Fast and Communication-Efficient Algorithm for Distributed Support Vector Machine Training

Jyotikrishna Dass, *Member, IEEE*, Vivek Sarin, *Member, IEEE*, and Rabi N. Mahapatra, *Senior Member, IEEE*

Abstract—Support Vector Machines (SVM) are widely used as supervised learning models to solve the classification problem in machine learning. Training SVMs for large datasets is an extremely challenging task due to excessive storage and computational requirements. To tackle so-called big data problems, one needs to design scalable distributed algorithms to parallelize the model training and to develop efficient implementations of these algorithms. In this paper, we propose a distributed algorithm for SVM training that is scalable and communication-efficient. The algorithm uses a compact representation of the kernel matrix, which is based on the QR decomposition of low-rank approximations, to reduce both computation and storage requirements for the training stage. This is accompanied by considerable reduction in communication required for a distributed implementation of the algorithm. Experiments on benchmark data sets with up to five million samples demonstrate negligible communication overhead and scalability on up to 64 cores. Execution times are vast improvements over other widely used packages. Furthermore, the proposed algorithm has linear time complexity with respect to the number of samples making it ideal for SVM training on decentralized environments such as smart embedded systems and edge-based internet of things, IoT.

Index Terms—Machine learning, support vector machines, classification algorithms, parallel programming, distributed computing, message passing, quadratic programming, iterative algorithms, optimization, multicore processing

1 Introduction

Machine learning is at the core of solving real-world challenges in sectors like energy, transportation, finance, business analytics, health-care and manufacturing. With the influx of huge amount of digital data from sensors, social media, mobile devices and online transactions, it has become increasingly challenging to store, process and analyze data for predictive analytics.

Support vector machines (SVMs) [1] fall under the realm of supervised machine learning wherein a mathematical model is trained on a prior dataset and associated class labels. SVMs are commonly used for classification and regression analysis. In binary SVM classification, the goal is to optimally compute a hyperplane that maximally separates the two classes. In regression, one attempts to find a function that is an optimal fit to the data with minimum deviation in the function value. Real-world applications give rise to datasets that have a non-linear structure for which kernel SVMs are used. In linear SVM problems, the training data is used in its original feature space, thereby, enabling the adoption of coordinate gradient methods [2] to achieve the above described optimization goal. In contrast, for non-linear SVM problems the data is transformed to a higher dimensional space which represents the feature space. Subsequently the classifier can be learned by simply computing the inner products between all pairs of datapoints in the feature space without explicitly calculating their transformed coordinates. This is commonly referred to as the kernel trick [1]. Since the higher dimensional coordinates of the datapoints are not explicitly computed, applying coordinate gradient methods is infeasible for non-linear SVMs.

Mathematically, SVM is a constrained convex optimization problem with a quadratic objective function. The Hessian of the quadratic function is the kernel matrix which poses major computational and scalability issues. Computation and storage of the kernel matrix grows as $O(n^2)$ for n datapoints, making it impractical to compute the matrix for large sized problems. To tackle these challenges, a variety of SVM algorithms, e.g., SMO [3], [4] have been proposed and implemented in packages such as LIBSVM [5] and SVM^{light}. While these packages have been used widely, they are not scalable to large datasets since they are based on sequential algorithms.

It has become necessary to design parallel and distributed algorithms to train kernel SVMs for large-scale problems. A number of attempts have been made to parallelize kernel SVM training. Cascade-SVM [6] is one of the earliest works that presents a parallel SVM training algorithm in which the global SVM problem is divided into local sub-problems. The cascade framework is designed in a hierarchical reduction-tree-like structure in which each layer uses independent SVM solvers. The drawback of the framework is that for large datasets it either leads to more resource requirements (SVM solvers) or longer training time. In addition, one has to always ensure that the SVM solver at the last stage is capable of handling the output of the previous layers (called support

• *The authors are with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77840. E-mail: {dass.jyotikrishna, sarin, rabi}@tamu.edu.*

Manuscript received 20 Feb. 2018, revised 31 Oct. 2018, accepted 3 Nov. 2018, Date of publication 7 Nov. 2018; date of current version 10 Apr. 2019. (Corresponding author: Jyotikrishna Dass.) Recommended for acceptance by Z. Chen. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2879950

vectors) which have trickled down through the cascade. Communication Avoiding SVM (CA-SVM) [7] is based on k-means clustering technique to partition datasets. The partitioned data are stored locally on the cluster nodes and solved independently. Since it uses the local solution from one of the nodes to predict a test sample, the methodology does not compute the global SVM solution. PSVM [8] and P-packSVM [9] are among the most popular parallel algorithms to solve global kernel SVM. PSVM exhibits limited scalability due to its quadratic dependence on the training sample size. Moreover, PSVM works with kernel matrix approximation using Incomplete Cholesky Factorization (ICF), which is difficult to parallelize, making it unfit for large datasets. In contrast, P-packSVM computes the primal form of SVM using a stochastic gradient descent method wherein the gradient is approximated at a single sample [9]. Since the primal SVM problem can be prohibitively large while its Wolfe dual problem is considerably smaller, the convergence rate of the primal solver is slower than the dual solver [10]. Hence, it is important to find alternative distributed approaches that are highly scalable and exhibit faster convergence for the dual form of kernel SVMs.

To exploit the potential of faster convergence of dual solvers we proposed a distributed QRSVM framework [11] that projects the problem onto a reduced subspace via QR factorization to minimize computation and storage needs. At present, however, it has significant communication overheads which limits its ability to achieve higher speedups in the training phase. In this paper, we propose a fast and communication-efficient algorithm for distributed QRSVM training that is scalable to large data sets and is memory-efficient. The paper has the following contributions.

1) We devise both memory and communication-efficient implementation for the distributed QRSVM framework [11] to train kernel SVM faster. The improved design significantly reduces the communication overhead by making it insignificant compared to the computation time. As a result, the resulting framework is feasible for efficient parallelism across multiple computing cores and trains SVM faster than the prior framework [11]. For instance, with the proposed approach and a computed optimal step size of 0.9 [11], we are able to train the SVM for covtype dataset on 16 cores in 18 seconds compared to 261 seconds in a prior implementation in [11], which represents an improvement of 14x.
2) We evaluate the performance of our algorithm on three real world benchmarks, covtype (geography), webspam (electronic) and SUSY (physics). The algorithm attains speed improvement of 45x, 29x and 136x, respectively, on these benchmarks on a 64 core multiprocessor with respect to sequential implementation.
3) We demonstrate through experiments that the proposed training algorithm outperforms state-of-the-art algorithms such as PSVM and P-packSVM. For instance, the distributed QRSVM algorithm was 71x faster than PSVM and 57x faster than P-packSVM on the covtype benchmark using 16 cores. In addition, our algorithm scales linearly with the sample size and hence can handle extremely large datasets such as SUSY (5M samples) with ease.

The rest of the paper is organized as follows. In Section 2, we introduce SVM as a quadratic optimization problem for non-linear classification. In Section 3 we review the QRSVM framework [11] and the dual ascent method to iteratively solve the above optimization problem. In Section 4, we present the distributed version of the QRSVM framework. In Section 5 we report the experiments on three benchmark datasets. Finally, we conclude in Section 6 with possible future research directions.

2 Preliminaries

2.1 Support Vector Machines

In a typical Support Vector Machine for the binary classification problem, the goal is to determine an optimal hyperplane that maximally separates the two classes. The hyperplane rests on a set of *support points* that determine the shape of the classifier.

For classifier training, one is given a training dataset $S = \{(x_i, y_i),\ \forall i = 1, \ldots, n\}$ with n number of samples. Each input data point $x_i \in \mathbb{R}^d$ ($\mathbb{R}^d$ is the $d-$dimensional input space) and $y_i \in \{-1, 1\}$ is the corresponding data label (or class).

An ℓ_2-regularized primal version [1] of this problem is

$$\min_w \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{n}\xi_i(w; x_i; y_i), \tag{1}$$

where $w \in \mathbb{R}^d$ represents the normal to the hyperplane separating the data points and $C > 0$ is the penalty parameter that determines the trade-off between margin maximization and training error minimization. The term $\xi_i(w; x_i; y_i)$ represents the squared *hinge* loss function associated with the optimization problem. A *bias* term, $b \in \mathbb{R}$ is typically associated with w representing the separating hyperplane, $w^T x + b = 0$.

The *dual* formulation of Equation (1) is

$$\min_\alpha \frac{1}{2}\alpha^T Z\alpha + e^T\alpha, \tag{2}$$

$$\text{subject to} \quad L \leq \alpha_i \leq U,$$

where, $Z = (G + D) \in \mathbb{R}^{n\times n}$ is dense and positive definite, $e = -\mathbf{1}_n$, L is the lower bound of each lagrangian multiplier α_i and U is its upper bound. $G = \{y_i y_j \kappa(x_i, x_j)\}$, where, $\kappa()$ represents the Mercer kernel function.

