Parallelizing Support Vector Machine Training Using GPU and MPI

Bobo Bai

December 8, 2025

Abstract

Support Vector Machines (SVMs) remain one of the most accurate and robust methods for supervised classification. However, classical training procedures suffer from high computational cost, especially with sequential minimal optimization where training complexity grows quadratically or cubically with the number of samples. This project investigates parallelization techniques for accelerating SVM training using (1) GPU parallelization of the Sequential Minimal Optimization (SMO) solver, and (2) distributed-memory parallelization via MPI using both the classical Cascade SVM architecture and a modified, more efficient two-layer Cascade SVM. Experiments on the MNIST dataset demonstrate substantial speedups. The GPU implementation achieved a $\approx 56\times$ acceleration relative to the serial solver, while the modified two-layer Cascade SVM achieved up to a $10.9\times$ speedup on 64 processes and consistently outperformed the classical Cascade SVM across all tested process counts. All methods produced identical accuracy (99.69%) and identical support-vector counts (1548), confirming the correctness of parallelism.

Contents

1 Introduction

Support Vector Machines (SVMs) have been widely used in machine learning, pattern recognition, computer vision, and scientific classification tasks. Despite the rise of deep neural networks, SVMs continue to provide strong performance on medium-scale problems, and they offer attractive theoretical properties such as margin maximization and well-understood generalization bounds [1].

The main drawback of nonlinear kernel SVMs is the computational burden of training. The optimization problem is convex but involves all pairwise kernel evaluations between training points. This results in at least quadratic time and memory complexity in the number of samples. The Sequential Minimal Optimization (SMO) algorithm introduced by Platt [2] reduces training to a sequence of very small quadratic subproblems, but repeated kernel evaluations and global reduction steps still dominate runtime on large datasets.

This project explores two complementary directions for speeding up SVM training. First, I use GPU parallelism to accelerate the heavy linear algebra and kernel computations in the SMO inner loop. Second, I use MPI to distribute data across multiple processes and combines local models using Cascade SVM ideas [3]. A modified two-layer Cascade SVM design is also proposed to reduce communication and redundant retraining. All methods are evaluated on the MNIST dataset, and performance is reported in terms of training time, speedup, and classification accuracy.

2 Support Vector Machines

2.1 Primal and Dual Formulations

Given training data (x_i, y_i) where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$, the soft-margin SVM in primal form is

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(w^\top \phi(x_i) - b) \geq 1 - \xi_i, \quad i = 1, \dots, n, \\ & \xi_i \geq 0, \quad i = 1, \dots, n. \end{aligned} \tag{1}$$

Introducing Lagrange multipliers and eliminating primal variables yields the dual formulation:

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(x_i, x_j) + \sum_{i=1}^{n}\alpha_i \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, n, \\ & \sum_{i=1}^{n}\alpha_i y_i = 0, \end{aligned} \tag{2}$$

where $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$ is the kernel function.

The decision function for a new point x is

$$\operatorname{sign}\left(\sum_{i=1}^{n}\alpha_i y_i K(x, x_i) - b\right). \tag{3}$$

In this project the Gaussian RBF kernel

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

is used, with $C = 10$ and $\gamma = 0.00125$ for MNIST.

The formulation is guaranteed to have a global maximum, and the prediction model in (3) depends only on support vectors with $\alpha_i > 0$.

2.2 Sequential Minimal Optimization

The SMO algorithm [2] decomposes the dual SVM problem into a sequence of two-variable subproblems. At each iteration, it selects a pair of indices $(i_{\text{high}}, i_{\text{low}})$ based on violations of the Karush–Kuhn–Tucker (KKT) conditions and updates the corresponding Lagrange multipliers while keeping all others fixed.

For each sample x_i, define the violation measure

$$f_i = \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) - y_i, \tag{4}$$

Following the first-order heuristic of Keerthi et al. [4], the two working-set indices are chosen as

$$i_{\text{high}} = \arg\min\{f_i : i \in I_{\text{high}}\}, \tag{5}$$

$$i_{\text{low}} = \arg\max\{f_i : i \in I_{\text{low}}\}, \tag{6}$$

where the feasible index sets are

$$I_{\text{high}} = \{i : \alpha_i < C,\, y_i = 1\} \cup \{i : \alpha_i > 0,\, y_i = -1\}, \tag{7}$$

$$I_{\text{low}} = \{i : \alpha_i > 0,\, y_i = 1\} \cup \{i : \alpha_i < C,\, y_i = -1\}. \tag{8}$$

Because the prevision limit of computer, these index sets are computed to within a tolerance $\epsilon = 10^{-12}$, e.g., $\{i : \alpha_i < C - \epsilon\}$ and $\{i : \alpha_i > \epsilon\}$.

Once $(i_{\text{high}}, i_{\text{low}})$ are selected, SMO optimizes only the two corresponding multipliers while enforcing the equality constraint $\sum_i \alpha_i y_i = 0$. Letting $i = i_{\text{high}}$ and $j = i_{\text{low}}$ for simplicity, the curvature of the resulting one-dimensional subproblem is

$$\eta = K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j). \tag{9}$$

The unconstrained update for α_j is expressed in terms of the threshold candidates $b_{\text{high}} := f_{i_{\text{high}}}$ and $b_{\text{low}} := f_{i_{\text{low}}}$:

$$\alpha_j^{\text{new}} \;=\; \alpha_j + y_j \frac{b_{\text{high}} - b_{\text{low}}}{\eta}. \tag{10}$$

This value is then clipped to the feasible interval

$$\alpha_j^{\text{new}} \in [U,\, V],$$

where U and V are determined by the box constraints $0 \le \alpha_i, \alpha_j \le C$ and the equality constraint linking α_i and α_j.

