SVM Project Parallel Design Proposal

Wesley Osogo, Tianfan Xu, Connor Buchheit, Peter Chen

Serial Implementation

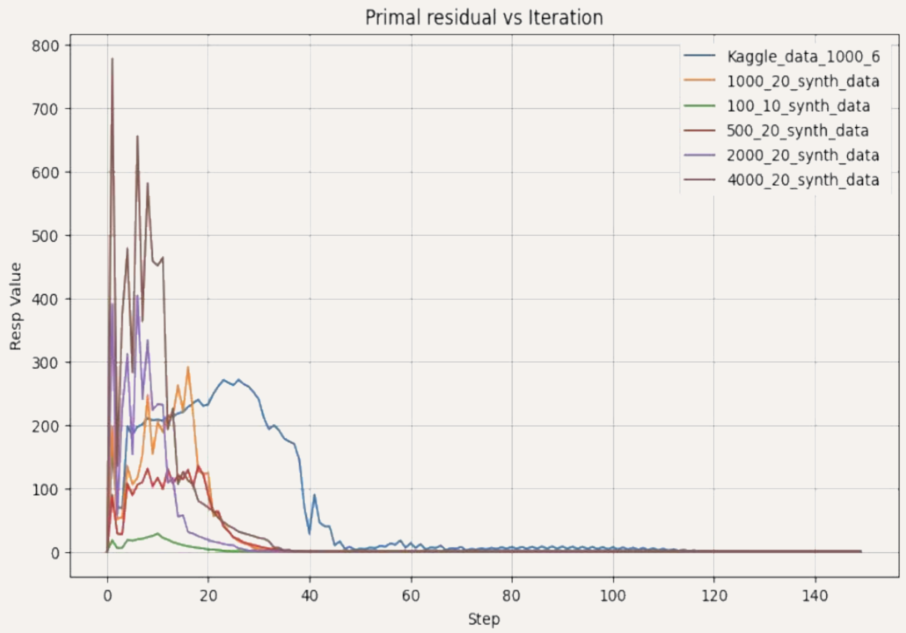
Input: $\mathbf{y}$ (labels), matrix $\mathbf{Q}$, parameters (params)
Output: solution vector $\mathbf{x}$

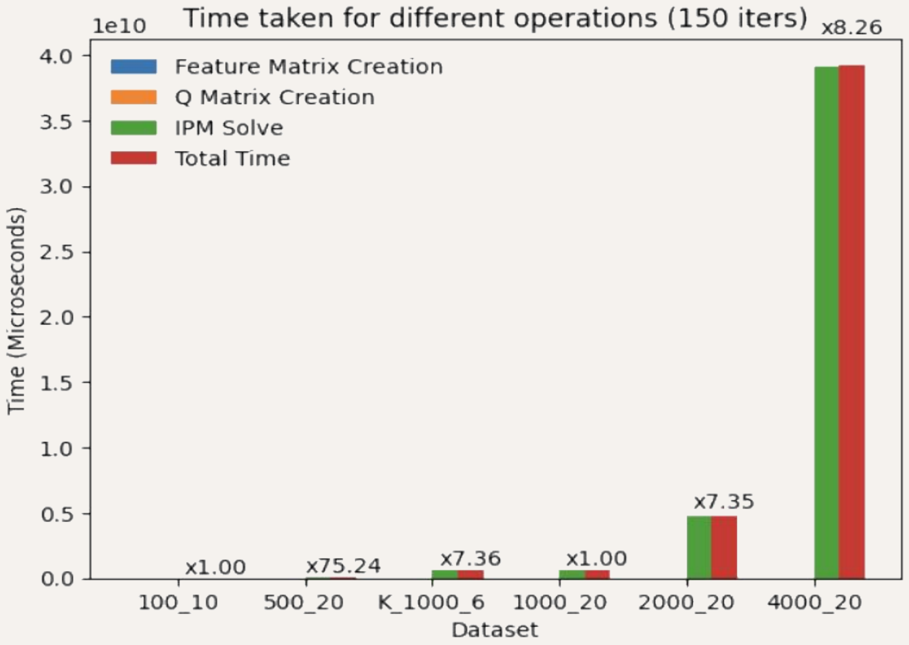
Variables:

- n: number of data points
- c_+, c_-: positive and negative class weights
- $\mathbf{x}$, λ, ξ: primal and dual variables
- $\mathbf{z}$: auxiliary vector for the objective gradient
- $\mathbf{y}$: input labels
- $\mathbf{C}$: upper bound for variables
- $\mathbf{\Sigma}$, $\mathbf{\Sigma}^{-1}$: matrices for the Newton step
- $\mathbf{\Sigma}^{-1}\mathbf{y}$, $\mathbf{\Sigma}^{-1}\mathbf{z}$: intermediate variables
- $\Delta\mathbf{x}$, $\Delta\lambda$, $\Delta\xi$: Newton step directions
- $\Delta\nu$: Newton step size for dual variable
- ν: dual variable
- η: surrogate duality gap
- r_{primal}, r_{dual}: primal and dual residuals
- t: barrier parameter
- α_{primal}, α_{dual}: primal and dual step sizes
- ϵ: small positive value to ensure numerical stability

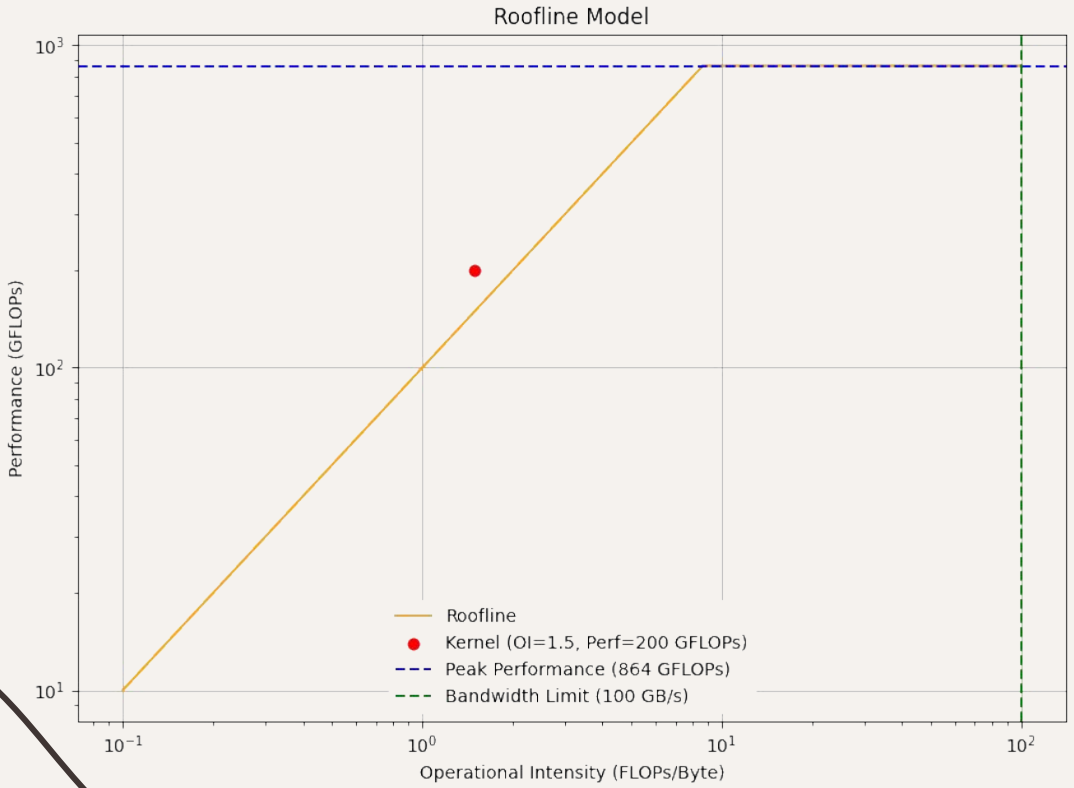
1. Initialize n to the size of $\mathbf{y}$.
2. Set c_+ and c_- using params.weight_positive, params.weight_negative, and params.hyper_parm.
3. Initialize $\mathbf{x}$, λ, ξ with zeros and $c_+/10.0$.
4. Set $\mathbf{y}$ to labels and $\mathbf{C}$ with zeros.
5. Initialize $\mathbf{\Sigma}$, $\mathbf{\Sigma}^{-1}$, $\mathbf{\Sigma}^{-1}\mathbf{y}$, $\mathbf{\Sigma}^{-1}\mathbf{z}$, $\Delta\mathbf{x}$, $\Delta\lambda$, $\Delta\xi$ with zeros.
6. Set $\Delta\nu$, ν, η, r_{primal}, r_{dual} to 0 and α_{primal}, α_{dual} to DBL_MAX.
7. Distribute data points to compute local fractions of $\mathbf{Q}$.
8. For $i = 0$ to $n - 1$ do:
   - Set $\mathbf{C}(i)$ to c_+ or c_- depending on the label.
   - Initialize $\lambda(i)$ and $\xi(i)$ to $c/10.0$.
9. For step $= 0$ to params.max_iter do:
   - Calculate η using ComputeSurrogateGap.
   - Set t using params.mu_factor, n, and η.
   - Calculate $\mathbf{D}$ using ComputeD with ξ, λ, $\mathbf{x}$, $\mathbf{C}$, and ϵ.
   - Calculate $\mathbf{z}$ using ComputeZ with $\mathbf{Q}$, $\mathbf{x}$, ν, $\mathbf{y}$, t, $\mathbf{C}$, and ϵ.
   - Update $\mathbf{\Sigma}$ using ComputeSigma with $\mathbf{D}$ and $\mathbf{Q}$.
   - Update $\mathbf{\Sigma}^{-1}$ by adding $\epsilon\mathbf{I}$ to $\mathbf{\Sigma}$ and inverting.
   - Update $\Delta\mathbf{x}$ using ComputeDeltaX.
   - Update $\Delta\lambda$ using ComputeDeltaLambda.
   - Update $\Delta\xi$ using ComputeDeltaXi.
   - Compute $\mathbf{\Sigma}^{-1}\mathbf{y}$ and $\mathbf{\Sigma}^{-1}\mathbf{z}$.
   - Update $\Delta\nu$ using ComputeDeltaNu.
   - Compute primal and dual residuals r_{primal} and r_{dual}.
   - Check for convergence and break if achieved.
   - Perform the updates for $\mathbf{x}$, ξ, λ, and ν.
   - Perform line search to find step sizes α_{primal} and α_{dual}.
10. Return $\mathbf{x}$ as the solution vector.