For SVM, G is derived from the kernel matrix, $K = \{\kappa(x_i, x_j), \forall i, j = 1, \ldots, n\}$ which is a positive semi-definite matrix. The diagonal matrix D, lower bound L and upper bound U are dependent on the type of loss function associated with the SVM problem. For L2-SVM with

$$\ell_2 - \mathbf{l}oss : \xi_i(w; x_i; y_i) = \max(0, 1 - y_i w^T x_i)^2,$$

we have $D = (2C)^{-1} I_n$, $L = 0$ and $U = \infty$.

There are two major types of SVMs based on the nature of the decision boundary to be learnt for any given dataset. Linear SVMs are used for finding the linearly separable hyperplane whereas Kernel (or non-linear) SVMs are used to learn complex and non-linear decision boundaries between the two classes present in the dataset. For kernel

(or non-linear) SVMs, it is advantageous to solve the problem using its *dual* form to leverage the benefits of the kernel trick [1]. In addition, by using the dual formulation, the loss function vanishes from the objective function making its optimization simpler.

For a kernel (or non-linear) SVM problem, the kernel function $\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$ is a similarity measure between a pair of data points, x_i and x_j in the dataset which has a non-linear decision boundary. Here, $\phi()$ represents a mapping that transforms the original data point, x_i from input space to the Reproducing Kernel Hilbert Space (RKHS) where it can be linearly separable. It should be noted that $\phi()$ need not be explicitly available as one has a representation of the above defined kernel function $\kappa(x_i, x_j)$. Some examples of non-linear kernel functions are polynomial kernel, radial basis function, etc.

In contrast, the kernel matrix K for linear SVM is directly separable in terms of original data-points, i.e., $K = XX^T$, where $X = \{x_i \in \mathbb{R}^d, i = 1, \ldots, n\}$. Our goal is to solve the non-linear problems since most of the datasets that exist have non-linear decision boundaries inherently.

We focus on the radial basis function (RBF) as the non-linear kernel function, i.e.,

$$\kappa(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2),$$

which transforms the data to infinite dimensional space. Unlike the linear SVM, the kernel matrix K for non-linear problems is not trivially separable. Moreover, K is associated with $O(n^3)$ computation for factoring the matrix and $O(n^2)$ memory to store the factors, which makes it challenging to scale to large n. A popular solution is to use a low-rank kernel approximation which speeds up kernel-based solvers while consuming limited memory.

Kernel approximation techniques attempt to find the best rank$-k$ approximation $K \approx AA^T$, with $A \in \mathbb{R}^{n \times k}$ $(k \ll n)$, which has the added benefit that the non-linear kernel matrix can now be written in a separable form just like the linear one. The L2-SVM formulation for kernel SVM in Equation (2) can be written as

$$\min_{\alpha} \frac{1}{2}\alpha^T\Big(\hat{A}\hat{A}^T\Big)\alpha + \frac{1}{2}\alpha^T\Big(\frac{1}{2C}I_n\Big)\alpha + e^T\alpha, \qquad (3)$$
$$\text{subject to} \quad -I_n\alpha \leq \mathbf{0}_n,$$

where, $\hat{A} = diag(y) \times A$ and $y = \{y_i \in \{-1, 1\}, i = 1, \ldots, n\}$. L2-SVM provides a simpler constraint formulation which specifies that each α_i corresponding to x_i must be non-negative. The data points corresponding to positive α_i's are the *support vectors* on which the separating hyperplane rests.

The QRSVM is based on the use of a rank-k approximation of the kernel matrix K for which we use MEKA, a "Memory Efficient Kernel Approximation" technique [12]. MEKA uses nearly same amount of storage as other approximation techniques like incomplete Cholesky decomposition [13], Greedy basis selection techniques [14] and Nyström [15] methods, while achieving lower approximation error.

2.2 Challenges

The matrix $\hat{A} \in \mathbb{R}^{n \times k}$ with $k \ll n$, has a tall and skinny (TS) structure. For SVM problems involving large n, $\hat{A}\hat{A}^T$ is a dense square matrix (Hessian component of the SVM objective function) requiring $O(n^2)$ memory to store and $O(n^3)$ computation. Furthermore, this square matrix can not be decomposed into independent and separable sub-matrices. Hence, it is difficult to parallelize the training while using this matrix. In addition, $\hat{A}\hat{A}^T$ is a rank deficient matrix that can lead to instability when minimizing the Lagrangian of the *dual* SVM problem in Equation (6).

QRSVM framework addresses these issues by employing the QR decomposition technique to efficiently transform the dense coefficient matrix into a sparse form followed by the dual ascent method to solve the resulting optimization problem iteratively until convergence is achieved.

3 QRSVM Framework

We review the fundamentals of a sequential QRSVM framework presented in [11]. In particular, we will discuss the optimization formulation based on QR decomposition factors and present the dual ascent method to solve such optimization problem. Finally, we will analyze the computational complexity of the QRSVM framework.

3.1 QRSVM Formulation

QR decomposition of the matrix $\hat{A}$ yields $\hat{A} = QR$. Here, Q is an orthogonal matrix of size $n \times n$ and R is a upper trapezoidal matrix of size $n \times k$. The cost function of the non-linear SVM problem in Equation (3) now becomes

$$\frac{1}{2}\alpha^T\Big(QRR^TQ^T\Big)\alpha + \frac{1}{2}\alpha^T\Big(\frac{1}{2C}I_n\Big)\alpha + e^T\alpha.$$

Defining $\hat{\alpha} = Q^T\alpha$, $\hat{e} = Q^Te$ and using $Q^TQ = I_n$, the L2-SVM quadratic programming (QP) problem becomes

$$\min_{\hat{\alpha}} \frac{1}{2}\hat{\alpha}^T\Big(RR^T + \frac{1}{2C}I_n\Big)\hat{\alpha} + (\hat{e})^T\hat{\alpha} \qquad (4)$$
$$\text{subject to} \quad -Q\hat{\alpha} \leq \mathbf{0}_n.$$

In Equation (4), RR^T is a symmetric sparse matrix of size n where the non-zeros are restricted to the first $k \times k$ submatrix. Thus, the Hessian of the cost function in Equation (4) is a block diagonal matrix comprising of two diagonal blocks:

1) a $k \times k$ symmetric and dense submatrix
$$(RR^T)_k + (1/2C)I_k.$$
2) a diagonal submatrix $(1/2C)I_{n-k}$.

The key benefits of the QRSVM formulation are:

Sparsity. It transforms the Hessian from a dense $n \times n$ matrix $\hat{A}\hat{A}^T + (1/2C)I_n$ to an overall sparse matrix which consists of a small dense $k \times k$ block. Thus, it requires lesser storage of $O(k^2)$ compared to earlier dense representation of $O(n^2)$.

Separability. It also renders the aforementioned non-separable Hessian matrix into a block separable form. One further leverages this separability to independently solve the two sub-problems in parallel using the dual ascent algorithm.

Stability. Training an SVM requires solving the quadratic programming problem defined in Equation (3) which is numerically stable [16]. QRSVM incorporates a stable MEKA [12] technique as a pre-processing stage followed by

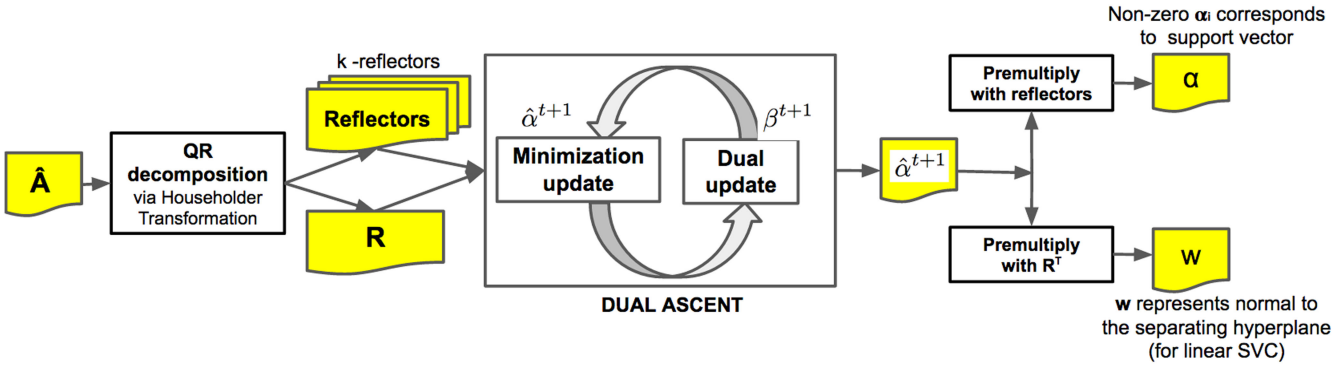


Fig. 1. QRSVM framework comprises of two main stages, namely, (1) QR decomposition of the approximated input matrix $\hat{A}$ that yields Householder reflectors and a matrix R, and (2) Dual ascent method to solve the QRSVM problem for obtaining w, which is the normal to the hyperplane (for linear SVM), and identifying set of support vectors.