After α_j is updated, the corresponding value of α_i^{new} is determined as

$$\alpha_i^{\text{new}} \;=\; \alpha_i + y_i y_j(\alpha_j - \alpha_j^{\text{new}}). \tag{11}$$

Two threshold candidates, b_{high} and b_{low}, are computed from the KKT conditions of the updated pair. Then the error vector f is incrementally updated using the changes in α_i and α_j. The program ends when $b_{\text{low}} \le b_{\text{high}} + 2\tau$, where τ is set to be 10^{-5}. The final threshold is taken as

$$b = \frac{b_{\text{high}} + b_{\text{low}}}{2}. \tag{12}$$

The serial implementation in this project follows this SMO procedure but computes kernel rows on demand and maintains an incremental error vector to avoid storing the full $n \times n$ kernel matrix.

3 Dataset and serial implementation

The MNIST dataset consists of 60,000 training images and 10,000 test images, each represented as a 784-dimensional vector of grayscale pixel intensities. Because MNIST is a ten-class dataset, but the SVM formulation used here is binary, a one-vs-rest strategy was used: a chosen digit "1" is assigned the positive label $+1$, and all remaining digits are grouped together and assigned the negative label -1. This produces a balanced and well-defined two-class classification problem suitable for SMO-based training.

Before optimization begins, all feature dimensions are scaled independently to the interval $[0, 1]$ using feature-wise minima and maxima computed from the training set. The SMO loop then proceeds by repeatedly selecting i_{high} and i_{low} according to the first-order working set heuristic, computing the corresponding kernel rows on demand, updating the two associated multipliers, and incrementally updating the error vector f. This continues until the duality gap falls below a tolerance or a maximum iteration limit is reached.

After training, only support vectors are retained for prediction. The final prediction phase evaluates the decision function on the test set using the learned multipliers, labels, kernel function, and bias term b, and the overall classification accuracy is computed. The serial SMO implementation is summarized in Algorithm 1.

Thus, the serial implementation is composed of three main components: data-reading and feature scaling, training by SMO, and prediction. Of all the processes, the SMO implementation is the most time-consuming part. This project will mainly focis on parallelizing the SMO algorithm. The parallelization is not trivial due to dependencies between the computation steps.

Algorithm 1 Serial SMO Training Procedure (Based on Implementation)

```
1: **Input:** training data $X$, labels $Y$, penalty $C$, kernel $K(\cdot,\cdot)$
2: Initialize $\alpha_i = 0$ for all $i$, set $b = 0$, and set $f_i = -y_i$
3: **while** $b_{\text{low}} > b_{\text{high}} + 2\tau$ **do**
4:    Select $i_{\text{high}}$ as the index in the active set with minimal $f_i$
5:    Select $i_{\text{low}}$ as the index in the active set with maximal $f_i$
6:    **if** $i_{\text{high}}$ or $i_{\text{low}}$ does not exist **then**
7:       break
8:    **end if**
9:    Compute kernel rows $K(x_{i_{\text{high}}}, x_j)$ and $K(x_{i_{\text{low}}}, x_j)$ for all $j$
10:   Compute $\eta = K_{ii} + K_{jj} - 2K_{ij}$
11:   Compute unconstrained update for $\alpha_{i_{\text{low}}}$ and clip it to $[U, V]$
12:   Update $\alpha_{i_{\text{high}}}$ from the equality constraint
13:   Compute $b_{\text{high}}$ and $b_{\text{low}}$ and set $b = (b_{\text{high}} + b_{\text{low}})/2$
14:   **for** each training point $j$ **do**
15:      Update
            $f_j \leftarrow f_j + (\Delta\alpha_{i_{\text{high}}}) y_{i_{\text{high}}} K(x_{i_{\text{high}}}, x_j) + (\Delta\alpha_{i_{\text{low}}}) y_{i_{\text{low}}} K(x_{i_{\text{low}}}, x_j)$
16:   **end for**
17: **end while**
18: **Return:** multipliers $\alpha$ and bias $b$
```

4 GPU Implementation

Many components of the serial SMO-based SVM can be parallelized on a GPU. The main parts for parallelism include:

- Prior to traning, finding the maximum and minimum values of each feature can be done in parallel. This takes $O(n)$ time and is non-trivial.
- Once the min and max values of each features were found, we can scale each x_i in parallel. This is trivial.
- In the SMO algorithm, finding i_{high} and i_{low} needs traversing f_i for $i \in I_{\text{high}}$ and $i \in I_{\text{low}}$, respectively. This takes $O(n)$ time and is the most tricky part to parallelize.
- In the SMO algorithm, the calculation of kernal rows and updating f_i can be done independently. Both take $O(n)$ time per iteration and are trivial.
- Finally, making prediction for each test sample can be done trivially.

Among all components, the only two parts that are not trivially parallel are (i) computing the global feature-wise minima and maxima, and (ii) selecting the SMO working-set indices i_{high} and i_{low}. Both tasks involve comparisons and reductions over non-contiguous subsets of indices, which makes them more difficult to parallelize efficiently.

Here I focus on the more difficult problem: parallelizing the selection of i_{high} and i_{low}. Although much of the GPU implementation is trivial, this component represents the main algorithmic challenge and is therefore discussed in detail.

