Hybrid Parallelization of Alternating Direction Method of Multipliers (ADMM) Algorithm for LASSO.

John Salako

April 22, 2024

—

Parallel Computing (CMSE 822)

—

Professor Sean Couch

Project Summary

The repository contains the serial implementation of the ADMM algorithm for solving LASSO and the hybrid parallelization of the ADMM algorithm using OpenMP and MPI parallelization techniques.

Domain decomposition (distribution of the data matrix and vector) was done using the MPI processes, and the parallelized computations were implemented using OpenMP.

INTRODUCTION

BACKGROUND ON LASSO

High-dimensional data are often computationally expensive, and an efficient optimization strategy is often necessary. Hence, we often like to determine a smaller subset of predictors that exhibit the strongest effects, which is where the Least absolute shrinkage and selection operator (lasso) plays an important role as a shrinkage method better than the ridge regression (Hastie et al., 2009).

LASSO is a regression technique that includes a penalty (L1 regularization penalty) equivalent to the absolute value of the magnitude of coefficients. This constraint leads to some coefficients being exactly zero, effectively selecting the most significant variables. By focusing on these key variables, LASSO enhances the model's interpretability and predictive accuracy by producing simpler models that incorporate only the most relevant variables (Boyd et al., 2011).

The lasso problem in the equivalent Lagrangian form is defined by

$$\widehat{x_{\text{lasso}}} = \arg\min_{x} \left\{ \frac{1}{2} |Ax - b|_2^2 + \lambda |\mathrm{x}|_1 \right\}. \qquad \text{...(1)}$$

Where A is the data matrix, b is the outcome vector, x are the coefficients to be determined, and λ is the regularization parameter.
The minimization of the above expression formulates the lasso problem. However, unlike the ridge regression, which has L2 regularization and is easily computed via differentiation and equating to zero, the L1 penalty leads to nonlinear solutions in terms of vector b. There is no closed-form expression as in the ridge regression, which is because there are an infinite number of slopes (or solutions) associated with the L1.

Alternating Direction Methods Of Multipliers (ADMM)

Here is where the implementation of the alternating direction methods of multipliers (ADMM) comes into action. The ADMM algorithm solves the lasso problem in the form.

$$\textit{minimize} \;\; f(x) + g(z) \quad \textit{subject to } Ax + Bz = c \qquad \ldots(2)$$

Where $f(x) = \frac{1}{2}|Ax - b|_2^2$ and $g(z) = \lambda |\text{x}|_1$

Creating an augmented Lagrangian using the $||\,.\,||_2$ norm

$$L_p(x, z, y) = f(x) + g(z) + y^T(Ax + Bz - c) + \frac{\rho}{2}|Ax + Bz - c|_2^2 \qquad \ldots(3)$$

We can then solve via ADMM by initializing the following Lagrange multipliers (x, z, u). Where the dual-scaled variable, $u = \frac{y}{\rho}$, where $\rho > 0$. With this information, the ADMM can be implemented using the following algorithm.

Algorithm 1 ADMM Algorithm

```
Initialization:
Initialize variables $x$, $z$, and $u$ (Lagrange multipliers).
Set convergence criteria and $\rho$, the penalty parameter.
Iterations:
while iteration process <= iter_max do
x-update (using least squares):
$x^{k+1} \leftarrow (A^TA + \rho I)^{-1}(A^Tb + \rho(z^k - u^k))$
z-update (using soft thresholding):
    $z^{k+1} \leftarrow S_{\lambda/\rho}(x^{k+1} + u^k)$ where $S$ is the soft thresholding operator.
  u-update:
    $u^{k+1} \leftarrow u^k + x^{k+1} - z^{k+1}$
end while
```

The **convergence (stopping) criterion** is that the primal and dual residuals must be small. Such that

$$||\,r^k||_2 \le \epsilon^{pri} \;\text{ and }\; ||\,s^k||_2 \le \epsilon^{dual} \qquad \ldots(4)$$

