Parallelizing Support Vector Machines

Tianfan Xu[1], Wesley Osogo[2], Peter Chen[1,2], and Connor Buchheit[1,2]

[1]Harvard University — CS 205

May 6th, 2024

Abstract

This project presents a comprehensive report and analysis of the performance differences between serialized and parallelized implementation of Support Vector Machines (SVMs). We explore the scalability and efficiency of parallelizing SVMs and employ strong and weak scaling methods to evaluate how the algorithms' performance evolves with increased number of machines in a distributed memory system. Our study uses data sets of different sizes to test the processing time and accuracy across both serialized and parallelized code, and parallelized algorithm with different number of machines. Additionally, we discuss the computational overhead and the trade-offs involved in parallelizing SVMs. Hopefully, this project offers insights into optimizing machine learning with distributed computing and memory systems.

1 Background and Significance

Support Vector Machine (SVM): Support Vector Machines are a method to predict upon binary classification problems, where given a set of inputs with n features and their labels, the algorithm is trained to classify the data, so that given future inputs with n features, the algorithm can classify the input into one of the two classes with high precision.

To perform this task, SVMs mathematically find the hyperplane that acts as a decision boundary between the data points. This is done by transforming the data with a kernel function, mapping the data into a higher-dimensional feature space where linear separation of classes is easier. In our kernel feature space, this is accomplished with a dual optimization problem, in which we both maximize the distance of our points from the hyperplane while optimizing for correct classifications, and we also optimize for our Lagrange multipliers in our constrained optimization task to determine how much we weigh the influence of each data point in finding the correct boundary; the "support vectors" are thus the points that lie closest to the boundary or are the ones that are misclassified, and they are the strongest determiners of the orientation of our hyperplane.

Mathematically, we formalize this optimization as:

$$\min P(\mathbf{w}, \xi) = \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{n} \xi_i \text{ s.t. } 1 - y_i(\mathrm{w}^T\phi(\mathrm{x_i}) + b) \leq \xi_i \ \xi_i > 0 \tag{1}$$

$$\min D(\alpha = \frac{1}{2}\alpha^T\mathbf{Q}\alpha - \alpha^T\mathbf{1} \text{ s.t.} \mathbf{0} \leq \alpha \leq \mathbf{C},\ \mathbf{y}^T\alpha = 0 \tag{2}$$

where equation (1) is our "primal" and $\mathbf{w}$ is a vector defining our hyperplane, ξ is our vector of slack variables defining that allow for misclassification if the data is not exactly separable, C is a regularization parameter that penalizes margin violations more strictly, $\mathbf{x_i}$ is our feature vectors of training samples, $\phi(\mathbf{x_i})$ is our transformed samples, y_i is the class label of our samples, and b is the bias of the hyperplane, and equation (2) is our "dual" where α are Lagrange multipliers and $\mathbf{Q}$ is our kernel matrix $\mathbf{Q_{ij}} = y_i y_j \phi(\mathbf{x_i})^\mathbf{T} \phi(\mathbf{x_i})$.

SVMs have largely fallen out of favor as a classification algorithm with the advent of neural networks, particularly due to how they do not scale well in terms of training time with respect to dataset size. However, in the context of high performance programming, the bottlenecks that caused it to fall out of favor can be alleviated with new technology. Thus, our project investigates the scalability of SVMs.

We were largely inspired by the work of Chang et. al (2011) [1] in investigating this problem. In their research, Chang et. al developed an algorithm to parallelize SVM and make use of distributed memory to work around the computational bottlenecks. In our work, we implemented their algorithm from scratch, using the parallelization methods we learned in class to determine how much of a speedup we achieve with more machines.

2 Scientific Goals and Objectives

The asymptotic runtime of traditional SVM algorithms is $O(n^3)$ with respect to the dataset size, as the dual optimization problem is solved with Quadratic Programming via the Interior Point Method (IPM) and involves matrix inversion as the bottleneck, which takes $O(n^3)$ time. At the highest level, our main objective is to improve upon this poor asymptotic scaling of SVMs through parallelization. By distributing the computational load across multiple machines with the parallel algorithm, we sought to test how parallel SVMs could handle larger datasets and reduce training times significantly, particularly by virtue of distributing our matrices such that each processor can approximate an inverted matrix using only a subset of the original matrix.. Furthermore, we also aimed to make use of OpenMP to take advantage of data parallelization, speeding up elements such as kernel computation and Cholesky factorization of matrices (discussed in Section 3).

