Analysis and Implementation of Communication Avoiding SGD for Logistic Regression

Md Asifur Rahman

Nikhil Rajkumar

ABSTRACT

This project explores the implementations of optimizing machine learning models using Stochastic Gradient Descent (SGD) for logistic regression and the potential of parallel computing to enhance the optimization process. Specifically, it focuses on communication Avoiding Stochastic Gradient Descent (CA-SGD) in comparison to row-major parallel SGD and local SGD. The project aims to provide an implementation and theoretical analysis of CA-SGD, using the alpha-beta-gamma model. The study seeks to demonstrate the effectiveness of parallel computing in improving model accuracy and reducing training time, while highlighting the importance of careful consideration of communication overhead in parallel implementations.

ACM Reference Format:

Md Asifur Rahman and Nikhil Rajkumar. 2023. Analysis and Implementation of Communication Avoiding SGD for Logistic Regression. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Machine learning (ML) has become increasingly important in a wide range of applications in our day-to-day life. ML models are designed to learn patterns and relationships in the data by minimizing a loss function. However, the optimization process, which involves finding the optimal set of model parameters that minimize the loss function, is one of the key challenges in training machine learning models. To address this challenge, Stochastic Gradient Descent (SGD) has become a widely used optimization technique in the field of machine learning, gaining a lot of attention in recent years. SGD solves the optimization problem by using a random subset of the data to compute the gradient at each iteration, making it much faster and more scalable. To further enhance the optimization process, the rise of parallel computing has allowed for the distribution of optimization problems among processors and nodes, leading to quicker and more accurate convergence compared to sequential implementations.

Parallel computing has the potential to speed up the convergence of SGD, leading to faster training times and improved model accuracy by distributing the SGD optimization problem among multiple processors and nodes. There are two ways to implement parallel SGD: data parallelism and model parallelism. Data parallelism involves assigning different subsets of the training data to different processors, which compute gradients independently. These gradients are then aggregated and averaged to produce a single update that is applied to the model parameters. Model parallelism, on the other hand, involves assigning different parts of the model to different processors, which communicate their results to each other. By using parallel computing, it is possible to train much larger and more complex models than would be possible with sequential SGD alone. However, parallel computing requires careful consideration of communication overhead, as it can create a bottleneck and render the entire parallel scaling ineffective.

This project aims to review some of the state-of-the-art data parallelism techniques of SGD for solving a binary classification problem using logistic regression. Specifically, we will focus on analyzing Communication Avoiding Stochastic Gradient Descent (CA-SGD) [2] in comparison to the baseline row major parallel SGD [4] and local SGD [3]. In pursuit of this goal, we will (i) provide a thorough overview of these optimization techniques, (ii) theoretically analyze their computation and communication using the alpha-beta-gamma model, and (iii) conduct a comparative empirical study by implementing these three techniques [2–4].

2 BACKGROUND

In this section, we will first give a brief overview of the optimization problem for logistic regression and later discuss the optimization techniques in parallel settings to solve this optimization problem in details.

2.1 Logistic Regression

Logistic regression is a popular statistical technique used for binary classification problems, where the goal is to predict the class of a new sample based on its features. It is widely used in machine learning applications such as fraud detection, spam filtering, and medical diagnosis. Logistic regression models the probability of a sample belonging to a particular class using a logistic function, also known as the sigmoid function denoted by

$$\sigma(\theta) = \frac{e^{\theta}}{1+e^{\theta}} \equiv \frac{1}{1+e^{-\theta}} \tag{1}$$

Given a dataset $A \in \mathbb{R}^{m\times n}$ with m samples and n features, and corresponding target labels $y \in \mathbb{R}^m$ specifying the class of each sample such that $y_i \in -1, +1; i = 1, \ldots, m$, the objective is to learn an optimal set of parameters $x \in \mathbb{R}^n$ that corresponds to each feature and maximizes the overall probability of correctly classifying the input data. So the probability can be modeled using the sigmoid function as

$$P(y_i|a_i x) = \begin{cases} \sigma(a_i x) & y_i = +1 \\ 1-\sigma(a_i x) & y_i = -1 \end{cases} \tag{2}$$