The primal residual, r^k at iteration k, is defined as

$$r^k = Ax^k + Bz^k - c \qquad \ldots(5)$$

and the dual residual, s^k at iteration, k is defined as

$$s^k = \rho A^T B(z^k - z^{k-1}) \qquad \ldots(6)$$

The data preparation for this project was developed as follows:

- Matrix A was produced with a multivariate normal random variable with its rows normalized with L2 norm.
- Vector b was produced by the expression $Ax^{true} + \epsilon$. where x^{true} was sampled from $N(0, 1)$ with 100 nonzero entries, and $\in \sim N(0, 10^{-3})$
- Usually, irrespective of the matrix and vector dimensions, the convergence occurs at about 15 – 20 iterations for both the primal and dual residual. However, a max iteration of 25 was used to capture the continuous decline of the primal and residual norm and ignored the stopping criteria.

Computational Challenges associated with ADMM

Applying the ADMM algorithm to the LASSO problem presents several difficulties, including selecting appropriate penalty parameters and guaranteeing the method's convergence. Tuning these parameters is essential because it influences the convergence rate and the precision of the results. However, these parameters have been optimally selected, which will not be an issue in this project.

The iterative process inherent to ADMM can result in prolonged convergence times, given that ADMM is an O(n²)—**Quadratic Time**, particularly in scenarios involving poorly conditioned problems or when managing many variables. **The goal of this project is to parallelize the ADMM algorithm and reduce each iteration time, which will lead to convergence.**

METHODS

The serial implementation of the code was done using Algorithm 1 above, and the results of the implementation are presented below.

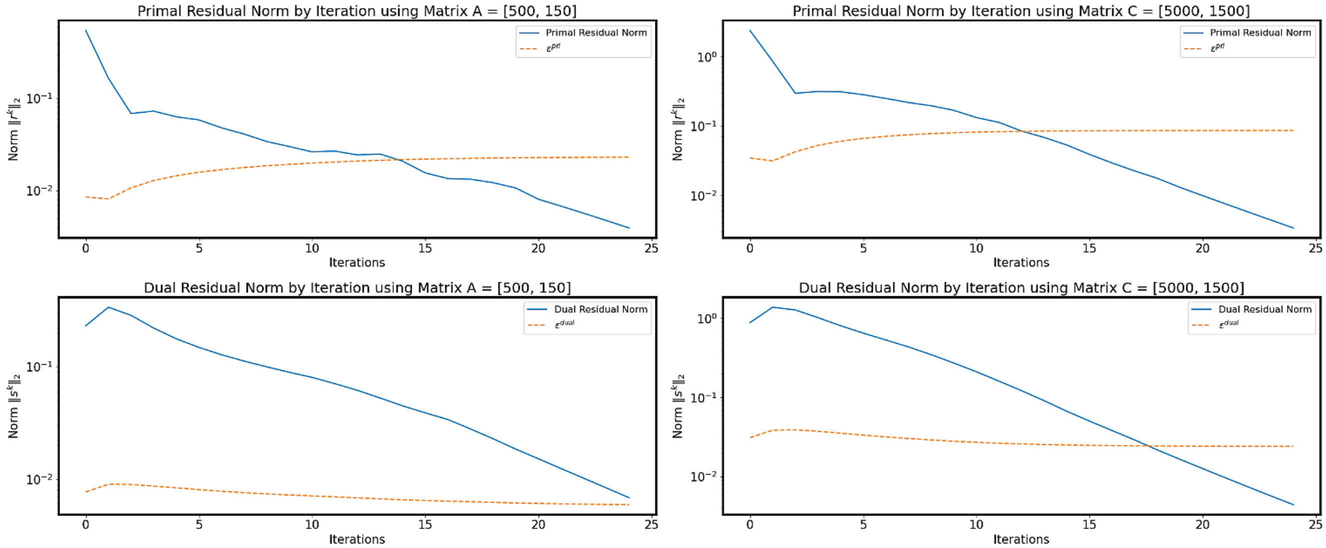

Figure 1: Showing the number of iteration leading to convergence for the ADMM.

