Parallel Computing of Support Vector Machines: A Survey

SHIRIN TAVARA, University of Borås and University of Skövde, Sweden

The immense amount of data created by digitalization requires parallel computing for machine-learning methods. While there are many parallel implementations for support vector machines (SVMs), there is no clear suggestion for every application scenario. Many factor—including optimization algorithm, problem size and dimension, kernel function, parallel programming stack, and hardware architecture—impact the efficiency of implementations. It is up to the user to balance trade-offs, particularly between computation time and classification accuracy. In this survey, we review the state-of-the-art implementations of SVMs, their pros and cons, and suggest possible avenues for future research.

CCS Concepts: • **Computing methodologies → Support vector machines**; *Parallel algorithms*; *Ensemble methods*; *Feature selection*; *Cross-validation*;

Additional Key Words and Phrases: Dual optimization, primal optimization, decomposition, CPU parallelism, GPU parallelism, speedup, data movement

ACM Reference format:
Shirin Tavara. 2019. Parallel Computing of Support Vector Machines: A Survey. *ACM Comput. Surv.* 51, 6, Article 123 (February 2019), 38 pages.
https://doi.org/10.1145/3280989

1 INTRODUCTION

High-Performance Computing (HPC) [1] and parallel computing are promising tools for improving the performance of machine-learning algorithms in terms of time, especially for large-scale problems. The use of Support Vector Machines (SVMs) [2] is a supervised machine-learning technique that can take advantage of HPC. SVM use is a popular technique because of the good generalization performance on much real-life data [3, 4]. Generalization performance shows how accurately unseen data can be classified by a pattern classifier [3, 5]. Despite the advantages of SVMs, their use suffers from a long training process and limited memory for solving large-scale problems [3, 6, 7, 8, 9, 10]. The reason is that the Quadratic Programming (QP) [11] problem addressed by SVMs contains computationally expensive tasks [11, 12] including matrix-related operations [11] for kernel computations, checking optimality conditions [13], and gradient updating tasks [14, 15].

In the digital era, the size of data has been growing exponentially; thus, the computation time and memory requirements regarding very large problems have been increasing and using HPC tools has become even more important [16]. The published results show that parallel SVMs can achieve considerable speedups compared to sequential SVM algorithms that use only a single CPU

This work is supported by the Universities of Borås, Skövde and Gothenburg in Sweden. The online tables are supported by Schliep's lab (https://schlieplab.org/).
Authors' address: S. Tavara, University of Borås, 50190 Borås, Sweden and University of Skövde, Högskolevägen, Box 408, Skövde, 541 28, Sweden; email: tavara.shirin@gmail.com.

0360-0300/2019/02-ART123
https://doi.org/10.1145/3280989

[10]. Parallelism has the potential to improve the performance of SVMs in terms of time and memory; however, parallel implementations of SVMs are far beyond easy tasks [17] and may become inefficient for solving very large problems or using a large number of processors, that is, they may not scale well to a large number of samples or processors. Problems such as communication overhead, computationally dependent steps, and memory limitations degrade the efficiency of parallelism [6]. In addition, coding part of a parallel SVM algorithm is difficult and requires considerable skills [3]. Due to the high space and time complexity of SVM algorithms, it is important to use the correct and appropriate algorithms and efficient heuristics for the given problem. Thereby, to identify the appropriate parallel approaches that have potential to obtain peak performance for the given large-scale problem, it is helpful to thoroughly study the efficiency of existing parallel approaches.

Tyree et al. [18] categorize the approaches used for the parallelization of SVMs into implicit and explicit parallelization. Inimplicit parallelization [18], an algorithm is written as a series of operations that can be computed using highly optimized parallel libraries, for example, the Intel Math Kernel Library (MKL) [19] and cuBLAS [20]. In explicit parallelization [18], programmers parallelize computationally expensive tasks using parallel programmings, for example, shared memory, distributed memory, and GPUs. The review conducted by Tyree et al. covers most of the existing parallel implementations of SVM and shows the importance of implicit parallelism. Despite many advantages of their review, it does not consider the important parallel algorithmic approaches and paradigms, for example, cascade and map-reduce, and it considers only SMO-type optimization methods. Lu et al. [21] review the mathematical optimization approaches used for accelerating the training process of SVM. The survey considers only the GPU-based parallel implementations of SVMs. To our knowledge, there is no survey that thoroughly reviews the parallel algorithmic approaches along with parallel tools for implementations of the SVM.

The objective of this article is to provide a summary of the parallel algorithmic approaches and parallel tools that have been used for implementations of the SVM in order to provide insights into efficient and potential approaches for solving large-scale problems. In addition, the article provides a brief summary of promising heuristics, their advantages, and challenges in parallelism. In this article, the dominant and promising parallel approaches that can be the target of future studies for further improvements are identified. We review parallel implementations of the SVM with respect to four focus lines that we have identified as the important goals of parallelism.

Two of the focus lines are speedup and memory, that is, the parallel implementations of an SVM may focus on improving the performance of SVM algorithms in terms of time and memory for large-scale problems [18, 21]. Parallel approaches that reduce problem size and handle memory issues may face deterioration in classification accuracy. Another focus line is accuracy, that is, parallel approaches may focus on improving or maintaining accuracy while reducing computation time. The fourth focus line is scalability, that is, parallel algorithms may focus on scaling well for a large number of training samples and a large number of processors or focus more on minimizing overheads.

The outline of the article is as follows. An overview of SVMs is described in Section 2. The methodology for choosing and reviewing the publications regarding parallel SVMs and the corresponding taxonomy are described in Section 3. A review of the parallel algorithmic techniques and models that have been used for SVMs is briefly described in Section 4. A discussion and brief summary are presented in Section 5. The conclusion and potential future work are presented in Section 6.

2 WHAT ARE SUPPORT VECTOR MACHINES?

An SVM is a supervised machine-learning technique developed by Vapnik et al. [2] from statistical learning theory to solve classification and regression problems [2, 11, 22]. The basic idea of an

SVM in a simple binary classification problem is to search for the hyperplane that is the farthest to the closest training data points from both sides of the hyperplane [23, 24]. This process has two phases, training and testing. In the training phase, the machine is trained to find a hyperplane that separates the given data samples into two classes with known labels, negative or −1 and positive or +1. After the machine is trained, the training model is extracted and then the testing phase is carried out. In the testing phase, the SVM model predicts which class label a new unseen test sample should have [22]. As mentioned in Section 1, an SVM gives a good generalization performance [22] and minimizes the upper bound of the generalization error [3]. An SVM has special characteristics that are used to implement efficient parallel algorithms in terms of time and memory. One characteristic is that the solution to a classification problem is obtained by only a few samples, called Support Vectors (SV) [2], that determine the maximum margin separating hyperplane [25]. Another characteristic of SVMs is to perform nonlinear mapping without knowing the mapping function using predefined functions called kernels for calculating the inner product of mapping functions [25]. Other characteristics of SVMs are the simple structure of constraints and the special definition of the kernel function in a linear case, that is, the inner product is a simple dot product [26]. The sparsity of solutions is the next characteristic of SVMs [27]. The primal optimization problem addressed by SVMs is as follows:

$$\begin{aligned} &min \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i \\ &s.t. \quad \forall i: \quad y_i(\mathbf{w}^T\Phi(\mathbf{x}_i)+b) \geq 1-\xi_i, \quad i=1,2,\ldots,N \\ &\qquad \forall i: \quad \xi_i \geq 0, \quad i=1,2,\ldots,N. \end{aligned} \tag{1}$$

Here, $\mathbf{w}$ is the weight vector for the hyperplane, $\mathbf{x}$ is a vector of observations, y_i is the class labels and $y_i \in \{+1,-1\}$, b is the bias parameter, and $\Phi(\mathbf{x})$ is the map function. In cases in which data can be classified by a linear classifier, $\Phi(\mathbf{x}) = \mathbf{x}$, but real-life data cannot always be classified by a linear classifier [3]. In non-linear cases, one can map the data from the input space into a high-dimensional feature space using a non-linear transformation, that is, $\Phi(\mathbf{x})$ maps the input vector $\mathbf{x}$ to the feature space [3]. In the feature space, the data can be linearly separable. Consequently, the dual form of Equation (1) is represented in Equation (2).

$$\begin{aligned} &min \quad D(\alpha) = \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \mathbf{1}^T\boldsymbol{\alpha} \\ &s.t. \quad \sum_{i=1}^{N} y_i\alpha_i = 0, \quad i=1,2,\ldots,N \\ &\forall i: \quad 0 \leq \alpha_i \leq C, \quad i=1,2,\ldots,N. \end{aligned} \tag{2}$$

Here, $\boldsymbol{\alpha}$ is the vector of Lagrangian multipliers ($\alpha_i \in \boldsymbol{\alpha}$), $\mathbf{1}^T$ is a vector of ones, Q is a matrix of size $N \times N$, and $Q_{ij} = y_i y_j \Phi^T(\mathbf{x}_i)\Phi(\mathbf{x}_j)$.

Kernel functions. In order to compute matrix Q, it is sufficient to compute the inner product of $\Phi(\mathbf{x}_i)$ and $\Phi(\mathbf{x}_j)$ without knowing the $\Phi(\mathbf{x})$ function. This is done through a predefined kernel function, $K(\mathbf{x}_i, \mathbf{x}_j) = \Phi^T(\mathbf{x}_i)\Phi(\mathbf{x}_j)$. The kernel matrix measures the similarity or the distance between vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ [22]. Examples of well-known kernel functions are presented in Table 1. The kernel function defines the feature space where the training samples are classified; therefore, the selection of an appropriate kernel function is important [28].

Multi-class classification. SVMs can solve multi-class classifications either by considering the multi-class classification in the optimization problem or through decomposing the multi-class classification into a series of binary-class classification [29]. The latter group is popular and includes

Table 1. Examples of Well-Known Kernel Functions

Kernel Function	Inner Product	Kernel Type
Linear kernel	$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$	Linear
Gaussian/Radial-Basis Function (RBF)	$K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma^2)$	Non-linear
Polynomial	$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i.\mathbf{x}_j + const)^d$	Non-linear
Laplacian	$K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|)$	Non-linear

One-Versus-One (OVO) [30] (also known as One-Against-One), One-Versus-All (OVA) [30] (also known as One-Against-All), and Directed Acyclic Graphs (DAGs) [29, 31]. In OVO, all binary combinations of N classes are created, thus $N(N-1)/2$ classifiers are built, where N is the number of classes [29]. A DAG is a directed graph that combines the results of OVO classifiers [31]. In OVA, the samples of a specific class (class i) will be considered as the positive class and all remaining samples will be considered as the negative class. Thereby, N different classifiers are built [29]. Multi-class classifications are computationally expensive and suffer from long training time because many classes are involved in classification and owing to passing through the training data many times [32]. For a large number of classes, OVO is more computationally expensive than OVA [29].

2.1 Challenges

The training phase of an SVM involves a dense QP; solving such a QP is computationally expensive since it involves computations of a Hessian/kernel matrix [33, 34]. Kernel evaluations include computationally expensive tasks [14, 35], that is, matrix-vector and matrix-matrix multiplications, gradient function updates [14] and optimality condition updates [32, 36]. The old and general-purpose QP solvers are no longer suitable for solving the QP addressed by SVMs [3, 11, 22, 33] since many QP solvers require computing and storing the kernel matrix in the memory, which is not always possible owing to memory limitations [37]. One efficient approach to improving the performance of SVMs is parallelism; however, the parallel approach used in the general-purpose QP solvers may not take advantage of the special characteristics of SVMs mentioned in Section 2 [14, 26, 38, 39] or they may not be easily parallelizable [26]. In addition, the immense size of real-life data can cause problems regarding computationally expensive tasks as follows.

— **Memory.** The whole dataset may not fit into the memory [34] or inefficient memory access slows down the training and testing phases [40].
— **Speedup.** The matrix operations might take too long time to be performed due to the computationally expensive tasks, for example, matrix-vector and matrix-matrix multiplications [14], and overheads [41].
— **Scalability.** Algorithms may not scale to a large number of processors or a large number of samples [26].
— **Accuracy.** Approximation methods [42] for reducing the size of the problem may lead to poor classification accuracy [43].

Some efficient heuristics along with parallel approaches have been used to speed up the optimization addressed by SVMs and to further handle the issues mentioned regarding memory, speedup, scalability, and accuracy. A brief overview of the common heuristics and tcorresponding challenges are mentioned below.

— **Grid Search And Cross-Validation.** One can improve the accuracy of an SVM model by selecting appropriate values for the model's parameters. This is done through grid search and cross-validation [44], in which a grid of different value sets of parameters is generated and, in each value set, cross-validation is conducted. In n-fold cross-validation, the training set is randomly divided into n subsets with almost equal sizes. In each fold, one subset is used as the validation set for testing the model and $(n-1)$ subsets are used as the training set [45]. The cross-validation process is computationally expensive owing to recomputing kernel matrix values at each iteration [44]. The process gets more computationally expensive for multi-class classifications since recomputations of the kernel elements may be repeated for each fold.

— **Caching.** Kernel evaluations are computationally expensive and every evaluation of $K(x_i, x_j)$ requires at least $O(d)$ flops, where d is the number of features [15]. Consequently, computing a submatrix of size $s \times m$ requires at least $O(smd)$ flops at each training step, where s is the number of rows and m is the number of columns [15]. Many computations in kernel evaluations are repeated or unused; therefore, one can reduce the memory requirements and computation time by avoiding recomputations of the kernel evaluations and storing only the previously computed values in the memory using caching [15]. To do so, caching stores a number of rows of the kernel matrix as the memory allows. One of the common updating strategies for caching is the Least Recently Used (LRU) [46] technique [46, 47]. Caching has been well studied and has been a popular heuristic in parallel implementations of SVMs [26, 27, 47–53]. Although it has advantages, caching is difficult and system dependent [3, 36] and might be inefficient for a large number of training samples [54, 55]. The reason is that the number of the cached rows of the kernel matrix is small owing to limited memory; thus, the size of the active sets will not be large enough to achieve fast optimization [54, 56] or, if an algorithm is memory bound, the memory access cost will be high and multi-threading might cause memory contention owing to limited bandwidth [55].

— **Shrinking.** Based on empirical studies and in practice, the number of SVs is much less than the total number of training samples. In order to solve the QP faster in each iteration, one can find SVs in advance and perform the training only on the SVs and discard the non-SVs in the optimization process. This makes the QP smaller and faster than the QP for all training samples in each iteration and obtains the same optimal result [57, 58]. Joachims et al. [57] proposed a strategy called shrinking that temporarily eliminates points that are unlikely to be selected in the working set in each iteration. To do this, Joachims et al. [57] temporarily discard samples for which the corresponding Lagrangian multipliers reach a predefined upper or lower bound. Shrinking reduces the computations regarding kernel evaluations since only parts of the Hessian matrix that correspond to SVs are calculated [48, 59]. Although shrinking has been used in parallel implementations of SVMs [13, 17, 46, 48, 55, 60, 61], except in a very low number of articles, the process of shrinking has not been parallelized [62]. Despite the advantages, shrinking requires rearranging the index, marshaling, and reconstruction, which may cause significant overhead; therefore, it may lead to performance loss. One should get enough information from datasets and the problem parameters to decide how to carefully use shrinking [55].

3 METHODOLOGY

This survey is based on reviewing the publications and information in technical books, journals, conference proceedings, technical reports, authentic websites, and libraries used for parallel implementations of SVMs. There has been no special focus on applications of SVMs in a specific field such as medicine, biomedical, or finance. The selection of journal and conference articles was conducted using well-known databases, for example, IEEE, Elsevier, ACM Digital Library, and

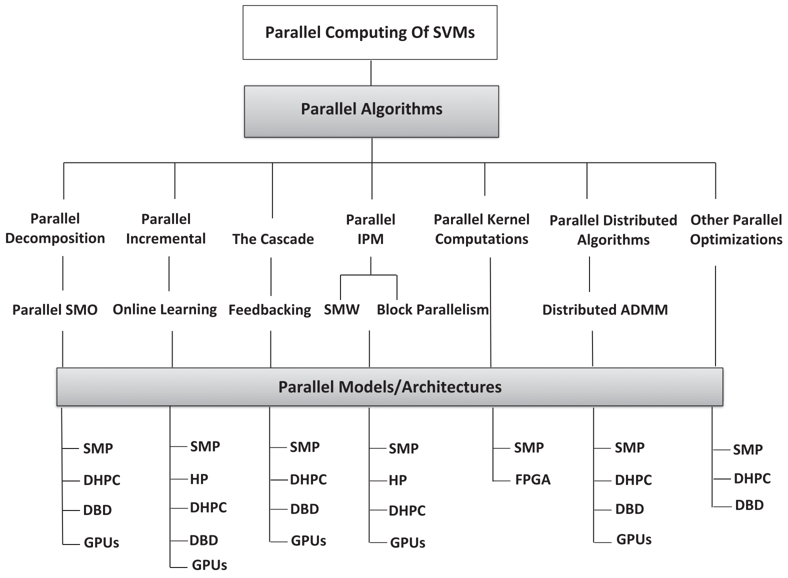

Fig. 1. The parallel computing of SVMs, including algorithms and tools. SMP: Shared Memory Parallelism. DHPC: Distributed HPC Architectures. HP: Hybrid shared-distributed memory Parallelism. GPUs: GPU-based Parallelism. DBD: Distributed Big Data architectures. FPGA: Field-Programmable Gate Arrays.

the Google Scholar search engine. Parallel SVM implementations that use parallel algorithmic approaches, parallel models, and frameworks are chosen regardless of their applications. In addition to the parallel approaches, the heuristics and strategies that improve the performance of the SVMs with respect to the four focus lines are mentioned in this survey. In addition, the parallel implementations of SVMs using Field-Programmable Gate Arrays (FPGAs) are briefly mentioned, although the customized and dedicated hardware for computational purposes is not the main focus of this survey. A review of publications regarding the sequential implementations of SVMs is excluded from this survey. The body of research regarding parallel computing of SVMs can be studied considering two aspects: parallel algorithms and parallel models related to parallel architectures. In this survey, the parallel SVMs are identified and categorized and, within each category, parallel models used for the parallel implementations of SVMs are reviewed. The resulting taxonomy of parallel SVMs is illustrated in Figure 1.