4.1 Parallel Reduction for i_{high} and i_{low}

This subproblem is restated as given y_i f_i, α_i, how to find i_{high} and i_{low} in parallel using (5), (6), (7) and (8). It is time-consuming because it takes $O(n)$ time in each single iteration. And the parallelization of this subproblem is nontrivial since the indices in I_{high} and I_{low} are not continuous, making parallel reduction using GPU challenging. To solve this issue, I defined two auxiliary device arrays $f_i^{\text{in\_I\_high}}$ and $f_i^{\text{in\_I\_low}}$, which defined as

$$f_i^{\text{in\_I\_high}} = \begin{cases} f_i & \text{if } i \in I_{\text{high}} \\ \infty & \text{otherwise} \end{cases} \tag{13}$$

and

$$f_i^{\text{in\_I\_low}} = \begin{cases} f_i & \text{if } i \in I_{\text{low}} \\ -\infty & \text{otherwise} \end{cases} \tag{14}$$

Then it is obvious that

$$\arg\min\{f_i^{\text{in\_I\_high}} : \forall i\} = \arg\min\{f_i : i \in I_{\text{high}}\} = i_{\text{high}}, \tag{15}$$

and

$$\arg\min\{f_i^{\text{in\_I\_low}} : \forall i\} = \arg\min\{f_i : i \in I_{\text{low}}\} = i_{\text{low}}, \tag{16}$$

In this case, the search scope for both i_{high} and i_{low} become continuous. The calculation of $f_i^{\text{in\_I\_high}}$ and $f_i^{\text{in\_I\_low}}$ can be done independently. Below I illustrate how to find i_{high}. Finding i_{low} is pretty similiar and is not illustrated here.

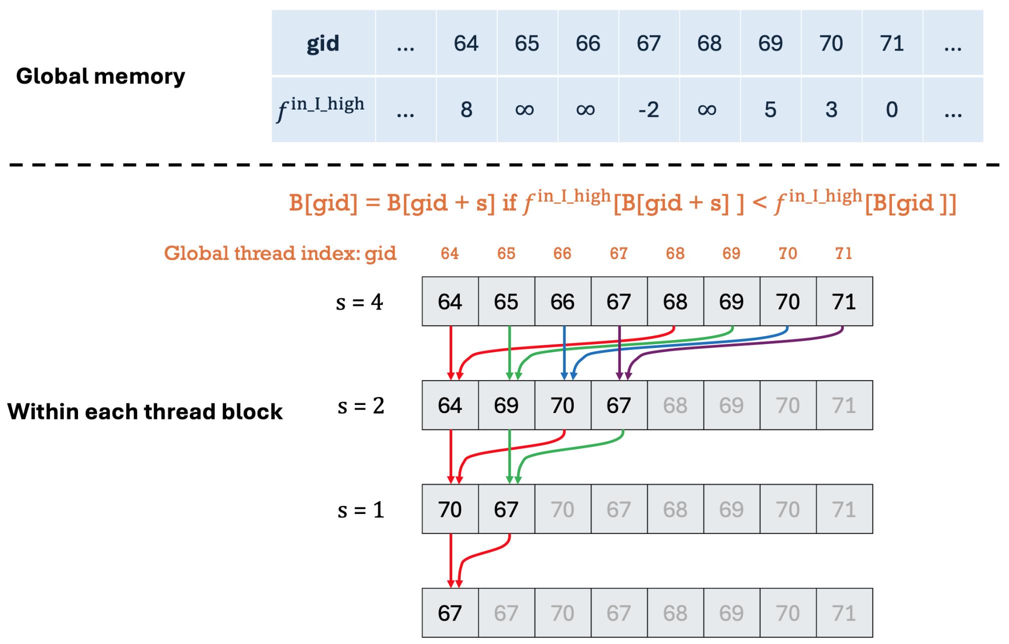
For i_{high}, the reduction kernel operates directly on an array of global indices rather than on the values of $f^{\text{in\_I\_high}}$ themselves. Each thread is assigned a global index `gid`, which points to an entry in the array $f^{\text{in\_I\_high}}$ stored in global memory. During the reduction, threads do not copy values into shared memory. Instead, they repeatedly compare their current candidate index with another index at a fixed offset and update their index if a better (i.e., smaller) objective value is found.

At the beginning of the reduction, the array `B` was assigned global indices. At stride s, thread `gid` compares the two values

$$f^{\text{in\_I\_high}}[\texttt{B[gid]}] \quad \text{and} \quad f^{\text{in\_I\_high}}[\texttt{B[gid + s]}],$$

and if the second is smaller, thread `gid` replaces its stored index `B[gid]` with `B[gid + s]`. Because each thread always carries an index rather than a raw value, only the index array `B` is updated. The actual comparisons always read the objective values from global memory using those indices.

This process is illustrated in Figure 1. For clarity, the figure shows a block of eight consecutive indices and three reduction steps corresponding to strides $s = 4, 2$, and 1. At each step, half of the threads become inactive, and the remaining threads update their indices based on the comparison. After $\log_2(\texttt{blockDim.x})$ stages, thread 0 holds the index of the best candidate within that block. After that, a new kernel reduces the remaining candidates in the same way, continuing until only one index is left. This final index is the global minimizer over all valid entries of $f^{\text{in\_I\_high}}$.



gid	...	64	65	66	67	68	69	70	71	...
$f^{\text{in\_I\_high}}$	...	8	∞	∞	-2	∞	5	3	0	...