Householder [18] method for QR decomposition. On applying QR decomposition, the low-rank Hessian matrix, $\hat{A}\hat{A}^T$ becomes block-separable where the two sub-diagonal blocks are now invertible. The first block $(RR^T)_k + (1/2C)I_k$ is full-rank and the second block $(1/2C)I_{n-k}$ is trivially invertible. The invertibility of the Hessian ensures stable computation of the minimization step, Equation (6), in the dual ascent stage. Overall, the QRSVM formulation is numerically stable.

3.2 Dual Ascent for QRSVM

Dual ascent is a gradient method which involves iterating through the update steps until convergence [17]. The Lagrangian $\mathcal{L}$ of the QP problem in Equation (4) is written as follows:

$$\mathcal{L}(\hat{\alpha}, \beta) = \frac{1}{2}\hat{\alpha}^T\left(RR^T + \frac{1}{2C}I_n\right)\hat{\alpha} + (\hat{e})^T\hat{\alpha} + \beta^T(-Q\hat{\alpha}), \quad (5)$$

where, $\beta \geq \mathbf{0}_n$ is the Lagrangian dual variable.

Dual ascent update steps for QRSVM are as follows.

Step 1. Minimization of Lagrangian

$$\begin{aligned}\hat{\alpha}^{t+1} &= \arg\min_{\hat{\alpha}} \mathcal{L}(\hat{\alpha}, \beta^t) \\ &= -\left(RR^T + \frac{1}{2C}I_n\right)^{-1}(-Q^T\beta^t + \hat{e}).\end{aligned} \quad (6)$$

Step 2. Dual variable update

$$\beta^{t+1} = \beta^t + \eta(-Q\hat{\alpha}^{t+1}). \quad (7)$$

Here $\eta > 0$ is the step size and the superscript $t = 0, 1, 2\ldots$ is the iteration counter. β^0 is initialized to $\mathbf{0}_n$. To satisfy the non-negativity constraint on each β_i, it is replaced with $\max(0, \beta_i)$ during every iteration.

3.3 Computational Complexity of QRSVM

In this section we present the computational complexity associated with the QRSVM framework. The framework is illustrated in Fig. 1 comprising of two major stages: QR decomposition and dual ascent.

For implementing the QR decomposition of $\hat{A}$, Householder transformation [18] is chosen since it has better numerical stability than the Gram-Schmidt process [19] and requires fewer arithmetic operations compared to Givens rotations. As discussed in [18], orthogonal matrix Q is never explicitly computed. Rather, it is stored as a set of k-Householder reflectors. The computational complexity for QR factorization is $O(nk^2)$.

In the dual ascent stage, the computational complexity of a single iteration of QRSVM is the combined cost of the two update steps defined in Equations (6) and (7). Premultiplying Q (or Q^T) to any vector v by using Householder reflectors requires $O(nk)$ operations, where n is the size of vector v and k is the number of Householder reflectors [20]. Thus, the cost of computing $(-Q^T\beta^t + \hat{e})$ in Equation (6) and $(-Q\hat{\alpha}^{t+1})$ in Equation (7) is $O(nk)$. The multiplication of the block diagonal structure of $(RR^T + \frac{1}{2C}I_n)^{-1}$ with the pre-computed $(-Q^T\beta^t + \hat{e})$ in Equation (6) can be split into

Subproblem 1. The first k components of $\hat{\alpha}^{t+1}$ are computed by solving a system with the $k \times k$ coefficient matrix $(RR_k^T + \frac{1}{2C}I_k)$. By computing and storing Cholesky factors of this matrix before starting the iterations, the system can be solved in $O(k^2)$ operations. Cholesky factorization of the coefficient matrix is a one time calculation that is carried out in the beginning of the dual ascent algorithm at the cost of $O(k^3)$.

Subproblem 2. Calculation of the remaining $(n-k)$ components of $\hat{\alpha}^{t+1}$ requires $O(n-k) \approx O(n)$ operations as it is reduced to scalar multiplication with $2C$.

Solving these subproblems in Equation (6) incurs $O(k^3 + k^2 + n)$ cost. Since, the pre-computation of $(-Q^T\beta^t + \hat{e})$ mentioned earlier requires $O(nk)$ cost, the overall computational cost per iteration incurred in Equation (6) is $O(nk + k^3 + k^2 + n) \approx O(nk)$, since $k \ll n$. Also, the computational cost of Equation (7) is trivially $O(nk)$ for a single iteration as mentioned earlier. Combining the cost for Equations (6) and (7), the effective computational complexity for dual ascent is $O(nk)$ per iteration.

The overall QRSVM [11] with both the stages requires $O(nk^2 + nkt_c)$ operations, where t_c is the number of iterations required for convergence of the dual ascent. The trend is empirically illustrated in [11] which depicts training time is linearly dependent on the number of samples, n.

To identify support vectors for prediction, the values of α are recovered from $\hat{\alpha}$ by simply pre-multiplying $\hat{\alpha}$ with Q via $k-$Householder reflectors in $O(nk)$ operations. For kernel SVMs where $k-$rank kernel approximation techniques such as the one in [21] are used, QRSVM finds the prediction for a test sample in $O(k^2)$ cost using the simplification $R_k^T\hat{\alpha}_k$.

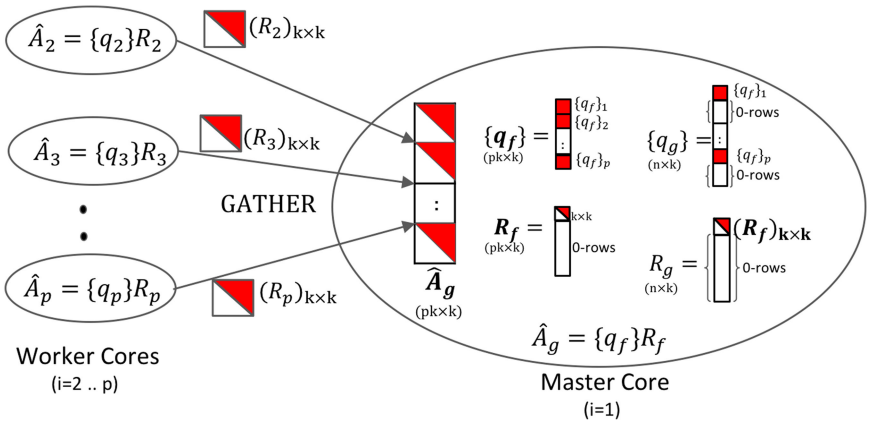


Fig. 2. A two-level implementation of distributed QR decomposition of $\hat{A}$. The orthogonal matrices are stored as sets of their Householder reflectors, denoted as $\{q\}$. $(R_i)_{k\times k}$ are gathered from all the cores and $\hat{A}_g$ is assembled at the master core. $\hat{A}_g = \{q_f\}R_f$ is computed which can be converted to original global factors, $\{q_g\}$ and R_g by appending appropriate rows of zeros as depicted [11]. However, in this work the proposed implementation uses memory-efficient representations of the global factors for $\hat{A}_g$, i.e., $\{q_f\}$ and $(R_f)_{k\times k}$.

4 Distributed QRSVM

With increase in the quantity of data and challenges associated with it in terms of computation and storage, it has become necessary to look for distributed algorithms that can solve SVM problems efficiently. For a given low-rank approximation of a kernel matrix, QRSVM scales linearly with the training dataset size n which makes it scalable to large datasets, unlike other state-of-the-art solvers such as PSVM which suffers due to quadratic time complexity. In this section, we first review the two stages of the distributed QRSVM [11] framework. Then, we describe the memory and communication-efficient implementation with improvements over the earlier distributed implementation [11].

4.1 Stage 1: Distributed QR Decomposition

In the QRSVM framework, $\hat{A}$ is decomposed into factors Q and R. To deal with large data sizes, one partitions the data matrix $\hat{A} = [\hat{A_1}^T, \ldots, \hat{A_p}^T]^T$ across p computing cores such that each core receives $\frac{n}{p}$ rows of $\hat{A}$ (assume n is a multiple of p). The number of cores p should be less than $\frac{n}{k}$ to ensure each block has more rows than columns. It is preferable to have $p \ll \frac{n}{k}$ to maintain a tall and skinny structure on each core. We now discuss how Q and R can be generated from the partitioned matrices $\hat{A}_i, i = 1, \ldots, p$ and stored in a distributed manner similar to parallel Tall-Skinny QR (TSQR) algorithm [22].