4 PARALLEL ALGORITHMIC APPROACHES AND PARALLEL MODELS

Parallelization of SVM algorithms is far beyond an easy task owing to problems such as dependencies between computation steps [17], high latency in memory access [40, 55], and limited memory [55]. In this section, the most common techniques and heuristics used for parallel implementations of SVMs are briefly described, each of which improves the efficiency of SVM algorithms in terms of memory, speedup, scalability, and accuracy.

4.1 Parallel Decomposition Techniques

Parallel decomposition implementations are popular for large-scale SVM problems since, at each iteration, it uses only a few samples in the working sets and skips the rest. However, decomposition techniques are inherently and essentially sequential. This is because the selection of working sets at each iteration depends on the results of the previous iteration, albeit some of the computationally expensive tasks/operations can be performed in parallel. In this section, the parallel decomposition-based SVMs are briefly described.

4.1.1 Parallel Decompositions Using Shared Memory Parallelism (SMP). SMP has been used for performing decomposition techniques in parallel [13, 15, 36, 60, 63–65]. To do so, computationally expensive tasks such as kernel evaluations, gradient updates, and working set selections [15, 65] are performed in parallel.

Memory and speedup. The advantage of parallel implementations of decomposition algorithms using SMP is that one copy of data is stored and shared between different processors/threads, avoiding multiple copies of data or communication overheads for transferring data [13]. Eitrich and Lang [65] use a simple loop-based parallelism for efficiently performing kernel evaluations, gradient updates, and selection of the working set in parallel. They employ BLAS routine libraries to accelerate matrix-vector and matrix-matrix multiplications. The advantage of parallel decomposition techniques is that, depending on the size of the working set, only a small subset of the large kernel matrix is computed in parallel [15]. However, if the appropriate size of the working set is large, the data may not fit in the limited shared memory. Going beyond a binary classification, Didiot et al. [15] reduce the memory requirements for multi-class classifications using kernel caching in which kernel elements are shared across all processors/threads. The proper data alignment in [15] allows vectorization of the kernel functions in which Single Instruction Multiple Data (SIMD) [66] instructions are used to apply the same operation on multiple components of large vectors. Unlike previous work, You et al. [55] remove the caching and reduce the shrinking frequency in SMO and focus on memory coverage using a two-level parallel mechanism in which tasks with independent computations and memory requirements, for example, kernel evaluations, are computed through task parallelism and tasks with dependency in the computations, for example, partial kernel evaluations, are computed through data parallelism. Unlike the previous works, this two-level parallelism supports both dense and sparse data formats. They were the first to introduce the use of modern processor technologies for vectorization.

Scalability and accuracy. Eitrich and Lang [36] introduce a data transformation in which an expensive operation in the kernel function is transformed into a less expensive operation, that is, a division is transformed to a multiplication. The data transformation along with the largest possible working set that fit in the memory leads to a speedup of 5.5 times using 7 threads for a binary classification. The data transformation allows the use of a Gaussian kernel without additional cost. However, the algorithm does not scale well to a large number of processors. In contrast, Gonçalves et al. [63] use Universal Kernel Function (UKF) [63], which has less computationally expensive operations than a Gaussian kernel. It yields a better speedup and a better or equal accuracy compared to a Gaussian kernel [63]. However, the impacts of using the UKF kernel on different applications are unknown. Chang et al. [64] take a step further and parallelize a decomposition-based algorithm for a multi-class classification, but they report only experiments using a linear kernel. Unlike previous work, Marzolla [60] parallelizes a decomposition-based algorithm using a specific multi-core processor called the cell processor. The algorithm obtains low speedups on datasets with few attributes due to transferring many small data blocks.

4.1.2 Parallel Decompositions Using Distributed HPC Architectures. Parallel implementations of SVMs using SMP can achieve considerable speedups. Still, owing to the limited capacity of shared memory, data may not fit in the memory. To overcome limited shared memory, distributed HPC architectures have been employed in which local memory is used without a single global memory space shared between processors. To do so, the computationally expensive tasks in a decomposition technique are distributed across multiple processors to be performed in parallel [14, 32, 44, 62, 67, 68]. In this regard, a particular decomposition algorithm called Sequential Minimal

Optimization (SMO) [22] has been popular [13, 25, 27, 32, 47, 64, 69–71] since it can analytically solve the QP addressed by SVM. The basic idea in SMO is that a large QP problem is divided into the smallest possible QP subproblems, each of which is solved using only two variables in their working set. SMO has been a target of distributed memory parallelism since around 90% of the total computation time is spent on updating the gradient function [32]. The most common approach for parallelization of SMO is to divide the training dataset into smaller subsets and then distribute the subsets into multiple processors where the local gradient functions can be calculated and updated in parallel [32, 72]. Shrinking [62, 67], caching [14], and two-level parallelism [55] are the most common strategies used for reducing the problem size and memory requirements.

Scalability and accuracy. The parallelization of SMO-based algorithms might achieve satisfactory speedups [32]; however, they may not show good scalability in terms of number of processors and training samples. This is because those algorithms often require going through the entire set of samples to select appropriate working sets [62] or processors may often need to communicate with each other to obtain the final result. This causes overhead. Narasimhan et al. [62] overcome scalability issues by adding adaptive shrinking, in which non-contributing training samples are eliminated; thus, the algorithm deals with only a part of the samples. Shrinking reduces the memory requirements along with accelerating the training process. The time complexity of the shrinking process is further reduced by performing the shrinking in parallel. In [62], the adaptive shrinking allows fast convergence. However, incorrect eliminations of samples in shrinking result in an inaccurate classification. Vishnu et al. [67] overcome this problem using several shrinking strategies ranging from early to late elimination of non-contributing samples, in which important data structures are synchronized using distributed memory parallelism to avoid false elimination. Their proposed algorithm shows good scalability, that is, it scales up to 4,096 cores (256 nodes) for a problem with 2.3 million samples. Although decomposition techniques have advantages, they may not converge if SVs do not fit in the memory owing to a phenomenon called *thrashing* in which a correction on a working set is canceled by the correction of another working set [72]. Yom-Tov [72] overcomes this problem using batch mode, in which every Lagrangian multiplier is updated at each iteration. Scalability of the proposed algorithm is unclear and the experiments report the performance for only 3 to 4 processors.

Memory and speedup. Shrinking might cause overhead if it is not handled carefully (see Section 2.1). One may skip inefficient shrinking and use caching instead (see Section 2.1). For instance, Brugger [14] uses parallel and distributed caching along with a sparse data presentation and large working sets in an SMO-based algorithm. The results of experiments show the linear and, in some cases, superlinear speedups for large problems. The approach in [14] supports different formulations of SVMs for classification and regression problems.

Speedup. While SMO is one of the most common decomposition methods used in SVMs, it may suffer from the existing dependencies between computation steps. To overcome this problem, Hazan et al. [73] proposed a different decomposition technique using Fenchel duality that is more suitable for distributed memory parallelism and for computing clusters of independent nodes with independent memory. The proposed algorithm uses a parallel block-update approach that can update and send back the results of all processors in an iterative manner with low communication overhead [73]. This leads to a speedup of $\frac{k}{2}$ using k processors.

4.1.3 Parallel Decompositions Using Distributed Big Data Architectures. These architectures have been used for parallel implementations of SMO [13, 43, 74].

Memory and scalability. The parallel algorithm proposed by Zhao et al. [13] reduces memory requirements using a map-reduce implementation of SMO along with caching and a sparse matrix representation. Caching reduces memory requirements since a unique copy of the cache is shared between all mappers, and tsparse data representation compresses the sample vectors to occupy the least amount of memory possible. Nevertheless, the sample vectors need to be re-formed back into the dense format to compute the corresponding dot product. The combination of caching, sparse data representation, and parallel implementation using map-reduce leads to a 4-fold speedup compared to Libsvm. Concerning memory usage, the algorithm could use a larger cache size and, concerning scalability, the algorithm shows only a slight time improvement for increasing number of cores. The reason is that the memory is not fully and efficiently used.

Accuracy. Distributed training of SVM may cause accuracy degradation. One can improve the accuracy using an ontology-based enhancement in which an end user corrects and modifies the training data [43]. To do so, the user adds some optimized instances (intelligence) as feedback to the weighted training data chunks and then all local SVMs are retrained in parallel. For instance, Caruana et al. [43] design a feedback-based approach for training and classification processes in which the optimized instances are added into the feedback loops, leading to an average of 5% accuracy improvement. The classification accuracy of this approach depends on the quality of provided intelligence by the end user, which requires expert knowledge in the domain.

The previous approaches mostly take homogeneous computing environments into consideration in which machines load equal amounts of data. The reason can be that most of the approaches perform the parallelism via Message Passing Interface (MPI) [75], and MPI was primarily designed for homogeneous computing environments [74]. In contrast, Hadoop implementation of map-reduce could use a cluster of distributed heterogeneous computers. However, it does not handle load balancing. One can balance loads for distributed heterogeneous computers using resource-aware algorithms. Alham et al. [74] balance the loads using a genetic algorithm scheme that distributes data chunks of different sizes into heterogeneous computers using map-reduce in which the processing times for all data chunks are equalized and the communication overhead is reduced.

4.1.4 Parallel Decompositions Using GPUs.

The computational power of GPUs has been used to speed up the computationally expensive kernel evaluations in decomposition techniques [10, 12, 27, 44, 49, 51, 76–82]. In GPU-based parallelism, using a sparse data format [77, 78, 80], selecting appropriate working sets [12, 47, 77] and adjusting data precision [27, 44, 47] are the most common strategies used for overcoming the limited memory bandwidth in GPUs and accelerating training/predicting processes for SVMs.

Memory and accuracy. The first GPU-based parallel SVM, gpuSVM, was implemented by Catanzaro et al. [47]. They reduced the memory requirements in gpuSVM and consequently accelerated the training/testing process in SVMs using single precision floating point arithmetic. Note that 32b arithmetic deteriorates accuracy [27, 44]. Carpenter [27] proposes a solution, cuSVM, to improve the accuracy using the mixture of single- and double-precision arithmetic. Although this mixture boosts the performance on dense samples, it does not duplicate the accuracy achieved by Libsvm and does not support cross-validation for improving accuracy [44]. In addition, cuSVM in many cases performs slower than gpuSVM.

Memory reduction through reducing precision may not lead to the desired results. Therefore, one may reduce memory requirements using a sparse format for storing large samples [77, 78, 80]. Sopyła et al. [78] show that simply using a compressed sparse row format leads to a speedup of 35 times; however, the algorithm in [78] supports those cases with sparse input data and not dense ones [28]. Cotter et al. [40] mention that the sparse format of data can reduce memory requirements, but it may not follow the certain restrictive memory access patterns on GPUs. Therefore, they use coalesced memory access for sparse data that fits the GPU architecture [40]. In contrast to previous works that focus on the parallelization of matrix-vector multiplication for updating gradients, Vaněk et al. [81] introduce an Optimized Hierarchical Decomposition SVM (OHD-SVM) [81], in which kernel values are computed via matrix-matrix multiplications that fit the GPU architecture better. OHD-SVM achieves significantly better performance than in [18, 40, 83] and achieves a speedup up to 12 times over the fastest published GPU-based SVM for binary classifications, gpuSVM [47]. It is the only implementation that supports both sparse and dense data formats with efficient loading of data from memory.

Cross-validation. It is used to enhance accuracy by finding appropriate values of corresponding parameters (see Section 2.1). It is computationally expensive; therefore, only a few parallel implementations support this process. Cross-validation contains independent tasks that can be computed in parallel using GPUs [44, 51, 53]. Kernel caching is used to accelerate cross-validation, although it needs to be performed carefully to avoid recomputation of the kernel at every iteration of cross-validation. One can avoid the recomputing of kernel values by calculating the kernel matrix only once for every fold of cross-validation [44, 51, 53, 76, 79]. This speeds up the process n times, where n is the number of folds in n-fold cross-validation [44]. Athanasopoulos et al. [44] parallelize the cross-validation process using the combination of CPU and GPU parallelism, which results in one order of magnitude faster processing time than using only CPU parallelism. One drawback of the proposed approach is that the algorithm loads the input data on the GPU memory and, owing to limited onboard memory on the GPU, a large dataset may not fit in the memory [44, 79].

Speedup. Size of working sets impact training/testing time in decomposition techniques [12, 47, 77]. The standard SMO chooses only two points for the working set. It is not clear that this is the best choice for GPU-based parallelism. One of the major differences of the work proposed by Liao et al. [77] with the standard SMO-based algorithms is that Liao et al. empirically choose around 16 to 32 points in the working set. The reason is that the proposed size is large enough to fit in the memory and small enough to accelerate the training process, and it gives a good balance between accuracy and solution time. Another difference is that a different kernel function, called *Tanimoto*, is used that shows a higher classification performance over widely used kernels for applications in molecular fingerprints. The algorithm can solve both classification and regression problems. The sparse format of storing input data, the data movement between GPU and CPU, caching, and mainly the change of working set size may describe why [77] achieves better speedups than in [47], even though the latter uses a highly parallel map-reduce model for binary classifications. Going beyond binary classifications, Herrero-Lopez et al. [76] implement the first OVA-based parallel SVM that improves the work proposed by Catanzaro et al. [47] for multi-class classifications and supports dense data formats.

Data storage format. One of the issues in parallel SVMs is that they mostly support one type of data format. For instance, the state-of-the-art SVM implementations in [80] and [47] support only compressed sparse row and dense data formats, respectively. You and Demmel [82] show that employing a unified data format for all datasets may significantly deteriorate the performance of the corresponding parallel algorithm in terms of storage, computation, and memory bandwidth. In [82], they use auto-tuning techniques to employ different formats for the same dataset. Their approach leads to the speedup of 4 (worst-case scenario) to 14 times (best-case scenario).

Multi-class classification. A multi-class classification is computationally expensive (see Section 2) and has been a target of GPU parallelism in decomposition techniques [49, 76, 80]. Herrero-Lopez et al. [76] use OVA to solve the N-class classification through N binary classifications in which a single SMO is solved similar to [47] and N(N× P) classifiers are constructed, where N is the number of class and P is the subsets. In [76], different tasks evoke only a single kernel, which causes task-controlling overhead owing to many memory transactions between the global memory. In contrast, Lin and Chien [80] use OVO (see Section 2) in which N(N-1)/2 classifiers are constructed. To overcome the task-controlling overhead in [76], Lin and Chien use a different caching strategy, in which every different task evokes a kernel at each iteration. Although this caching approach along with sparse matrix format gives a speedup of around 134 times over Libsvm, the sequential computation of the operations in each task is less efficient than those computed in parallel. Unlike previous works, Zhang et al. [49] reduce data transaction for a binary classification by simply moving data from global memory to shared memory since data access from shared memory on GPUs is faster than global memory. To further reduce the computational time, they use a scatter-parallel-reduce-gather reduction method. This simple modification significantly improves the access speed on GPUs; however, the limited shared memory can still be an issue for large working sets.

Prediction. Unlike previous works that focus on the training process, Peng et al. [12] mention that an appropriate size for working sets accelerates the predicting process more than the training process. This is because the training process is more complicated and its time expenditure depends on many factors beyond the working set size. In addition, some parts of the training process may not be parallelizable, whereas the predicting process is computationally intensive, in which almost all parts are easily parallelizable. Peng et al. [12] improve on the work initiated by Liao et al. [77] for parallelization of both training and predicting processes in SVMLight. We highlight the findings, pros, and cons of the parallel decomposition-based SVMs mentioned in this subsection in online table Decomposition.