Using a large dataset always results in faster convergence in terms of the number of iterations required for both the Primal and Dual Residual Norms. However, the time it takes to calculate each iteration when the input data is large increases quadratically, given by the O(n²) nature of the ADMM algorithm (graphically presented in Figure 2.

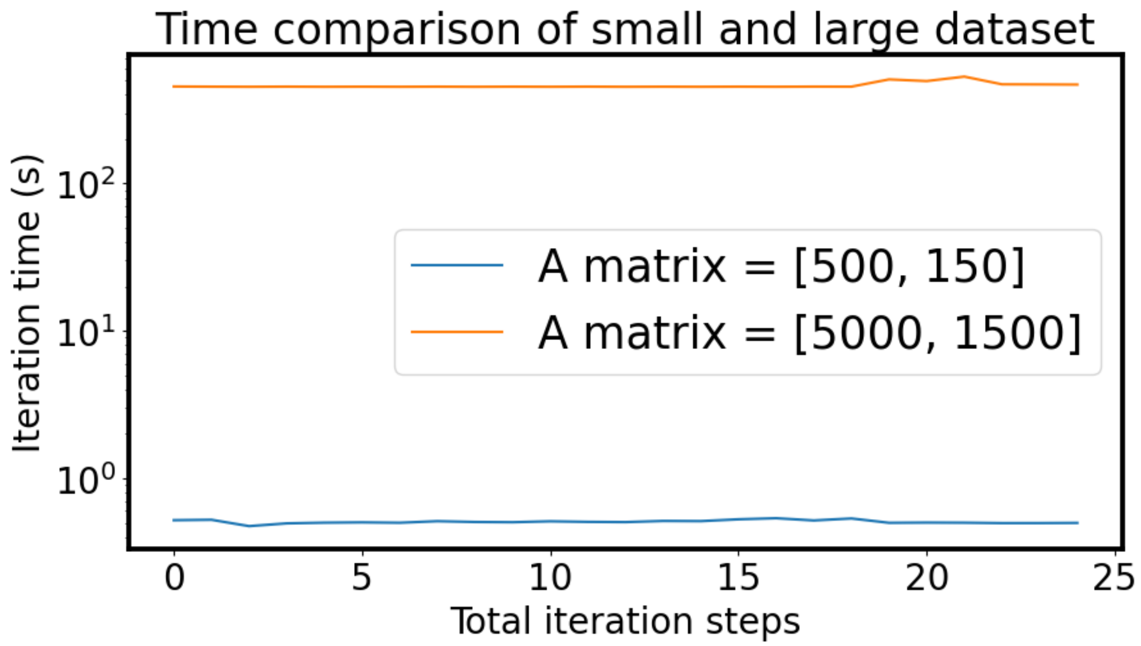

Figure 2: Showing the time taken for the completion of each iteration.

Recognizing that an increase in the input value leads to an enormous increase in the time it takes to complete each iteration, Parallelization of the ADMM code becomes imperative.

Parallelization Strategy

In this section, I employed the hybrid parallelization of the ADMM algorithm, where I distributed the Matrix and vector data among MPI processes using domain decomposition. Each process handles a subset of rows from matrix 'A' and the corresponding elements of vector 'b'. I then used OpenMP to parallelize operations such as z-update and u-update, which benefit from thread-level parallelism within each MPI process. The algorithm and pseudocode explaining these processes are captured in Algorithm 2, which are in exact sync with the parallel code (admm_parallel_hybrid.cpp) in the Github repository.