Assume $\hat{A}_i = Q_i R_i$ is the QR decomposition of $\hat{A}_i$ in the core i. Define $A_g = [R_1^T, \ldots, R_p^T]^T$ where $A_g \in \mathbb{R}^{n\times k}$ and denote the block diagonal matrix by $diag(Q_1, \ldots, Q_p)$. Further, let $A_g = Q_g R_g$ be the QR factorization of A_g. It is easy to see that

$$\hat{A} = diag(Q_1, \ldots, Q_p)A_g = diag(Q_1, \ldots, Q_p)Q_g R_g.$$

Thus, $\hat{A} = QR$, where

$$Q = diag(Q_1, \ldots, Q_p) \times Q_g, \quad \text{and} \quad R = R_g.$$

The partitioning strategy for $\hat{A}$ guides the parallelization of the QR decomposition. Core i computes $A_i = Q_i R_i$. Core 1 is designated as the master core, which is responsible for assembling A_g and computing $A_g = Q_g R_g$. However, for the sake of implementation we focus on memory-efficient representation of A_g which is $\hat{A}_g$ and computing its QR decomposition, $\hat{A}_g = Q_f R_f$. Implementation details are provided in Section 4.3 along with the pseudocode in Algorithm 1.

Algorithm 1. Distributed QR Decomposition of $\hat{A}$

1: $k \leftarrow$ `rank`
2: **for** `each core` i **do**
3: $\quad \hat{A}_i \leftarrow$ `local partitioned data`
4: $\quad$ `Parallel Compute` $\{q_i\}, R_i \leftarrow \hat{A}_i$
5: $\quad$ `GATHER` $(R_i)_{k\times k}$ `at Master core`
6: **end for**
7: $\hat{A}_g \leftarrow$ `gathered or stacked` $(R_i)_{k\times k}$
8: `Compute` $\{q_f\}, R_f \leftarrow \hat{A}_g$ `at Master core`
9: `Use` $(R_f)_{k\times k}$

This distribution is a two-level approach where the first level involves QR factorization at each core locally and in parallel, followed by a second level which computes a single QR factorization at the master core. The approach is illustrated in Fig. 2. Alternatively, one can use a multi-level approach TSQR [22] in the form of binary reduction tree with $\log_2 p$ levels in which, at each level, pairs of R_i's from the preceding level are combined into a single matrix. QR factorizations of these matrices are computed in a distributed manner using half as many cores compared to the preceding level in the hierarchy. In principle, this scheme yields the maximum amount of parallelism and highest processor utilization, albeit at the expense of greater algorithmic complexity. We opted for the two-level implementation approach due to its simplicity as depicted in Fig. 2. As seen from the experimental results in Section 5.2, the impact of our choice of implementation on the distributed QR factorization (Fig. 4a) is negligible. Specifically, the time spent by the master core at the second level is minimal compared to the local QR computations at the first level.

The overall Q is represented in terms of local Q_i's in the worker cores ($i > 1$) and Q_g in the master core ($i = 1$). This leads to the following distributed formulation of matrix-vector multiplication involving the matrix Q.

For a vector $v \in \mathbb{R}^n$, computing Qv in the distributed QRSVM framework can be formulated as

$$Qv = \text{diag}(Q_1, \ldots, Q_p) \times (Q_g v).$$

The vector v exists as partitions across the p cores in a manner identical to $\hat{A}$. Thus, core i owns block v_i of size $(\frac{n}{p})$ elements. To compute Qv, v is assembled at the master core by gathering v_i from all the cores. Multiplication with Q_g is computed by the master core, which is followed by distribution (*scattering*) of the resulting vector across cores. Subsequently, each core applies its Q_i to the sub-vector assigned to it. Pseudocode for such computation is presented in Algorithm 3.

Multiplication with Q^T is done in a reverse flow. First, each core applies its ${Q_i}^T$ to v_i. This is followed by assembling (*gathering*) the resulting sub-vectors at the master core and finally multiplying with ${Q_g}^T$. Pseudocode for such computation is presented in Algorithm 4

$$Q^T v = {Q_g}^T \times (\text{diag}({Q_1}^T, \ldots, {Q_p}^T)v).$$

As observed from Algorithms 3 and 4, each of the above two calculations inherently requires communication (*gather* and *scatter*) across the distributed network. Again, the computation of the product between the Householder reflectors, representing Q_g (or ${Q_g}^T$) and Q_i (or ${Q_i}^T$), and the vector in these formulations are local to the respective cores in the cluster.

4.2 Stage 2: Parallel Dual Ascent

With the distributed QR decomposition technique, it is possible to calculate the dual ascent steps, Equations (6) and (7), in parallel across the computing cores.

Let us define the Hessian matrix in Equation (5) as

$$F = -\Big(R_g {R_g}^T + \frac{1}{2C} I_n\Big),$$

where we substitute, $R = R_g$. F is sparse and invertible. Due to its separable structure, it can be block-partitioned as $F = diag(F_1, \ldots, F_p)$ such that the diagonal block $F_i \in \mathbb{R}^{\frac{n}{p} \times \frac{n}{p}}$ is allocated to core i. Since $k \ll \frac{n}{p}$ due to choice of p, the dense and symmetric block $\Big((R_g {R_g}^T)_k + \frac{1}{2C} I_k\Big)$ of size $k \times k$ lies entirely within F_1 at the master core ($i = 1$). It is also worth noting that there is no need to actually partition the Hessian F at the master core and communicate it to other worker cores $i = 2, \ldots, p$. In fact, the other diagonal blocks F_i in the respective worker cores are simply the constant diagonal matrix $-(\frac{1}{2C})I_{\frac{n}{p}}$ which is never constructed, rather directly used as scalar multiplication in Equation (8).

The parallel version of the dual ascent method at iteration $(t+1)$ for any core i is given below. Here, it is assumed that vectors $\hat{\alpha}$ and $\hat{\beta}$ have been partitioned and distributed across cores, analogous to the approximation matrix $\hat{A}$.

Step 1. Minimization of Lagrangian

$$\hat{\alpha_i}^{t+1} = F_i^{-1}\big(-\hat{\beta_i}^t + \hat{e_i}\big), \tag{8}$$

where, $\hat{\beta}^t = Q^T \beta^t$ and

$${F_i}^{-1} = \begin{cases} F_1^{-1} & \text{if} \quad i = 1 \\ diag(-2C) & \text{if} \quad i = 2..p \end{cases}.$$

Step 2. Dual variable update

$$\hat{\beta_i}^{t+1} = \hat{\beta_i}^t + \eta^* (-\hat{\alpha_i}^{t+1}). \tag{9}$$

Here, the optimal step size η^* defined in [11] is used for faster convergence. The iterative updates continue until the stopping criterion, ($\|\hat{\beta_i}^{t+1} - \hat{\beta_i}^t\|_1 \leq threshold$) is met.

It is worth noting that by changing the dual variable from β to $\hat{\beta}$, the updates to $\hat{\alpha_i}^{t+1}$ and $\hat{\beta_i}^{t+1}$ occur locally (in parallel) without requiring any communication among the computing cores. However, after every iteration, the original dual variable β has to be checked for non-negativity as discussed earlier in Section 3.2. This requires transformation from $\hat{\beta}$ to $\beta = Q\hat{\beta}$, then zeroing out the negative β values and finally transforming back to $\hat{\beta} = Q^T \beta$. Such transformations from $\hat{\beta}$ to β and back to $\hat{\beta}$ require communication across the computing cores as described in Section 4.1. The pseudo-code for this stage of parallel dual ascent is given in Algorithm 2.

Algorithm 2. Parallel Dual Ascent

```
1: iteration t ← 0
2: for each core i do
3:   while error > threshold do
4:     Parallel Compute Equation (8): α̂_i^{t+1}
5:     Parallel Compute Equation (9): β̂_i^{t+1}
6:     Compute β_i ← Qβ̂ //Algorithm 3
7:     Parallel Compute β_i ← max{0, β_i}
8:     Compute β̂_i ← Q^T β //Algorithm 4
9:     error ← |β̂_i^{t+1} − β̂_i^t|
10:    t ← t + 1
11:  end while
12: end for
```

4.3 Implementation of Distributed QRSVM

The distributed implementation of QRSVM presented in [11] performs well for small and medium-sized problems than the competing algorithms, but like others it is infeasible for large datasets. The major limitations of [11] affecting its scalability to large datasets have been identified as follows

- Large memory needed to construct and store global factors at the master core, namely, Householder reflectors for Q_g and the upper trapezoidal matrix R_g.
- Significant communication overhead incurred during $Q\hat{\beta}$ and $Q^T\beta$ operations in each iteration of the parallel dual ascent.

In this section, we describe the efficient implementation for distributed QRSVM along with the improvements done over [11] to address the above limitations. These improvements are significant in both memory savings and communication efficiency thereby ensuring faster speedup and highly scalable framework for training SVM on large datasets.

(a) Distributed Implementation of QR Decomposition

We propose a novel implementation of the distributed QR decomposition technique (Stage 1) wherein memory-efficient representations of the global factors at the master core are generated. Fig. 2 illustrates the above implementation and Algorithm 1 presents the pseudo-code.