4.2 Parallel Incremental SVMs

Incremental learning is another approach that has been used for parallel implementations of SVMs [10, 30, 84–93]. The basic idea in incremental learning is that a partition of data is processed and corresponding SVs are identified. At each step, the SVs of the previous step together with a new data partition are added as inputs of the current step. The process is terminated when no partitions of data are left. Incremental learning can be used for online SVM training when new data are added [9]. Note that one may consider online learning as a solution for incremental learning [90]; in a few cases, they have been used interchangeably [84]. Incremental learning has the potential to solve problems regarding scalability and memory for large-scale problems since it divides large samples into several subsets, each of which fits into the main memory [9, 92]. The main focus line considered in incremental learning is memory handling for large-scale problems.

4.2.1 Parallel Incremental SVMs Using SMP. An incremental learning approach has been used for data that arrives in a streaming fashion. To our knowledge, the algorithm proposed by Matsushima et al. [87] is the only SMP-based incremental learning of an SVM for a linear classification. They use POSIX multi-threaded programming for streaming data. The algorithm identifies insignificant data, that is, non-SVs, as data is added. The memory requirements are further reduced by processing data in blocks and by trading off the file I/O using data expand on the fly; that is, features are generated on demand. Furthermore, data is cached using software that provides a thread-safe in-memory hash table called *Kyoto Cabinet* [94]. One drawback is that the proposed algorithm requires multiple passes through the data to achieve optimal performance. This may cause considerable overhead, although after one or two passes, a close to optimal performance is achieved.

4.2.2 Hybrid Parallelism. SMP may be unable to handle multi-class classifications due to limited memory, although incremental learning to some extent handles limited memory issues by processing a partition of data at a time. To overcome limited memory issues, Doan et al. [29] use a hybrid of shared and distributed memory parallelism for implementing a multi-class classification problem. This is because using only distributed memory via MPI requires loading the whole subset into the memory for the learning process, and this may not be possible for very large subsets. They add SMP to avoid loading the whole subset for each MPI process. The algorithm handles the class imbalance problem that occurs when the number of examples in classes is very unbalanced. They overcome the problem using a balanced bagging incremental approach in which the number of samples in the majority class is reduced, known as *undersampling*.

4.2.3 Parallel Incremental SVMs Using Distributed HPC Architectures. Incremental learning of SVMs has been performed in parallel using distributed HPC architectures [85, 88, 89, 93].

Scalability. One of the effective strategies for training large SVM problems is using row- or column-based distributed algorithms in which data is split into blocks of rows or columns and then the corresponding matrix multiplications are computed incrementally in parallel. Do and Poulet [88] study both row- and column-based block distribution of data. The row-based approach has linear dependencies on the number of examples, the number of machines, and a second order depending on the number of dimensions. Therefore, the row-based approach is suitable for a large number of samples with a small enough number of dimensions. This is the opposite of the column-based approach. To adapt the row-based algorithm in [88] to solve problems with high-dimensional input space, the Sherman-Morrison-Woodbury (SMW) [95] formula has been applied in which the inverse of a large matrix is computed in an efficient and inexpensive way that reduces memory requirements and the runtime. To our knowledge, the algorithm in [88] is one of a few algorithms that can handle the biggest samples reported in the experiments.

Incremental learning can be further accelerated by avoiding retraining old data when new data is added [84, 85], that is, the classifier is incrementally updated only when a new point is added. Unlike in [88], Tveit and Engum [85] use a tree-based structure to allow the distributed nodes to access their input data efficiently [85, 89]. They use a heap-based tree topology for efficient data access for linear classifications. The heap-numbering of the nodes creates efficient communications between the nodes in which each node of the tree efficiently calculates the addresses of the corresponding child and parent nodes. The results of experiments regarding the proposed algorithm show that the reading on all nodes in the tree outperforms the reading on only leaf nodes. The algorithm scales linearly with the number of cores. This is because increasing the increment size leads to increasing speedups and each node deals with heavy processing with minor communications [85].

Scalability for different network topologies. Incremental learning is an efficient technique to reduce communications between processors by avoiding retraining of old data. In this regard, choosing an appropriate network topology with efficient communications of processors can reduce overhead. Incremental learning of SVMs has been implemented using various networks: a centralized network [92], a fully decentralized network [89], and a strongly connected network [90].

Centralized networks. In this type of network, distributed nodes solve their associated local SVMs independently; then, the resulting local SVs are aggregated in a central processing center. The parallel algorithm proposed by Syed et al. [92] uses this approach, but it does not succeed in obtaining the global optimum. Caragea et al. [93] improve on the work in [92] through sending SVs back from the center node to the distributed nodes and repeat the process until the global optimal solution is achieved. The centralized network may suffer from communication and synchronization overhead if the slave nodes regularly need to communicate with the master.

Decentralized networks. In this type of network, data are distributed across several nodes, each of which communicates only with its neighboring nodes, without needing to communicate with the master or central node. The advantage of a decentralized network is that, if a node, including the master, fails for any reason, the network remains connected; thus, the algorithm remains operational [89]. To further reduce overhead, the Alternating Direction Method of Multipliers (ADMM) [89] has been designed to handle exchanging messages only between neighboring nodes (see Section 4.6.2). To do this, Forero et al. [89] cast a centralized SVM as a set of decentralized optimization subproblems using consensus constraints. One of the advantages of this approach is that the proposed SVM is fully distributed in which nodes do not exchange training data. This leads to minor communication overhead and good scalability in terms of the number of computing nodes. Note that the topology of the network can affect the efficiency of the algorithm. Although the convergence of this method is guaranteed, poor choices of values for the corresponding parameters have negative impacts on convergence and the algorithm may require many iterations to obtain the desired result. The algorithm supports only binary classifications and has not been evaluated for large samples.

Strongly connected networks. In this type of network, each node receives its input data from its ancestor. The received data together with the new data are used to train the local subproblems and then the resulting SVs are sent to the descendent nodes. For instance, Lu et al. [90] show that their incremental learning on a strongly connected network converges in a small number of iterations and the computing time is independent of the number of SVs at each iteration; thus, the algorithm scales to a large size of training samples. One should take into the consideration that increasing the size of the network causes high communication overheads [90].

4.2.4 *Parallel Incremental SVMs Using Distributed Big Data Architectures.*

Map-reduce has been used for the parallelization of kernel computations and has been combined with incremental learning to reduce memory requirements and accelerate the corresponding processes. For instance, He et al. [84] use incremental learning along with a map-reduce technique to handle large-scale problems in which the computations regarding new data are performed without redoing the computations regarding the old data. The computational complexity of the proposed algorithm increases linearly in terms of the amount of new data, where the size of the new data is much smaller than the size of the training samples. The proposed algorithm does not take the multi-class classifications into consideration and does not scale to a large number of cores owing to the overhead.

4.2.5 *Parallel Incremental SVMs Using GPUs.*

GPUs have been used for the parallelization of incremental learning of SVMs [10, 91]. In this regard, one may need to minimize memory requirements due to the limited memory on GPUs. To do so, one can avoid loading the entire set of training samples into the main memory and update the solution with a growing set. In each incremental step, a block of rows/columns is loaded into the CPU memory and then the block is copied from the CPU memory into the GPU memory. After all the results are computed in parallel using the GPU, they are uploaded from the GPU memory back to the CPU memory. Do et al. [91] use Newton SVM formulation, which requires only the solution of a system of linear equations instead of QP. They incrementally compute the corresponding gradient function and Hessian matrix for each iteration. The algorithm in [91] linearly classifies billions of data points on a standard workstation if the dimension of the input space is small enough, i.e., less than 10^3. For problems with large training samples and high dimensional input space, one may first use dimension reduction strategies such as Principle Components Analysis (PCA) [96]. However, reducing the dimension of large-scale problems is difficult and computationally expensive. Do et al. in [10] improve the algorithm in [91] by exchanging the Newton SVMs optimization into a least square optimization formulation (LS-SVM) since the proposed LS-SVM converges faster than the Newton method and, thus, leads to better performance. The proposed LS-SVM is 1,000 times faster than Libsvm for linear classifying of 5 million data points. We show brief comparisons of the parallel incremental SVMs in the online table Incremental.

4.3 The Cascade

The cascade [17, 97] is a parallel scheme that has been used for the parallelization of SVMs [97–109] in which SVM subproblems are trained in different layers. In the first layer, training samples are split into smaller subsets, each of which is individually and independently trained by an SVM subproblem to find the corresponding SVs. The results of subproblems are combined pairwise and sent as input for the next layer of the cascade. This process continues until only one set of training samples remains. In order to check global convergence, SVs together with non-SVs in the last layer of the cascade are fed back into the first layer. The cascade finds the global solution if SVs that are fed to the first layer are the same as the SVs coming out of the first layer [17].

4.3.1 Parallel Cascade Using SMP. SMP has been used for the parallelization of the cascade of SVMs [97, 99].

Memory and accuracy. Three points impact the training time and accuracy of the cascade; (1) how a problem can be efficiently divided into smaller subproblems, (2) how SVs are identified in each layer [97], and (3) how and which feedbacks can be given to get the maximum improvement in accuracy and time [97, 99]. Regarding the second point, Zhang et al. [97] and Hu and Hao [99] filter out insignificant data using the distance mean of samples so that if the distance of a data point from the hyperplane is less than the mean, the point is chosen as an SV [97, 99]. Zhang et al. [97] improve the accuracy of the cascade using various types of feedback. They use alternating feedback strategy in which the feedback is not added to each subset of the first layer; instead, it is added in a crossed way (for more information, refer to [97]). Hu and Hao [99] follow the same approach but use .NET for the parallelization of the algorithm, which achieves a superlinear speedup compared to the standard cascade. However, it is unclear how many passes through the cascade are needed to achieve the desired accuracy.

Scalability. The parallel cascade-based SVMs scale to large numbers of training samples [17] and the required communication between processors is minor since processors need to exchange only the SVs, thus latency is low [50]. Low latency makes the cascade suitable for parallelization using a loosely coupled network of processors [110]. One drawback of the cascade is poor scalability in terms of the number of processors, that is, speedup saturates for a small number of processors, around 16. The reason is that one machine should combine all SVs in the last layer, thus overheads are increased [110]. Another drawback is the poor performance of the cascade for highly imbalanced samples [111]. These problems still exist in [97] and [99].

4.3.2 Parallel Cascade Using Distributed HPC Architectures. Distributed HPC architectures have been used for performing the cascade in parallel [100–103, 112].

Accuracy and the number of SVs. Classification accuracy of the cascade can be improved using feedback (see Section 4.3.1). One strategy that has been used is feedbacking in a crossed manner [97, 99, 100]. For instance, Yang et al. [100] add the resulting SVs of each subproblem of the first layer as feedback to the rest of the samples in the first layer. New subproblems are trained until the last layer of the cascade. This leads to improved accuracy and a faster training process compared with the standard cascade. Meyer et al. [98] go a step further and improve the accuracy and minimize communication overhead in [100]. In the proposed cascade, instead of generating one single SVM in the last layer of the cascade, they use a bagging SVM in which SVs regarding one of the subproblems are chosen as the final results using vote aggregation between the subproblems. This combination approach is promising for accelerating the cascade process; however, it does not achieve optimal results for all of the evaluated datasets and it misses an efficient parameter tuning. Unlike the previous works, Wen et al. [112] improve the accuracy and reduce the number of SVs in the cascade for multi-class classifications. They use a hierarchical approach similar to a 3-layer cascade in which the samples of each class are divided into k partitions and instead of the pairwise combining of resulting SVs, they combine SVs regarding k subproblems. The number of SVs obtained by the algorithm is less than that in the traditional cascade. The training time in [112] is reduced when the value of k increases. However, it is unclear how large the value of k can be, that is, how many partitions a dataset can be divided into without degrading the accuracy. This work is one of the few that study the impact of the number of partitions in the performance of the cascade.

Scalability. The previously mentioned cascade SVMs may deal with a large number of SVs that may not fit in the memory for large-scale problems. To overcome this problem, the cascade has been combined with approaches such as incremental learning [103] and a divide-and-conquer scheme [102]. Combination approaches can handle large problems without requiring a large amount of memory. For instance, Du et al. [103] combine a cascade SVM with incremental learning in which the new training data together with the resulting SVs in the last layer are fed back to the first layer. Their combination approach achieves considerable speedups compared with the standard cascade since the number of SVs in the proposed cascade is fewer than those in the standard cascade. You et al. [102] combine the cascade with a divide-and-conquer scheme to achieve a balance between accuracy and training time.In contrast to previous works, they minimize the communication between distributed computing nodes using initial clustering of data in which large samples are divided into subsets using k-means clustering. The algorithm in [102] removes the internode communication, which resolves the scalability issue of the cascade. Consequently, it scales well to large numbers of processors with a good load balance.

Peer-To-Peer environment (P2P). The cascade has a potential for further improvements in terms of communication, computations, and memory costs. For instance, Ang et al. [101] introduce the first P2P network for training a cascade SVM in which an upper bound on the communication overhead is identified. They further improve the communication between peers and reduce the overhead of sending SVs from a peer to another peer using a reduced-SVM strategy. The basic idea of the reduced-SVM strategy is that the number of SVs is reduced as much as possible. To do so, Ang et al. [101] generate a small random subset of the training samples as the representative of the entire set of samples [113]. In the proposed algorithm, the accuracy is not degraded as the number of peers increases and the algorithm scales well with the size of the network.

4.3.3 Parallel Cascade Using Distributed Big Data Architectures. Hadoop implementation of map-reduce has been used for the parallelization of the cascade SVMs [106–108, 114, 115] in which each map function trains a subproblem and each reduce function combines the resulting SVs from two subproblems. A map-reduce implementation of the cascade reduces the overhead as the number of mappers increases [106], but it may not balance loads of different computers. Guo et al. [114] propose a solution for load balancing issues using a genetic-based algorithm in a heterogeneous environment. Despite the advantages, the proposed solution is complicated to implement and may lose performance due to the fact that computation time in the genetic algorithm may get longer than time spent on training the subproblems [116, 117].

Accuracy. One may further reduce memory requirements and accelerate the training process in the cascade using feature selection/extraction strategies in which important features of samples are selected while the others are discarded [107, 108]. However, these strategies may lead to the deterioration of accuracy [108]. One can improve the accuracy of the cascade by combining it with other strategies (see Section 4.3.2). For instance, Pouladzadeh et al. [115] combine the cascade with incremental learning in which the database is periodically updated to improve the accuracy in each step. They further improve the accuracy using ontology-based enhancement, in which an end user confirms the classification results. This approach has been suitable for small-scale problems.

Multi-class classifications. Map-reduce has been used for parallelization of the cascade for multi-class classifications [106, 107]. The algorithms used for multi-class classifications may require iterative map/reduce tasks; however, Hadoop-based map-reduce may not support iterative algorithms and iterative map/reduce tasks. To overcome this problem, Sun et al. [107] use the Twister implementation of map-reduce, which supports iterative tasks.

4.3.4 Parallel Cascade Using GPUs. The computational power of GPUs has been used for accelerating the cascade of SVMs for large-scale problems [104, 105, 109]. Data reduction and data chunking are employed to overcome the limited memory on GPUs [105]. The parallel cascade algorithm proposed by Li et al. [105], along with data chunking and data reduction, shows superior performance in terms of training time. The basic idea of data reduction is that insignificant samples are eliminated in order to fit significant ones in the memory. The basic idea of data chunking is that the number of training samples is reduced by grouping them into chunks/groups based on their similarities. In the proposed algorithm in [105], data reduction with a single GPU outperforms, in terms of training time, data chunking with multi-GPUs when accuracy is not an issue. The proposed algorithm reduces the memory requirements for large-scale problems with slight deterioration of the training accuracy. Unlike previous works, Tarapore et al. [109] study the impact of different numbers of partitions on training time, speedups, and accuracy. The results of experiments in [109] show that the number of SVs may decrease as the number of partitions increases; thus, the communication cost between processors becomes more dominant than the training time. The speedup increases as the number of partitions increases while maintaining a high accuracy. This trend deteriorates as the number of partitions increases and the speedup is reduced. Again, the reason can be the increase of communication overhead.

In contrast to the above approaches that use public datasets from well-known repositories, Wimer et al. [104] use parallel cascade SVMs for a specific application: video-based pedestrian recognition and tracking at intersections. The detection rate of the corresponding algorithm is the same as that reported in other published articles for pedestrian detection, but the computation time of their approach is much less than that of others. The online table Cascade summarizes the findings, pros, and cons of parallel cascade SVMs mentioned in this section.