I employed dynamic load balancing and memory usage when distributing the data using domain decomposition, as seen in the code snippet below:

```
// shared variable
int row_size = m/(size);  // rows per rank
int rows_rem = m % (size);
// Dynamically assign the remainder rows to the last process
if (rank == size-1){
        row_size += rows_rem;
    }
```

The total number of tasks (rows, `m`) is divided by the total number of processes (`size`). Each process gets a base number of rows (`row_size`) to process. Since dividing `m` by `size` might not result in an even distribution, the remainder (`rows_rem`) is calculated. This is the number of rows left after distributing an equal share to all processes. The last process (when `rank == size - 1`) receives the remaining rows. This decision balances the load by ensuring all tasks are handled and leverages the last process to absorb any imbalance caused by the remainder.

This approach ensures that all processes are utilized efficiently, with the last process potentially having a slightly heavier load due to the extra rows. This implementation fulfills the project's dynamic load balancing requirement, which is also an important part of the data-parallel tasks by effectively balancing the load and memory usage.

Most of the parameters used in the code have been explained in the introduction section.

The algorithm detailing the hybrid parallelization is written below.

Algorithm 2 Hybrid MPI/OpenMP ADMM for Lasso

```
Initialize $x^{(0)}$,$z^{(0)}$,$u^{(0)}$ and parameters $\rho,\alpha$
Split data among MPI processes
for each iteration $k$ do
    parallel MPI sections
    for each subproblem in MPI process do
        Update $x_i^{(k+1)} \leftarrow \text{argmin} L_\rho(x_i, z^{(k)}, u^{(k)})$
        parallel OpenMP for
            Update $z_{(k+1)} \leftarrow \text{argmin} L_\rho(x_{(k+1)}, z, u_{(k)})$
        end parallel
        parallel OpenMP for
            Update $u_i^{(k+1)} \leftarrow u_i^{(k)} + \rho(x_i^{(k+1)} - z^{(k+1)})$
        end parallel
    end for
end for
```

Verification Test

From the algorithms presented above, one of the main reasons why the stopping (convergence) criteria were not used to stop the process was to present a similar chart, as we will see below. The plots showing the primal and dual residual norms, accompanied by their respective errors indicating at what iteration step the convergence actually happened, is a key approach to verifying the parallelization accuracy.

The success of the parallelization, as presented in the Results section below, shows that no matter the number of threads, latency, or communication overhead, the solution is independent of the number of parallel tasks and the specific parallelization strategy used.

Scaling Studies

This study utilized a robust scaling approach. Three matrix sizes were tested with a range of processing steps and on a range of nodes from 1 to 15 nodes to capture the influence of nodes' communication latency on the results.

The measurement performance used for the scaling studies are the speedup(S) and Efficiency (E).
As explained in the class and from selected literature:

- **Speedup (S)** measures the improvement in performance when using multiple processors compared to a single processor for the same task (Malony, 2011). It is defined as the ratio of the execution time on a single processor (T1) to the execution time on p processors (T_p).

$$\mathrm{S(p)} = \frac{T_1}{T_p} \qquad \text{...(7)}$$

- **Efficiency (E)** is the ratio of speedup to the number of processors, representing how effectively the parallel system utilizes the processors (Moreland & Oldfield, 2015). It is a measure of the fraction of time for which a processor is usefully employed.

$$\mathrm{E(p)} = \frac{S_p}{P} = \frac{T_1}{P \times T_p} \qquad \text{...(7)}$$

The analysis and presentation of the scaling approaches are presented in the next section.

RESULTS

Three matrix sizes were used to test the scalability and the success of the parallelization technique as compared to the serial results:

- Matrix A: [row=500, col=150], with 10 non-zero values for coefficients
- Matrix B: [row=1000, col=500], with 50 non-zero values for coefficients
- Matrix C: [row=5000, col=1500], with 100 non-zero values for coefficients.

The results presented below are from parallelization using MPI and Hybrid parallelization that integrates both OpenMP and MPI.