First, local QR decompositions are done in parallel across all the cores (Step 4 in Algorithm 1) using Householder transformation to generate two local factors. The first local factor is an orthogonal matrix Q_i in each of the computing cores. It is stored as a set of $k-$Householder reflectors [18], which we denote as $\{q_i\}$ of size $(n/p) \times k$. Here, each column is a reflector in the set. The second local factor of the decomposition is the upper trapezoidal matrix R_i of size $(n/p) \times k$. It comprises of $k \times k$ upper triangular block followed by zero-blocks. We denote the upper triangular block as $(R_i)_{k\times k}$. Next, as per the formulation described in Section 4.1, the local factor, R_i, from all the cores needs to be *gathered* at the master core. However, to ensure communication-efficient implementation, each of the worker cores communicates only its local upper triangular block $(R_i)_{k\times k}$ to the master core (see Step 5 in Algorithm 1) rather than sending the complete matrix R_i comprising of redundant zero-blocks. Through this strategy, we reduce the communication volume across the distributed network from $O(\frac{n}{p})$ to $O(k)$ per core. Then, at the master core, all these local upper triangular blocks $(R_i)_{k\times k}$ which have been gathered from all the computing cores p are stacked to generate $\hat{A}_g$ of size $pk \times k$ (see Step 7 in Algorithm 1). This is significantly smaller memory footprint than storing the original A_g of size $n \times k$ which would have required gathering the complete R_i with redundant zero-blocks. Finally in the master core, QR decomposition of $\hat{A}_g$ using Householder transformation generates memory-efficient global factors; a set of Householder reflectors $\{q_f\}$ and an upper trapezoidal matrix R_f, as illustrated in Fig. 2 and in Step 8 of Algorithm 1.

Memory Improvement Over [11]. In the prior implementation of the distributed QRSVM as described in [11], large memory was required to construct and store these global factors (set of Householder reflectors and upper trapezoidal matrix) at the master core. First, the set of $k-$Householder reflectors for the orthogonal matrix Q_g, denoted as $\{q_g\}$ of size $n \times k$, was constructed by appending $(\frac{n}{p} - k)$ rows of zeros to each $\{q_f\}_i$ block. In addition, the second global factor, namely, the upper trapezoidal matrix, R_g, was generated by appending $(n - k)$ rows of zeros to $(R_f)_{k\times k}$. Fig. 2 illustrates these constructions at the master core. For large sample size n, the memory consumption for $\{q_g\}$ and R_g increases linearly with n, posing a serious challenge to its performance. Such large memory requirement at the master core limits the capabilities of [11] to small and medium-sized datasets only. However, in the proposed implementation we avoid constructing these global factors, $\{q_g\}$ and R_g. Rather, we retain their memory-efficient representations, denoted as $\{q_f\}$ and $(R_f)_{k\times k}$, respectively. Fig. 2 depicts the construction of these memory-efficient representations at the master core while Algorithm 1 lists those as Steps 8-9. By retaining the global reflector set as $\{q_f\}$ of size $pk \times k$ rather than constructing $\{q_g\}$ of size $n \times k$ at the master core, we now achieve significant memory savings worth $(\frac{n}{pk})$ times compared to [11], where, $p \ll \frac{n}{k}$. Henceforth, the distributed QRSVM framework will be designed only using these memory-efficient representations, $\{q_f\}$ and $(R_f)_{k\times k}$.

(b) Parallel Implementation of Dual Ascent

As mentioned in Section 4.2, update Equations (8) and (9) are trivially parallelized across the cores (see Step 4 and Step 5 in pseudo-code in Algorithm 2). However, transformations from $\hat{\beta}$ to β and back needs to be done in every iteration to ensure the non-negativity of β for guaranteed convergence of the algorithm. Such transformations require communication across the computing cores. So, we focus on the calculation of the following two distributed matrix-vector products that involve inter-core communication processes *gather* and *scatter*

$$\beta = Q\hat{\beta} = \text{diag}(Q_1, \ldots, Q_p) \times (Q_f\hat{\beta}),$$

and

$$\hat{\beta} = Q^T\beta = {Q_f}^T \times (\text{diag}({Q_1}^T, \ldots, {Q_p}^T)\beta).$$

In each iteration of the Stage 2 (Steps 3-11 in Algorithm 2), these two multiplication modules are invoked to transform $\hat{\beta}_i$ to β_i and vice-versa, respectively (see Step 6 and Step 8 in Algorithm 2). This can lead to large communication overhead. Algorithms 3 and 4 describe how to transform $\hat{\beta}$ to β and β to $\hat{\beta}$, respectively, in an efficient manner. It is worth restating that Q_i's and Q_f are never calculated explicitly, rather, are stored as sets of $k-$Householder reflectors; as $\{q_i\}$'s in the worker cores and as $\{q_f\}$ in the master core. The implicit matrix-vector multiplication involving these reflectors is described in [20].

Algorithm 3. `Compute` $\beta_i \leftarrow Q\hat{\beta}$, `in Each Core` i

1: `GATHER` $\hat{\beta}_i$ `at Master core`
2: $\hat{\beta} \leftarrow$ `gathered` $\hat{\beta}_i$
3: `Compute` $(Q_f\hat{\beta})$ `at Master core`
4: `SCATTER` $(Q_f\hat{\beta})$ `to all cores`
5: $(Q_f\hat{\beta})_i \leftarrow$ `scattered` $(Q_f\hat{\beta})$
6: `Parallel Compute` $\beta_i \leftarrow Q_i \times (Q_f\hat{\beta})_i$

Algorithm 4. `Compute` $\hat{\beta}_i \leftarrow Q^T\beta$, `in Each Core` i

1: `Parallel Compute` $({Q_i}^T\beta_i)$
2: `GATHER` $({Q_i}^T\beta_i)$ `at Master core`
3: $(\text{diag}({Q_1}^T, \ldots, {Q_p}^T)\beta) \leftarrow$ `gathered` $({Q_i}^T\beta_i)$
4: `Compute` $\hat{\beta} \leftarrow {Q_f}^T \times (\text{diag}({Q_1}^T, \ldots, {Q_p}^T)\beta)$ `at Master`
5: `SCATTER` $\hat{\beta}$ `to all cores`
6: $\hat{\beta}_i \leftarrow$ `scattered` $\hat{\beta}$

To compute $\beta = Q\hat{\beta}$ in Algorithm 3, we first gather local $\hat{\beta}_i$ from the worker cores and calculate $(Q_f\hat{\beta})$ vector at the master core. Next, the partitioned vector $(Q_f\hat{\beta})_i$ is scattered to the cores. Core i uses its local Q_i in the form of reflector set $\{q_i\}$ that was pre-computed during the QR decomposition of the partitioned data $\hat{A}_i$ to multiply Q_i to $(Q_f\hat{\beta})_i$ in order to obtain β_i. These multiplications with Q_i proceed concurrently at each core.

To compute $\hat{\beta} = Q^T\beta$ in Algorithm 4, however, we first calculate local $({Q_i}^T\beta_i)$ in each core i. This computation can be trivially parallelized. Then, these local vectors are gathered at the master core. The gathered vector $(\text{diag}({Q_1}^T, \ldots, {Q_p}^T)\beta)$ is multiplied with ${Q_f}^T$ to generate $\hat{\beta}$ at the master core. Finally, we scatter it to all the cores such that each core gets its $\hat{\beta}_i$.

Communication Improvement Over [11]. In the prior implementation [11], significant communication overhead was incurred during $Q\hat{\beta}$ and $Q^T\beta$ operations in each iteration of the parallel dual ascent. Specifically, it required communicating large volume of data of size $(\frac{n}{p})$ per core via gather and scatter processes in the distributed network. Increasing the number of cores resulted in more interactions across the

TABLE 1
Benchmark Dataset Description

Dataset	#training samples (n)	#features (d)	k-rank approx.
covtype.binary	464,810	54	64
webspam(unigram)	350,000	254	128
SUSY	5,000,000	18	128

network. As a result, the training time became prohibitive for large datasets thereby limiting the benefits of the distributed QRSVM framework. In the proposed implementation which is based on memory-efficient representation $\{q_f\}$, it is sufficient to communicate (via gather and scatter) only the first $k-$elements of the partitioned vector, β_i or $\hat{\beta}_i$ in each of the above computation operations (Algorithms 3 and 4). This is a key step to achieve near-negligible communication overhead. As a result, in each iteration of the parallel dual ascent, the communication volume is reduced by a factor of $(\frac{n}{pk})$ per core compared to [11], without incurring any computation error. Moreover, the proposed implementation is rendered computation-bound only which can be handled efficiently by employing more number of parallel computing cores. Due to above communication improvements, we achieve large speedup in the SVM training and can handle significantly larger workloads, thereby making the framework scalable than [11].