In order to fully test the benefits from implementing the algorithm, we sought to optimize performance over a variety of factors. For starters, we wanted consistently-high accuracy across a variety of datasets, so we both created code to generate synthetic datasets by fetching data from the UCI repo and scaling it such that it has mean=0 and variance=1; we additionally tested on pre-existing datasets. Furthermore, we wanted to greatly increase the size of the datasets, both in terms of dimensionality and in terms of datapoints. For instance, a dataset with 40,000 points and 54 features took hours to train upon in the sequential algorithm; our goal was to be attain results on accuracy and training time with such a dataset in parallel.

Thus, the need for compute hours on an HPC architecture emerges as we wish to train our model and examine the impact of distributing computation across machines and parallelizing computations without data dependencies with OpenMP. With access to compute hours, we can efficiently train SVMs on larger datasets than would typically be manageable, allowing us to create powerful predictive models and facilitating further research on the properties of SVMs, including how prediction accuracy varies with training size and how training time scales when done in parallel.

3 Algorithms and Code Parallelization

For our code, our implementation revolved around two main algorithms — Parallel IPM (Interior Point Method) for solving the Quadratic Programming problem, and ICF (Incomplete Cholesky Factorization), which gives us a low-rank approximation for our Q matrix, which is the matrix $\mathbf{Q_{ij}} = y_i y_j \phi(\mathbf{x_i})^{\mathbf{T}} \phi(\mathbf{x_i})$. ICF is useful in our implementation because it reduces the dimension, and thus the computational load, of our Q matrix, which is in turn used in our Parallel IPM. We are main developers for this code.

The training process is roughly:

1. Set up MPI and initialize our parameters.
2. Calculate the Q matrix and distribute subsets of its rows across nodes.
3. Perform ICF and use the results to perform IPM to solve the optimization problem, yielding the optimal hyperplane used to separate our training data.
4. Make predictions and compare with the true labels, storing the equation for the optimal hyperplane to use on testing data.

For our matrix representations and linear algebra operations in our code, we extensively utilized the Eigen library. We chose the Eigen library for a few reasons. First of all, its optimized performance for matrix operations, along with the built-in vectorization that facilitates SIMD parallelization, lending itself to high performance and reduced times. Additionally, it is very flexible in its representation of matrices, allowing us to dynamically size matrices when we do not know their size in advance, allowing our code to easily accommodate different dataset sizes. Finally, the library is extensive in the features it provides, allowing us to handle many complicated linear algebra tasks in one method, rather than many lines of computation. For the kernel function, we used a linear kernel, as it achieved comparable results with the other choices of kernel functions we tried, such as RBF kernels, and it was the fastest choice.

To better understand the interplay between these algorithms and our use of both OpenMP shared memory parallelism and MPI distributed parallelism, we shall dissect each algorithm in detail.

1) ICF Algorithm:
This algorithm makes use of both OpenMP and MPI. Before the algorithm is run, the Q matrix is calculated and distributed across nodes using the function call
"MPI_Scatterv(Q.data(), Qsendcounts.data(), Qdispls.data(), MPI_DOUBLE, localQ.data(), Qsendcounts[rank], MPI_DOUBLE, 0, MPI_COMM_WORLD)" where Q.data() contains the contents of the Q matrix, Qsendcounts.data() contains the number of elements to be sent to each process, and Qdispls.data() contains the starting index in the send buffer for each process.

As arguments, the function takes in the matrix **localQ**, which contains the distributed Q matrix that each node will take, int n, which specifies the number of rows and columns of the original Q matrix, int p, which specifies the approximate rank of the ICF, and matrix **localH**, which contains the ICF matrix computed by the algorithm. We begin with initializing our MPI environment, as well as calculating the amount of work that is done by each machine using our inputs n, p, and the amount of

machines we are using, in order to ensure that the amount of work done on each machine is roughly equal.

