C}-\boldsymbol{\alpha})_{nn} & \mathbf{0}_{nn} & \mathbf{0}_n \\ \text{diag}(\boldsymbol{\xi})_{nn} & \mathbf{0}_{nn} & \text{diag}(\boldsymbol{\alpha})_{nn} & \mathbf{0}_n \\ \mathbf{y}^T & \mathbf{0}_n^T & \mathbf{0}_n^T & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \\ \Delta\boldsymbol{\xi} \\ \Delta\nu \end{pmatrix}, \\
&= -\begin{pmatrix} \mathbf{Q}\alpha - \mathbf{1}_n + \nu\mathbf{y} + \boldsymbol{\lambda} - \boldsymbol{\xi} \\ \text{vec}(\lambda_i(C-\alpha_i) - \frac{1}{t}) \\ \text{vec}(\xi_i\alpha_i - \frac{1}{t}) \\ \mathbf{y}^T\boldsymbol{\alpha} \end{pmatrix}
\end{aligned} \tag{10.5}$$

where $\text{diag}(\mathbf{v})$ means generating an $n \times n$ square diagonal matrix whose diagonal element in the $i$th row is $v_i$; $\text{vec}(\alpha_i)$ means generating a vector with the $i$th component as $\alpha_i$; $I_{nn}$ is an identity matrix.

IPM boils down to solving the following equations in the Newton step iteratively.

$$\Delta\boldsymbol{\lambda} = -\boldsymbol{\lambda} + \text{vec}\left(\frac{1}{t(C-\alpha_i)}\right) + \text{diag}\left(\frac{\lambda_i}{C-\alpha_i}\right)\Delta\mathbf{x} \tag{10.6}$$

$$\Delta\boldsymbol{\xi} = -\boldsymbol{\xi} + \text{vec}\left(\frac{1}{t\alpha_i}\right) - \text{diag}\left(\frac{\xi_i}{\alpha_i}\right)\Delta\mathbf{x} \tag{10.7}$$

$$\Delta\nu = \frac{\mathbf{y}^T\boldsymbol{\Sigma}^{-1}\mathbf{z} + \mathbf{y}^T\boldsymbol{\alpha}}{\mathbf{y}^T\boldsymbol{\Sigma}^{-1}\mathbf{y}} \tag{10.8}$$

$$\mathbf{D} = \text{diag}\left(\frac{\xi_i}{\alpha_i} + \frac{\lambda_i}{C-\alpha_i}\right) \tag{10.9}$$

$$\Delta\mathbf{x} = \boldsymbol{\Sigma}^{-1}(\mathbf{z} - \mathbf{y}\Delta\nu), \tag{10.10}$$

where $\boldsymbol{\Sigma}$ and $\mathbf{z}$ depend only on $[\boldsymbol{\alpha}, \boldsymbol{\lambda}, \boldsymbol{\xi}, \nu]$ from the last iteration as follows:

$$\boldsymbol{\Sigma} = \mathbf{Q} + \text{diag}\left(\frac{\xi_i}{\alpha_i} + \frac{\lambda_i}{C-\alpha_i}\right) \tag{10.11}$$

$$\mathbf{z} = -\mathbf{Q}\boldsymbol{\alpha} + \mathbf{1}_n - \nu\mathbf{y} + \frac{1}{t}\text{vec}\left(\frac{1}{\alpha_i} - \frac{1}{C-\alpha_i}\right). \tag{10.12}$$

The computation bottleneck is on matrix inverse, which takes place on $\boldsymbol{\Sigma}$ for solving $\Delta\nu$ in (10.8) and $\Delta\mathbf{x}$ in (10.10). We will mainly focus on this part as the other computations are trivial. Obviously, when the data set size is large, it is virtually infeasible to compute inversion of an $n \times n$ matrix due to resource and time constraints. It is beneficial to employ matrix factorization to factorize $\mathbf{Q}$. As $\mathbf{Q}$ is positive semi definite, there always exists an exact Cholesky factor: a lower-triangular matrix $\mathbf{G}$ that $\mathbf{G} \in \mathbb{R}^{n*n}$ and $\mathbf{Q} = \mathbf{G}\mathbf{G}^T$. If we truncate $\mathbf{G}$ to $\mathbf{H}(\mathbf{H} \in \mathbb{R}^{n*p}$ and $p \ll n)$ by keeping only the most important $p$ columns (i.e., minimizing trace$(\mathbf{Q} - \mathbf{H}\mathbf{H}^T)$, this will become incomplete Cholesky factorization and $\mathbf{Q} \approx \mathbf{H}\mathbf{H}^T$. In other words, $\mathbf{H}$ is somehow "close" to $\mathbf{Q}$'s exact Cholesky factor $\mathbf{G}$.

If we factorize $\mathbf{Q}$ this way and $\mathbf{D}$ is an identity matrix, according to SMW (the *Sherman–Morrison–Woodbury formula*) [15], we can write $\boldsymbol{\Sigma}^{-1}$ as

$$\begin{aligned}\Sigma^{-1} &= (\mathbf{D} + \mathbf{Q})^{-1} \approx (\mathbf{D} + \mathbf{H}\mathbf{H}^T)^{-1} \\ &= \mathbf{D}^{-1} - \mathbf{D}^{-1}\mathbf{H}(\mathbf{I} + \mathbf{H}^T\mathbf{D}^{-1}\mathbf{H})^{-1}\mathbf{H}^T\mathbf{D}^{-1},\end{aligned}$$

where $(\mathbf{I}+\mathbf{H}^T\mathbf{D}^{-1}\mathbf{H})$ is a $p \times p$ matrix. As $p$ is usually small, it is practically feasible to compute it. In the following section, we will introduce how to parallelize the key steps of IPM to further speed it up.