An alternative implementation for $Q\hat{\beta}$ and $Q^T\beta$ in the parallel dual ascent method (Stage 2) can be used to avoid the frequent gathering and scattering of the $k-$sized vectors β_i and $\hat{\beta}_i$ from each of the cores. One can *scatter* the reflector set $\{q_f\} \in \mathbb{R}^{pk\times k}$ present in the master core across all the worker cores as $\{q_f\}_i$ on completion of distributed QR decomposition (Stage 1). Each core can now locally compute $(Q_{f_i}\hat{\beta}_i)$ and $({Q_{f_i}}^T\beta_i)$ via implicit multiplication described in [20]. With distributed $\{q_f\}_i$ in each core, it is possible to parallelize the implicit matrix-vector (Qv) multiplication while simply using *All_Reduce* on the locally computed $\{q_f\}_i^T\beta_i$ (scalar) values with *SUM* operation. In every iteration of the update steps, *All_Reduce* will be invoked $k-$times within every core for computing $(Q_{f_i}\hat{\beta}_i)$ and $({Q_{f_i}}^T\beta_i)$. This alternative is expected to be beneficial for sufficiently large k. The experimental evaluation with this alternate design is beyond the scope of the current work.

5 Results and Discussions

5.1 Experimental Setup

For our experiments, we use datasets which are available in LIBSVM datasets repository[1] for binary classification. Specifically, we focus on large datasets with number of training samples in hundreds of thousand and above to justify the need for a distributed and scalable SVM framework. The RBF kernel function is used for SVM training to represent the Kernel matrix. The dataset description is provided in Table 1. We also report the k-rank approximation selected for the kernel matrix computed by MEKA. The distributed QRSVM framework has been implemented in C/C++ using the Armadillo library [23] integrated with LAPACK/BLAS for linear algebra calculations. We run our parallel implementation on the *Ada* supercomputing cluster at the Texas A&M High Performance Research Computing[2] facility. The cluster consists of 792 compute nodes each equipped with two 10-core Intel Xeon E5-2670 v2 (Ivy Bridge) processors and 64 GB memory. The interconnect used is Infiniband. We use Message-Passing Interface (MPI) [24] for interprocess communication.

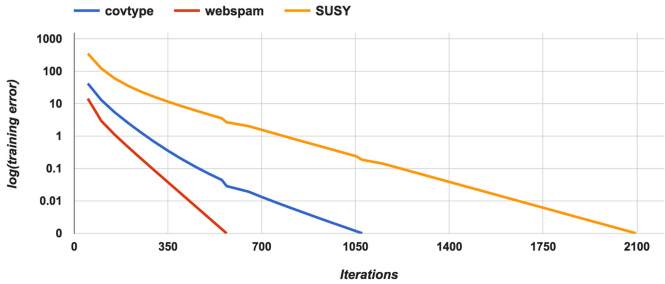


Fig. 3. Convergence rate of *distributed QRSVM* algorithm: Training error on benchmark datasets (covtype, webspam and SUSY) approaches the stopping threshold (10^{-3}) in $t_c = 1075$, $t_c = 569$, and $t_c = 2096$ iterations, respectively, using the optimal step size $\eta^* = 0.9$ [11].

TABLE 2
Distributed-QRSVM

Parameters	covtype	webspam	SUSY
C	1	1	1
γ	2^3	2^0	2^{-3}
#cores, p	16	32	64
stopping threshold	10^{-3}	10^{-3}	10^{-3}
optimal step size, η^*	0.9	0.9	0.9
#iterations, t_c	1,075	569	2,096

5.2 Discussions for Distributed QRSVM

Here, we present our discussion on the various performance measures, namely, convergence, scalability, computation time, communication time and overall training time, of the improved and efficient implementation of the distributed QRSVM framework.

Convergence. The convergence of the distributed QRSVM technique is guaranteed by the parallel dual ascent method discussed in Section 4.2. The authors in [11] have derived an optimal step size for faster convergence, which has been adopted in our experiments. Fig. 3 illustrates the convergence trend of the proposed algorithm for the three benchmarks. It exhibits a sharp decrease in the training error $\|\hat{\beta}^{t+1} - \hat{\beta}^t\|_1$ as the dual ascent algorithm converges to the optimal solution. A stopping threshold of 10^{-3} has been used for convergence. Various parameters for the selected benchmarks are shown in Table 2.

Scalability. Table 3 lists the training time and speedup achieved by distributed QRSVM on p cores, where, $p \in \{2, 4, 8, 16, 32, 64\}$. Since a processing node on *Ada* consists of two 10-core processors, we use one node for $p \in \{2, 4, 8, 16\}$, two nodes for $p = 32$, and four nodes for $p = 64$. When using more than a single node, the number of cores used on each node was identical. The only

1. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html

2. http://hprc.tamu.edu/

TABLE 3
Scalability of Distributed-QRSVM

Dataset	sequential	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
covtype	268 (1x)	132 (2x)	64 (4x)	33 (8x)	18 (15x)	10 (27x)	6 (45x)
webspam	258 (1x)	120 (2x)	58 (4x)	32 (8x)	19 (14x)	11 (23x)	9 (29x)
SUSY	28,614 (1x)	11,284 (2x)	4,405 (7x)	1,686 (17x)	804 (36x)	380 (75x)	210 (136x)

Training time (in seconds) and Speedup, S_p wrt sequential-QRSVM (S_p is shown in parenthesis).

exception to this rule is the execution of SUSY, where the number of cores was restricted to 2 per node in order to leverage caches effectively on each node for large dataset. It is found that SVM on covtype, webspam and SUSY can be trained in as fast as 6 seconds, 9 seconds, and 210 seconds, respectively, on 64 cores. This results in speedup of around 45x, 29x and 136x over sequential QRSVM implementation on a single core. Speedup (S_p) is computed as the ratio of training time (T_{train}) on p cores to that on a single core, i.e.,

$$S_p = \frac{T_{train}|_p}{T_{train}|_{p=1}},$$

where, $T_{train}|_p$ is the QRSVM training time on p cores. Since Stage 1 (distributed QR decomposition) and Stage 2 (parallel dual ascent) of the distributed technique have been parallelized efficiently, we are able to achieve very high speedups with respect to the sequential execution. The two smaller benchmarks show near-linear speedup on up to 16 cores. The speedup observed for SUSY is highly dependent on cache utilization by the cores due to large problem size. SUSY demonstrates near linear speedup for $p = \{16, 32, 64\}$ when $p = 8$ is used as the base case since the subproblem on each core fits within the local cache. However, when $p < 8$ cores are used, the subproblem size on each core is too large to fit the local cache which results in relatively larger training time. As a result, one observes super-linear speedup as we increase the number of cores when $p = 1$ (sequential implementation) is used as the base case.

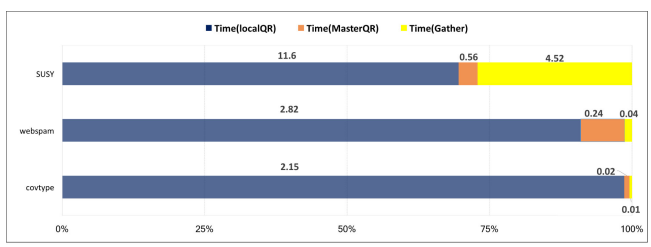
Computation Time, T_{comp}. The distributed QRSVM framework for parallel training has three major computational requirements:

1) Low-rank kernel approximation
2) Distributed QR decomposition (Stage 1)
3) Parallel Dual ascent (Stage 2)

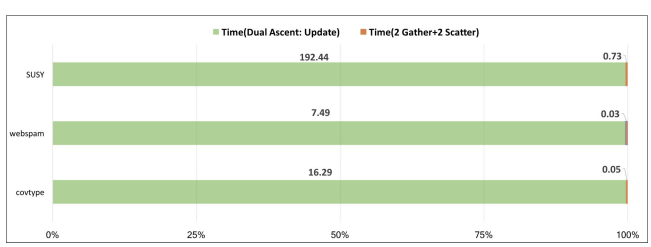
For computing the low-rank kernel approximation we use MEKA [12] which is a sequential code. The time to compute these approximations, denoted as T_{LRA}, has values 2.1 seconds, 5.93 seconds and 29.66 seconds for the benchmarks covtype, webspam and SUSY, respectively. Kernel approximation is a one-time pre-processing step prior to the other two more computationally dominant stages of the proposed framework. Therefore, we do not include T_{LRA} in the overall training time, and instead focus on the time spent on the other two stages. As a side note, it is also possible to use alternative kernel approximation techniques like [13], [14], [15], etc., but we recommend MEKA for its superior properties of memory-efficiency and low approximation error [12].

In the distributed QR decomposition stage (Algorithm 1), partitioned data $\hat{A}_i$ is decomposed locally in each core into Q_i and R_i concurrently (Step 4 in Algorithm 1). We denote its worst-case computation time as $T_{localQR}$. Then, at the master core the gathered data $\hat{A}_g$ is further factorized (see Step 8 in Algorithm 1), and its computation time is denoted as $T_{masterQR}$. Fig. 4a depicts the timing distribution of the distributed QR decomposition, where $T_{localQR} = \{2.15, 2.82, 11.6\}$ seconds and $T_{masterQR} = \{0.02, 0.24, 0.56\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. It can be seen that most of the total execution time for Stage 1 is spent on computing the local QR decomposition, $T_{localQR}$.