4.4 Parallel Interior Point Methods

One of the popular solutions used in parallel SVM implementations is the Interior Point Method (IPM) [11, 28, 61, 118, 119], in which an interior point traverses on the feasible region until reaching the optimal solution of the optimization problem. The IPM has been an attractive method for solving SVM problems for the following reasons. It fits the special formulation of the kernel function in the linear case and uses the simple structure of the optimization constraint addressed by an SVM problem [26]. The number of iterations in the IPM is constant or increases very slowly as the problem dimension grows [28]. The IPM has polynomial-time complexity and the major computations in the IPM involve solving one or two systems of linear equations with constant and predictable structures [28]. The primal-dual IPM is the most effective IPM algorithm, in which the inequality constraints are removed using a barrier function and the iterative Newton method is used to solve the linear system corresponding to a Hessian matrix [28]. The computational cost and memory requirements of the standard dual-primal IPM algorithms are $O(n^3)$ and $O(n^2)$, respectively. High computational costs and large memory requirements of the algorithm limit the use of the IPM for training large-scale problems. To overcome these problems, approximate matrix factorizations along with parallelization approaches have been used [11, 28, 33, 41, 61, 86, 119–121]. Incomplete Cholesky factorization (ICF), in which the kernel matrix is approximated by low-rank approximations, and Kronecker are two factorization schemes that have been used in the IPM [86]. ICF and Kronecker have the computational cost of $O(p^2n)$ and $O(2n^2)$, respectively, where p is the reduced matrix dimension after factorization and n is the number of samples [86].

The efficiency of other approximate matrix factorizations that can reduce memory requirements without significant trading off of accuracy is still an open question.

4.4.1 Parallel IPM Using SMP. Parallelization of the IPM using SMP can improve training time. However, one may need to employ strategies to further reduce computational costs for solving large-scale problems. These strategies often confront trade-offs between speedups and accuracy [86].

Accuracy and speedup. Coarse-grained and fine-grained approximate algorithms are two examples of strategies for reducing computational costs. Coarse-grained approximations achieve good speedups at the expense of poor training accuracy, whereas fine-grained approximations achieve good training accuracy at the expense of long training time. In order to find a balance between accuracy and speedups, Wu et al. [86] combine two approximations in a parallel incremental approximate matrix factorization in which the approximate IPM solution of a coarse-grained factorization initiates the IPM of a fine-grained factorization. The warm start for an IPM algorithm leads to fast convergence, thus high speedups. One can further reduce computational costs in IPMs by only one-time computing of approximate matrix factorizations before IPM iterations are started. After the factorization, only a reduced matrix needs to be stored at each iteration. Wu et al. [86] parallelize the ICF matrix factorization scheme using SMP, in which the memory requirement is reduced from $O(n^2)$ to $O(np)$, where p is the reduced matrix dimension after factorization, n is the number of training points, and $p \ll n$.

4.4.2 Parallel IPM Using Hybrid Parallelism. The computational capabilities of shared and distributed memory parallelism have been combined for the parallelization of IPM algorithms in which the distributed HPC architecture handles the communication between processors and SMP handles the computations inside processors. For instance, Woodsend and Gondzio [33] use BLAS routines for computing matrix-vector and matrix-matrix multiplications in shared memory and they handle communications between the processors using the distributed HPC architecture. The proposed algorithm achieves high classification accuracy compared with LibLinear for linear classifications (LibLinear is an open source library for linear SVMs [122]). A drawback of the proposed algorithm is that it requires loading all the samples into memory. Therefore, the algorithms cannot tackle large-scale problems.

4.4.3 Parallel IPM Using Distributed HPC Architectures. IPM-based SVMs have been performed in parallel using distributed HPC architectures [11, 41, 119–121] in which training samples are distributed across multiple processors. Approximate matrix factorizations have been used to speed up the process [119, 121].

Memory and speedup. One of the computationally expensive tasks in IPM-based algorithms is the calculation of the inverse of a large matrix. The SMW formula has been used to compute the inverse of a large matrix in an inexpensive way that reduces memory requirements and the runtime of the IPM [41, 119–121]. For instance, Gerts et al. [119] employ SMW for an iterative technique called the preconditioned linear conjugate gradient method. This method does not explicitly require the computation of the kernel matrix, but it requires the computation of matrix-vector products. The results of experiments in [119] show that the proposed parallel IPM outperforms the standard IPM algorithms. However, it supports linear kernels only for problems with a limited number of features. In contrast, Woodsend and Gondzio [11] mention that SMW can cause numerical instabilities; to overcome this problem, they used the Cholesky decomposition. The proposed decomposition was applied to all features at once, which facilitated efficient use of the memory cache of the processors.

To further reduce memory requirements for large-scale problems, linear algebra operations have been used to exploit the block structures of the matrix addressed by SVMs [11, 33]. In this regard, the order of distributing the matrix across machines has an impact on memory usage, that is, one should take into consideration which row or column distribution of the matrix fits the model chosen for parallelization [41, 121, 123].

Column- and row-based data partitioning. A column-based approach is more suitable for SMP than distributed memory. The reason is that in the distributed memory parallelism, each machine needs to reach all of the training data to perform its corresponding computations and, therefore, it should store a local copy of all data that is inefficient and memory intensive. Moreover, in the column-based approach, only calculations regarding the inner product can be parallelized [121]. Chang et al. in [121] and [41] propose a parallel IPM-based algorithm using parallel row-based ICF in which only essential data are loaded into multiple machines and each machine performs the corresponding computations in parallel. Thereby, the memory requirements are reduced from $O(n^2)$ to $O(np/m)$ and the computational time is reduced from $O(n^3)$ to $O(np^2/m)$, where p is the reduced matrix dimension after factorization, n is the number of training samples, m is the number of machines and p is much smaller than n [41, 120, 121]. The results show that the computation time of parallel ICF is reduced as the problems size increases since the communication overheads are low. The computation speedup is sublinear owing to the unparallelizable step in ICF that has the computation time of $O(p^2)$ [121] and, according to Amdahl's law, even a small sequential part can deteriorate the speedup. One should keep in mind that communication overheads are minor for problems that use only a few machines for the parallelization [41, 120, 121].

4.4.4 Parallel IPM Using GPUs. Despite the fact that the computational powers of GPUs make this hardware attractive for solving computationally expensive tasks, GPU-based parallelism is used only in a few IPM-based SVMs [28, 61, 124]. In those algorithms, low-rank matrix approximations and SMW have been the target of GPU parallelism. For instance, Li et al. [28] use GPUs to perform the computationally expensive tasks such as ICF matrix factorization, matrix-matrix and matrix-vector multiplications in parallel. They perform the Cholesky factorization on a CPU using the master process in a serial code and optimize the data transfer between the host and the device by allocating a contiguous memory space for transferring the matrix in the host memory. The matrix is copied from the host to the global memory of the device by only one function call; thus, communication and synchronization overheads are reduced as the size of samples increases. The algorithm outperforms the CPU-based parallel IPM; however, it does not take maximum advantage of new GPU features, for example, the pinned memory that provides transfer speed and enlarges memory space of the GPU and unified virtual address that provides one address space for CPU and GPU memory. In another work, Li et al. [124] improve the data transfer and memory access speed in [28] by exploiting the heterogeneous hierarchical memory on a CPU-GPU cluster using the dual buffer 3-stage pipeline mechanism and the pinned memory. In the dual 3-stage pipeline stream scheduling, data send, calculation, and receive operations are performed concurrently. We briefly mention the comparisons of the parallel IPM-based algorithms in the online table IPM.

4.5 Parallel Kernel Computations

Kernel matrix calculations contain calculating dot products regarding large vectors in linear classifications and a Hessian matrix in non-linear classifications [33, 34]. These computations dominate the total computation time in training and testing/predicting processes [110]. One straightforward approach to accelerating these processes for large-scale problems is to parallelize the kernel computations and kernel matrix factorizations [42, 46, 54, 110, 125, 126].

4.5.1 Parallel Kernel Matrix Using SMP. Kernel matrix calculations contain computationally expensive tasks.

Memory. One can reduce the memory requirements for kernel matrix calculations using matrix approximations via block diagonal matrices [46, 54] and reduced-set SVMs [42]. In the matrix approximation strategy, the kernel matrix is approximated with block diagonal matrices that can be computed in parallel. Dong et al. [54] propose parallel and sequential optimization steps in a parallel algorithm in which non-SVs are eliminated in the parallel optimization step. Thus with only remaining SVs, the training time for the sequential optimization step is reduced. Regarding scalability, the analysis of the algorithm proposed by Dong et al. shows that the runtime complexity linearly scales with the size of the datasets and the number of classes. Unlike the previous strategy, Diaz-Morales et al. [42] use the reduced-SVM strategy to eliminate non-significant samples, that is, non-SVs. In this method, the kernel matrix is approximated and the training samples are reduced based on the sparse greedy rule in which only candidates with the highest possible error descent are chosen at each iteration. Diaz-Morales et al. propose a parallel reduced-set SVM along with a matrix decomposition and block matrix scheme for computing the kernel matrix inversion in multiprocessors [42]. The results show that almost twice the speedup is achieved by doubling the number of cores; note, however, that the efficiency deteriorates around 10%. This is because launching parallel processes on the increasing number of cores may take longer than the actual computation time. One should take into the consideration that a parallel reduced-set SVM using SMP may not perform well for small-scale problems or problems with fewer SVs since the time for launching the tasks for/from different cores may take longer than computing parallel tasks [42].

Accuracy. One needs to consider that reducing the size of datasets in the previous works may cause class imbalance problems, that is, the number of samples in one class may get much larger (the majority class) or smaller (the minority class) than those in the other class. To overcome the class imbalance, Severyn and Moschitti [125] use a resampling strategy in which examples corresponding to the importance weights are chosen iteratively, thus no important information is lost as it may have been in [42, 54]. They define weights using the cost-proportionate rejection resampling strategy in which examples from both the majority and minority classes are chosen in the desired proportion. The SVM algorithm proposed by Severyn and Moschitti [125] is a modular algorithm that is easy to be performed in parallel and uses tree-based kernels. They use a Cutting Plane Algorithm (CPA) [125] in which the constraints of the original optimization problem are replaced with linear combinations of the constraints from the original optimization problem. Although the algorithm is parallel friendly, the kernel evaluations at each iteration are computationally expensive for non-linear SVMs regarding large problems. To reduce the kernel evaluations in a tree-based kernel, they use an approximate CPA along with DAGs (mentioned in multi-class classification in Section 2) in which a fraction of training examples is chosen and many examples share the common substructures in the tree. This reduces the number of kernel evaluations at each iteration from $O(tn^2)$ to $O(tr^2/p)$, where p is the number of processors, r is the number of chosen examples, and n is the number of total training examples.

4.5.2 Parallel Kernel Matrix Using FPGAs. Kernel computations can be implemented in hardware using FPGAs [56, 127–130]. FPGAs are digital integrated circuits that contain programmable blocks of logic and programmable interconnects between the blocks [131]. The disadvantage is that FPGA may suffer from limited RAM blocks [130].

Low power dissipation. One of the reasons for a growing interest in employing dedicated architectures for computing-intensive operations is that those architectures can be designed with the aim of reducing power dissipation. For instance, Graf et al. [129] build more compact circuits for fixed-point arithmetic. With this choice of arithmetic, faster computations trade off the accuracy, although a slight deterioration of accuracy might be acceptable.

Algorithm adjustments. A dedicated coprocessor consisting of a grid of cores can compute several columns of kernel matrix in parallel [56]. In this regard, one may need to perform adjustments and modifications to fit the corresponding algorithm for the hardware characteristics. For instance, for the parallelization of a decomposition technique, one may use strategies to reduce the number of iterations at the expense of more cost per iteration and the cost per iteration can be reduced using FPGA at each iteration. In order to take advantage of FPGAs, one should consider the availability of cached kernel values and a fast convergence criterion [56].

FPGA versus GPU. Papadonikolakis [130] compares the dedicated high-performance architecture using FPGA to the GPU-based parallelism for training SVMs. The results show that FPGA outperforms GPUs only for datasets that fit in the FPGA RAM blocks and not otherwise. The reason is that the GPU needs to transfer data from the device global memory to the CPU's shared memory, which may cause high overheads [130].

The parallel implementations of kernel computations using distributed memory parallelism (see Section 4.4.3), GPU-based parallelism (see Sections 4.1.4 and 4.4.4) and map-reduce (see Sections 4.1.3 and 4.2.4) have been excluded from this section since their main contributions fit better in the other sections. We show the brief comparisons, pros and cons, of the parallel kernel algorithms in online table ParallelKernel.

4.6 Parallel Distributed Algorithms

A large SVM problem can be performed in parallel by dividing the problem into multiple smaller problems using the following approaches. One approach is training of the combined results of individually trained subproblems [26, 38, 68, 116, 132–136]. Other approaches are combining the results of ensemble SVMs [117, 137, 138] and initial clustering of data before the training [40, 137, 139].

4.6.1 Parallel Distributed Algorithms Using SMP.

One can use initial clustering of data to fit large-scale problems into limited shared memory before the training phase starts. To do so, large training samples can be clustered into smaller chunks of similar data, all of which are trained in parallel [139].

Memory. The advantage of initial clustering is that the memory requirements are reduced by reducing the number of data into clusters of data and then each data cluster becomes a representative of a group of samples that have predefined similarities. In this regard, k-means clustering has been used to train many local SVMs instead of one global SVM [137, 139]. One may expect that the training time of n clusters using n machines is dropped by a factor of n times, but this may not be achieved owing to the overhead. For instance, Shrivastava et al. [139] use parallel k-means clustering in order to use the minimum number of training examples as abstracts of a large dataset. The proposed algorithm achieves a speedup of 3 times using SMP compared to the sequential SVMs, but detailed information is needed to analyze the experiments more clearly, for example, it is unclear that the algorithm can support non-linear or multi-class classifications.

Accuracy. The results of experiments in [139] show that there is a trade-off between the number of clusters and training accuracy. One may control accuracy by selecting an appropriate starting point for k-means clustering. Do [137] follows a similar approach for parallelizing Libsvm for non-linear and multi-class classifications. Another approach for solving SVMs regarding large-scale problems is to use ensemble SVMs. For instance, Claesen et al. [117] employ ensemble learning to train several SVMs. Unlike in [137, 139], the ensemble SVM [117] gives higher accuracy competitive with Libsvm using ensembles of SVMs with a bagging strategy in which local SVMs are trained on bootstrap subsamples and the results of all local SVMs are aggregated based on majority voting. This approach reduces the training complexity.

4.6.2 Parallel Distributed Algorithms Using Distributed HPC Architectures. Two key points play a significant role in the efficiency of parallel distributed SVMs using distributed HPC architectures. One point is to find an efficient strategy to minimize overhead with regard to combining the results of subproblems [140]. The second point is to find an efficient strategy to divide the data into subsets or to distribute data across processors.

Scalability. One strategy to reduce overhead is that the number of computations that need to be sent to the processors is reduced at each optimization iteration. For instance, Bickson et al. [133] reduce communication overhead for problems with high dimensions by reducing the number of messages sent between processors from $O(n^2)$ to $O(n)$, where n is the number of samples. To do this, they shift from an algebraic to probabilistic domain using Gaussian distribution. At each iteration, a message that contains only two real numbers is sent to neighboring nodes through mutual edges, if available. The algorithm is the largest parallel implementation of the belief propagation algorithm that scales up to 1,024 computing nodes for 150,000 data points with comparable accuracy to a well-known SVM software called SVM^{light}. One drawback is that they use an implementation of MPI—MPICH2—that lacks the support of asynchronous communications for heterogeneous systems. In addition, the scalability in terms of the number of samples is questionable and it is unclear whether the algorithm can solve large-scale problems owing to the fact that the full kernel matrix needs to be computed.

Divide and conquer. Note that the method of dividing a large dataset into subsets and the method of aggregating results have impacts on accuracy. One can use a divide-and-conquer strategy to break down a large problem into smaller problems. The basic idea is that multiple local SVMs are trained and the results are combined based on an aggregation scheme [68, 116, 117, 132–135, 137]. A parallel mixture of SVMs [132], parallel modular SVMs [116] and parallel ensemble SVMs [117] are examples of divide and conquer. Collobert et al. [132] use a mixture of SVMs in which each SVM is trained on part of the samples and the results of subproblems are aggregated using a neural network. Although it leads to high prediction accuracy, the corresponding computation time is longer than the time spent on training the subproblems [116, 117]. Huang et al. [116] improve the work in [132] using a region-computing modular network to train several SVMs, each of which is trained on only a small subregion of the sample space. This approach is easy to perform in parallel because of its modular nature and does not have the overhead and complications of [132] since the neural network is replaced with neural quantizer modules. These modules allow the local SVMs to be fired if their inputs belong to their specialized subregion, otherwise; the neural quantizer modules inhibit the output of the local SVMs [116]. In contrast, Claesen et al. [117] use a much simpler aggregation model than in [132] and [116] with competitive performance (see Section 4.6.1).

Accuracy. Another approach to partitioning data is to randomly partition and distribute the data in order to solve class imbalance that occurs when the local data contain mostly the same label [141]. Qiu and Lane [140] distribute training samples across processors using a dimension-wise data partition in a distributed memory parallelism. This strategy is simpler than the previous ones and reduces the communication overhead between the nodes since it follows the memory access pattern in distributed memory parallelism, thus reduces the data transfer. They further reduce communication overhead using an approximation of the kernel matrix.