Figure 1: Illustration of an in-block parallel reduction with block size 8. At each stage, the range of active threads halves until only a single value per block remains.

5 MPI Parallelization: Cascade SVM

The MPI implementation adopts a coarse-grain parallelization strategy. Instead of accelerating a single-node solver, it partitions the training set across multiple MPI processes (ranks) and coordinates them to produce a single global SVM model.

Rank 0 reads the full training dataset, assigns each sample a unique global ID, partitions the data into approximately equal chunks, and sends one chunk to each rank. It then computes global feature-wise minima and maxima on the full dataset and broadcasts them so that every rank can scale its local partition to $[0, 1]$ in a consistent way.

The distributed training procedure is inspired by the Cascade SVM framework of [3]. In Cascade SVM, each process first trains an SVM on its local subset of data, extracts support vectors, and then participates in a hierarchy of merges where only support vectors are propagated downward. The idea is that non-support points can be safely discarded without changing the final decision function. In this project, I implemented two variants:

- a classical Cascade SVM that uses a binary reduction tree over MPI ranks, and
- a modified two-layer Cascade SVM in which all workers send support vectors directly to rank 0 in each round.

Both variants are iterative. They perform multiple cascade rounds and stop only when the set of global support vector IDs on rank 0 no longer changes between consecutive rounds.

5.1 Implementation of Classical Cascade SVM

The classical Cascade SVM follows a divide-and-conquer strategy. The number of MPI processes P is required to be a power of two, which allows a binary reduction tree to form naturally from the rank indices. A schematic example for $P = 8$ is shown in Figure 2, and the full pseudocode is given in Algorithm 2. The original paper [3] does not provide implementation details, hence the version presented here reflects my own design choices and modifications.

At the beginning of round 1, each rank holds only its own data partition $(x_{\text{part}}, y_{\text{part}})$ together with the corresponding global IDs (id_{part}). Rank 0 begins with an empty global support-vector set. In every subsequent round, rank 0 broadcasts its current global support vectors (features, labels, alphas, and IDs) to all ranks. These broadcasted vectors represent the current global model and serve as warm-start information for all processes.

After receiving the global support vectors, each rank constructs its local training set as the union of the broadcast SVs and its local partition. The main difficulty in this merging step is eliminating duplicates, since a support vector may appear both in the local partition and in the global set. To handle this safely, each process performs deduplication by global ID using an `unordered_set`. A second challenge lies in initializing the α values. Initializing all α's to zero discards useful information and significantly slows convergence. Instead, the implementation warm-starts the alphas of previously known support vectors using the broadcast values, while newly added local samples receive $\alpha = 0$. Each rank then trains the serial SMO solver, which recomputes the error vector and runs until convergence on this augmented dataset. After convergence, the rank extracts its current support vectors ($\alpha_i > \text{tol}$) together with their features, labels, and global IDs.

Within each round, these locally extracted support vectors are merged following a tree-structured reduction over ranks. The code implements merges at step sizes $1, 2, 4, \ldots, P/2$. At a given step size s, ranks with indices congruent to $0 \pmod{2s}$ are receivers, and ranks congruent to $s \pmod{2s}$ are senders. Each sender transmits its support-vector set to its corresponding receiver and becomes inactive for the remainder of the round. The receiver deduplicates the incoming vectors by ID, initializes any newly arrived samples with $\alpha = 0$ if needed, and retrains the SMO solver on the merged set to obtain a new, reduced support-vector set. This reduction pattern continues with doubling step sizes until only rank 0 remains active and holds the final support-vector set for that round.

At the end of each round, rank 0 compares the global support-vector IDs from the current round with those from the previous round. If the two sets are identical, the algorithm has converged: the support vectors and the associated bias b define the final global model. Rank 0 then evaluates this model on the test set (scaled using the same global minima and maxima) and reports the test accuracy, final number of support vectors, and training/prediction times. If the ID sets differ, rank 0 broadcasts the updated global support vectors and a new round begins.