For the rest of the algorithm, we loop from $i = 0$ to $i = p-1$ and iteratively solve. The vector **local_v** is created to store the diagonal elements of each submatrix held in each machine. The code then calculates the trace of the Q matrix by calculating each row's contribution to the trace by summing elements in each **local_v**, and uses **MPI_Allreduce** to efficiently calculate the global trace with each local trace. We iterate through this process p times, stopping if the global trace is below a certain threshold (in this case, we used 5 as it empirically was efficient and effective), as this determines whether our approximation is close enough to a Cholesky decomposition. If this is not the case, we then find the maximum element on the diagonal of Q among all the locally-stored rows of the Q matrix using **MPI_Allreduce** and store the index and value.

Next, we locate the process that contains the maximum value on the diagonal by checking whether the current process contains the maximum diagonal. Upon locating the row containing the maximum diagonal value (called the "pivot row), we iteratively update its values below and on the diagonal, copying values from **localH** into the below-diagonal elements and setting the diagonal element of the pivot row to the square root of the maximum diagonal. Upon doing so, we broadcast the pivot row to all other processes, using MPI_Bcast t transmit the $k + 1$ columns we update. We use blocking communication because this point is a critical point, as the rest of the processes need the information from the updated row before performing any other operations.

Finally, we use OpenMP in our final step, which is updating **localH** (our updated matrix) and **local_v** (our matrices containing our diagonals). In the first parallelized for loop, we loop across the number of rows that each process works on (which was calculated in the beginning of this algorithm), calculating the sum of the products of the entries in the pivot row and the corresponding entry of **localH**, storing the result back in the H matrix. In the second parallelized for loop, we iterate through all rows, reducing the diagonal value by substracting the square of the pivot value for the kth column from **localH**. If we go through p iterations without achieving a trace value that is less than our threshold of 5, we simply return p, otherwise we return k (e.g. the iteration number) and localH as an ICF matrix for Q.

2) IPM Algorithm

This algorithm is primarily handled with the method **PrimalDualIPM::Solve**, which takes in the arguments **local_labels**, which contains the ground truth labels for the datapoints we use in training, **local_num_rows**, which corresponds to the number of rows each processor handles of the H matrix, **global_num_rows**, which describes the total number of samples in the training instance, and H, which is the parallel ICF result for each process, distributed evenly by rows in each machine. Our parameters described in Section 1) are also either passed as arguments or randomly initialized to a starting value, and they are initialized such that the size is evenly distributed among each processor, allowing us to split the workload evenly.

We then distribute the lower-triangular matrix H that we calculate with our ICF algorithm, and MPI_Barrier is called to ensure that each process has completed its initial setup before continuing. Then, we iterate from $i = 0$ to $i =$max_iter (a hyperparameter we set to 50), sequentially updating our parameters as follows: We begin by computing the surrogate gap, η, which measures the closeness of the current solution from optimality with respect to the KKT conditions. We use this gap to update the parameter t, which controls our tradeoff between quick optimization and the tightness of our slack variables ξ. We then compute $z = HH^T\alpha$ using **MPI_Allreduce** and use these values to calculate the primal and dual

residuals, measuring how far our solutions are from the KKT conditions once more. If our values of η and our two residuals are below certain thresholds (10^{-6} for η, 10^{-4} for our residuals), we end our training. Next, we calculate our margins and calculate our factors using the Sherman-Morrison-Woodbury formula. We then perform more parallel reductions with **MPI_Allreduce** to aggregate results from all processors, ensuring global views on our variables.

Next, we perform line search for each variable, calculating the maximum step size that keeps the updates within a feasible region with respect to our constraints, and we make our adjustments with slightly smaller sizes to prevent overshooting the optimal solution. After updating our variables, we start at the beginning of the loop, once more checking whether we have converged; upon either convergence or exhausting max_iters, we return the labels corresponding to the solutions to our optimization problem.

Validation, Verification

Our models that we created were validated for accuracy with both our synthetic datasets which we generated with our code data_generate.py as well as datasets we found and processed. For our datasets, we did a 75/25 split of train and test data. We consistently found that our model performed at 80% accuracy on the test data, indicating that our model is quite good at making predictions, even on datasets with up to 54 features. Thus, we have checked that the accuracy of our model is maintained when parallelizing the code.