## 10.3 PSVM Algorithm

The key step of PSVM is parallel ICF (PICF). Traditional column-based ICF [16, 17] can reduce computational cost, but the initial memory requirement is $O(np)$, and hence not practical for very large data set. PSVM devises parallel row-based ICF (PICF) as its initial step, which loads training instances onto parallel machines and

**Input**: $n$ training instances; $p$: rank of ICF matrix $H$; $m$: number of machines
**Output**: $H$ distributed on $m$ machines
**Variables**:
$\mathbf{v}$: fraction of the diagonal vector of $Q$ that resides in local machine
$k$: iteration number
$\mathbf{x}_i$: the $i^{th}$ training instance
$M$: machine index set, $M = \{0, 1, \ldots, m-1\}$
$I_c$: row-index set on machine $c$ ($c \in M$), $I_c = \{c, c+m, c+2m, \ldots\}$

1: **for** $i = 0$ to $n-1$ **do**
2: Load $\mathbf{x}_i$ into machine $i\%m$.
3: **end for**
4: $k \leftarrow 0$; $H \leftarrow 0$; $\mathbf{v} \leftarrow$ the fraction of the diagonal vector of $Q$ that resides in local machine. ($\mathbf{v}(i)(i \in I_m)$ can be obtained from $\mathbf{x}_i$)
5: Initialize *master* to be machine 0.
6: **while** $k < p$ **do**
7: Each machine $c \in M$ selects its local pivot value, which is the largest element in $\mathbf{v}$:

$$\mathbf{lpv}_{k,c} = \max_{i \in I_c} \mathbf{v}(i),$$

and records the local pivot index, the row index corresponds to $\mathbf{lpv}_{k,c}$:

$$\mathbf{lpi}_{k,c} = \arg\max_{i \in I_c} \mathbf{v}(i)$$

.
8: Gather $\mathbf{lpv}_{k,c}$'s and $\mathbf{lpi}_{k,c}$'s ($c \in M$) to *master*.
9: The *master* selects the largest local pivot value as global pivot value $\mathbf{gpv}_k$ and records in $i_k$, row index corresponding to the global pivot value.

$$\mathbf{gpv}_k = \max_{c \in M} \mathbf{lpv}_{k,c}$$

.
10: The *master* broadcasts $\mathbf{gpv}_k$ and $i_k$.
11: Change *master* to machine $i_k\%m$.
12: Calculate $H(i_k, k)$ according to Eq.(10.13) on *master*.
13: The *master* broadcasts the pivot instance $\mathbf{x}_{i_k}$ and the pivot row $H(i_k, :)$. (Only the first $k+1$ values of the pivot row need to be broadcast, since the remainder are zeros.)
14: Each machine $c \in M$ calculates its part of the $k^{th}$ column of $H$ according to Eq.(10.14).
15: Each machine $c \in M$ updates $\mathbf{v}$ according to Eq.(10.15).
16: $k \leftarrow k+1$
17: **end while**

**Fig. 10.2** Row-based PICF

performs factorization simultaneously on these machines. Once PICF has loaded $n$ training data distributedly on $m$ machines, and reduced the size of the kernel matrix through factorization, IPM can be solved on parallel machines simultaneously. We present PICF first, and then describe how IPM takes advantage of PICF (Fig. 10.2).

### *10.3.1 Parallel ICF*

ICF can approximate $Q$ ($Q \in R^{n \times n}$) by a smaller matrix $H$ ($H \in R^{n \times p}$, $p \ll n$), i.e., $Q \approx HH^T$. ICF, together with SMW (the *Sherman–Morrison–Woodbury for-*

*mula*), can greatly reduce the computational complexity in solving an $n \times n$ linear system. The work of [16] provides a theoretical analysis of how ICF influences the optimization problem in (10.2). They proved that the error of the optimal objective value introduced by ICF is bounded by $C^2 l\epsilon/2$, where $C$ is the hyperparameter of SVM, $l$ is the number of support vectors, and $\epsilon$ is the bound of ICF approximation (i.e. trace$(Q - HH^T) < \epsilon$). Experimental results in Sect. 10.4 show that when $p$ is set to $\sqrt{n}$, the error can be negligible.

Our row-based parallel ICF (PICF) works as follows: Let vector $\mathbf{v}$ be the diagonal of $Q$ and suppose the pivots (the largest diagonal values) are $\{i_1, i_2, \ldots, i_k\}$, the $k$th iteration of ICF computes three equations:

$$H(i_k, k) = \sqrt{\mathbf{v}(i_k)}, \tag{10.13}$$

$$H(J_k, k) = (Q(J_k, k) - \sum_{j=1}^{k-1} H(J_k, j)H(i_k, j))/H(i_k, k), \tag{10.14}$$

$$\mathbf{v}(J_k) = \mathbf{v}(J_k) - H(J_k, k)^2, \tag{10.15}$$

where $J_k$ denotes the complement of $\{i_1, i_2, \ldots, i_k\}$. The algorithm iterates until the approximation of $Q$ by $H_k H_k^T$ (measured by trace$(Q - H_k H_k^T)$) is satisfactory, or the predefined maximum iterations (or say, the desired rank of the ICF matrix) $p$ is reached.