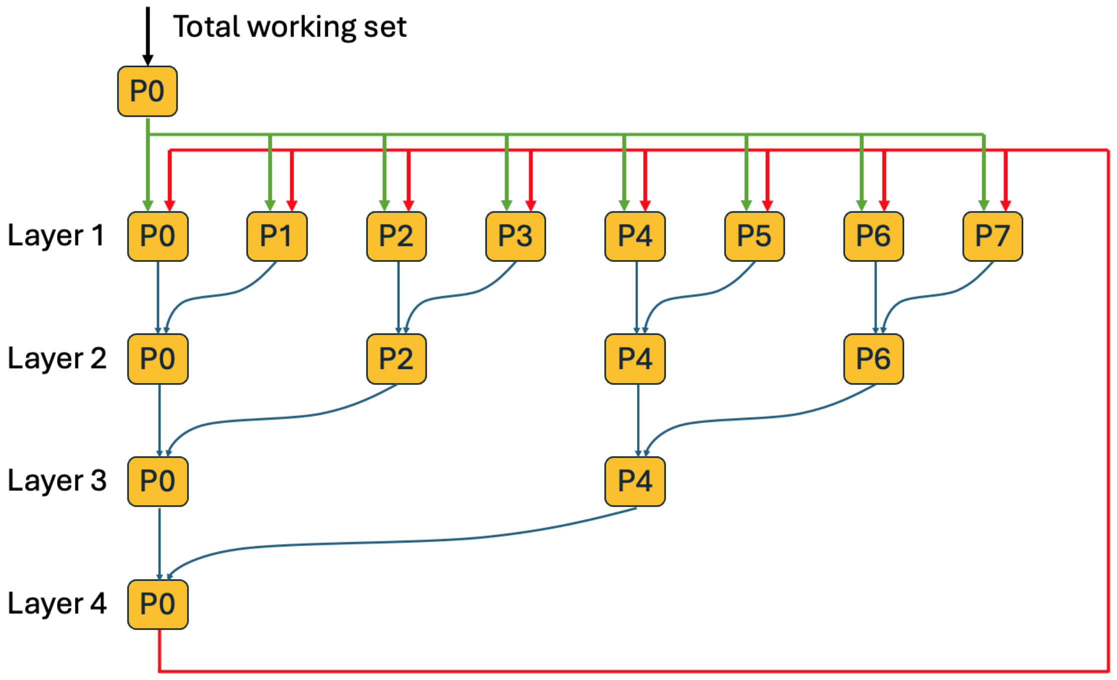


Figure 2: Binary Cascade SVM tree with $P = 8$ processes. The data are split into subsets and each one is evaluated individually for support vectors in the first layer. The results are combined two-by-two and entered as training sets for the next layer. The resulting support vectors are tested for global convergence by feeding the result of the last layer into the first layer, together with the non-support vectors.

Algorithm 2 Classical Cascade SVM with MPI (multi-round, tree-structured)

```
 1: Rank 0 reads full training set, assigns global IDs, partitions and distributes data
 2: Rank 0 computes global feature-wise min/max and broadcasts them
 3: Each rank scales its local partition using the global min/max
 4: Initialize global SV set on rank 0 to empty, and global_ID_sv to empty; Set converged $\leftarrow$
    false
 5: repeat
 6:    if rank = 0 then
 7:       Copy current global SVs into recv_X_sv, recv_Y_sv, recv_alpha_sv, recv_ID_sv
 8:       Set recv_sv_count to the size of this set
 9:    end if
10:    Broadcast recv_sv_count from rank 0 to all ranks
11:    if recv_sv_count > 0 then
12:       Nonzero ranks resize recv_X_sv, recv_Y_sv, recv_alpha_sv, recv_ID_sv accordingly
13:       Rank 0 broadcasts recv_X_sv, recv_Y_sv, recv_alpha_sv, recv_ID_sv
14:    else
15:       Nonzero ranks clear their recv_ buffers
16:    end if
17:    Initialize X_sv $\leftarrow$ local partition x_part, Y_sv $\leftarrow$ y_part, ID_sv $\leftarrow$ id_part, sv_count $\leftarrow$ my_n
18:    for step = 1; step $\leq P$; step = 2step do
19:       if  rank % step == 0 then
20:          Build X_train, Y_train, ID_train, alpha_local as the union of broadcast SVs and
             current local SVs, deduplicated by global ID
21:          Run SMO on this training set to get updated alpha_local and bias b_local
22:          Extract support vectors (indices with $\alpha_i >$ tol_sv), rebuild X_sv, Y_sv, alpha_sv, ID_sv,
             and set sv_count accordingly
23:       end if
24:       if step < $P$ then
25:          if rank % (2step) == step then
26:             Send sv_count, X_sv, Y_sv, alpha_sv, ID_sv to rank rank - step
27:          else if rank % (2step) == 0 then
28:             Receive recv_sv_count and corresponding arrays from rank rank + step into
                recv_X_sv, etc.(These received SVs will be merged at the next value of step)
29:          end if
30:       end if
31:    end for
32:    if rank = 0 then
33:       Compare current ID_sv with global_ID_sv
34:       if the two ID sets are identical then
35:          converged $\leftarrow$ true
36:          Evaluate final model (X_sv, Y_sv, alpha_sv, b_local) on the test set
37:       else
38:          converged $\leftarrow$ false
39:          Update global_ID_sv with the current ID_sv
40:       end if
41:    end if
42:    Broadcast converged from rank 0 to all ranks
43: until converged is true or maximum number of rounds is reached
```

5.2 Implementation of Modified Two-Layer Cascade SVM

The classical Cascade SVM reduces the amount of data processed at each merge stage but may still perform lots of redundant work. For example, intermediate retraining step leads to additional SMO iterations and communication. To reduce this overhead, I transformed the classic Cascade SVM into a two-layer Cascade SVM that uses a star topology instead of a tree as shown in Figure 3.

In the modified design, each round consists of only two layers. In the first layer, every rank trains an SVM on the union of its local partition and the current global support vectors, exactly as in the classical case. Besides, the training set is deduplicated by global ID, previously known global support vectors are warm-started with their existing α values, and new local samples start with $\alpha = 0$. Each rank runs SMO and extracts its local support vectors.

In the second layer of the round, all nonzero ranks send their local support vectors (features, labels, alphas, and global IDs) to rank 0 in a single communication phase. Rank 0 then merges all received support vectors with its own local support vectors, deduplicates by global ID, resets the merged alphas (or keeps them, depending on the chosen warm start), and runs another SMO solve on this merged set. The resulting support vectors become the new global support-vector set for the next round.

As in the classical version, rank 0 checks convergence after each round by comparing the set of global support-vector IDs with that of the previous round. If the ID sets are unchanged, the algorithm terminates and the final model is evaluated on the test set. Otherwise, rank 0 broadcasts the updated global support vectors, and the next round begins. Conceptually, the topology is always "local–global": there are no tree levels beyond rank 0, but the two-layer pattern is repeated until the global SV set remains unchanged.