4 Performance Benchmarks and Scaling Analysis

For scaling experiments, we used a large open source data set (cover_type) which has up to $580,000$ samples with 54 dimensions. However, for convenience and careful use of resource we sampled different number of samples from the datasets and used for them for the scaling experiments that follow.

Strong Scaling

Given that a single machine cannot efficiently factorize the kernel matrix in its local memory, we are not able to conveniently obtain running time of the implementation on a single machine. We use 5 machines as baseline to measure speedup of using more than 10 machines for 10000 samples. The inherent assumption is that speedup of 5 machines to 1 is perfectly linear which we deem reasonable. We observe that as we increase the number of available machines we enjoy a superlinear speed up but this diminishes when number of machines exceeds 35. This is likely explained by increase in communication overheads with increase in number of machines

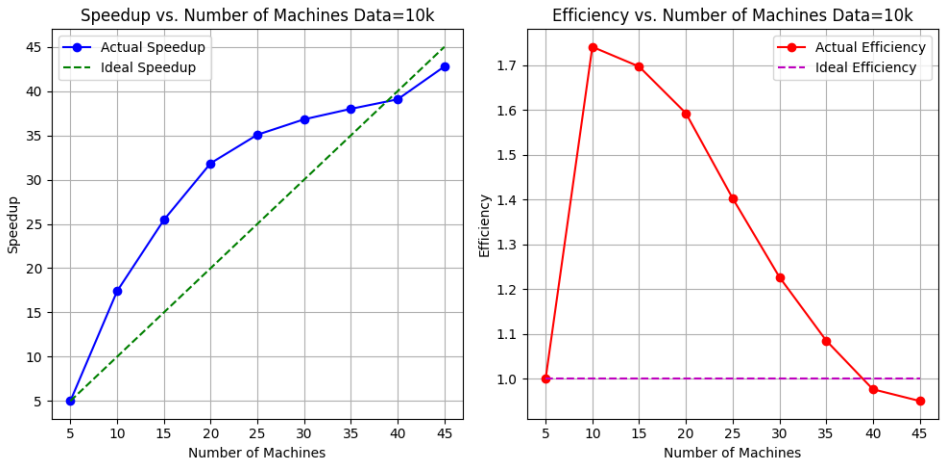

Figure 1: 10k Strong speed up

For 50000 samples however, we get less than ideal strong speedup with even small number of machines showing effect of overhead in limiting scalability. The efficiency plots illustrate this fact.

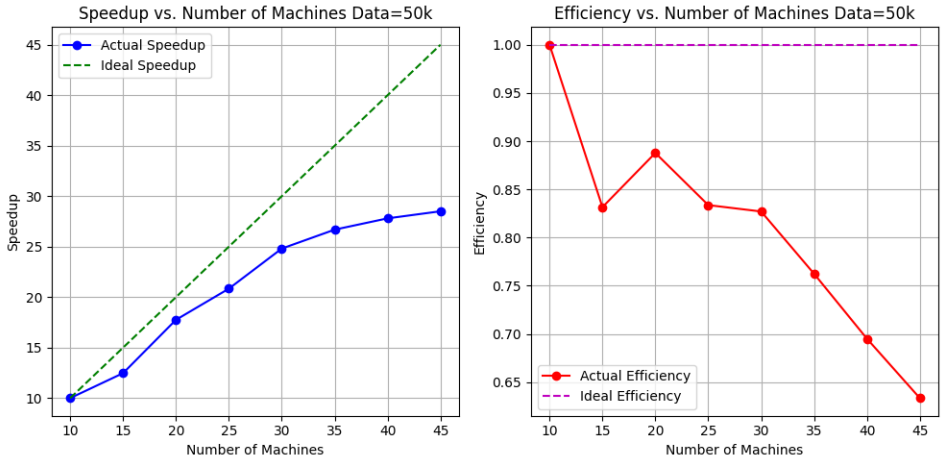

Figure 2: 50k Strong speed up

Communication Overhead

The figures below illustrate how communication overhead limits speed up that can be achieved by increasing number of machines. We benchmark this only for the ipm solver thus this effect is even more pronounced with inclusion of ICF. We observe that as we increase number of machines communication takes a larger and larger percentage of time compared to computation because an increasing number of machines need to synchronize parameters.Thus ideally we want to figure out a good point in number of machines where communication overhead is not that significant.