As suggested by Golub, a parallelized ICF algorithm can be obtained by constraining the parallelized Cholesky Factorization algorithm, iterating at most $p$ times. However, in the proposed algorithm [15], matrix $H$ is distributed by columns in a round-robin way on $m$ machines (hence we call it column-based parallelized ICF). Such column-based approach is optimal for the single-machine setting, but cannot gain full benefit from parallelization for two major reasons:

1. Large memory requirement. All training data are needed for each machine to calculate $Q(J_k, k)$. Therefore, each machine must be able to store a local copy of the training data.
2. Limited parallelizable computation. Only the inner product calculation

$$\sum_{j=1}^{k-1} H(J_k, j)H(i_k, j)$$

in (10.14) can be parallelized. The calculation of pivot selection, the summation of local inner product result, column calculation in (10.14), and the vector update in (10.15) must be performed on one single machine.

To remedy these shortcomings of the column-based approach, we propose a row-based approach to parallelize ICF, which we summarize in Fig. 10.3. Our row-based approach starts by initializing variables and loading training data onto $m$ machines in a round-robin fashion (steps 1–5). The algorithm then performs the ICF main loop

until the termination criteria are satisfied (e.g., the rank of matrix $H$ reaches $p$). In the main loop, PICF performs five tasks in each iteration $k$:

1. Distributedly find a pivot, which is the largest value in the diagonal $\mathbf{v}$ of matrix $Q$ (steps 7–10). Notice that PICF computes only needed elements in $Q$ from training data, and it does not store $Q$.
2. Set the machine where the pivot resides as the *master* (step 11).
3. On the *master*, PICF calculates $H(i_k, k)$ according to (13) (step 12).
4. The *master* then broadcasts the pivot instance $\mathbf{x}_{i_k}$ and the pivot row $H(i_k, :)$ (step 13).
5. Distributedly compute (10.14) and (10.15) (steps 14 and 15).

At the end of the algorithm, $H$ is stored distributedly on $m$ machines, ready for parallel IPM (presented in the next section). PICF enjoys three advantages: parallel memory use ($\mathcal{O}(np/m)$), parallel computation ($\mathcal{O}(p^2n/m)$), and low communication overhead ($\mathcal{O}(p^2 \log(m))$). Particularly on the communication overhead, its fraction of the entire computation time shrinks as the problem size grows. We will verify this in the experimental section. This pattern permits a larger problem to be solved on more machines to take advantage of parallel memory use and computation.

*Example*

We use a simple example to explain how PICF works. Suppose we have three machines (or processors) and eight data instances, PICF first loads the data in a round-robin fashion on the three machines (numbered as #0, #1, and #2).

| Processor | Data ($label\ id : value\ [id : value \cdots]$) | Row index |
|---|---|---|
| # 0 | − 1 1:0.943578 2:0.397088 | 0 |
| # 1 | − 1 1:0.397835 2:0.097548 | 1 |
| # 2 | 1 1:0.821040 2:0.197176 | 2 |
| # 0 | 1 1:0.592864 2:0.452824 | 3 |
| # 1 | 1 1:0.743459 2:0.605765 | 4 |
| # 2 | − 1 1:0.406734 2:0.687923 | 5 |
| # 0 | − 1 1:0.398752 2:0.820476 | 6 |
| # 1 | − 1 1:0.592647 2:0.224432 | 7 |

Suppose the Laplacian kernel is used:

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma ||\mathbf{x}_i - \mathbf{x}_j||},$$

and we set $\gamma = 1.000$. The first five columns of $Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$ is

$$Q = \begin{pmatrix} 1.000000 & 0.429436 & -0.724372 & -0.666010 & -0.666010 \\ 0.429436 & 1.000000 & -0.592839 & -0.576774 & -0.425776 \\ -0.724372 & -0.592839 & 1.000000 & 0.616422 & 0.614977 \\ -0.666010 & -0.576774 & 0.616422 & 1.000000 & 0.738203 \\ -0.664450 & -0.425776 & 0.614977 & 0.738203 & 1.000000 \\ 0.437063 & 0.549210 & -0.404520 & -0.656240 & -0.657781 \\ 0.379761 & 0.484884 & -0.351485 & -0.570202 & -0.571542 \\ 0.592392 & 0.724919 & -0.774414 & -0.795640 & -0.587344 \end{pmatrix}_{8\times 5}$$