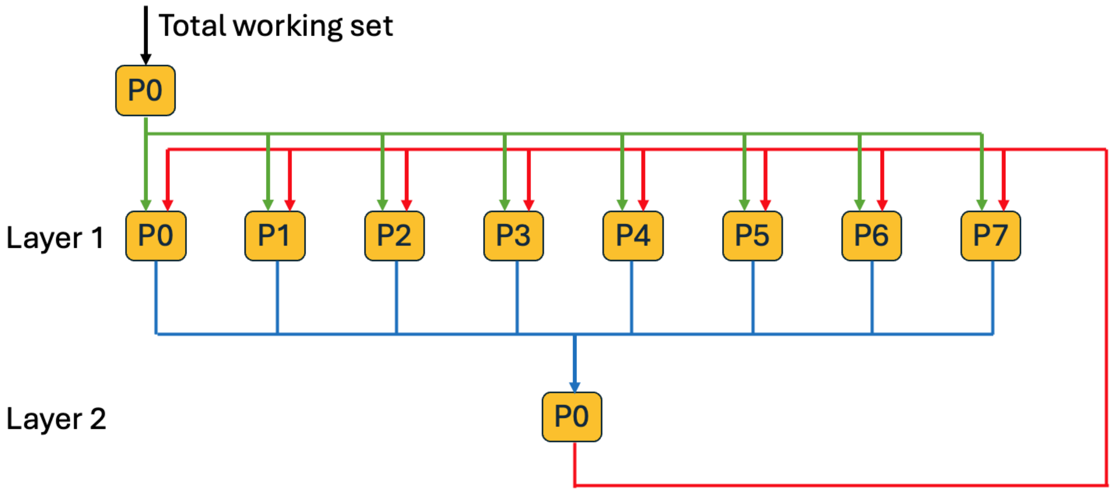


Figure 3: Modified two-layer Cascade SVM. In each round, every process trains on the union of its local partition and the current global SVs, then sends its local SVs directly to rank 0. Rank 0 merges, retrains on the union, and checks whether the global SV ID set has stabilized.

Algorithm 3 summarizes the modified implementation. Compared with the classical Cascade SVM, it replaces the multi-level reduction tree with repeated local–global rounds and a single global retraining step per round.

Algorithm 3 Modified Two-Layer Cascade SVM with MPI

```
1: **Input:** global training set, number of ranks $P$
2: Rank 0 reads all data, assigns global IDs, partitions and distributes data
3: Rank 0 computes global feature-wise min/max and broadcasts them
4: Each rank scales its local partition using the global min/max
5: Initialize the global SV set on rank 0 to empty; set round $\leftarrow 0$, converged $\leftarrow$ false
6: **repeat**
7:   round $\leftarrow$ round $+ 1$
8:   **if** rank $= 0$ **then**
9:     Broadcast current global SVs (features, labels, alphas, IDs)
10:  **else**
11:    Receive current global SVs from rank 0
12:  **end if**
13:  Each rank forms a local training set by combining: (i) its local partition and (ii) the broadcast
     SVs, deduplicating by global ID and warm-starting alphas
14:  Each rank runs SMO on its local training set and extracts local SVs
15:  **if** rank $= 0$ **then**
16:    Initialize merged SV set with its own local SVs; track seen IDs
17:    **for** each rank $r = 1, \ldots, P - 1$ **do**
18:      Receive local SVs (features, labels, alphas, IDs) from rank $r$
19:      Insert unseen SVs into the merged set, deduplicating by ID
20:    **end for**
21:    Run SMO on the merged SV set to obtain updated global SVs
22:    Extract global SVs and their IDs
23:    Compare current global SV IDs with those from the previous round
24:    **if** ID sets are identical **then**
25:      Set converged $\leftarrow$ true
26:      Evaluate the final model on the test set and report accuracy
27:    **else**
28:      Update stored global SV IDs and set converged $\leftarrow$ false
29:    **end if**
30:  **else**
31:    Send local SVs (features, labels, alphas, IDs) to rank 0
32:  **end if**
33:  Broadcast converged from rank 0 to all ranks
34: **until** converged is true or maximum number of rounds is reached
```

6 Experimental Results

All experiments use the MNIST handwritten digit dataset, detailed in section 3. All models use an RBF kernel with $\gamma = 0.00125$ and penalty parameter $C = 10$.

To verify the correctness of the implementations, all runs achieve the same test accuracy of 0.9969 (9969 out of 10000) with 1548 support vectors. The final thresholds b also differ by less than 0.003% across implementations, confirming numerical consistency.

The overall computation consists of two main phases: training and prediction. Training involves iterative SMO updates, making it difficult to isolate the runtime of individual subcomponents. Prediction, by contrast, is embarrassingly parallel and is accelerated only in the GPU implementation.

Consequently, I report both training and prediction times for the GPU solver, while the MPI-based methods focus solely on training time.

6.1 GPU implementation

Table 1 compares the serial SMO solver with the GPU-based version. With identical hyperparameters and accuracy, the GPU solver achieves a 56× speedup over the serial baseline.

Table 1: Serial vs. GPU SMO on MNIST (60k training / 10k test).