DataSize (10k)			
# Machines	Communication(s)	Computation(s)	Total(s)
10	0.137346	20.893	21.030346
20	0.0746704	2.78609	2.8607604
30	0.028086	0.891897	0.919983
40	0.0105262	0.440084	0.4506102
48	0.0497279	0.285863	0.3355909

Table 1: 10k Data Statistics

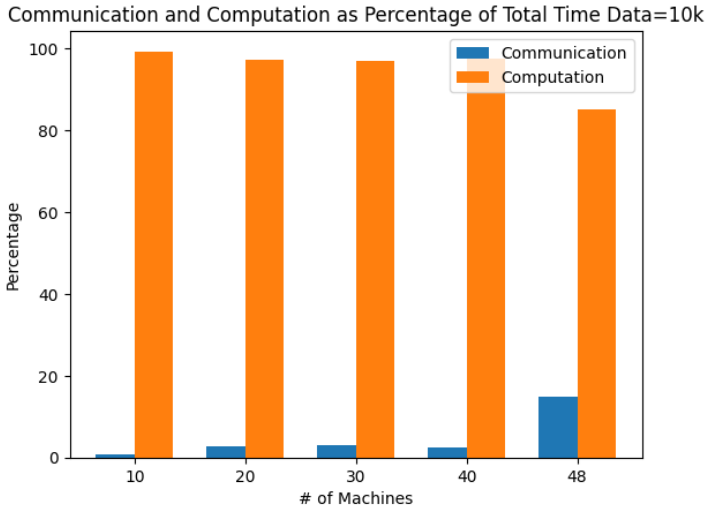

Figure 3: 10k Data Time

DataSize (50k)			
# Machines	Communication(s)	Computation(s)	Total(s)
20	2.37357	118.539	120.91257
30	0.518076	30.8502	31.368276
40	0.0771924	12.6438	12.7209924
48	0.0661727	7.28403	7.3502027

Table 2: 50k Data Statistics

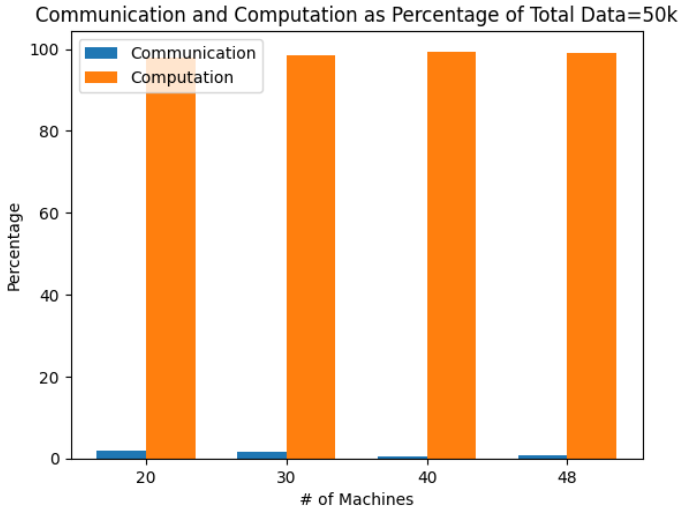

Figure 4: 50k Data Time

Table 3 provides two sample cases of typical job submissions during development and testing. In Test case A we have a sample size of $10,000$ and in Test case B we have a sample size of $50,000$ trained on average for 10 iterations.

	Test case A	Test case B
Typical wall clock time (hours)	0.2	0.2
Typical job size (nodes)	10	30
Memory per node (GB)	1	18
Maximum number of input files in a job	8	8
Maximum number of output files in a job	1	1
Library used for I/O	Standard	Standard

Table 3: Workflow parameters of the two test cases used during project development.