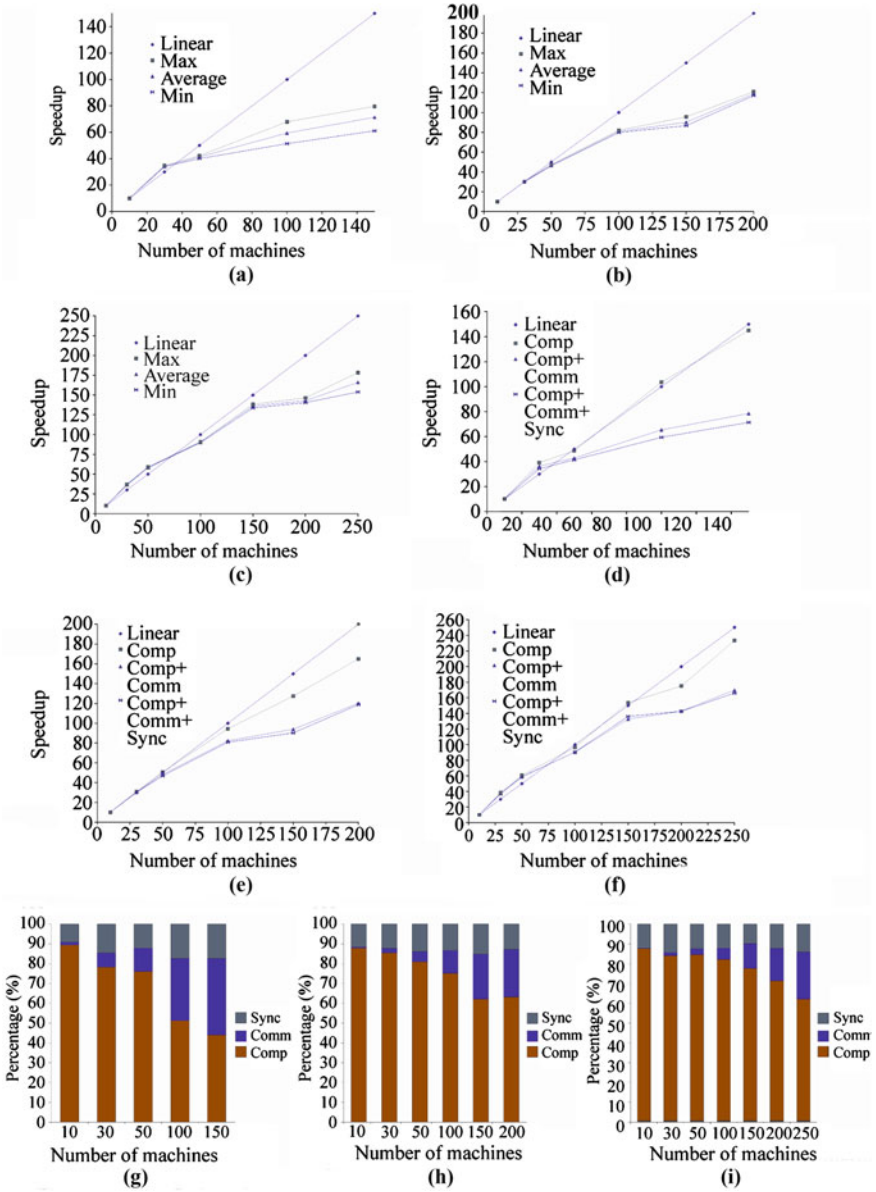


**Fig. 10.3** Speedup and overheads of three datasets (see color insert). **a** Image (200k) speedup, **b** Covertype (500k) speedup, **c** RCV (800k) speedup, **d** Image (200k) overhead, **e** Covertype (500k) overhead, **f** RCV (800k) overhead, **g** Image (200k) fraction, **h** Covertype (500k) fraction, **i** RCV (800k) fraction

Note that the kernel matrix doesn't reside in memory, it is computed on-demand according to (10.6).

#### 10.3.1.1 Iteration k = 0

PICF initializes $\boldsymbol{v}$ , whose elements are all one at the start. The elements of $\boldsymbol{v}$ are stored on the same machines as their corresponding $\mathbf{x}_i$.

processor local diagonal vector $v$ row index

$$\#0\ v = \begin{pmatrix} 1.000000 \\ 1.000000 \\ 1.000000 \end{pmatrix} \begin{matrix} 0 \\ 3 \\ 6 \end{matrix}$$

processor local diagonal vector $v$ row index

$$\#1\ v = \begin{pmatrix} 1.000000 \\ 1.000000 \\ 1.000000 \end{pmatrix} \begin{matrix} 1 \\ 4 \\ 7 \end{matrix}$$

processor local diagonal vector $v$ row index

$$\#1\ v = \begin{pmatrix} 1.000000 \\ 1.000000 \end{pmatrix} \begin{matrix} 2 \\ 5 \end{matrix}$$

PICF next chooses the pivot. Each machine finds the maximum pivot and its index, and then broadcasts to the rest of the machines. Each machine then finds the largest value, and its corresponding index is the index of the global pivot. PICF sets the machine where the pivot resides as the *master* machine. In the first iteration, since all elements of $v$ are one, the *master* can be set to machine #0. The global pivot value is 1, and its index 0.

Once the global pivot has been identified, PICF follows (10.13) to compute $H(i_0, 0) = H(0, 0) = \sqrt{v(i_0)} = \sqrt{1} = 1$. The *master* broadcasts the pivot instance and the first $k + 1$ value (in iteration $k = 0$, the *master* broadcasts only one value) of the pivot row of $H$ (the $i_0$th row of $H$). That is, the *master* broadcasts pivot instance $x_0 = -11 : 0.9435782 : 0.397088$ and 1.