Serial SVM Experiments





Sequential Baseline Bottlenecks



I/O

Large datasets require intense I/O operations.

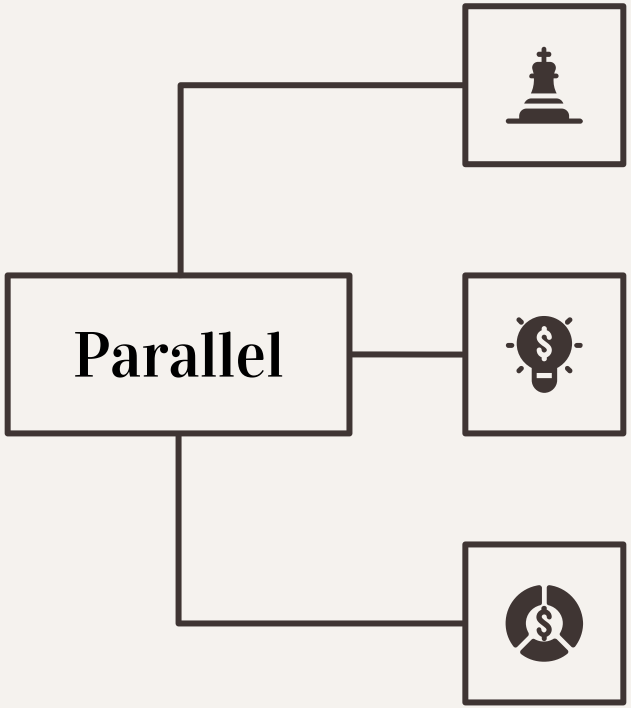
Q matrix

O(n^2) time complexity

Hyperparameter Tuning

Train models in parallel

Proposed Forms of Parallelism



Thread-Level Parallelism

In many of our IPM calculations, we can perform OpenMP reductions for loops without data dependencies and create parallel sections to parallelize different computations.

Distributed Memory Parallelism

Forming the Q matrix for the optimization step can be parallelized using MPI, as storing it is very memory intensive and distributing the work across many nodes reduces this major bottleneck. We can also use it for I/O parallelization.

Data Parallelism

Our kernel calculations involve many matrix operations. Thus, as a bonus, we can utilize CUDA for GPU programming to greatly speed up this process.

Parallel Implementation

1. Process and distribute the data.
2. Calculate the Q matrix (MPI) using kernel calculations and data (CUDA).
3. Perform the optimization algorithm (OpenMP), construct our model, and tune hyperparameters.

We have a synchronization point to ensure all nodes have received their input chunks. Keep Q distributed to minimize synchronization. For datasets that are heterogeneous in complexity, we will use dynamic scheduling.

Computational Steps

MPI

01 —— 02 —— 03 —— 04

OpenMP

CUDA

We have synchronization points when we update shared variables. We can use reductions and minimize use of critical sections to reduce communication overhead.

We have synchronization points to ensure the kernel calculations are complete before they are used in the Q matrix or optimization steps.

Thanks