Finally, for each iteration in the parallel dual ascent stage, we locally compute the update steps given in Equations (8) and (9) in parallel across the cores. They are denoted as Step 4 and Step 5 of Algorithm 2, respectively. Later, we compute the transformation from $\hat{\beta}$ to β and back to $\hat{\beta}$ in Step 6 and Step 8 of Algorithm 2 which are separately tagged as Algorithms 3 and 4, respectively. Note that, Step 3 and Step 6 in Algorithm 3 and Step 1 and Step 4 in Algorithm 4 are the computation steps in the above transformations. We combine the above mentioned computation times for all the iterations of the parallel dual ascent until convergence and denote as T_{update}. Fig. 4b presents the total execution time for the Stage 2 which includes both the computation and the communication time. Here, the computation time is $T_{update} = \{16.29, 7.49, 192.44\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. It is observed that



(a) Stage 1: Distributed QR decomposition



(b) Stage 2: Parallel Dual ascent

Fig. 4. Timing (in seconds) analysis for computation and communication in different stages of distributed QRSVM. (a) Stage 1: $T_{localQR}$ and $T_{masterQR}$ are the computation time, whereas T_{gather} is the communication time. (b) Stage 2: T_{update} denotes the computation time for all iterations of parallel dual ascent, while T_{2g+2s} refers to the communication time spent in transformations $\hat{\beta}$ to β. Datasets: Covtype (p = 16), webspam (p = 32), SUSY (p = 64).

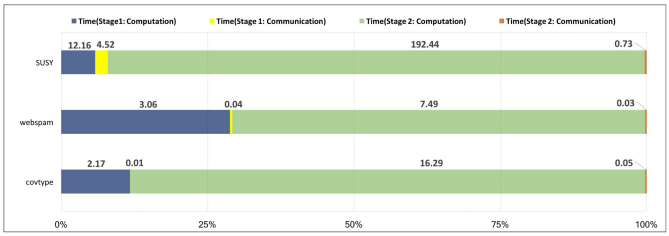


Fig. 5. Overall training time, T_{train} analysis (in seconds) for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$). A majority of the training time is spent in iterative computations in Stage 2 (parallel dual ascent), while communication overhead is negligible.

more than 99 percent of the execution time of parallel dual ascent (Stage 2) is spent on computation. Hence, the proposed algorithm is amenable for full parallelism by incorporating more cores.

Communication Time, T_{comm}. The distributed QRSVM framework has two necessary communication requirements. The first communication occurs during Stage 1, i.e., distributed QR decomposition, when local $(R_i)_{k\times k}$ are gathered at the master core (Step 5 in Algorithm 1). This is also depicted in Fig. 2. Let us denote this communication time as $T_{gatherR}$. As seen in Fig. 4a the communication overhead, $T_{gatherR} = \{0.01, 0.04\}$ seconds hardly impacts the Stage 1 execution time for benchmarks covtype and webspam on $p = 16$ and $p = 32$ cores, respectively. It is due to minimal inter-node communication where the number of *Ada* nodes required are 1 and 2, respectively based on our experimental setup. In the case of larger benchmark SUSY, the inter-node communication is higher as 32 *Ada* nodes are used. Here, the communication requirement in Stage 1 is $T_{gatherR} = 4.52$ seconds on $p = 64$ cores, which is around 27 percent of the execution time of Stage 1. However, for such large datasets requiring more number of nodes, the increase in inter-node communication time for Stage 1 is largely compensated by the computation time for Stage 2, as seen in Fig. 5.

The second communication occurs during each iteration of the parallel dual ascent method (Step 6 and Step 8 in Algorithm 2). There are two gather and two scatter processes in each iteration as discussed in Sections 4.2 and 4.3. These communication processes, gather and scatter, first occur in Step 1 and Step 4 of Algorithm 3, respectively and again in Step 2 and Step 5 of Algorithm 4, respectively. Let T_{2g+2s} denote the time involved for gathering (g) and scattering (s) twice during all the iterations of the parallel dual ascent until convergence. From Fig. 4b, we observe that $T_{2g+2s} = \{0.05, 0.03, 0.73\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively, has insignificant impact ($< 1\%$) on execution time of Stage 2 in comparison to its computation time, T_{update}.

Training Time, T_{train}. The overall training time, denoted as $T_{train} = T_{comp} + T_{comm}$, for the distributed QRSVM algorithm is observed to be $T_{train} = \{18.52, 10.62, 209.85\}$ seconds on benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. In the proposed framework, T_{train} comprises of computation time, i.e., $T_{comp} = T_{localQR} + T_{masterQR} + T_{update} = \{18.46, 10.55, 204.60\}$ seconds, and communication time, i.e., $T_{comm} = T_{gatherR} + T_{2g+2s} = \{0.06, 0.07, 5.25\}$ seconds. Fig. 5 depicts the analysis of T_{train} where the time for communication is around 0.3, 0.6 and 2.5 percent of the total training time for benchmarks covtype, webspam and SUSY respectively. Moreover, in Fig. 5, it is also observed that parallel dual ascent (Stage 2) in the QRSVM algorithm is more computationally dominant stage than the distributed QR decomposition (Stage 1). On Ivybridge processor, parallel dual ascent achieves around 700 MFLOP per second per core for the chosen benchmarks. These results validate our communication-efficient implementation of distributed QRSVM where negligible amount of SVM training time is spent in communicating data across the network. Hence, our framework is a step towards distributed training on the edge for applications in IoT without the need to transfer all the data to a central server for training. In essence, our framework offers decentralization of SVM training with guaranteed convergence.

TABLE 4
Comparing *dis*-QRSVM with PSVM, P-packSVM and [11] on T_{train} (in Seconds) for *Covtype* Dataset

Algorithm	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
PSVM	8,562	4,396	2,352	1,270	635	341
P-packSVM	-	-	2,019	1,022	295	110
dis-QRSVM [11]	390	309	271	261	256	454
dis-QRSVM	**132**	**64**	**33**	**18**	**10**	**6**

- data unavailable

5.3 Comparison with Other Parallel Methods

In Table 4, we compare the training time of the proposed framework with state-of-the-art parallel solvers, namely, PSVM [8] and P-packSVM [9] on covtype benchmark. Moreover, we also evaluate the performance improvement over the prior implementation of distributed QRSVM framework [11]. It is observed that for a given benchmark, the proposed implementation of the distributed QRSVM trains much faster than PSVM, P-packSVM and prior implementation [11]. Both PSVM and the distributed QRSVM optimize the dual form of the SVM cost function, making our comparison fair. We have used the default setting for PSVM parameters as reported in [8]. In our experiments, the dual residual threshold for convergence was set to 10^{-3}.

PSVM is the foremost parallel SVM solver available as open source. It employs parallel incomplete cholesky factorization (PICF) to approximate $n \times n$ kernel matrix with a $k-$rank representation. PSVM then performs parallel Interior-Point Method to solve the quadratic optimization problem. As per [8], the computational complexity of PSVM is $O(nk^2/p)$. To ensure good accuracy, the authors in PSVM [8] recommend the rank of the ICF of kernel matrix as $k = \sqrt{n}$, where n represents the number of training samples. As a result, the cost of executing PSVM becomes quadratic in n which results in large training time and hinders its scalability to large datasets.

In contrast to PSVM, the proposed distributed QRSVM framework builds on the current state-of-the-art kernel approximation, MEKA [12], which is both memory-efficient and has the least kernel approximation error amongst various approximation methods such as ICF. Since the accuracy of our framework is directly related to the quality of kernel approximation, we can safely argue that accuracy of our MEKA-based distributed QRSVM will be at least at par or

TABLE 5
Comparing Proposed *dis*-QRSVM with [11] on Stage 2 Computation Time, T_{update} and Communication Time, T_{2g+2s} (in Seconds) for *Covtype* Dataset

Time	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
T_{update}[11]	379	304	269	259	252	448
T_{update}(proposed)	**122**	**59**	**30**	**16**	**8**	**5**
T_{2g+2s}[11]	1.63	1.81	0.34	0.05	1.19	3.02
T_{2g+2s}(proposed)	**0.02**	**0.02**	**0.14**	**0.05**	**0.03**	**0.11**

better than the traditional ICF-based PSVM. In the light of above, the k-rank approximation of the kernel matrix is chosen empirically through MEKA [12]. We ensure rank $k \ll n$ as discussed in Section 2.1. Since the dominant computations in both stages of the distributed QRSVM framework can be fully parallelized, the time complexity of the proposed framework is $O(nk^2/p + nkt_c/p)$. With the above choice of $k \ll n$, the computational cost for the distributed QRSVM is linear in number of samples n unlike PSVM which shows quadratic behavior. Therefore, the proposed framework converges faster and is more scalable than PSVM. For instance, it can be observed from Table 4 that on $p = \{2, 4, 8, 16, 32, 64\}$ cores for covtype benchmark, PSVM takes $T_{train} = \{8562, 4396, 2352, 1270, 635, 341\}$ seconds while the proposed distributed QRSVM framework trains in $T_{train} = \{132, 64, 33, 18, 10, 6\}$ seconds with performance improvement of $\{65\text{x}, 69\text{x}, 71\text{x}, 71\text{x}, 63\text{x}, 57\text{x}\}$.