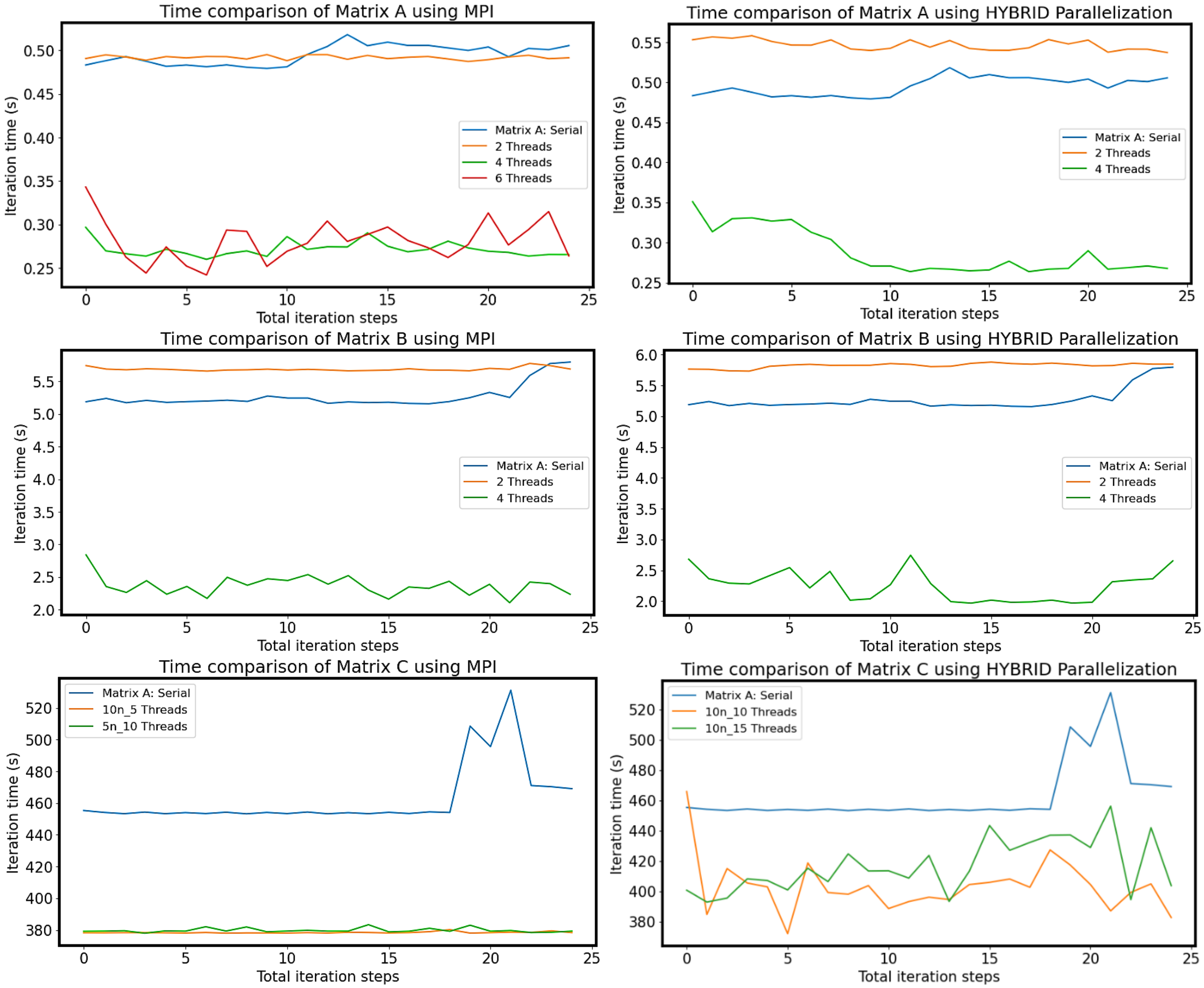

Figure 3: MPI and Hybrid Parallelization results using multiple Threads and multiple nodes

From the results above, strong scaling is observed across boards when the processes are increased from np = 2 to np =4. However, as the number of multiprocessors increases, the time begins to increase, and this can be attributed to the **communication cost** being a major limiter.