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Here, the sigmoid function $\sigma(.)$ denotes the probability of the i-th sample a_i from the dataset A belonging to a particular class while evaluated with the learnable weight parameter x. Furthermore, since $1 - \sigma(a_i x) = \sigma(-a_i x)$, the logistic function in Eq (2) can be written as

$$P(y_i|a_i x) = \sigma(y_i a_i x) \tag{3}$$

So the optimization problem for logistic regression can be defined as

$$\underset{x}{argmax} \prod_{i=1}^{m} \sigma(y_i a_i x) = \underset{x}{argmax} \prod_{i=1}^{m} \frac{1}{1 + exp(-y_i a_i x))} \tag{4}$$

Here, the learning problem is to find optimal weights x that maximize the likelihood function. By taking the negative log of likelihood Eq (4) can be written as

$$\underset{x}{argmin}\ F(A, x, y) \tag{5}$$

where,

$$F(A, x, y) = \frac{1}{m} \sum_{i=1}^{m} log(1 + exp(-y_i a_i x)) \tag{6}$$

But this does not have a closed-form solution like linear regression and cannot be solved using direct methods. However, we can iteratively update the gradient until the solution converges, where the gradient with respect to x is denoted as:

$$\nabla F(A, x, y) = \frac{1}{m} \sum_{i=1}^{m} \frac{-\tilde{a_i}^T}{1 + exp(\tilde{a_i} x)} \tag{7}$$

Where, $\tilde{a_i} = y_i a_i \ \forall i = 1, ..., m$, this means that the rows of A are scaled by their corresponding labels. We use the following representation for the scaled matrix, $\tilde{A} = A \circ y$, where $\circ$ represents scaling the i-th row of A by the same i-th element of vector y. This allows us to rewrite Eq (7) as:

$$\nabla F(A, x, y) = -\frac{1}{m} \tilde{A}^T (\vec{1} \oslash (\vec{1} + exp(\tilde{A}x)) \tag{8}$$

Here $\oslash$ denotes elementwise division, exp(.) is the exponential function applied elementwise to the vector $\tilde{A}x$, and the sigmoid function is represented as $sig(\tilde{A}x) = \vec{1} \oslash (\vec{1} + exp(\tilde{A}x))$, which is applied to vector the $\tilde{A}x$. We can now rewrite Eq (8) as:

$$\nabla F(A, x, y) = -\frac{1}{m} \tilde{A}^T sig(\tilde{A}x) \tag{9}$$

Finally weight x can be learned iteratively by updating the weight x_i at iteration i as:

$$x_i = x_{i-1} - \eta_i \nabla F(A, x_{i-1}, y) \tag{10}$$

x_{i-1} is the solution vector from the previous update iteration and η is the learning rate which determines by how much we want to move the solution in the direction of the gradient. For the purpose of this project, we are considering a fixed learning rate η over all iterations.

This is the same math that goes into Gradient Descent (GD). SGD computes this same gradient but only for a subset of rows (data points) in A. Technically speaking, SGD requires only a single row per iteration. However, we can generalize it to a tunable batch size of b rows that are sampled at random from the dataset A. Thus, the update in Eq (10) with respect to randomly sampled b rows can be written as:

$$x_i = x_{i-1} - \eta_i \nabla F(\mathbb{I}_i A, x_{i-1}, \mathbb{I}_i y) \tag{11}$$

Here, $\mathbb{I}_h \in \mathbb{R}^{bxm}$ denotes the b rows sampled uniformly at random without replacement from the identity matrix, $\mathbb{I}_m$. If b=m, SGD becomes equivalent to GD where all rows of A are used in computation at each iteration.

In applications where low accuracy solution is sufficient SGD is preferable over GD as it can provide faster convergence. This is because, when $b << m$, SGD requires $\frac{m}{b}$ factor less computation in each iteration than GD. In a parallel computing setting where data is distributed across multiple processors, both GD and SGD algorithms require communication during each iteration. However, SGD requires a factor of $\frac{m}{b}$ more rounds of communication compared to GD as it requires $\frac{m}{b}$ times more iterations. This means that, on modern parallel hardware where communication cost is often high, SGD requires orders of magnitude more communication than GD. Several research over the years has proposed various algorithm to perform SGD in parallel computing setting. Next, we will discuss three particular parallel algorithms.