Next, each machine can compute rows of the first column of $H$ according to (10.14). Take $H(4, 0)$ as an example, which is located at machine #1. $Q(4,0)$ can be computed by the Laplacian kernel function using the broadcast pivot instance $x_0$ and $x_4$ on machine #1 :

$$Q(4, 0) = y_4 y_0 K(x_4, x_0) = y_4 y_0 \exp(-\gamma ||x_4 - x_0||) = -0.664450.$$

$H(0,0)$ can be obtained from the pivot row of $H$, which has been broadcast in the previous step. We thus get

$$H(4,0) = (Q(4,0) - \sum_{j=0}^{-1} H(4,j)H(0,j))/H(0,0) = Q(4,0)/H(0,0) = -0.664450.$$

Similarly, the other elements of the first column of $H$ can be calculated on their machines. The result on machine #0 is as follows:

$$H_0 = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 \end{pmatrix}$$
$$\downarrow$$
$$\begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 \\ -0.666010 & 0.000000 & 0.000000 & 0.000000 \\ 0.37961 & 0.000000 & 0.000000 & 0.000000 \end{pmatrix}$$

The final step of the first iteration updates $v$ distributedly according to (10.15).

$$v = \begin{pmatrix} v(0) - H(0,0)^2 \\ v(3) - H(3,0)^2 \\ v(6) - H(6,0)^2 \end{pmatrix} = \begin{pmatrix} 1.000000 - 1.000000^2 \\ 1.000000 - (-0.666010)^2 \\ 1.000000 - 0.379761^2 \end{pmatrix} = \begin{pmatrix} 0.000000 \\ 0.556430 \\ 0.855782 \end{pmatrix}$$

#### 10.3.1.2 Iteration k = 1

PICF again, obtains local pivot values (the largest element of $v$ on each machine, and their indexes.

$$\#0\ localPivotValue_{1,0} = 0.855782\ localPivotIndex_{1,0} = 6$$
$$\#1\ localPivotValue_{1,1} = 0.815585\ localPivotIndex_{1,1} = 3$$
$$\#2\ localPivotValue_{1,2} = 0.808976\ localPivotIndex_{1,2} = 5$$

After the above information has been broadcast and received, the global pivot value is identified as 0.855782, and the global pivot index $i_1 = 6$. The id of the *master* machine is $6\%3 = 0$. Next, PICF calculates $H(i_1, 1)$ on the *master* according to (10.13). $H(6,1) = \sqrt{v(i_6)} = \sqrt{0.855782} = 0.925085$. PICF then broadcasts the pivot instance $x_6$, and the first $k+1$ elements on the pivot row of $H$, which are 0.379761 and 0.925085. Each machine then computes the second column of $H$ according to (10.14). The result on machine #0 is as follows:

$$H_0 = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 \\ -0.666010 & 0.000000 & 0.000000 & 0.000000 \\ 0.379761 & 0.000000 & 0.000000 & 0.000000 \end{pmatrix}$$
$$\downarrow$$
$$\begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 \\ -0.666010 & -0.342972 & 0.000000 & 0.000000 \\ 0.37961 & 0.925085 & 0.000000 & 0.000000 \end{pmatrix}$$

In the final step of the second iteration, PICF updates $\boldsymbol{v}$ distributedly according to (10.15).

$$v = \begin{pmatrix} v(0) - H(0,1)^2 \\ v(3) - H(3,1)^2 \\ v(6) - H(6,1)^2 \end{pmatrix} = \begin{pmatrix} 0.000000 - 0.000000^2 \\ 0.556430 - (-0.342972)^2 \\ 0.855872 - 0.925085^2 \end{pmatrix} = \begin{pmatrix} 0.000000 \\ 0.438801 \\ 0.000000 \end{pmatrix}$$

#### 10.3.1.3 Iteration k = 3

We fast-forward to show the end result of the fourth and final iteration of this example. The ICF matrix is obtained as follows:

| Computer | ICF matrix H | | | | Row index |
|---|---|---|---|---|---|
| # 0 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0 |
| # 1 | 0.429436 | 0.347862 | 0.833413 | 0.000000 | 1 |
| # 2 | − 0.724372 | − 0.082584 | − 0.303618 | 0.147541 | 2 |
| # 0 | − 0.666010 | − 0.342972 | − 0.205731 | 0.260080 | 3 |
| # 1 | − 0.664450 | − 0.345060 | − 0.024483 | 0.662451 | 4 |
| # 2 | 0.437063 | 0.759837 | 0.116631 | − 0.154472 | 5 |
| # 0 | 0.379761 | 0.925085 | 0.000000 | 0.000000 | 6 |
| # 1 | 0.592392 | 0.247443 | 0.461294 | − 0.146505 | 7 |

### *10.3.2 Parallel IPM*

Solving IPM can be both memory and computation intensive. The computation bottleneck is on matrix inverse, which takes place on $\boldsymbol{\Sigma}$ for solving $\Delta v$ in (10.8) and $\Delta \mathbf{x}$ in (10.10). Equation 10.11 shows that $\boldsymbol{\Sigma}$ depends on $Q$, and we have shown that $Q$ can be approximated through PICF by $HH^T$. Therefore, the bottleneck of the Newton step can be sped up from $\mathcal{O}(n^3)$ to $\mathcal{O}(p^2 n)$, and be parallelized to $\mathcal{O}(p^2 n/m)$.

#### 10.3.2.1 Parallel Data Loading

To minimize both storage and communication cost, PIPM stores data distributedly as follows:

- *Distribute matrix data.* $H$ is distributedly stored at the end of PICF.
- *Distribute* $n \times 1$ *vector data.* All $n \times 1$ vectors are distributed in a round-robin fashion on $m$ machines. These vectors are $\mathbf{z}$, $\boldsymbol{\alpha}$, $\boldsymbol{\xi}$, $\boldsymbol{\lambda}$, $\Delta\mathbf{z}$, $\Delta\boldsymbol{\alpha}$, $\Delta\boldsymbol{\xi}$, and $\Delta\boldsymbol{\lambda}$.
- *Replicate global scalar data.* Every machine caches a copy of global data including $v$, $t$, $n$, and $\Delta v$. Whenever a scalar is changed, a broadcast is required to maintain global consistency.