ADMM. ADMM is a popular distributed scheme for solving SVM problems in networks of interconnected nodes [142] in which each node has a private cost function and private constraints [89, 111, 141–144]. The goal is to minimize the sum of all cost functions with respect to the intersection of all constraints [142]. Despite the fact that ADMM is a promising scheme for reducing memory requirements, it may suffer from slow convergence and high time complexity [144, 145]. In ADMM, one can reduce the communication between local processors using a coloring scheme of networks [142] and hash table [141]. A coloring scheme of networks is to assign colors to the nodes of networks so that no adjacent nodes have the same color. Coloring controls the order of communications between nodes; the nodes with the same color can asynchronously perform their computations in parallel. For instance, Mota et al. [142] use a generalization of distributed ADMM with a coloring scheme that leads to less communication between nodes than that in the state-of-the-art algorithms. The theoretical explanation of why this algorithm is more efficient than similar algorithms is still unclear. In contrast to the work in [142], Zhang et al. [141] use the combination of efficient strategies in which the algorithm integrates random sampling, a warm start of the subproblems, inexact minimization, normalizing test data, over-relaxation, and cross-validation of multi-class classifications. One requirement of the algorithm is that data should fit in distributed memory; otherwise, one should use batching strategies to do so. This algorithm outperforms similar approaches in [146] and [147] in terms of training time and convergence rate, respectively. In a similar fashion, Deist et al. [144] introduce a systematic multi-centric data-sharing framework based on ADMM applied to personalized medicine, in which patient data privacy is preserved. To do this, SVM models are learned on data from separated and distributed databases using a customized information technology infrastructure in which different sites communicate via file-based and asynchronous messaging [144]. Although the ADMM approaches are promising for solving large-scale distributed SVMs, they may suffer from slow convergence, weak global consensus, supporting only linear classifications, and high time cost. To overcome slow convergence, Wang et al. [148] divide the slave nodes into groups. The local results of the slaves are gathered in the corresponding group to update the global result. The group-based ADMM converges faster and saves up to 30% of the total time compared to the non-grouped ADMM [148].

Unlike the previous algorithms that only focus on linear classifications, Chen and Fan [111] combine ADMM with multiple kernel learning [111] for accelerating the parallel implementation of SVM using multiprocessors. In multiple kernel learning, the global SVM problem with multiple kernels is divided into multiple local problems, each of which is optimized with a single kernel in a local processor. The approach in [111] is the first hybrid ADMM and multiple kernel learning implementation that coordinates the communications between processors to obtain a global solution. The algorithm outperforms the standard cascade [17] and SVM ensembles [106].

4.6.3 Parallel Distributed Algorithms Using Distributed Big Data Architectures. These architectures reduce memory requirements by dividing a large training dataset into many smaller subsets. Dividing the dataset into subsets can have an impact on training/testing accuracy [43, 138]. For instance, if the distribution of samples in each subset is very different, accuracy is degraded [138].

Accuracy. Parallel balanced bootstrapping has been used to improve classification accuracy in which training samples are resampled into subsets so that each sample appears the same number of times in all bootstrap sets. For instance, Alham et al. [138] use this strategy to resample data for ensembles of SVMs in which multiple weak learners are combined to create a strong learner. The proposed approach supports only binary classifications and does not balance loads for heterogeneous computing environments.

Memory and speedup. In contrast to [138], which supports non-linear classifications, Pechyony et al. [149] use block minimization to reduce memory requirements and accelerate map-reduce-based parallel distributed algorithms for linear classifications in which a kernel matrix is divided into blocks and optimization regarding each block is performed in parallel by map functions. At each iteration, all of the distributed blocks are computed in parallel by slave nodes and the results coming from slaves are combined in a master node (centralized computing) using averaging and line search strategies. The results of the experiments concerning the scalability focus line regarding a line search strategy show that the algorithm can solve large problems containing training and testing samples around 80 million each in only 11 minutes. The communication complexity of the proposed algorithm is independent of the number of training data points. Besides the advantages, the algorithm conducts only linear classifications and the global convergence is not proven.

4.6.4 Parallel Distributed Algorithms Using GPUs. One can take advantage of GPU parallelism if the algorithm in use is designed for GPU architectures. Two drawbacks of the previous GPU-based SVMs are as follows. (1) They perform the matrix multiplications addressed by the standard SVM algorithms using GPU-based libraries without paying extra attention to modifying the algorithms to better fit the GPU architectures. (2) They often do not support sparse datasets [40]. The problem regarding sparse datasets is that they may not follow a certain pattern of memory access on GPUs, which is a hurdle to overcome to get the maximum performance from GPUs. One needs to employ techniques such as coalescing for efficiently accessing data from memory [35, 40]. In this regard, Cotter et al. [40] propose a novel sparsity clustering that takes advantages of the sparsity of datasets and GPU parallelism. Sparsity clustering groups the training examples of similar sparsity patterns, thus overcomes the limited memory in GPUs. The algorithm in [40] is specially designed for GPUs and supports both dense and sparse datasets. At the time of publication, it was the only GPU-based algorithm that supported both binary and multi-class classifications. The algorithm improves the runtime from $O(n^2)$ to $O(nk)$ for a small k, where n is the number of examples and k is the number of active clusters. Active clusters are candidate clusters that are considered for each training example. The results show that further decreasing k can improve runtime with only minor deterioration of accuracy. In a similar fashion, Codreanu et al. [35] show that changing the uncoalesced memory access to coalesced memory access results in 3 times speedup. Furthermore, they show that the algorithms similar to [40] can achieve around 10-fold speedups by changing the input data structure from arrays of structures to structures of arrays. This is because structures of arrays fit the memory access pattern on GPU architectures better, which leads to higher memory throughput [35]. The algorithm is the fastest GPU-based SVM for problems with high-dimensional input spaces. Note that scalability has not been discussed in any GPU-based implementations of SVMs. The comparisons of parallel distributed SVMs are shown in online table DistributedSVMs.

4.7 Other Parallel Optimizations

Gradient-based [68, 135], gradient projection-based [26, 38, 52], Gaussian belief propagation [133], Iterative Re-Weighted Least Squares (IRWLS) [150, 151] and semiparametric [152] algorithms are other types of algorithms used for solving SVMs that have been parallelized to accelerate the training and testing/predicting processes.

4.7.1 Parallel Optimizations Using SMP. SMP has been used for performing IRWLS methods for training SVMs in parallel [150, 151]. IRWLS reformulates the primal optimization addressed by SVMs in the form of weighted least squares that can be independent of slack variables that provide tolerance for misclassifications. IRWLS solves one linear system in every iteration in which the inverse of a large matrix containing kernel evaluations is calculated. This is computationally expensive for large-scale problems. Therefore, Morales and Vázquez [150] use Cholesky factorization to reduce the kernel evaluations for solving full SVMs in IRWLS. Solving full SVMs for large-scale problems is computationally expensive. To solve this problem, Morales and Vázquez use semiparametric IRWLS in which the complexity of the resulting model is under control. This results in the speedup of the classifications of new examples. In another work, Morales and Vázquez [151] improve their work in [150] using the budgeted IRWLS to control the number of SVs in which a set of basis elements is chosen and the approximation of the weight vector is calculated using sparse greedy matrix approximation and random sampling. Comparing the results of the two articles shows that the first algorithm PSIRWLS outperforms the second algorithm LIBIRWLS for the same datasets. One possible explanation is the efficiency of Cholesky factorization in PSIRWLS for reducing the kernel evaluations. PSIRWLS outperforms the similar parallel semiparametric SVM called PS-SVM [152] in which quadtrees, a parallel block matrix inversion, instead of Cholesky factorization is used. We show the detailed comparisons in online table IRWLS.

4.7.2 Parallel Optimizations Using Distributed HPC Architectures. These architectures have been the most common parallel architectures used for performing optimization algorithms in parallel.

Gradient-based algorithms. These algorithms are the standard SVM algorithms that have been parallelized using distributed HPC architectures [68, 135]. Gradient-based algorithms often compute the gradient using the whole set of training samples, which is a hurdle to overcome for solving large-scale problems. To overcome this problem, Zhu et al. [135] propose P-packSVM, a Stochastic Gradient Descent (SGD) algorithm in which the gradient is stochastically approximated using only a single training example. The advantage of P-packSVM is that it handles an arbitrary kernel and develops the parallelism using a distributed hash table. The table stores the key-value pairs and then the computationally heavy tasks are performed in parallel via distributed storage of inputs. Gradient-based SVMs often require a larger number of iterations than those for IPM-based SVMs before convergence occurs, which causes higher communication cost for gradient methods. To reduce the communication cost, Zhu et al. [135] use a packing strategy in which communication overheadis minimized by packing a number of iterations into a single iteration. This reduces the number of times that processors need to communicate by the factor of $O(r)$, where r is the number of iterations packed into a single iteration. Considering memory, packing along with hash table makes the algorithm highly parallelizable with the memory requirements of $O(m/p)$ for each processor, while the memory requirements for PSVM [41] that is IPM-based is $O(m^{3/2}/p)$, where m is the number of input space dimension and p is the number of machines. Considering speedup, Zhu et al. [135] have shown that stochastic gradient-based SVM algorithms outperform the IPM- and SMO-based SVMs in which the training time for a dataset with 800k samples reduces to only 13 minutes with 95% accuracy, while the parallel IPM-based PSVM proposed by Chang [41] trains the same dataset in 5 hours with 92% accuracy. This shows that packing may compensate a large number of iterations required by gradient-based methods to converge. Other advantages of the algorithm proposed by Zhu et al. is that it scales to a large number of examples, that is, 8 million data points for multi-class classifications, while PSVM focuses mainly on binary classifications with fewer samples. As mentioned in Section 2, the multi-class classification procedure is computationally expensive and, to reach considerable speedups, one may need to avoid training on a full dataset. To do so, a parallel bagging SVM with a sampling strategy has been used. For instance, Nghi et al. [153] propose an undersampling strategy in which the majority class is resampled to get an equal size the same as the other class(es). This approach achieves more than 1,000 times speedup compared to Libsvm, but it requires loading the whole set of training data into memory.

In contrast, Ferreira et al. [68] ease the parallelization of linear and non-linear SVMs with gradient-based neural networks that compute only the lower triangular matrix addressed by QP in which each processor computes only a part of the lower triangular matrix needed for its local computations. Although the parallel algorithm proposed by Ferreira et al. outperforms Libsvm and SVMlight for datasets with around 50k samples in non-linear and binary classifications, the work by Nghi et al. [153] performs better for large-scale problems. The article lacks the details of implementations for the fairer comparisons with previous algorithms.

Gradient projection-based algorithms. These algorithms are the standard SVM algorithms that have been performed in parallel using distributed HPC architecture [26, 38, 52] in which subproblems are optimized based on an iterative projection of the gradient. Caching, shrinking, a sparse format for storing data, and block-wise distribution of kernel matrix are some of the efficient approaches used in these algorithms [26, 38]. Although the existing gradient projection-based algorithms are efficient, they suffer from some issues. For instance, the algorithm proposed by Zanghirati and Zanni [26] is effective only for a specific kernel function, that is, the Gaussian kernels. In the algorithm proposed by Zanni et al. [38], speedup deteriorates for the increasing number of processors. This happens due to the increased communication overhead. Gradient projection-based SVMs algorithms are iterative methods based on chunking techniques in which strong data points—that is, SVs—are retained from chunk to chunk in an iterative manner [136]. At each iteration, the chunks can be computed by multiple processors. If the number of SVs is large, communication overhead will be increased since SVs from different processors need to be combined to obtain the final result [136, 154]. To overcome the overhead, Winters-Hilt and Armond [154] reduce the number of SVs for each chunk by forcing some of the corresponding low-value multipliers to zero, that is, cutting down the SV vector and forcing some SVs to be a non-SV. The proposed algorithm handles both the regression and classification problems. However, it considers only binary classifications.

4.7.3 Parallel Optimizations Using Distributed Big Data Architectures. The process of multi-class classification is computationally expensive (see Section 2). In this regard, one can extract important features to reduce the data size. Extracting features for large problems, such as large-scale image classifications with 1,000-class SVM classifiers, is computationally expensive. To reduce the computational costs, Lin et al. [155] propose a parallel feature extraction using map-reduce in which the Hadoop distributed file system (HDFS) [156] distributes the images through all machines, all of which can perform the extraction tasks independently on the local images. The feature extraction process is easily parallelizable since the extraction task on each machine is independent. After the feature extraction, they use Averaging Stochastic Gradient Descent (ASGD) [155] to train SVMs. Concerning the speedup focus line, adding the averaging scheme to SGD leads to fast computation for large redundant samples. The reason is that the averaging is much simpler to compute than the second-order SGD, which requires computing the inverse of a computationally expensive matrix. Lin et al. [155] reduce the traffic for loading a large training dataset by minimizing the file I/O. To do so, the memory is shared on each multi-core machine. The memory sharing enables the algorithm for loading a non-sparse training dataset containing 1.37TB of data and reduces memory requirements since multiple programs, which train the same data chunk, need to load data only once for a multi-class classification. Although it has advantages, the algorithm in [155] may suffer from communication overhead and the update of the corresponding parameters one by one. To reduce the communications between computing units, Chu et al. [157] modify the optimization problem addressed by SVMs in order to obtain computations that can be performed independently and in parallel. Chu et al. rewrite the summation formula addressed by the Batch Gradient Descent (BGD) [157] in a certain summation form in which each piece of the summation can be easily and independently performed in parallel and the results are averaged to obtain final results. The advantage of the BGD in the proposed algorithm is that the data are combined in batches, thus the communications between the processors are minimized since the result corresponding to each batch needs to be communicated instead of communicating the result of a single data. Concerning the scalability focus line, the proposed algorithm by Chu et al. [157] does not scale to a large number of cores, that is, it gets only around 13% speedup using 16 cores. Another disadvantage of a BGD is that it uses the whole examples in each iteration, which is a hurdle to overcome for solving large-scale problems. Thereby, Tao et al. [158] use a Mini-Batch Gradient Descent (MBGD) [158] in which a part of examples is used in each iteration. Unlike the earlier algorithms, while MBGD is suitable for solving large-scale problems, it may suffer from the curse

Table 2. The Comparison of Well-Studied SVM Algorithms

Algorithms	Differences	Problem Type	Pros	Cons
SGD-based SVMs	Focus on primal optimization	Non-linear and multi-class classifications	It is the fastest for linear SVMs on a single machine, easier to be parallelized.	A large number of iterations until convergence, accuracy fluctuates.
IPM-based SVMs	Focus on dual optimization	Non-linear and binary classifications	Requires few iterations until converges, has the lowest communication cost.	The Cholesky factorization lacks theoretical error bound and may be inaccurate for some datasets, slow convergence, difficult to be parallelized.
SMO-based SVMs	Focus on dual optimization	Non-linear	Good accuracy, fastest for non-linear SVMs on a single machine.	Slow convergence, needs modification to be parallelized.

of dimensionality that causes unbounded linear growth for model size and the update time with data size [158]. To overcome this problem, Tao et al. use a budget maintenance strategy for MBGD to keep the number of SVs under control by removing some SVs, which results in constant space and time complexity in each update. Another advantage of the algorithm proposed by Tao et al. is that they use the Spark data processing engine for solving the iterative algorithm in which the working set is saved and cached in memory. This overcomes the problem of reading and writing data repeatedly from the HDFS. The online table Gradient shows a brief comparison of gradient-based SVMs. We highlight the findings, pros and cons, of the parallel optimization algorithms in online table OtherParallelAlg.

5 DISCUSSION

One of the challenges we have faced for comparing parallel SVM implementations is the difficulty of replicating the results. Except for a few cases, the source codes are not publicly available or the settings of the experiments are not clarified. Some of the pressing issues are the size or dimension of samples, user-defined values of parameters, the number of processors, the type of classifications, and the efficiency of algorithms for non-linear classifications, or the type of parallelism. In the following sections, we summarize and further highlight some of the other challenges in parallel SVM implementations.

5.1 Summary

We survey parallel SVM implementations, including parallel algorithms and architectures that have been used for solving large-scale problems. Our discussion generated from this survey is with respect to the goals of parallelism, including the four mentioned focus lines. Figure 1 shows our categorization of parallel SVM implementations that we have reviewed. We have also briefly surveyed efficient heuristics, including grid search, cross-validation, caching, shrinking, sparse formats, feature extraction/selection, working set size/selection, data movement, data reordering, and memory access pattern in this article. Table 2 shows a brief comparison of the well-studied SVM algorithms. Table 3 summarizes the characteristics of parallel SVMs with the top 10 largest training examples and Table 4 summarizes the characteristics of parallel SVMs with the top 5 largest speedups reported in the experiments.