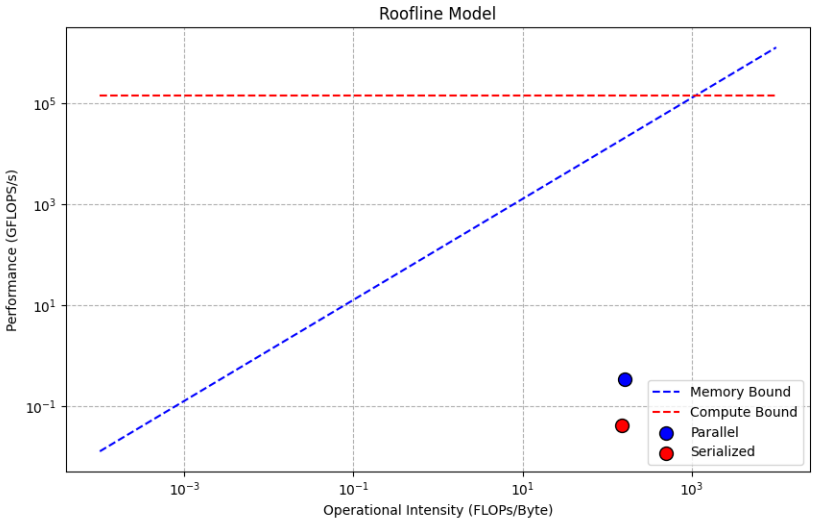

Figure 5: Roofline Model

The roofline model is as shown in the figure. The compute bound is calculated by

$$\text{peak GFlops/s} = 18 \text{ cores} \times 3.0 \text{ GHz} * 1000 * 16 \text{ operations/FMA} = 864000 \text{ GFlops/s} \tag{3}$$

with the default memory bandwidth of 128 GB/s. The parallelized benchmark is performed on our algorithm with 8 ntasks-per-node and 3 cpus-per-task. The serialized is benchmarked on the serialized performance with no parallelization. As shown in the graph, there is significant improvement in performance from serialized to parallelized implementation. Both implementations fall under **memory bound** region, since SVM requires large data matrices, and the computation often is stalled waiting for data.

Despite parallel algorithm outperforming serialized implementation, both are still below the peak performance given the operational intensity. This is due to and not limited to these several factors:

1. Computational and communication overhead
2. Despite using loop unrolling and vectorized calculations, the eigen library calculations might not fully maximize CPU performance when doing matrix calculations
3. Large amount of data set, which requires significant time in reading and outputing data
4. Given the large amount of data, there are often cache misses that require additional overhead