#### 10.3.2.2 Parallel computation of $\Delta v$

Rather than walking through all equations, we describe how PIPM solves (10.8), where $\Sigma^{-1}$ appears twice. An interesting observation is that parallelizing $\Sigma^{-1}z$ (or $\Sigma^{-1}y$) is simpler than parallelizing $\Sigma^{-1}$. Let us explain how parallelizing $\Sigma^{-1}z$ works, and parallelizing $\Sigma^{-1}y$ can follow suit.

According to SMW (the *Sherman–Morrison–Woodbury formula*), we can write $\Sigma^{-1}z$ as

$$\begin{aligned}\Sigma^{-1}z &= (D+Q)^{-1}z \approx (D+HH^T)^{-1}z \\ &= D^{-1}z - D^{-1}H(I+H^TD^{-1}H)^{-1}H^TD^{-1}z \\ &= D^{-1}z - D^{-1}H(GG^T)^{-1}H^TD^{-1}z.\end{aligned}$$

$\Sigma^{-1}z$ can be computed in four steps:

1. Compute $D^{-1}z$.
   $D$ can be derived from locally stored vectors, following (10.9). $D^{-1}z$ is an $n \times 1$ vector, and can be computed locally on each of the $m$ machines.
2. Compute $t_1 = H^TD^{-1}z$.
   Every machine stores some rows of $H$ and their corresponding part of $D^{-1}z$. This step can be computed locally on each machine. The results are sent to the *master* (which can be a randomly picked machine for all PIPM iterations) to aggregate into $t_1$ for the next step.
3. Compute $(GG^T)^{-1}t_1$.
   This step is completed on the *master*, since it has all the required data. $G$ can be obtained from $I + H^TD^{-1}H$ by Cholesky Factorization. Computing $t_2 = (GG^T)^{-1}t_1$ is equivalent to solving the linear equation system $t_1 = (GG^T)t_2$. PIPM first solves $t_1 = Gy_0$, then $y_0 = G^Tt_2$. Once it has obtained $y_0$, PIPM can solve $G^Tt_2 = y_0$ to obtain $t_2$. The *master* then broadcasts $t_2$ to all machines.
4. Compute $D^{-1}Ht_2$.
   All machines have a copy of $t_2$, and can compute $D^{-1}Ht_2$ locally to solve for $\Sigma^{-1}z$.

Similarly, $\Sigma^{-1}y$ can be computed at the same time. Once we have obtained both, we can solve $\Delta\nu$ according to (10.8).

### *10.3.3 Computing Parameter b and Writing Back*

When the IPM iteration stops, we have the value of $\alpha$ and hence the classification function

$$f(x) = \sum_{i=1}^{N_s} \alpha_i y_i \mathbf{k}(s_i, x) + b.$$

Here $N_s$ is the number of support vectors and $s_i$ are support vectors. In order to complete this classification function, $b$ must be computed. According to the SVM model, given a support vector $s$, we obtain one of the two results for $f(s) : f(s) = +1$, if $y_s = +1$ or $f(s) = -1$, if $y_s = -1$.

In practice, we can select $M$, say 1,000, support vectors and compute the average of the $b$s:

$$b = \frac{1}{M} \sum_{j=1}^{M} (y_{s_j} - \sum_{i=1}^{N_s} \alpha_i y_i \mathbf{k}(s_i, s_j)).$$

Since the support vectors are distributed on $m$ machines, PSVM collects them in parallel to compute $b$. For this purpose, we transform the above formula into the following:

$$b = \frac{1}{M} \sum_{j=1}^{M} y_{s_j} - \frac{1}{M} \sum_{i=1}^{N_s} \alpha_i y_i \sum_{j=1}^{M} \mathbf{k}(s_i, s_j).$$

The $M$ support vectors and their labels $y$s are first broadcast to all machines. All $m$ machines then compute their local results. Finally, the local results are summed up by a reduce operation [18]. When $b$ has been computed, the last task of PSVM is to store the model file on GFS [19] for later classification use.

## 10.4 Experiments

We conducted experiments on PSVM to evaluate its: (1) class-prediction accuracy, (2) scalability on large datasets, and (3) overheads. The experiments were conducted on up to 500 machines in our data center. Not all machines are identically configured; however, each machine is configured with a CPU faster than 2 GHz and memory larger than 4 GB.

### *10.4.1 Class-Prediction Accuracy*

PSVM employs PICF to approximate an $n \times n$ kernel matrix $Q$ with an $n \times p$ matrix $H$. This experiment evaluated how the choice of $p$ affects class-prediction accuracy. We set $p$ of PSVM to $n^t$, where $t$ ranges from 0.1 to 0.5 incremented by 0.1, and compared its class-prediction accuracy with that achieved by LIBSVM. The first two columns of Table 10.1 enumerate the datasets[1] and their sizes with which we

[1] *RCV* is located at http://jmlr.csail.mit.edu/papers/volume5/lewis04a/lyrl2004_rcv1v2_README.ht. The *image* set is a binary-class image dataset consisting of 144 perceptual features. The others are obtained from http://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets. We separated the datasets into training/testing (see Table 10.1 for the splits) and performed cross validation.