5.2 The Dominant Approach

Among the parallel SVMs reviewed in Section 4, parallel decomposition is the dominant approach for parallel implementations of SVM. The reason is that the decomposition methods use only a

Table 3. Characteristics of Parallel SVMs with Top 10 Largest Training Examples Evaluated in the Experiments

Settings	Approach	Tool	#Samples	Reference
Block partitioning, LS-SVM, linear and non-linear	Parallel incremental	DMP[a]	1 billion	Do and Poulet [88]
Block minimization	Map-Reduce	Hadoop	80 million	Pechyony et al. [149]
Random sampling, inexact minimization, normalization, cross-validation, multi-class	ADMM	DMP	~20 million and 30 million features	Zhang et al. [141]
Caching, dual coordinate descent	Parallel incremental	SMP[b]	20 million	Matsushima et al. [87]
Newton SVM	Parallel Incremental	GPU	10 million	Do et al. [91]
Multi-class, gradient-based, packing, arbitrary kernel, hash table	Parallel Optimization	DMP	8 million	Zhu et al. [135]
LS-SVM	Parallel Incremental	GPU	5 million	Do et al. [10]
Data reordering, SMO	Parallel Kernel Computation	DMP	4 million	Durdanovic et al. [110]
Various shrinking	Parallel Decomposition	DMP	2.3 million	Vishnu et al. [67]
Gradient projection-based	Parallel Optimization	DMP	2 million	Zanni et al. [38]

[a]DMP denotes Distributed Memory Parallelism
[b]SMP denotes Shared Memory Parallelism

Table 4. Top 5 Speedups. Characteristics of Parallel Algorithms with Top 5 Largest Speedups in the Experiments

Settings	Parallel Approach	Parallel Tool	Speedups	Reference
Multi-class, balanced class, bagging SVMs	Parallel Incremental	Hybrid SMP[a]-DMP[b]	1193× LibLinear, and 732× original algorithm, 160 cores	Doan et al. [29]
Block partitioning, LS-SVM, linear and non-linear classifications	Parallel incremental	GPU	1000×	Do et al. [10]
LS-SVM, column incremental	Parallel incremental	SMP	190× over LibSVM	Do and Fekete [159]
Data reordering	Parallel Kernel Computation	DMP	100× with 48 machines	Durdanovic et al. [110]
Random sampling, inexact minimization, normalization, cross-validation, multi-class	ADMM	DMP	60× with 8 machines over LibLinear	Zhang et al. [141]

[a]SMP denotes Shared Memory Parallelism
[b]DMP denotes Distributed Memory Parallelism

fraction of input data in the working set. For instance, the standard SMO as the most common decomposition technique has the working set size of 2, that is, SMO calculates only two rows/columns of the kernel matrix. One drawback of decomposition methods is that they are inherently sequential owing to the dependent computation steps; thus, they are not the best option for parallelization. In order to decrease the number of dependent steps, parallel decomposition-based SVMs use large working sets at the expense of increased cost per step. The cost per iteration can then be reduced by parallelizing each step. The size of working sets has an impact on training time and accuracy (section 4.1). In this regard, parallel decompositions have had different strategies for choosing an appropriate size of the working set. However, there is a lack of agreement on the optimal size and partitioning data on available memory. An empirical study of the efficiency of different working set sizes is needed to better understand possible improvements of decomposition methods in terms of the speed of convergence and accuracy.

5.3 Four Focus Lines

We have identified four main focus lines that have been investigated in the parallel approaches reviewed in this article. In this section, we briefly discuss aspects of these focus lines.

5.3.1 Memory. One focus line of parallel approaches is reducing the memory requirements for large-scale problems so that data can fit into the available memory. This is particularly important for shared memory and GPU-based memory parallelism owing to limited memory or restricted memory access pattern on GPUs. While there are effective approaches to reducing memory requirements (see Sections 2.1 and 4), there is still no clear suggestion for every SVM implementation. In addition, the resulting speedup magnitude and parallelizability of these approaches need further investigation.

Parallel incremental SVMs. Incremental learning among parallel SVM implementations is one of the efficient and promising approaches to handling limited memory restriction on standard workstations. Factors including the size of increments and dimension of the input spaces impact the efficiency of the implementations (see Section 4.2.5). The efficiency and scalability of parallel incremental SVMs are still open questions for large-scale problems with high-dimensional input spaces. For such problems, although one can use dimension reduction techniques, these techniques are difficult to perform [96].

Parallel IPMs. The large size of input data requires employing approximate matrix factorizations for IPM-based algorithms in order to reduce memory requirements (see Section 4.4). The efficiency of approximate matrix factorization schemes is studied on sequential IPM-based algorithms in [118]. However, to our knowledge, there is no benchmarking and empirical study of the schemes for parallel settings. A computational comparison of these schemes would be beneficial to identify possibly simple, computationally inexpensive and easily parallelizable schemes for improving the performance of IPM-based algorithms. One suggestion could be Jacobi factorization since it is easy to compute and parallelize.

The parallel IPM-based SVMs in Section 4.4 agree that there is a trade-off between training/testing time and training/testing accuracy owing to approximations. One challenge is the number of samples that can, through the approximation, be reduced without the deterioration of accuracy. Although some suggestions for an optimal size of reduced ICF have been given, they are optimal for only a few scenarios [123]. To our knowledge, there is no clear suggestion for an optimal size of reduced data without trading off taccuracy. It seems that an optimal size depends on the problem size and dimension of input spaces, opening the opportunity for future research.

5.3.2 Speedup. Another focus line of parallel approaches is accelerating the training and testing processes using available parallel architectures. This can be done through two approaches. One is solving the problem through training a single SVM in which only the computationally expensive tasks are performed in parallel. The second approach is to divide the large SVM problem into several smaller SVM subproblems, all of which are performed in parallel (see Section 1). On one hand, solving one single SVM problem leads to higher accuracy since the main optimization problem stays unchanged. However, owing to the sequential parts of the algorithm, the speedup deteriorates. Another issue is that a single problem-based algorithm may suffer from memory limitations since the algorithm may require computing the sequential parts using the whole training samples (see Sections 4.1 and 4.4). On the other hand, solving multiple SVM subproblems can solve the memory issues, but since the problem is divided into multiple simpler problems, the original problem is changed, which may cause the deterioration of accuracy (see Section 4.3). To our knowledge, except for the cascade approach, efficiency, and possible trade-offs of training, a single SVM versus training multiple SVMs has not been explicitly investigated for parallel settings. However, distributed approaches of parallel SVM implementations show a tendency toward training multiple SVMs in which several independent subproblems are solved in parallel and the computing powers are added on demand.

5.3.3 Scalability. One can consider scalability both in terms of the number of machines employed for parallelization and/or the size of samples.

Scalability in terms of the number of machines. Increasing the number of computing machines does not always lead to better performance owing to communication and synchronization overheads. For SMP, while there are many strategies for reducing overhead (see Section 4), there are very few implementations that investigate synchronization between threads and measure scalability. For distributed memory parallelism, although more works take scalability into consideration, only a very few parallel SVM approaches have evaluated scalability for a large number of machines (e.g. [67, 102, 133]). In addition to measuring scalability, it would be beneficial to find a theoretical bound for scalability in order to find the optimal number of machines usable in parallel. For instance, [123] shows that such a theoretical bound exists, although it lacks the proof. To our knowledge, [123] is the only parallel SVM that investigates the theoretical bound for scalability.

The parallel implementations of SVMs using P2P networks have shown good scalability in terms of number of peers (Section 4.3.2). P2P networks are promising for solving large-scale problems since more peers can be added when a large amount of memory and computing powers are demanded. However, a few parallel SVMs have explored the impact of P2P networks on the efficiency of SVM implementations. A further investigation of efficiency and minimizing corresponding overhead would be beneficial to exploit the computing power of distributed networks.

Scalability in terms of the number of samples. Parallel SVM implementations have considered scalability (see Section 4), but only a few parallel implementations have been evaluated for large training or testing samples with/without high-dimensional input spaces. In addition, most of the parallel SVMs focus on large training samples rather than large testing samples. Except in incremental learning, very few parallel SVMs use samples that do not fit into memory. It would be useful to investigate the scalability of parallel implementations regarding both training and testing samples and the dimension of the input spaces for large-scale problems. This would allow us to handle a wide range of datasets without limited size range.

5.3.4 Accuracy. There are many strategies to reduce data in order to fit it into the available memory (see Section 4.6.2). The benefit comes at the expense of poor accuracy or a slight deterioration of accuracy. The parallel SVMs reviewed in this article show a trade-off between the training time and accuracy regarding how much data can be reduced (see Section 4.4). In addition, parallel SVMs often require updating or retraining the algorithms to iteratively improve accuracy. Promising models such as a Hadoop implementation of map-reduce may not support algorithms with an iterative manner or sequential nature (see Section 4.3.3). The sequential-nature algorithms require updating at each iteration, which may lead to high communication overhead. Iterative algorithms may require moderate numbers of iterations to reach the global solution. Each iteration is solved using a map-reduce job. Thus, many iterations require many map-reduce jobs, which are expensive to launch [160]. Therefore, starting up parallel routines at each iteration is inefficient and, if the problem is not large enough, it leads to high overhead. In this regard, it would be useful to implement a map-reduce framework for iterative algorithms and investigate the possible trade-offs between accuracy and training time.

5.4 Promising Parallel Approaches

In this section, we discuss the parallel approaches that have shown promising performance and efficiency for solving large-scale problems (for detailed information, refer to Section 4).

5.4.1 Map-reduce. As we discussed in Section 5.3.1, map-reduce-based algorithms have good scalability in terms of the number of machines and samples. Nevertheless, they may not perform well for sequential or iterative-nature algorithms. There is an extension of the Hadoop implementation of map-reduce called Twister that supports iterative algorithms [160]. To our knowledge, only a few parallel map-reduce-based SVMs, including [107], use Twister. Further exploration of the map-reduce framework in Twister can be useful for SVM algorithms that solve large-scale problems in an iterative manner. Another point that has not been explored sufficiently is the impacts of the number of mappers and reducers on the performance of parallel SVMs since, after some numbers, the execution time for an increasing number of mappers/reducers is saturated [114]. A further investigation of optimal numbers of mappers and reducers is required to improve the efficiency of a map-reduce framework.

5.4.2 Incremental Learning. It can overcome memory limitations for large-scale problems. Incremental learning shows good scalability in terms of the number of samples and number of machines because of low overhead. In spite of that, good scalability happens to samples with small enough dimensional input spaces [91] and reducing dimensions for large-scale problems is difficult (see Section 4.2.5). Therefore, it seems that the size of samples is not an issue for the scalability of incremental learning, but the dimension is. Incremental learning can be further explored in order to find possible avenues for solving problems with high-dimensional input spaces.

5.4.3 Combination Approaches. Parallel incremental learning, map-reduce, the cascade, and distributed approaches—particularly ADMM—are promising to reduce memory requirements and accelerate the training process for large-scale problems. A combination of these approaches may lead to further improvements since they can complement each other (see Sections 4.1.3, 4.2, 4.3, and 4.6.2). It would be interesting to find out which combinations match and give the optimal results and performance.

One suggestion can be a combination of ADMM with map-reduce for multi-class and non-linear classifications of large-scale problems. ADMM is one of the promising distributed schemes for reducing memory requirements. In spite of that, it suffers from slow convergence and high time complexity [145]. A combination of ADMM with methods that reduce time complexity and accelerate the convergence rate is beneficial. Map-reduce has this potential and matches the distributed structure of ADMM.

5.4.4 Network Architecture. The dominant architectures used for parallel SVMs are centralized networks. On these networks, a master node often has the duty of distributing data among slaves, each of which often needs to communicate or to be synced by the master to obtain the final result. One drawback is that communication and synchronization overheads in centralized networks reduce the efficiency of parallel implementations. In contrast, decentralized computing or P2P computing models can minimize the overhead and reduce memory requirements (see Section 4.2.3). In these models, each node receives a part of data only from neighbors, thus skips communicating with the master. The parallel algorithms proposed by Bickson et al. [133], Fei et al. [134], and Ang et al. [101] show the attractiveness and potential of P2P models in which large-scale problems can be solved with a classification accuracy comparable to the centralized model. Further research is required to solve issues such as handling asynchronous communications for decentralized networks.

6 CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

Parallel computing of SVMs is becoming a necessity for improving the performance of SVMs for big data and already has demonstrated promising results for improving large-scale problems. This survey presents a summary of state-of-the-art techniques and tools used to solve SVMs in parallel. As we outline, there is still space for further improvement regarding computation time, accuracy, scalability, and memory issues owing to the immense and increasing size of real-life data requiring judicious choice for end users. With existing challenges and trade-offs in mind, designers have a great deal of flexibility in designing and implementing SVMs by taking their goals of parallelism into consideration. For instance, considerable speedups can be achieved by a slight deterioration of classification accuracy, which might be acceptable in some applications. One important point is to identify the efficient parallel tools, heuristics, and strategies that fit the characteristics of SVM algorithms in mind; otherwise, one should be prepared for modifications and manipulations of the corresponding algorithms or strategies to take maximum advantage of parallelism. In the current trend of parallelizing SVMs, it seems that the parallel implementations of SVM solvers are still not sufficient to handle large-scale problems and their challenges open up directions for future work. Here, we mention some of the potential open questions that result from our review of parallel approaches of SVMs on large-scale problems.

Use of the four focus lines. We have identified the four focus lines of parallelism that have been targeted in the parallel SVM implementations, that is, memory, speedup, accuracy, and scalability. These four focus lines of parallelism are not independent and there are trade-offs between them. Consequently, there has not been much work addressing all four focus lines at the same time. One direction of future work is to develop robust algorithms that are capable of addressing all four focus lines of parallelism and their possible impacts. Another aspect is improving the transparency of the field by focusing on the reproducibility and reliability of the experiments. These give the possibility of comparing the performance of the developed algorithms with the existing ones independent of the specific size of data and dimension of the input spaces.

Use of combinational approaches. As shown in Table 3, combinational approaches have been successful in handling large samples since the matching techniques improve the possible weaknesses. To take maximum advantage of parallelism, a future direction likely to be successful would be the development of an integrated framework that combines complementing and matching techniques and gives the possibility of combining several techniques. A thorough investigation of combining the state-of-the-art techniques can open up new frontiers for solving large-scale problems.

Use of available modern processor technologies. Modern processors already contain technologies that designers of parallel algorithms can take advantage of. These technologies are employed by high-performance and parallel libraries and software, for example, MKL. Indirectly, using these libraries and software helps the users to improve the performance of developed algorithms. In addition, one can directly use modern processor technologies in SVM algorithms to further improve the performance, for example, in [55, 123, 161]. However, the restrictive design of algorithms is a hurdle and may not allow using these technologies to the full extent [123]. A future research direction can be to implement non-restrictive parallel SVMs that allow using the available modern processor technologies, for example, SIMD instructions, SSE, and AVX.

Use of decentralized computing. Solving large-scale problems requires a large amount of memory, which can be provided by adding more computing resources from different physical locations. However, the majority of the parallel SVMs have their main focus on a centralized computing in which the slaves communicate more or less regularly with a master to obtain the final results.

This can cause overhead, thus is a hurdle to overcome for effective parallel implementations. An interesting future research direction is to develop algorithms that are suitable for P2P and decentralized computing to take the maximum advantages of distributed computing resources without major overhead and loss of performance.

ACKNOWLEDGMENTS

The author thanks Alexander Schliep from the University of Gothenburg for his excellent support.

REFERENCES

[1] L. T. Yang and M. Guo. 2005. *High-Performance Computing: Paradigm and Infrastructure.* Wiley. https://books.google.se/books?id=qA4DbnFB2XcC

[2] Vladimir Vapnik. 2013. *The Nature of Statistical Learning Theory.* Springer Science & Business Media. 314 pages.

[3] Hyeran Byun and Seong-Whan Lee. 2002. Applications of support vector machines for pattern recognition: A survey. In *Pattern Recognition with Support Vector Machines.* Springer, 213–236.

[4] Zhao Bin, Liu Yong, and Xia Shao-Wei. 2000. Support vector machine and its application in handwritten numeral recognition. In *Proceedings of the 15th International Conference on Pattern Recognition.* Vol. 2. 720–723. DOI: http://dx.doi.org/10.1109/ICPR.2000.906176

[5] Peter Bartlett and John Shawe-Taylor. 1999. Generalization performance of support vector machines and other pattern classifiers. *Advances in Kernel Methods-Support Vector Learning*, C. Burges and B. Scholkopf (Eds.). MIT Press, 43–54.

[6] Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. 2012. Recent advances of large-scale linear classification. *Proceedings of IEEE* 100, 9 (2012), 2584–2603.

[7] Janmenjoy Nayak, Bighnaraj Naik, and H. S. Behera. 2015. A comprehensive survey on support vector machine in data mining tasks: Applications & challenges. *International Journal of Database Theory and Application* 8, 1 (2015), 169–186.

[8] R. Ravinder Reddy, B. Kavya, and Y. Ramadevi. 2014. A survey on SVM classifiers for intrusion detection. In *International Journal of Computer Applications* 98, 19 (2014), 34–44.