Method	b	Training Time (s)	Speedup
Serial SMO	-5.9026206	3,285.662	1.00
GPU SMO	-5.9027319	58.570	56.09

To examine how runtime scales with training-set size, the GPU solver was run using $n = 10{,}000, 20{,}000, 30{,}000, 40{,}000, 50{,}000$, and 60,000 training samples while keeping the test set fixed. The results are presented in Table 2 and Figure 4. As expected, prediction time increases almost linearly with the number of support vectors, since evaluating $\sum_{i \in SV} \alpha_i y_i K(x_i, x)$ is embarrassingly parallel.

In contrast, training time grows superlinearly with n. Each SMO iteration requires computing two kernel rows of length n, and the number of SMO iterations also increases with dataset size. Thus, the overall complexity is

$$\text{Training time} \sim (\#\text{ iterations}) \times (\text{cost per iteration}) \propto O(n) \times O(n) = O(n^2),$$

up to constants and GPU efficiency. Because the GPU is already fully utilized for moderate n, additional data mainly increases the number of kernel evaluations rather than parallel efficiency, explaining the superlinear scaling behavior.

Table 2: Scaling of training and prediction time with training-set size on GPU.

Size of training set	Training Time (s)	Prediction Time (s)
10,000	3.555	6.854
20,000	6.719	13.140
30,000	10.164	19.439
40,000	16.270	25.720
50,000	29.790	32.011
60,000	58.570	38.297

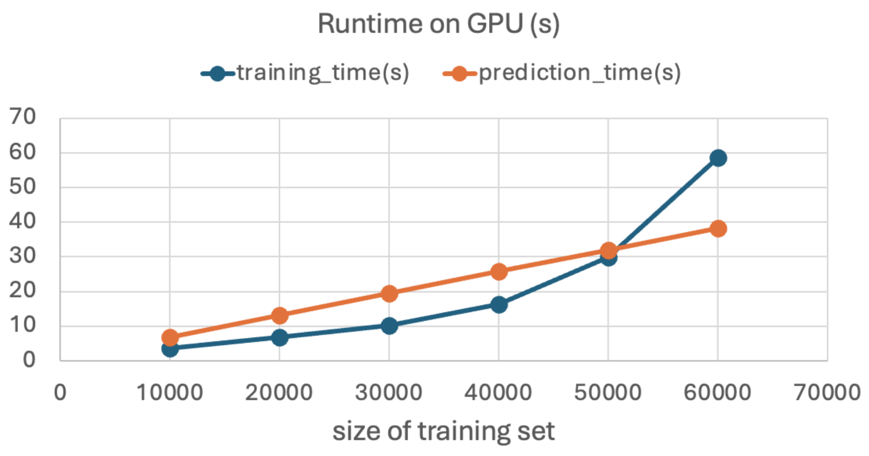


Figure 4: Scaling of training and prediction time on the GPU as a function of the number of training samples n.

6.2 MPI implementation

The MPI implementation uses blocking communication; non-blocking variants were tested but provided no improvement for this work. Table 3 reports results for the classical Cascade SVM. Although the method consistently produces correct models, its speedup saturates quickly, with the best performance around 16 processes, consistent with observations in [3]. And the efficiency decreases with the number of processes.

Table 3: Classical Cascade SVM performance for different process counts (60k training / 10k test).

Processes	b	Training Time (s)	Speedup	Efficiency
4	-5.9025395	1,194.269	2.75	0.69
8	-5.9025786	839.406	3.91	0.49
16	-5.9025665	662.153	4.96	0.31
32	-5.9025757	671.448	4.89	0.15
64	-5.9026030	673.580	4.88	0.08

The modified two-layer Cascade SVM, shown in Table 4, exhibits significantly better scalability. By eliminating intermediate levels in the cascade tree and sending all local support vectors directly to rank 0, it avoids many redundant retraining stages. Speedup increases nearly linearly up to 16 processes and continues to improve at 64 processes.

Table 4: Modified two-layer Cascade SVM performance (60k training / 10k test).

Processes	b	Training Time (s)	Speedup	Efficiency
4	-5.9026390	886.733	3.71	0.93
8	-5.9025582	649.773	5.06	0.63
16	-5.9025660	440.705	7.46	0.47
32	-5.9025403	333.696	9.85	0.31
64	-5.9026032	301.263	10.91	0.17