**Table 10.1** Class-prediction accuracy with different $p$ settings

| Dataset | Samples (train/test) | LIBSVM | $p = n^{0.1}$ | $p = n^{0.2}$ | $p = n^{0.3}$ | $p = n^{0.4}$ | $p = n^{0.5}$ |
|---|---|---|---|---|---|---|---|
| *svmguide1* | 3,089/4,000 | 0.9608 | 0.6563 | 0.9 | 0.917 | 0.9495 | 0.9593 |
| *mushrooms* | 7,500/624 | 1 | 0.9904 | 0.9920 | 1 | 1 | 1 |
| *news20* | 18,000/1,996 | 0.7835 | 0.6949 | 0.6949 | 0.6969 | 0.7806 | 0.7811 |
| *Image* | 199,957/84,507 | 0.849 | 0.7293 | 0.7210 | 0.8041 | 0.8121 | 0.8258 |
| *CoverType* | 522,910/58,102 | 0.9769 | 0.9764 | 0.9762 | 0.9766 | 0.9761 | 0.9766 |
| *RCV* | 781,265/23,149 | 0.9575 | 0.8527 | 0.8586 | 0.8616 | 0.9065 | 0.9264 |

experimented. We use Gaussian kernel, and select the best $C$ and $\sigma$ for LIBSVM and PSVM, respectively. For *CoverType* and *RCV*, we loosed the terminate condition (set -e 1, default 0.001) and used shrink heuristics (set -h 1) to make LIBSVM terminate within several days. The table shows that when $t$ is set to 0.5 (or $p = \sqrt{n}$), the class-prediction accuracy of PSVM approaches that of LIBSVM.

We compared only with LIBSVM because it is arguably the best open-source SVM implementation in both accuracy and speed. Another possible candidate is CVM [12]. Our experimental result on the *CoverType* dataset outperforms the result reported by CVM on the same dataset in both accuracy and speed. Moreover, CVM's training time has been shown unpredictable by [20], since the training time is sensitive to the selection of stop criteria and hyper-parameters. For how we position PSVM with respect to other related work, please refer to our disclaimer in the end of Sect. 10.1.

### *10.4.2 Scalability*

For scalability experiments, we used three large datasets. Table 10.2 reports the speedup of PSVM on up to $m = 500$ machines. Since when a dataset size is large, a single machine cannot store the factorized matrix $H$ in its local memory, we cannot obtain the running time of PSVM on one machine. We thus used 10 machines as the baseline to measure the speedup of using more than 10 machines. To quantify speedup, we made an assumption that the speedup of using 10 machines is 10, compared to using one machine. This assumption is reasonable for our experiments, since PSVM does enjoy linear speedup when the number of machines is up to 30.

We trained PSVM three times for each dataset-m combination. The speedup reported in the table is the average of three runs with standard deviation provided in brackets. The observed variance in speedup was caused by the variance of machine loads, as all machines were shared with other tasks running on our data centers. We can observe in Table 10.2 that the larger is the dataset, the better is the speedup. Figure 10.3a–c plot the speedup of *Image*, *CoverType*, and *RCV*, respectively. All

**Table 10.2** Speedup ($p$ is set to $\sqrt{n}$); LIBSVM training time is reported on the last row for reference

| | Image (200k) | | CoverType (500k) | | RCV (800k) | |
|---|---|---|---|---|---|---|
| Machines | Time (s) | Speedup | Time (s) | Speedup | Time (s) | Speedup |
| 10 | 1,958 (9) | 10* | 16,818 (442) | 10* | 45,135(1373) | 10* |
| 30 | 572 (8) | 34.2 | 5,591 (10) | 30.1 | 12,289 (98) | 36.7 |
| 50 | 473 (14) | 41.4 | 3,598 (60) | 46.8 | 7,695 (92) | 58.7 |
| 100 | 330 (47) | 59.4 | 2,082 (29) | 80.8 | 4,992 (34) | 90.4 |
| 150 | 274 (40) | 71.4 | 1,865 (93) | 90.2 | 3,313 (59) | 136.3 |
| 200 | 294 (41) | 66.7 | 1,416 (24) | 118.7 | 3,163 (69) | 142.7 |
| 250 | 397 (78) | 49.4 | 1,405 (115) | 119.7 | 2,719 (203) | 166.0 |
| 500 | 814 (123) | 24.1 | 1,655 (34) | 101.6 | 2,671 (193) | 169.0 |
| LIBSVM | 4,334 NA | NA | 28,149 NA | NA | 184,199 NA | NA |

datasets enjoy a linear speedup[2] when the number of machines is moderate. For instance, PSVM achieves linear speedup on *RCV* when running on up to around 100 machines. PSVM scales well till around 250 machines. After that, adding more machines receives diminishing returns. This result led to our examination on the overheads of PSVM, presented next.

### *10.4.3 Overheads*

PSVM cannot achieve linear speedup when the number of machines continues to increase beyond a data-size-dependent threshold. This is expected due to communication and synchronization overheads. Communication time is incurred when message passing takes place between machines. Synchronization overhead is incurred when the *master* machine waits for task completion on the slowest machine. (The *master* could wait forever if a child machine fails. We have implemented a checkpoint scheme to deal with this issue.)