Due to lack of availability of P-packSVM code as open source, we estimate its results based on its training time ratio with respect to PSVM obtained from [9]. We report the training performance of P-packSVM on $p = \{8, 16, 32, 64\}$ cores since the base number of cores used in [9] is 8. As observed in Table 4, our implementation performs $\{61\text{x}, 57\text{x}, 30\text{x}, 18\text{x}\}$ faster than P-packSVM for training covtype on $p = \{8, 16, 32, 64\}$ cores, respectively.

We also compare and evaluate the performance of the proposed implementation with our prior work [11]. Section 4.3 discusses the communication improvement over [11] during the parallel implementation of the dominant stage of iterative dual ascent (Stage 2). Hence, we compare Stage 2 computation time, T_{update} and communication time, T_{2g+2s} of both the implementations in Table 5. It can be empirically observed that the proposed implementation significantly improves over [11] in computation of Stage 2 with relatively lower communication overhead, leading to faster overall training time. From Table 4, we achieve performance improvement of $\{3\text{x}, 5\text{x}, 8\text{x}, 14\text{x}, 25\text{x}, 75\text{x}\}$ in training time compared to [11] on $p = \{2, 4, 8, 16, 32, 64\}$ cores, respectively. While [11] could train datasets just as large as covtype with $n = 464,810$ samples, the proposed implementation is demonstrated to work for same and even larger datasets such as SUSY with $n = 5$M samples. Moreover, the results in Table 5 also validate that [11] was unable to scale on large number of cores while the proposed approach exhibits better parallel speedup, owing to both memory and communication-efficient implementation.

6 Conclusion

We propose a fast and efficient implementation to improve the distributed QR decomposition framework [11] for training kernel support vector machines. The framework comprises of two stages: distributed QR decomposition and parallel dual ascent, to accelerate the training. Householder transformation is used to convert a dense and low-rank approximation of the kernel matrix into a highly sparse and separable structure, with easily invertible diagonal blocks. Moreover, we efficiently implement these two stages of the framework to achieve significant memory and communication benefits over [11]. We empirically demonstrate that the proposed distributed implementation of the framework achieves considerable speedup over its sequential counterpart. In addition, it substantially reduces the communication overhead to a negligible fraction of the total training time. These characteristics make the framework suitable to handle large data problems while being scalable across large number of computing cores, unlike [11]. We also achieve performance benefit on training time compared to state-of-the-art parallel SVM solvers. As a future work, extension of this work to incremental SVMs for real-time training can be explored. We are also designing efficient hardware accelerator based on distributed QRSVM for applications in edge computing and smart embedded systems.

Acknowledgments

This research was conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

References

[1] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining Knowl. Discovery*, vol. 2, pp. 121–167, Jun. 1998.

[2] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 408–415.

[3] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," p. 21, Apr. 1998, https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fastalgorithm-for-training-support-vector-machines/

[4] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in Kernel Methods*. Cambridge, MA, USA: MIT Press, 1999, pp. 169–184.

[5] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, pp. 27:1–27:27, 2011. Software. [Online]. Available: http://www.csie.ntu.edu.tw/ cjlin/libsvm

[6] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade SVM," in *Proc. 17th Int. Conf. Neural Inf. Process. Syst.*, 2004, pp. 521–528.

[7] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "CA-SVM: Communication-avoiding support vector machines on distributed systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 847–859.

[8] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "PSVM: Parallelizing support vector machines on distributed computers," in *Proc. 20th Int. Conf. Neural Inf. Process. Syst.*, 2007, pp. 257–264. Software. [Online]. Available: http://code.google.com/p/psvm

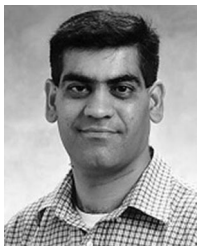
[9] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen, "P-packSVM: Parallel primal gradient descent kernel SVM," in *Proc. 9th IEEE Int. Conf. Data Mining*, Dec. 2009, pp. 677–686.

[10] D. Hush, P. Kelly, C. Scovel, and I. Steinwart, "QP algorithms with guaranteed accuracy and run time for support vector machines," *J. Mach. Learn. Res.*, vol. 7, no. May, pp. 733–769, 2006.

[11] J. Dass, V. N. S. P. Sakuru, V. Sarin, and R. N. Mahapatra, "Distributed QR decomposition framework for training support vector machines," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 753–763, doi: 10.1109/ICDCS.2017.222.

[12] S. Si, C.-J. Hsieh, and I. S. Dhillon, "Memory efficient kernel approximation," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 701–709.
[13] S. Fine and K. Scheinberg, "Efficient SVM training using low-rank kernel representations," *J. Mach. Learn. Res.*, vol. 2, pp. 243–264, Mar. 2002.
[14] A. J. Smola and B. Schökopf, "Sparse greedy matrix approximation for machine learning," in *Proc. 17th Int. Conf. Mach. Learn.*, 2000, pp. 911–918.
[15] P. Drineas and M. W. Mahoney, "On the Nyström method for approximating a gram matrix for improved kernel-based learning," *J. Mach. Learn. Res.*, vol. 6, pp. 2153–2175, Dec. 2005.
[16] O. Bousquet and A. Elisseeff, "Stability and generalization," *J. Mach. Learn. Res.*, vol. 2, no. Mar., pp. 499–526, 2002.
[17] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends® Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2011.
[18] A. S. Householder, "Unitary triangularization of a nonsymmetric matrix," *J. ACM*, vol. 5, pp. 339–342, Oct. 1958.
[19] W. Gander, "Algorithms for the QR decomposition," *Res. Rep.*, vol. 80, no. 02, pp. 1251–1268, 1980.
[20] S. Johnson, "18.335J - Introduction to Numerical Methods, Fall 2010, " *Massachusetts Institute of Technology: MIT OpenCourseWare*, License: Creative Commons BY-NC-SA, 2010, https://ocw.mit.edu
[21] C.-J. Hsieh, S. Si, and I. S. Dhillon, "Fast prediction for large-scale kernel machines," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 3689–3697.
[22] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. A206–A239, 2012.
[23] C. Sanderson, et al., "Armadillo: a template-based C++ library for linear algebra," *J. Open Source Softw.*, vol. 1, no. 2, p. 26, 2016, doi: 10.21105/joss.00026.
[24] M. P. Forum, "MPI: A message-passing interface standard," University of Tennessee, Knoxville, TN, USA, 1994, http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Autk_cs%3Ancstrl.utk_cs%2F%2FUT-CS-94-230

Jyotikrishna Dass received the BTech degree in electronics and communication engineering with minor in computer science and engineering from the Indian Institute of Technology (IIT) Guwahati, India, in 2014. He is currently working toward the PhD degree advised by Prof. Rabi N. Mahapatra in Embedded Systems and CoDesign Lab, Department of Computer Science and Engineering, Texas A&M University, College Station. His research is in devising efficient machine learning algorithms and hardware accelerators for edge analytics in clustered embedded devices. He has published papers in IPDPS'16, ICDCS'17 and HiPC'17. He received Teaching Assistant Excellence award in 2018. He is a student member of the IEEE Computer Society and IEEE Big Data Community. He is a member of the IEEE.



Vivek Sarin received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, India, in 1990, the MS degree in computer science from the University of Minnesota, Minneapolis, in 1993, and the PhD degree in computer science from the University of Illinois at UrbanaChampaign, Urbana, in 1997. He is an associate professor with the Department of Computer Science, Texas A&M University, College Station. His research interests include numerical methods, parallel algorithms, and their applications. He is a member of the IEEE.

Rabi N. Mahapatra is a professor with the Department of Computer Science and Engineering, Texas A&M University. He was a faculty with the Indian Institute of Technology, Kharagpur, India and a faculty fellow with IBM T.J Watson Research Center. His principal areas of research are embedded systems, system on chip, low-power system design, and data analytic accelerators. He directs the Embedded System Codesign Research Group, Texas A&M University. He has been in the editorial boards of the *ACM Transactions on Embedded Computing*, the *IEEE Transactions on Parallel Distributed Systems*, and the *EUROSIP Journal on Embedded Systems*. He has published three books and more than 160 research articles in the refereed International Journals and Conference Proceedings. He is a ford fellow, BOYS-CAST fellow, senior-member of the IEEE Computer Society, and was a distinguished visitor of the IEEE Computer Society. He received Undergraduate Teaching Excellence award in 2010.