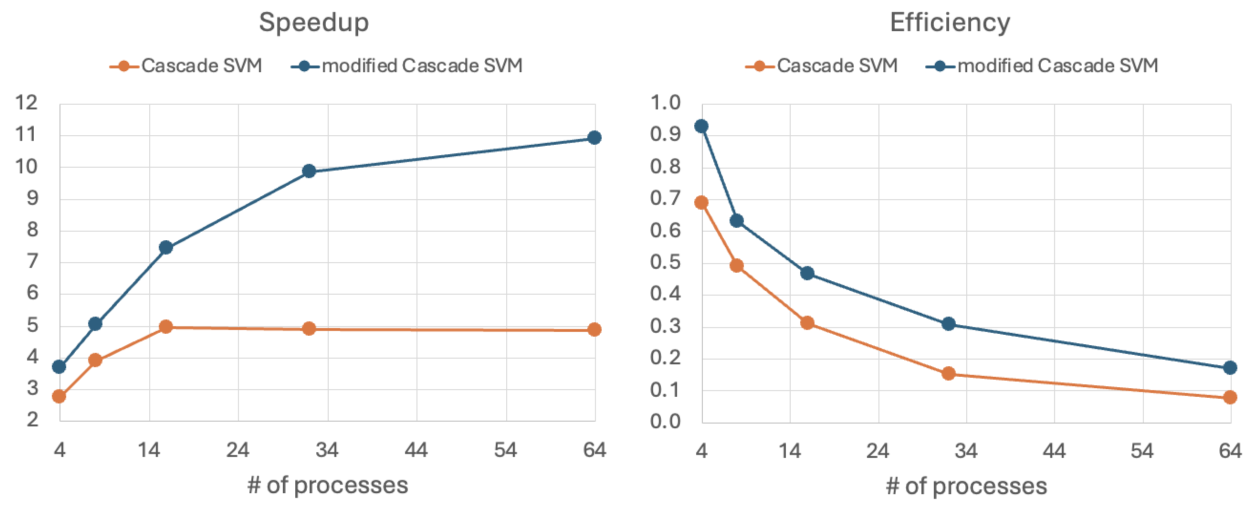


Figure 5: Speedup and efficiency of classical and modified Cascade SVM relative to the serial implementation.

Figure 5 shows the speedup and efficiency for both methods. The classical method stops scaling early, but the modified method keeps scaling as I add more processes. However, in both cases, parallel efficiency decreases as the number of processes increases. This effect arises from several interacting factors.

First, as the dataset is divided among more processes, each rank receives a smaller initial partition (e.g., 15,000 samples at $P = 4$ versus only 937 at $P = 64$). Although this reduces early-round computation, the communication, merging, and synchronization overheads do not reduce proportionally and eventually dominate runtime. Second, the number of cascade rounds required for convergence remains constant across all process counts: both the classical and modified methods converge in 6–7 rounds, regardless of P. Thus, adding more processes does not reduce the number of global merging and retraining steps.

Figure 6 further illustrates how the number of support vectors evolves in the modified Cascade SVM. For all process counts, the algorithm identifies about 97% of the final support vectors in the first first round. As a result, the size of the support-vector messages exchanged between processes is nearly independent of P. Combined with the fact that the final global support-vector set is fixed at 1548, this implies that, in later rounds, the global SVs constitute a large fraction of each rank's training set—for example, at $P = 32$ a rank holds 1875 local samples and 1548 global SVs, and at $P = 64$ the global SVs actually outnumber the local samples. Because all ranks are repeatedly training on almost the same set of support vectors while communication volume continues to grow roughly linearly with P, much of the later-round computation becomes redundant across processes. This mismatch explains why both the classical and modified Cascade SVMs exhibit diminishing efficiency at higher process counts, even though the overall training time continues to decrease.

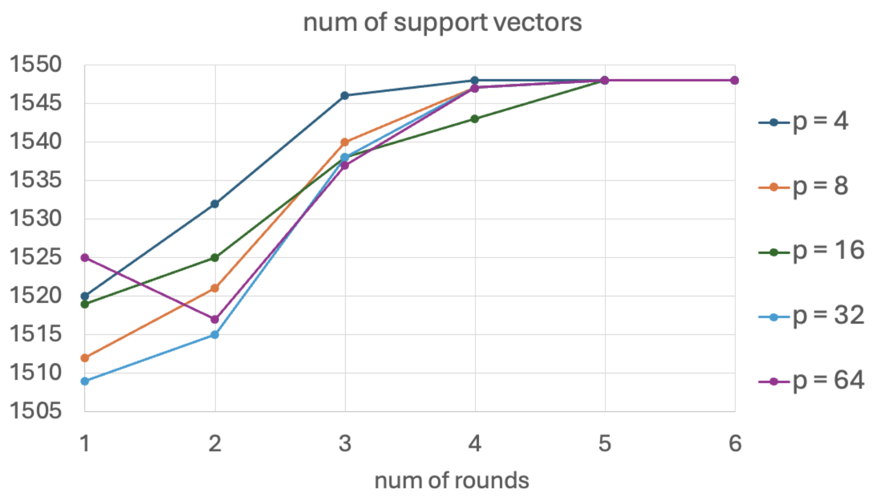


Figure 6: Number of support vectors identified in each cascade round for 4, 8, 16, 32, and 64 processes in the modified Cascade SVM. Across all process counts, the algorithm recovers approximately 97% of the final support vectors in the first round, indicating that the size of support-vector messages exchanged between processes is nearly independent of P.

7 Future Work

As a natural next step, I would like to combine GPU and MPI parallelism. Each MPI rank could run a GPU-accelerated SMO solver on its local partition, thereby reducing local training time while still benefiting from the data-parallel structure of Cascade SVM. Rank 0 would then gather support vectors as in the modified design and perform a final GPU-based training step. This hybrid approach would be capable of handling much larger datasets while still exploiting fast device kernels.

8 Conclusion

This project investigated two complementary methods for accelerating SVM training on the MNIST dataset: a serial SMO implementation, a GPU-based SMO solver, and two MPI-based Cascade SVM designs. The GPU implementation achieved nearly a 56× speedup over the serial baseline by parallelizing feature scaling, kernel evaluations and error updates, etc. The modified two-layer Cascade SVM achieved up to a 10.9× speedup on 64 processes and consistently outperformed the classical Cascade SVM across all tested process counts. All methods produced the same test accuracy (99.69%) and the same support-vector count, confirming that performance improvements were achieved without sacrificing model quality.

References

[1] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[2] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. *Technical Report MSR-TR-98-14*, Microsoft Research, 1998.

[3] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik. Parallel support vector machines: The cascade SVM. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 521–528, 2005.

[4] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.

[5] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011.