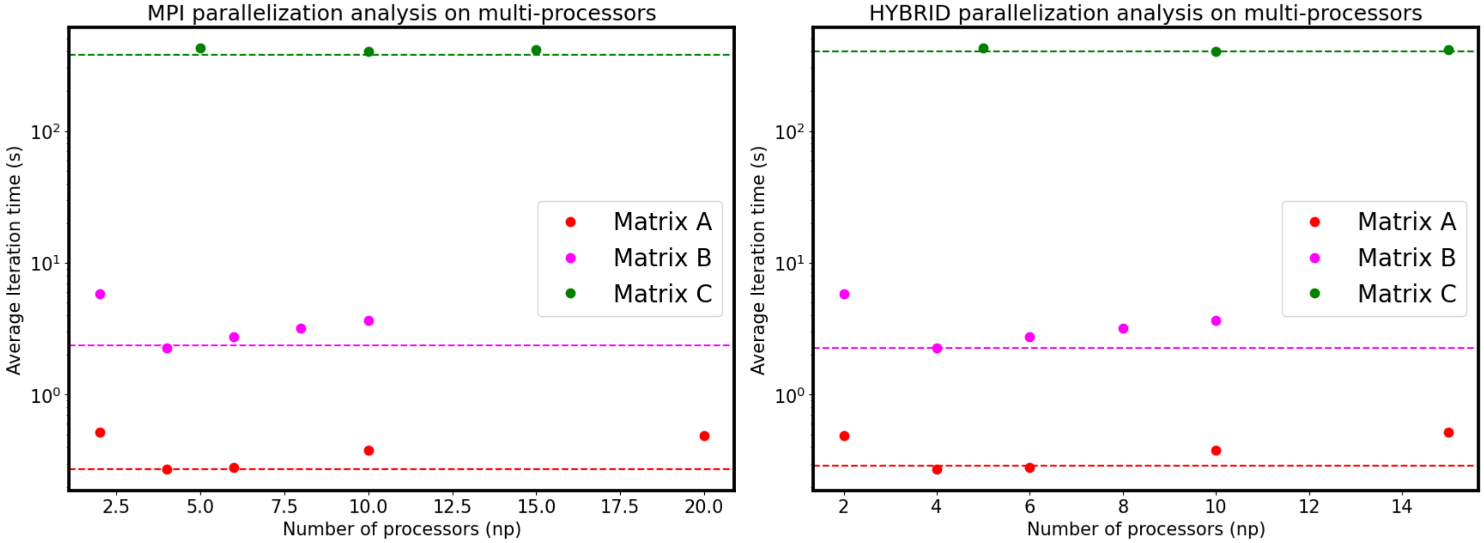

Figure 4: Showing Weak Scaling of the parallelization as the multi-processors increases.

The horizontal line drawn in the figure above was the minimum time value of the individual matrix.
As seen from both figures, np=4 had the lowest parallelization time and would be used to calculate the speed up and efficiency for both the Hybrid Parallelization and the MPI parallelization. The detailed analysis and calculation containing all figures and the memory usage calculations can be found in the Serial and parallel analysis Jupyter notebook. The figures provided in the table below are approximate average values.

Table 1: Showing the Memory Usage Calculations of the run time for the different input data

Parameters	**MPI MEMORY USAGE**			**HYBRID MEMORY USAGE**		
	Matrix A	Matrix B	Matrix C	Matrix A	Matrix B	Matrix C
T1	0.50	5.27	462.80	0.50	5.27	462.80
Tp	0.27	2.80	379.74	0.29	2.77	416.74
Speedup	1.82	1.88	1.22	1.72	1.90	1.11
Efficiency	0.46	0.47	0.30	0.43	0.48	0.28

While we had about twice the speed for most of the scenarios when np=4 as compared to the single processing, the efficiency still needs improvement.