The running time consists of three parts: computation (Comp), communication (Comm), and synchronization (Sync). Figure 10.3d–f show how Comm and Sync overheads influence the speedup curves. In the figures, we draw on the top the computation only line (Comp), which approaches the linear speedup line. Computation speedup can become sublinear when adding machines beyond a threshold. This is because the computation bottleneck of the unparallelizable step 12 in Fig. 10.3 (which computation time is $\mathcal{O}(p^2)$). When $m$ is small, this bottleneck is insignificant in the total computation time. According to the Amdahl's law; however, even a small fraction of unparallelizable computation can cap speedup. Fortunately, the

[2] We observed super-linear speedup when 30 machines were used for training *Image* and when up to 50 machines were used for *RCV*. We believe that this super-linear speedup resulted from performance gain in the memory management system when the physical memory was not in contention with other processes running at the data center. This benefit was cancelled by other overheads (explained in Sect. 10.4.3) when more machines were employed.

larger the dataset is, the smaller is this unparallelizable fraction, which is $\mathcal{O}(m/n)$. Therefore, more machines (larger $m$) can be employed for larger datasets (larger $n$) to gain speedup.

When communication overhead or synchronization overhead is accounted for (the Comp + Comm line and the Comp + Comm + Sync line), the speedup deteriorates. Between the two overheads, the synchronization overhead does not impact speedup as much as the communication overhead does. Figure 10.3g–i present the percentage of Comp, Comm, and Sync in total running time. The synchronization overhead maintains about the same percentage when $m$ increases, whereas the percentage of communication overhead grows with $m$. As mentioned in Sect. 10.3.1, the communication overhead is $\mathcal{O}(p^2 \log(m))$, growing sub-linearly with $m$. But since the computation time per node decreases as $m$ increases, the fraction of the communication overhead grows with $m$. Therefore, PSVM must select a proper $m$ for a training task to maximize the benefit of parallelization.

## 10.5 Concluding Remarks

In this chapter, we have shown how SVMs can be parallelized to achieve scalable performance. PSVM distributively loads training data on parallel machines, reducing memory requirement through approximate factorization on the kernel matrix. PSVM solves IPM in parallel by cleverly arranging computation order. Through empirical studies, we have shown that PSVM does not sacrifice class-prediction accuracy significantly for scalability, and it scales well with training data size. Open source code of PSVM was made available at [2].

## References

1. E.Y. Chang, K. Zhu , H. Wang, H. Bai, J. Li, Z. Qiu, H. Cui, Parallelizing support vector machines on distributed computers, in *Proceedings of NIPS*, 2007
2. E.Y. Chang, H. Bai, K. Zhu, H. Wang, J. Li, Z. Qiu, Google PSVM open source. http://code.google.com/p/psvm/
3. V. Vapnik, The Nature of Statistical Learning Theory. (Springer, New York, 1995)
4. S. Mehrotra, On the implementation of a primal-dual interior point method. SIAM J. Optim. **2**, 575–601 (1992)
5. T. Joachims, Making large-scale SVM learning practical, in *Advances in Kernel Methods—Support Vector Learning*, ed. by B. Schölkopf, C. Burges, A. Smola (MIT Press, 1999)
6. C.C. Chang, C.J. Lin, LIBSVM: a library for support vector machines, 2001
7. J. Platt, Sequential minimal optimization: a fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, 1998
8. S. Vishwanathan, A.J. Smola, M.N. Murty, Simple SVM, in *Proceedings of ICML*, 2003, pp. 760–767
9. T. Joachims, Training linear SVMs in linear time, in *Proceedings of ACM KDD*, 2006, pp. 217–226
10. C.T. Chu, S.K. Kim, Y.A.. Lin, Y. Yu, G. Bradski, A.Y. Ng, K. Olukotun, Map reduce for machine learning on multicore, in *Proceedings of NIPS*, 2006, pp. 281–288

11. Y.J. Lee, S.Y. Huang, Reduced support vector machines: a statistical theory. IEEE Trans. Neural Networks **18**(1), 1–13 (2007)
12. I.W. Tsang, J.T. Kwok, P.M. Cheung, Core vector machines: fast svm training on very large data sets. J Mach. Learn. Res. **6**, 363–392 (2005)
13. H.P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, V. Vapnik, Parallel support vector machines: The cascade svm, in *Proceedings of NIPS*, 2005, pp. 521–528
14. S. Boyd, *Convex Optimization* (Cambridge University Press, Cambridge, 2004)
15. G.H. Golub, C.F.V. Loan, *Matrix Computations* (Johns Hopkins University Press, Baltimore, 1996)
16. S. Fine, K. Scheinberg, Efficient svm training using low-rank kernel representations. J. Mach. Learn. Res. **2**, 243–264 (2001)
17. F.R. Bach, M.I. Jordan, Predictive low-rank decomposition for kernel methods, in *Proceedings of International Conference on Machine Learning* (*ICML*), 2005, pp. 33–40
18. J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in *Proceedings of OSDI: Symposium on Operating System Design and Implementation*, 2004, pp. 137–150
19. S. Ghemawat, H. Gobioff, S.T. Leung The Google file system, in *Proceedings of 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43
20. G. Loosli, S. Canu, Comments on the core vector machines: fast SVM training on very large data sets. J. Mach. Learn. Res. **8**, 291–301 (2007)