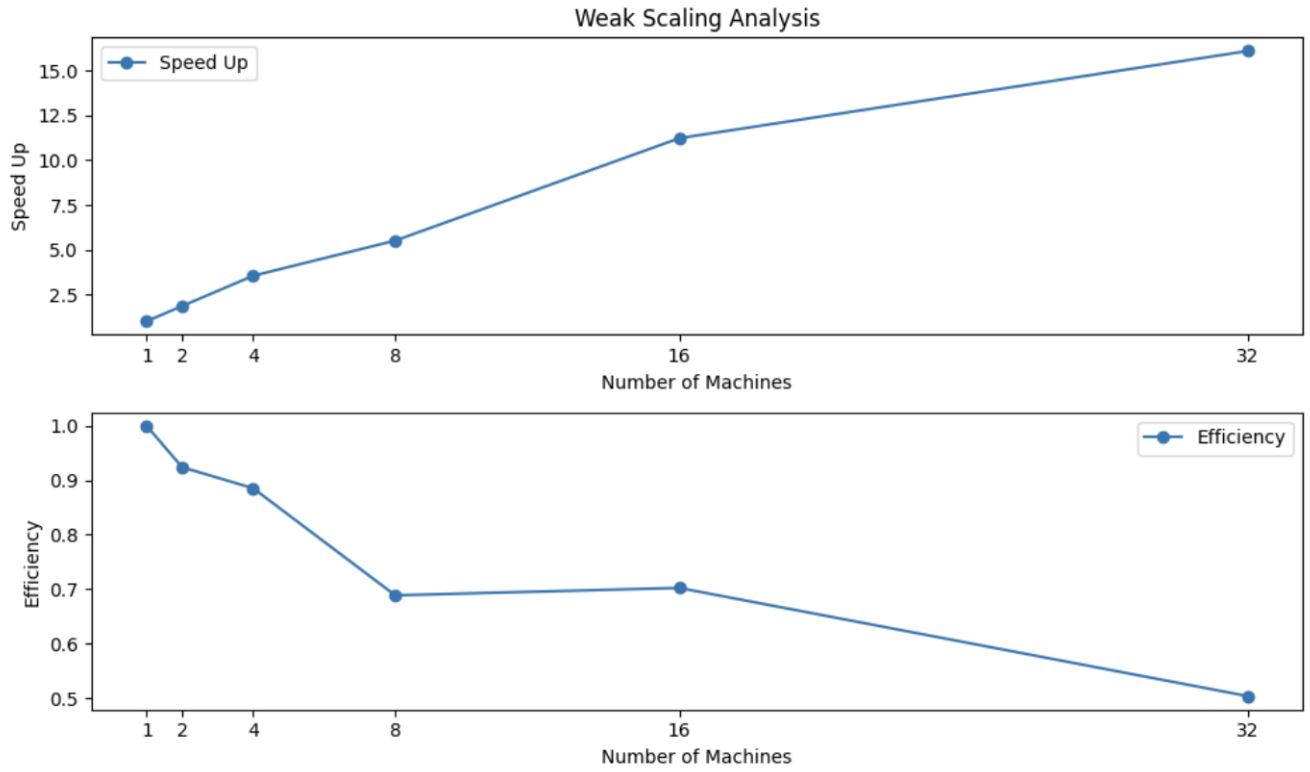

Figure 6: Weak Scaling

Weak Scaling

The weak scaling is shown in the figure. Given that the implementation uses a distributed memory system across the machines, in an ideal situation, the speedup should approach p where $p =$ the number of machines. Similarly, the ideal efficiency is 1.0 throughout. However, this weak scaling makes sense, given that there is computational overhead, communication overhead and a memory bound. At 32 machines, we are at about 50% efficiency. The data is given from the table below:

Weak Scaling Data					
Machines	**Work Size**	**Time 1(s)**	**Time 2(s)**	**Time 3(s)**	**Avg Time(s)**
1	**1600**	462.292	460.79	443.29	**455.4573**
2	**3200**	494.647	490.527	493.555	**492.9097**
4	**6400**	515.204	514.625	513.368	**514.399**
8	**12800**	660.103	662.108	661.896	**661.369**
16	**25600**	649.446	648.785	647.518	**648.583**
32	**51200**	904.972	904.657	903.967	**904.532**

Figure 7: Weak Scaling Data

5 Resource Justification

From Figure 6 and Figure 7, we see that we reach an efficiency of about 50% with 32 machines and a work size of 51200, yielding an average time of 904.532 seconds.

We can thus calculate:

$$32 \text{ nodes} \times \frac{904.532s}{3600\frac{s}{\text{hour}}} = 8.0403 \text{ node hours}$$

For both of the test cases below, assume that we are maintaining a work size of 51200.

Say in Test Case A, we want to increase the max number of iterations from 500 (in our code—described in Section 3) to 2000 in our training, and we want to train 30 models to fine tune the initialization of different hyperparameters. In this case, our simulations per task is 10, whereas our number of iterations per simulation increases from our base of 500 to 2000, which is a four-fold increase from the wall time we calculated, effectively making our iterations per simulation 4.

Meanwhile, say that in Test Case B, we want to measure the effects of class imbalances on SVM performance, so we use 5 training datasets of work size 51200, each with a different balance of classes, and we want to train two models on each, resulting in 10 simulations per task. Furthermore, we want to increase the iterations per training to 1500, resulting in a 3-times increase.

The total node hours that we are requesting is reflected in the table below:

	Test case A	Test case B
Simulations per task	30	10
Factor of Increase in iterations per simulation	4	3
node hours per iteration	8.0403	8.0403
Total node hours	965	241

Table 4: Justification of the resource request

References

[1] Edward Y. Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, & Hang Cui. textitPSVM: Parallelizing Support Vector Machines on Distributed Computers. Google Research, Beijing, China