As observed in the analysis above, following the scaling/performance studies and the memory usage analysis, the MPI parallelization seems to be just enough, and the hybrid parallelization does not seem to add many advantages except in Matrix B. The reason for this can be seen in the serial version of the written

ADMM algorithm. Here, I utilized the Eigen library, which is an optimized linear algebra library parallelized using OpenMP (Source: Link).

VERIFICATION TEST

Here, the simple test is to compare the Primal and Dual Residual plot for the serial version (Figure 1) with the MPI and the Hybrid Parallelization results. The holistic plots for all the data utilized in this research are in the Jupyter Notebook referenced in the analysis folder of this project. For this verification test, an example was selected to show the exact results obtained from the overall analysis of both MPI and OpenMP.

MPI Results

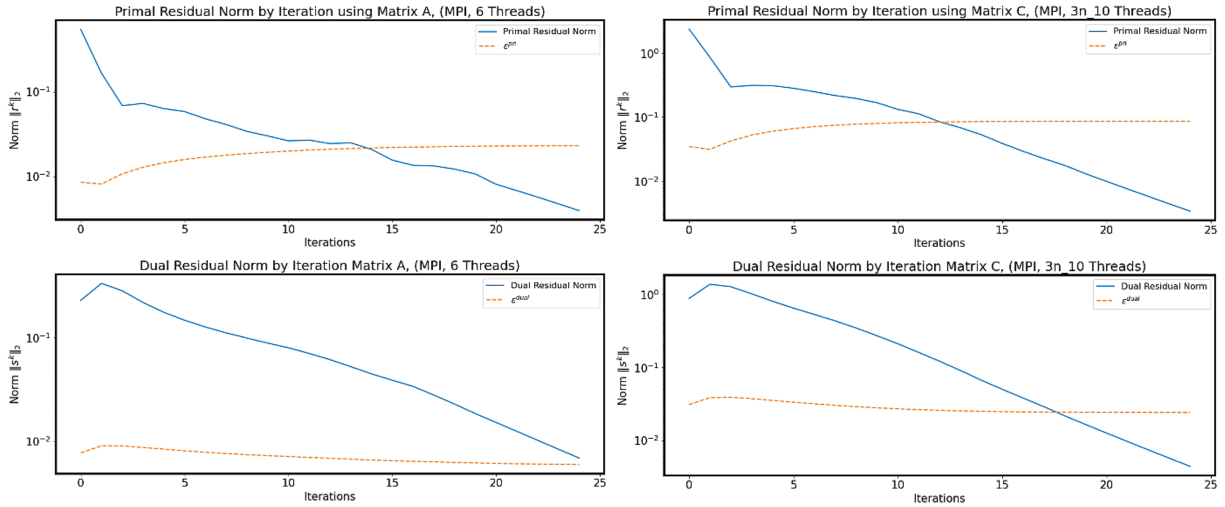

Figure 5: Validation Test for the MPI Parallelization technique showing similar with the serial code.