[9] G. Wang. 2008. A survey on training algorithms for support vector machine classifiers. In *4th International Conference on Networked Computing and Advanced Information Management (NCM'08).* Vol. 1. 123–128. DOI: http://dx.doi.org/10.1109/NCM.2008.103

[10] Thanh-Nghi Do, Van-Hoa Nguyen, and François Poulet. 2008. Speed up SVM algorithm for massive classification tasks. In *Advanced Data Mining and Applications.* Springer, 147–157.

[11] Kristian Woodsend and Jacek Gondzio. 2009. High-performance parallel support vector machine training. In *Parallel Scientific Computing and Optimization.* Springer, 83–92.

[12] Nanbo Peng, Yanxia Zhang, and Yongheng Zhao. 2011. CUDA-accelerated SVM for celestial object classification. In *Astronomical Data Analysis Software and Systems Xx*, Vol. 442. 119–122. Astronomical Society of the Pacific.

[13] H. X. Zhao and Frédéric Magoules. 2011. Parallel support vector machines on multi-core and multiprocessor systems. In *11th International Conference on Artificial Intelligence and Applications (AIA'11)*, R. Fox (Ed.). IASTED, 75–81.

[14] Dominik Brugger. 2006. *Parallel Support Vector Machines.* Arbeitsbereich Technische Informatik, Universität Tübingen. 27 pages. https://publikationen.uni-tuebingen.de/xmlui/bitstream/handle/10900/49015/pdf/tech_21.pdf?sequence=1&isAllowed=y

[15] Emmanuel Didiot and Fabien Lauer. 2015. Efficient optimization of multi-class support vector machines with MSVMpack. In *Modelling, Computation and Optimization in Information Systems and Management Sciences.* Springer, 23–34.

[16] Ana Gainaru, Emil Slusanschi, and Stefan Trausan-Matu. 2011. Mapping data mining algorithms on a GPU architecture: A study. In *Foundations of Intelligent Systems.* Springer, 102–112.

[17] Hans P. Graf, Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. 2004. Parallel support vector machines: The cascade SVM. In *Proceedings of the 17th International Conference on Neural Information Processing Systems.* MIT Press, 521–528.

[18] Stephen Tyree, Jacob R. Gardner, Kilian Q. Weinberger, Kunal Agrawal, and John Tran. 2014. Parallel support vector machines in practice. *arXiv preprint arXiv:1404.1066* (2014).

[19] Evans Data Software Developer surveys 2011-2013. Performance: Ready to Use. Retrieved October 11, 2015 from https://software.intel.com/en-us/intel-mkl.

[20] NVIDIA Corporation. 2015. cuBLAS. Retrieved October 11, 2015 from https://developer.nvidia.com/cublas.

[21] Yunmei Lu, Yun Zhu, Meng Han, Jing (Selena) He, and Yanqing Zhang. 2014. A survey of GPU accelerated SVM. In *Proceedings of the ACM Southeast Regional Conference (ACM SE'14).* ACM, Article 15, 7 pages. DOI: http://dx.doi.org/10.1145/2638404.2638474

[22] John Platt. 1998. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.* Technical Report MSR-TR-98-14. Microsoft Research. 21 pages. Retrieved December 12, 2018 from http://research.microsoft.com/apps/pubs/default.aspx?id=69644
[23] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory*. ACM, 144–152.
[24] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.
[25] Ovidiu Ivanciuc. 2007. Applications of support vector machines in chemistry. *Reviews in Computational Chemistry* 23 (2007), 291–400.
[26] Gaetano Zanghirati and Luca Zanni. 2003. A parallel solver for large quadratic programs in training support vector machines. *Parallel Computing* 29, 4 (2003), 535–551.
[27] AUSTIN Carpenter. 2009. cuSVM: A CUDA implementation of support vector classification and regression. Retrieved December 12, 2018 from http://www.patternsonascreen.net/cuSVMDesc.pdf.
[28] Tao Li, Hua Li, Xuechen Liu, Shuai Zhang, Kai Wang, and Yulu Yang. 2013. GPU acceleration of interior point methods in large scale SVM training. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'13)*. IEEE, 863–870.
[29] Thanh-Nghi Doan, Thanh-Nghi Do, and François Poulet. 2013. Large scale visual classification with parallel, imbalanced bagging of incremental LIBLINEAR SVM. In *Proceedings of the International Conference on Data Mining (DMIN'13)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 1.
[30] W. W. Hwu. 2011. *GPU Computing Gems Emerald Edition*. Elsevier Science. https://books.google.se/books?id=lGMzmbUhpiAC
[31] Volkan Vural and Jennifer G. Dy. 2004. A hierarchical method for multi-class support vector machines. In *Proceedings of the 21st International Conference on Machine Learning*. ACM, 105–112.
[32] L. J. Cao, S. S. Keerthi, Chong-Jin Ong, J. Q. Zhang, and H. P. Lee. 2006. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Transactions on Neural Networks,* 17, 4 (July 2006), 1039–1049. DOI:http://dx.doi.org/10.1109/TNN.2006.875989
[33] Kristian Woodsend and Jacek Gondzio. 2009. Hybrid MPI/OpenMP parallel linear support vector machine training. *Journal of Machine Learning Research* 10 (Dec. 2009), 1937–1953. http://dl.acm.org/citation.cfm?id=1577069.1755850.
[34] John C. Platt. 1999. Using analytic QP and sparseness to speed training of support vector machines. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*. MIT Press, 557–563.
[35] Valeriu Codreanu, Bob Dröge, David Williams, Burhan Yasar, Po Yang, Baoquan Liu, Feng Dong, Olarik Surinta, Lambert R. B. Schomaker, Jos B. T. M. Roerdink, et al. 2016. Evaluating automatically parallelized versions of the support vector machine. In *Concurrency and Computation: Practice and Experience* 28, 7 (2016), 2274–2294.
[36] Tatjana Eitrich and Bruno Lang. 2005. *Efficient Implementation of Serial and Parallel Support Vector Machine Training with a Multi-parameter Kernel for Large-scale Data Mining*. Citeseer.
[37] Fernando Pérez-Cruz, Anıbal R. Figueiras-Vidal, and Antonio Artés-Rodríguez. 2004. Double chunking for solving SVMs for very large datasets. In *Proceedings of Learning* (2004).
[38] Luca Zanni, Thomas Serafini, and Gaetano Zanghirati. 2006. Parallel software for training large scale support vector machines on multiprocessor systems. *Journal of Machine Learning Research* 7, 7 (2006), 1467–1492.
[39] Thomas Serafini, Gaetano Zanghirati, and Luca Zanni. 2005. Gradient projection methods for quadratic programs and applications in training support vector machines. *Optimization Methods and Software* 20, 2-3 (2005), 353–378.
[40] Andrew Cotter, Nathan Srebro, and Joseph Keshet. 2011. A GPU-tailored approach for training kernelized SVMs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 805–813.
[41] Edward Y. Chang. 2011. *PSVM: Parallelizing Support Vector Machines on Distributed Computers*. Springer, Berlin, 213–230. DOI:http://dx.doi.org/10.1007/978-3-642-20429-6_10
[42] Roberto Díaz-Morales, Harold Y. Molina-Bulla, and Angel Navia-Vázquez. 2011. Parallel semiparametric support vector machines. In *International Joint Conference on Neural Networks (IJCNN'11)*. IEEE, 475–481.
[43] Godwin Caruana, Maozhen Li, and Man Qi. 2011. A MapReduce based parallel SVM for large scale spam filtering. In *8th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'11)*, Vol. 4. IEEE, 2659–2662.
[44] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. 2011. GPU acceleration for support vector machines. In *12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS'11), Delft, the Netherlands, April 13-15, 2011.* TU Delft; EWI; MM; PRB.
[45] W. C. C. Chu, H. C. Chao, and S. J. H. Yang. 2015. *Intelligent Systems and Applications: Proceedings of the International Computer Symposium (ICS) Held at Taichung, Taiwan, December 12–14, 2014.* IOS Press. https://books.google.se/books?id=dOqbCAAAQBAJ

[46] Jian-xiong Dong, Adam Krzyżak, and Ching_Y. Suen. 2003. A fast parallel optimization for training support vector machine. In *Machine Learning and Data Mining in Pattern Recognition*. Springer, 96–105.

[47] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. 2008. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 104–111.

[48] Léon Bottou and Chih-Jen Lin. 2007. Support vector machine solvers. In *Large Scale Kernel Machines*. MIT Press, 301–320.

[49] Xueqin Zhang, Yifeng Zhang, and Chunhua Gu. 2014. GPU implementation of parallel support vector machine algorithm with applications to detection intruder. *Journal of Computers* 9, 5 (2014), 1117–1124.

[50] Léon Bottou. 2007. *Large-scale Kernel Machines*. MIT Press.

[51] Qi Li, Raied Salman, Erik Test, Robert Strack, and Vojislav Kecman. 2011. GPUSVM: A comprehensive CUDA based support vector machine package. *Open Computer Science* 1, 4 (2011), 387–405.

[52] Lingfeng Niu and Ya-xiang Yuan. 2011. A parallel decomposition algorithm for training multiclass kernel-based vector machines. *Optimization Methods and Software* 26, 3 (2011), 431–454.

[53] Qi Li, Raied Salman, Erik Test, Robert Strack, and Vojislav Kecman. 2013. Parallel multitask cross validation for support vector machine using GPU. *Journal of Parallel and Distributed Computing* 73, 3 (2013), 293–302.

[54] Jian-xiong Dong, Adam Krzyżak, and Ching Y Suen. 2005. Fast SVM training algorithm with decomposition on very large data sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27, 4 (2005), 603–618.

[55] Yang You, S. L. Song, Haohuan Fu, A. Marquez, M. M. Dehnavi, K. Barker, K. W. Cameron, A. P. Randles, and Guangwen Yang. 2014. MIC-SVM: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures. In *IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 809–818. DOI: http://dx.doi.org/10.1109/IPDPS.2014.88

[56] Sriram Venkateshan, Alap Patel, and Kuruvilla Varghese. 2015. Hybrid working set algorithm for SVM learning with a kernel coprocessor on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 10 (2015), 2221–2232.

[57] T. Joachims. 1998. *Making Large-Scale SVM Learning Practical*. LS8-Report 24. Universität Dortmund, LS VIII-Report.

[58] Yunmei Lu, Yun Zhu, Meng Han, Jing Selena He, and Yanqing Zhang. 2014. A survey of GPU accelerated SVM. In *Proceedings of the ACM Southeast Regional Conference*. ACM, 15.

[59] Thorsten Joachims. 1999. *Making Large Scale SVM Learning Practical*. Technical Report. Universität Dortmund.

[60] Moreno Marzolla. 2011. Fast training of support vector machines on the cell processor. *Neurocomputing* 74, 17 (2011), 3700–3707.

[61] Jing Jin, Xianggao Cai, and Xiaola Lin. 2013. Efficient SVM training using parallel primal-dual interior point method on GPU. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'13)*. IEEE, 12–17.

[62] Jeyanthi Narasimhan, Abhinav Vishnu, Lawrence Holder, and Adolfy Hoisie. 2014. Fast support vector machines using parallel adaptive shrinking on distributed systems. *arXiv preprint arXiv:1406.5161* (2014).

[63] João Gonçalves, Noel Lopes, and Bernardete Ribeiro. 2012. Multi-threaded support vector machines for pattern recognition. In *Neural Information Processing*. Springer, 616–623.

[64] Pengfei Chang, Zhuo Bi, and Yiyong Feng. 2014. Parallel SMO algorithm implementation based on OpenMP. In *IEEE International Conference on System Science and Engineering (ICSS'14)*. 236–240. DOI: http://dx.doi.org/10.1109/ICSSE.2014.6887941

[65] Tatjana Eitrich and Bruno Lang. 2006. Data mining with parallel support vector machines for classification. In *Advances in Information Systems*. Springer, 197–206.

[66] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. 2012. Performance of SSE and AVX instruction sets. *arXiv preprint arXiv:1211.0820* (2012).

[67] Abhinav Vishnu, Jeyanthi Narasimhan, Lawrence Holder, Darren Kerbyson, and Adolfy Hoisie. 2015. Fast and accurate support vector machines on large scale systems. In *IEEE International Conference on Cluster Computing (CLUSTER'15)*. IEEE, 110–119.

[68] L. V. Ferreira, E. Kaszkurewicz, and Amit Bhaya. 2006. Parallel implementation of gradient-based neural networks for SVM training. In *International Joint Conference on Neural Networks (IJCNN'06)*. IEEE, 339–346. DOI: http://dx.doi.org/10.1109/IJCNN.2006.246701

[69] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. 2000. Improvements to the SMO algorithm for SVM regression. *IEEE Transactions on Neural Networks* 11, 5 (Sep 2000), 1188–1193. DOI: http://dx.doi.org/10.1109/72.870050

[70] F. Aiolli and A. Sperduti. 2002. An efficient SMO-like algorithm for multiclass SVM. In *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*. IEEE, 297–306. DOI: http://dx.doi.org/10.1109/NNSP.2002.1030041

[71] Peng Peng, Qian-Li Ma, and Lei-Ming Hong. 2009. The research of the parallel SMO algorithm for solving SVM. In *International Conference on Machine Learning and Cybernetics*, Vol. 3. 1271–1274. DOI:http://dx.doi.org/10.1109/ICMLC.2009.5212348

[72] Elad Yom-Tov. 2005. A parallel training algorithm for large scale support vector machines. In *Neural Information Processing Systems (NIPS'05), Workshop on Large Scale Kernel Machines*. Whistler, Canada.

[73] T. Hazan, A. Man, and A. Shashua. 2008. A parallel decomposition solver for SVM: Distributed dual ascend using fenchel duality. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'08).*IEEE, 1–8. DOI:http://dx.doi.org/10.1109/CVPR.2008.4587354

[74] Nasullah Khalid Alham, Maozhen Li, and Yang Liu. 2014. Parallelizing multiclass support vector machines for scalable image annotation. *Neural Computing and Applications* 24, 2 (2014), 367–381.

[75] MPI Forum. 2015. Message Passing Interface Forum. Retrieved October 18, 2015 from http://www.mpi-forum.org/.

[76] Sergio Herrero-Lopez, John R. Williams, and Abel Sanchez. 2010. Parallel multiclass classification using SVMs on GPUs. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, 2–11. DOI:http://dx.doi.org/10.1145/1735688.1735692

[77] Quan Liao, Jibo Wang, Yue Webster, and Ian A. Watson. 2009. GPU accelerated support vector machines for mining high-throughput screening data. *Journal of Chemical Information and Modeling* 49, 12 (2009), 2718–2725.

[78] Krzysztof Sopyła, Paweł Drozda, and Przemysław Górecki. 2012. SVM with CUDA accelerated kernels for big sparse problems. In *Artificial Intelligence and Soft Computing*. Springer, 439–447.

[79] Qi Li. 2011. *Fast Parallel Machine Learning Algorithms for Large Datasets Using Graphic Processing Unit.* Ph.D. thesis. AAI3489482.

[80] Tsung-Kai Lin and Shao-Yi Chien. 2010. Support vector machines on GPU with sparse matrix format. In *9th International Conference on Machine Learning and Applications*. IEEE, 313–318.

[81] J. Vaněk, J. Michálek, and J. Psutka. 2017. A GPU-architecture optimized hierarchical decomposition algorithm for support vector machine training. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (Dec 2017), 3330–3343. DOI:http://dx.doi.org/10.1109/TPDS.2017.2731764

[82] Yang You and James Demmel. 2017. Runtime data layout scheduling for machine learning dataset. In *46th International Conference on Parallel Processing (ICPP'17).* IEEE, 452–461.

[83] Noel Lopes and Bernardete Ribeiro. 2011. GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications* 3 (2011), 355–362.

[84] Qing He, Changying Du, Qun Wang, Fuzhen Zhuang, and Zhongzhi Shi. 2011. A parallel incremental extreme SVM classifier. *Neurocomputing* 74, 16 (2011), 2532–2540.

[85] Amund Tveit and Havard Engum. 2003. Parallelization of the incremental proximal support vector machine classifier using a heap-based tree topology. In *Parallel and Distributed Computing for Machine Learning: Proceedings of the 14th European Conference on Machine Learning and the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD'03), Cavtat-Dubrovnik, Croatia.* Citeseer.

[86] Gang Wu, Edward Chang, Yen Kuang Chen, and Christopher Hughes. 2006. Incremental approximate matrix factorization for speeding up support vector machines. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 760–766.

[87] Shin Matsushima, S. V. N. Vishwanathan, and Alexander J. Smola. 2012. Linear support vector machines via dual cached loops. In *Proceedings of the 18th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining.* ACM, 177–185.

[88] Thanh-Nghi Do and François Poulet. 2006. Classifying one billion data with a new distributed SVM algorithm. In *International Conference on Research, Innovation and Vision for the Future (RIVF'06).* IEEE, 59–66.

[89] Pedro A. Forero, Alfonso Cano, and Georgios B. Giannakis. 2010. Consensus-based distributed support vector machines. *Journal of Machine Learning Research* 11 (2010), 1663–1707.