HYBRID Results

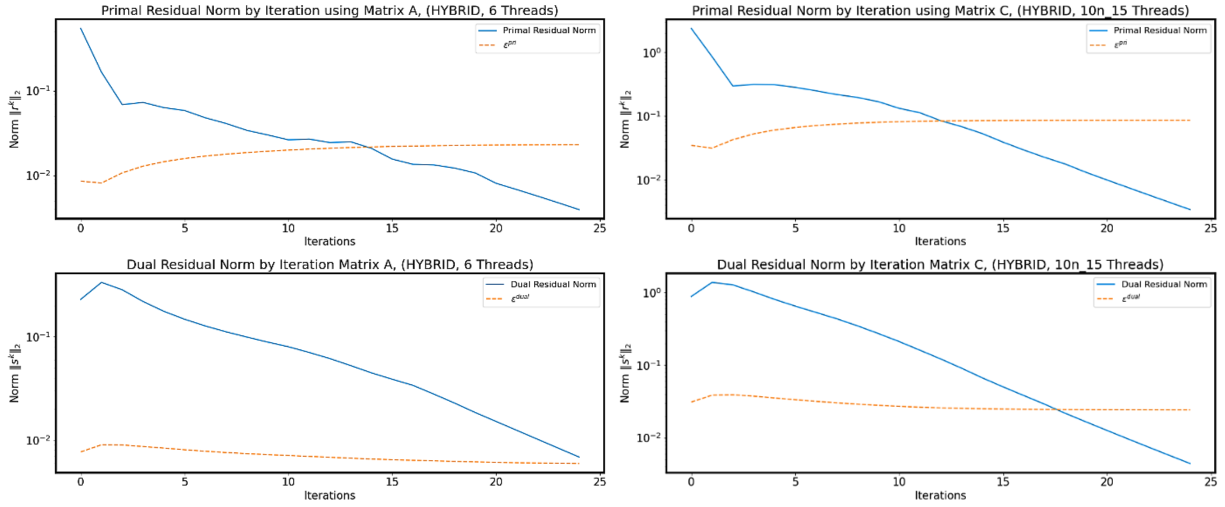

Figure 6: Validation Test for the HYBRID Parallelization technique showing similar the convergence with the iteration step in the serial code.

The results obtained from both the MPI parallelization and the HYBRID parallelization illustrate the correctness of my parallel solution, as seen in Figures 5 and 6. These parallelized codes were run on different multi-processors and nodes, yet they converged at the same iteration steps as the serial code, as displayed in Figure 1.

The detailed results for all the multi-processors used, including the different threads, are detailed in the Notebook. These results are important because they demonstrate that regardless of the number of parallel tasks and specific parallelization strategy used (MPI and HYBRID), the solutions remain the same.

CONCLUSIONS

The overall goal for this project was to parallelize the ADMM algorithm with the aim of reducing the individual iteration time required for convergence. I have presented the ADMM algorithm clearly, explaining the crucial parameters needed to solve LASSO via ADMM. Moreover, two parallelization strategies were employed, including strengths and weaknesses, and a comparison of both methods was itemized. The results of the parallelization strategy proved highly successful and showed that the algorithm both had weak and strong scaling.

The memory usage analysis showed the speedup time for the average iteration time (np=4) to increase by about twice the speed with a singular processor, fulfilling the goal of the project. On the other end, the efficiency of the code was approximately 50%, which can be attributed to the fact that the serialized version of the code was written in parallel, using a very optimized library like the Eigen C++ library that integrates the OpenMP parallelization.

Several scaling studies were conducted, such as using multi-processors in the same node, using multi-processors in multiple nodes, and also variating the input data size, which captures the success in parallelizing the code. The different parallelization techniques resulted in a higher average time as compared to the serialized version of the code.

The project was not absolutely perfect, as a more careful outlook can still be incorporated to ensure adequate scaling of the parallelized code, ensuring only strong scaling, surmounting the communication overhead hurdles.

Finally, this has been an interesting class for me, having never written C++ code before or understood parallelization. So, I say a big thank you, Professor Sean, for being such an astute and erudite professor. Also, thank you for assisting in debugging my codes as that gave me the needed confidence to complete this project.

REFERENCES

Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, *3*(1), 1–122.

Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction* (Vol. 2). Springer.

Malony, A. D. (2011). Metrics. In D. Padua (Ed.), *Encyclopedia of Parallel Computing* (pp. 1124–1130). Springer US. https://doi.org/10.1007/978-0-387-09766-4_69

Moreland, K., & Oldfield, R. (2015). Formal metrics for large-scale parallel performance. *International Conference on High Performance Computing*, 488–496.