[90] Yumao Lu, Vwani Roychowdhury, and Lieven Vandenberghe. 2008. Distributed parallel support vector machines in strongly connected networks. *IEEE Transactions on Neural Networks* 19, 7 (2008), 1167–1178.

[91] Thanh-Nghi Do, Van-Hoa Nguyen, and François Poulet. 2008. A fast parallel SVM algorithm for massive classification tasks. In *Modelling, Computation and Optimization in Information Systems and Management Sciences.* Springer, 419–428.

[92] Nadeem Ahmed Syed, Syed Huan, Liu Kah, and Kay Sung. 1999. Incremental learning with support vector machines. In *Proceedings of the Workshop on Support Vector Machines at the International Joint Conference on Artificial Intelligence.* CiteSeer.

[93] Cornelia Caragea, Doina Caragea, and Vasant Honavar. 2005. Learning support vector machines from distributed data sources. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, Vol. 4. AAAI Press, 1602–1603.

[94] FAL Labs. 2011. Bukkoji Cabinet: a straightforward implementation of DBM. Retrieved October 2, 2016 from http://fallabs.com/kyotocabinet/.

[95] R. Bekkerman, M. Bilenko, and J. Langford. 2012. *Scaling Up Machine Learning: Parallel and Distributed Approaches.* Cambridge University Press. https://books.google.se/books?id=c5v5USMvcMYC

[96] Xue-Qiang Zeng and Guo-Zheng Li. 2014. Incremental partial least squares analysis of big streaming data. *Pattern Recognition* 47, 11 (2014), 3726–3735.

[97] Jian-Pei Zhang, Zhong-Wei Li, and Jing Yang. 2005. A parallel SVM training algorithm on large-scale classification problems. In *Proceedings of International Conference on Machine Learning and Cybernetics.* Vol. 3. IEEE, 1637–1641.

[98] Oliver Meyer, Bernd Bischl, and Claus Weihs. 2014. Support vector machines on large data sets: Simple parallel approaches. In *Data Analysis, Machine Learning and Knowledge Discovery.* Springer, 87–95.

[99] Mingsheng Hu and Weixu Hao. 2010. A parallel approach for SVM with multi-core CPU. In *International Conference on Computer Application and System Modeling (ICCASM'10)*, Vol. 15. IEEE, V15–373–V15–377. DOI: http://dx.doi.org/10.1109/ICCASM.2010.5622595

[100] Jing Yang. 2006. An improved cascade SVM training algorithm with crossed feedbacks. In *1st International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06).* Vol. 2. IEEE, 735–738.

[101] Hock Hee Ang, Vivekanand Gopalkrishnan, Steven CH Hoi, and Wee Keong Ng. 2008. Cascade RSVM in peer-to-peer networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer, 55–70.

[102] Yang You, J. Demmel, K. Czechowski, Le Song, and R. Vuduc. 2015. CA-SVM: Communication-avoiding support vector machines on distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'15).* IEEE, 847–859. DOI: http://dx.doi.org/10.1109/IPDPS.2015.117

[103] Hongle Du, Shaohua Teng, Xiufen Fu, Wei Zhang, and Yuanfang Pu. 2009. A cooperative intrusion detection system based on improved parallel SVM. In *Joint Conferences on Pervasive Computing (JCPC'09).* IEEE, 515–518.

[104] Daniel Weimer, Sebastian Köhler, Christian Hellert, Konrad Doll, Ulrich Brunsmann, and Roland Krzikalla. 2011. GPU architecture for stationary multisensor pedestrian detection at smart intersections. In *IEEE Intelligent Vehicles Symposium (IV'11),*. IEEE, 89–94.

[105] Qi Li, Raied Salman, and Vojislav Kecman. 2010. An intelligent system for accelerating parallel SVM classification problems on large datasets using GPU. In *10th International Conference on Intelligent Systems Design and Applications (ISDA'10).* IEEE, 1131–1135.

[106] Nasullah Khalid Alham, Maozhen Li, Yang Liu, Suhel Hammoud, and Mahesh Ponraj. 2011. A distributed SVM for scalable image annotation. In *8th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'11)*, Vol. 4. IEEE, 2655–2658.

[107] Zhanquan Sun and Geoffrey Fox. 2012. Study on parallel SVM based on MapReduce. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12).* 16–19.

[108] Ke Xu, Cui Wen, Qiong Yuan, Xiangzhu He, and Jun Tie. 2014. A MapReduce based parallel SVM for email classification. *Journal of Networks* 9, 6 (2014), 1640–1647.

[109] N. Z. Tarapore, D. B. Kulkarni, and V. K. Prasad. 2016. Implementation of parallel algorithm for support vector machine applied to intrusion detection systems. In *International Conference on Computing, Analytics and Security Trends (CAST'16).* IEEE, 179–184. DOI: http://dx.doi.org/10.1109/CAST.2016.7914962

[110] Igor Durdanovic, Eric Cosatto, and Hans-Peter Graf. 2007. *Large Scale Parallel SVM Implementation.* MIT Press, 105–138.

[111] Zhen-Yu Chen and Zhi-Ping Fan. 2014. Parallel multiple kernel learning: A hybrid alternating direction method of multipliers. *Knowledge and Information Systems* 40, 3 (2014), 673–696.

[112] Yimin Wen and Baoliang Lu. 2005. A hierarchical and parallel method for training support vector machines. In *Proceedings of the 2nd International Conference on Advances in Neural Networks - Volume Part I (ISNN'05).* Springer, Berlin, 881–886. DOI: http://dx.doi.org/10.1007/11427391_141

[113] Yuh-Jye Lee and Olvi L. Mangasarian. 2001. RSVM: Reduced support vector machines. In *Proceedings of the 2001 Society for Industrial and Applied Mathematics (SIAM) International Conference on Data Mining.* 1–17.

[114] Wenming Guo, Nasullah Khalid Alham, Yang Liu, Maozhen Li, and Man Qi. 2015. A resource aware MapReduce based parallel SVM for large scale image classifications. *Neural Processing Letters* 44, 1 (2015), 161–184.

[115] Parisa Pouladzadeh, Shervin Shirmohammadi, Aslan Bakirov, Ahmet Bulut, and Abdulsalam Yassine. 2014. Cloud-based SVM for food categorization. *Multimedia Tools and Applications* 74, 14 (2014), 5243–5260.

[116] Guang-Bin Huang, KZ Mao, Chee-Kheong Siew, and De-Shuang Huang. 2005. Fast modular network implementation for support vector machines. *IEEE Transactions on Neural Networks,* 16, 6 (2005), 1651–1663.

[117] Marc Claesen, Frank De Smet, Johan AK Suykens, and Bart De Moor. 2014. EnsembleSVM: A library for ensemble learning using support vector machines.*Journal of Machine Learning Research* 15, 1 (2014), 141–145.

[118] Anirban Chatterjee, Kelly Fermoyle, and Padma Raghavan. 2010. Characterizing sparse preconditioner performance for the support vector machine kernel. *Procedia Computer Science* 1, 1 (2010), 367–375.

[119] E. Michael Gertz and Joshua D. Griffin. 2010. Using an iterative linear solver in an interior-point method for generating support vector machines. *Computational Optimization and Applications* 47, 3 (2010), 431–453.

[120] Kaihua Zhu, Hang Cui, Hongjie Bai, Jian Li, Zhihuan Qiu, Hao Wang, Hui Xu, and Edward Y. Chang. 2007. Parallel approximate matrix factorization for kernel methods. In *IEEE International Conference on Multimedia and Expo*. IEEE, 1275–1278.

[121] Edward Y. Chang, Hongjie Bai, and Kaihua Zhu. 2009. Parallel algorithms for mining large-scale rich-media data. In *Proceedings of the 17th ACM International Conference on Multimedia*. ACM, 917–918.

[122] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research* 9, Aug (2008), 1871–1874.

[123] S. Tavara, H. Sundell, and A. Dahlbom. 2015. Empirical study of time efficiency and accuracy of support vector machines using an improved version of PSVM. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'15)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 177.

[124] T. Li, X. Liu, Q. Dong, W. Ma, and K. Wang. 2016. HPSVM: Heterogeneous parallel SVM with factorization based IPM algorithm on CPU-GPU cluster. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'16)*. IEEE, 74–81. DOI: http://dx.doi.org/10.1109/PDP.2016.29

[125] Aliaksei Severyn and Alessandro Moschitti. 2011. Fast support vector machines for structural kernels. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 175–190.

[126] D. Steinkraus, I. Buck, and P. Y. Simard. 2005. Using GPUs for machine learning algorithms. In *Proceedings of the 8th International Conference on Document Analysis and Recognition,* Vol. 2. IEEE Computer Society, 1115–1120. DOI: http://dx.doi.org/10.1109/ICDAR.2005.251

[127] R . A. Reyna-Rojas, D. Dragomirescu, D. Houzet, and D. Esteve. 2003. Implementation of the SVM generalization function on FPGA. In *Proceedings of the International Signal Processing Conference (ISPC'03), Dallas, TX*. 147–153.

[128] Ian Biasi, Andrea Boni, and Alessandro Zorat. 2005. A reconfigurable parallel architecture for SVM classification. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN'05)*. Vol. 5. IEEE, 2867–2872.

[129] Hans P. Graf, Srihari Cadambi, Venkata Jakkula, Murugan Sankaradass, Eric Cosatto, Srimat Chakradhar, and Igor Dourdanovic. 2009. A massively parallel digital learning processor. In *Proceedings of the 21st International Conference on Neural Information Processing Systems (NIPS'08)*. Curran Associates Inc, 529–536.

[130] Markos Papadonikolakis, Christos-Savvas Bouganis, and George Constantinides. 2009. Performance comparison of GPU and FPGA architectures for the SVM training problem. In *International Conference on Field-Programmable Technology (FPT'09)*. IEEE, 388–391.

[131] Clive Maxfield. 2011. *FPGAs: Instant Access*. Newnes.

[132] Ronan Collobert, Samy Bengio, and Yoshua Bengio. 2002. A parallel mixture of SVMs for very large scale problems. *Neural Computation* 14, 5 (2002), 1105–1114.

[133] Danny Bickson, Elad Yom-Tov, and Danny Dolev. 2008. A Gaussian belief propagation solver for large scale support vector machines. *arXiv preprint arXiv:0810.1648* (2008).

[134] Liu Fei, Zhang Wen-Ju, Yu Shui, Ma Fan-Yuan, and Li Ming-Lu. 2004. A peer-to-peer hypertext categorization using directed acyclic graph support vector machines. In *Parallel and Distributed Computing: Applications and Technologies*. Springer, 54–57.

[135] Zeyuan Allen Zhu, Weizhu Chen, Gang Wang, Chenguang Zhu, and Zheng Chen. 2009. P-packSVM: Parallel primal gradient descent kernel SVM. In *IEEE International Conference on Data Mining (ICDM'09)*. IEEE, 677–686.

[136] Angel Navia-Vázquez and Emilio Parrado-Hernandez. 2006. Distributed support vector machines. *IEEE Transactions on Neural Networks* 17, 4 (2006), 1091–1097.

[137] Thanh-Nghi Do. 2015. Non-linear classification of massive datasets with a parallel algorithm of local support vector machines. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 231–241.

[138] Nasullah Khalid Alham, Maozhen Li, Yang Liu, and Man Qi. 2013. A MapReduce-based distributed SVM ensemble for scalable image classification and annotation. *Computers & Mathematics with Applications* 66, 10 (2013), 1920–1934.

[139] Naveeen Kumar Shrivastava, Praneet Saurabh, and Bhupendra Verma. 2011. An efficient approach parallel support vector machine for classification of diabetes dataset. *International Journal of Computer Applications* 36, 6 (2011), 19–24.

[140] Shibin Qiu and Terran Lane. 2005. Parallel computation of RBF kernels for support vector classifiers. In *SDM*. SIAM, 334–345.

[141] Caoxie Zhang, Honglak Lee, and Kang G Shin. 2012. Efficient distributed linear classification algorithms via the alternating direction method of multipliers. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, Neil D. Lawrence and Mark Girolami (Eds.), and In *Proceedings of Machine Learning Research (PMLR'12)*, Vol. 22. 1398–1406.

[142] João F. C. Mota, João M. F. Xavier, Pedro M. Q. Aguiar, and Markus Puschel. 2013. D-ADMM: A communication-efficient distributed algorithm for separable optimization. *IEEE Transactions on Signal Processing* 61, 10 (2013), 2718–2723.

[143] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning* 3, 1 (2011), 1–122.

[144] Timo M. Deist, A. Jochems, Johan van Soest, Georgi Nalbantov, Cary Oberije, SeÃ¡n Walsh, Michael Eble, Paul Bulens, Philippe Coucke, Wim Dries, Andre Dekker, and Philippe Lambin. 2017. Infrastructure and distributed learning methodology for privacy-preserving multi-centric rapid learning health care: euroCAT. *Clinical and Translational Radiation Oncology* 4, Supplement C (2017), 24–31. DOI: http://dx.doi.org/10.1016/j.ctro.2016.12.004

[145] H. Wang, Y. Gao, Y. Shi, and R. Wang. 2016. Group-based alternating direction method of multipliers for distributed linear classification. *IEEE Transactions on Cybernetics* PP, 99 (2016), 1–15. DOI: http://dx.doi.org/10.1109/TCYB.2016.2570808

[146] Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. 2012. Large linear classification when data cannot fit in memory. *ACM Transactions on Knowledge Discovery from Data* 5, 4 (2012), 23:1–23:23.

[147] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. 2010. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta (Eds.). Curran Associates, 2595–2603.

[148] H. Wang, Y. Gao, Y. Shi, and R. Wang. 2017. Group-based alternating direction method of multipliers for distributed linear classification. *IEEE Transactions on Cybernetics* 47, 11 (Nov 2017), 3568–3582. DOI: http://dx.doi.org/10.1109/TCYB.2016.2570808

[149] Dmitry Pechyony, Libin Shen, and Rosie Jones. 2011. Solving large scale linear SVM with distributed block minimization. In *NIPS 2011 Workshop on Big Learning: Algorithms, Systems and Tools for Learning at Scale*. Citeseer.

[150] Roberto Díaz Morales and Ángel Navia Vázquez. 2016. Improving the efficiency of IRWLS SVMs using parallel Cholesky factorization. *Pattern Recognition Letters* 84, Supplement C (2016), 91–98. DOI: http://dx.doi.org/10.1016/j.patrec.2016.08.015

[151] Roberto Díaz Morales and Ángel Navia Vázquez. 2017. LIBIRWLS: A parallel IRWLS library for full and budgeted SVMs. *Knowledge-Based Systems* 136 (2017), 183–186. DOI: http://dx.doi.org/10.1016/j.knosys.2017.09.007

[152] Roberto Díaz Morales, H. Y. Molina-Bulla, and Ángel Navia Vázquez. 2011. Parallel semiparametric support vector machines. In *International Joint Conference on Neural Networks*. IEEE, 475–481. DOI: http://dx.doi.org/10.1109/IJCNN.2011.6033259

[153] Doan Thanh Nghi. 2015. Parallel, imbalanced bagging power mean SVM for large scale visual classification with million images and thousand classes. *Journal of Science* 4, 4 (2015), 26–40.

[154] Stephen Winters-Hilt and Kenneth Armond Jr. 2008. Distributed SVM learning and support vector reduction. *International Journal of Computing and Optimization* 4, 1 (2017), 91–114.

[155] Yuanqing Lin, Fengjun Lv, Shenghuo Zhu, Ming Yang, Timothee Cour, Kai Yu, Liangliang Cao, and Thomas Huang. 2011. Large-scale image classification: Fast feature extraction and SVM training. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1689–1696.

[156] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (Jan. 2008), 107–113. DOI: http://dx.doi.org/10.1145/1327452.1327492

[157] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. 2007. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems* 19 (2007), 281.

[158] H. Tao, B. Wu, and X. Lin. 2014. Budgeted mini-batch parallel gradient descent for support vector machines on spark. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. IEEE, 945–950. DOI: http://dx.doi.org/10.1109/PADSW.2014.7097914

[159] Thanh-Nghi Do and Jean-Daniel Fekete. 2007. Large scale classification with support vector machine algorithms. In *6th International Conference on Machine Learning and Applications (ICMLA'07)*. IEEE, 7–12.

[160] Katarina Grolinger, Michael Hayes, Wilson A. Higashino, Alexandra L'Heureux, David S. Allison, and Miriam A. M. Capretz. 2014. Challenges for MapReduce in big data. In *IEEE World Congress on Services*. IEEE, 182–189.

[161] Yang You, Haohuan Fu, Shuaiwen Leon Song, Amanda Randles, Darren Kerbyson, Andres Marquez, Guangwen Yang, and Adolfy Hoisie. 2015. Scaling support vector machines on modern HPC platforms. *Journal of Parallel and Distributed Computing* 76 (2015), 16–31. DOI: http://dx.doi.org/10.1016/j.jpdc.2014.09.005

Received June 2017; revised July 2018; accepted September 2018