2.2 Row Major Parallel SGD

Row major parallel SGD was introduced in [4]. This technique splits the data A over P processors so each processor has $\frac{m}{P}$ rows and n columns. At each iteration, each processor updates its local weight x by sampling data (of batch size 'b' from its local data split) and calculating the corresponding gradient. Afterward, a globally averaged weight is evaluated at every iteration by communicating the local weights from each processor and is then conveyed to every processor to be used in the next iteration. The pseudocode for parallel row major SGD is given in Algorithm 1.

Algorithm 1 Row major parallel SGD

Input: Data $A \in \mathbb{R}^{m \times n}$, label $y \in \mathbb{R}^m$, batch size b , max no of iteration T , learning rate η
$P :=$ number of processors
Scatter y to P processor i.e $y_k = \mathbb{R}^{\frac{m}{P} \times 1}; k = [1, .., P]$
Scatter samples of A across P processors such that the local copy $A_k \in \mathbb{R}^{\frac{m}{P} \times n}; k = [1, .., P]$
Initialize local weights in each processor as $x_k \in \mathbb{R}^{n \times 1}; k = [1, .., P]$
for $t = 1, 2, \ldots T$ **do**
    $idx_k = \text{random}(0, \frac{m}{P} - b)$
    $B_k = A_k[idx : (idx + b), :]$
    $s_k = B_k \circ y_k[idx_k : (idx_k + b)]$
    $u_k = s_k \cdot x_k$
    $h_k = sigmoid(u_k)$
    Evaluate $\nabla_k = B_k^T \cdot h_k$
    Update local weight $x_k = x_k + \frac{\eta}{b} \nabla_k$
    All reduce to globally update weight $x_k = \frac{1}{P} \sum_k^P x_k$
end for
Output: Weight x

Clearly, there is a factor of T communication required in row-major parallel SGD.

2.3 Local SGD

As we have discussed previously that row major SGD suffers from the problem of high communication overhead, as each iteration requires communication among all processors. Local SGD has been proposed in [3] as a solution to this problem by reducing communication overhead while maintaining convergence guarantees. This approach allows each processor to update its weights based on its local subset of the data and then periodically synchronize with other processors to share information. In this way, local SGD strikes a balance between the convergence speed of traditional parallel SGD and the communication efficiency of sequential SGD. Pseudocode of local SGD is given in Algorithm 2.

Algorithm 2 Local SGD

Input: Data $A \in \mathbb{R}^{m \times n}$, label $y \in \mathbb{R}^m$, batch size b , max no of iteration T , local iteration τ, learning rate η
$P :=$ number of processors
Scatter y to P processor i.e $y_k = \mathbb{R}^{\frac{m}{P} \times 1}; k = [1, .., P]$
Scatter samples of A across P processors such that the local copy $A_k \in \mathbb{R}^{\frac{m}{P} \times n}; k = [1, .., P]$
Initialize local weights in each processor as $x_k \in \mathbb{R}^{n \times 1}; k = [1, .., P]$
for $t = 1, 2, \ldots T$ **do**
  for $k = 1, 2, \ldots \tau$ **do**
    $idx_k = \text{random}(0, \frac{m}{P} - b)$
    $B_k = A_k[idx : (idx + b), :]$
    $s_k = B_k \circ y_k[idx_k : (idx_k + b)]$
    $u_k = s_k \cdot x_k$
    $h_k = sigmoid(u_k)$
    Evaluate $\nabla_k = B_k^T \cdot h_k$
    Update local weight $x_k = x_k + \frac{\alpha}{b} \nabla_k$
  end for
  All reduce to globally update weight $x_k = \frac{1}{P} \sum_k^P x_k$
end for
Output: Weight x

Since local SGD performs the gradient descent over its local data for τ step before establishing communication to compute the global weight, it actually reduces the communication requirement by a factor of $\frac{1}{\tau}$. Further discussion on the computation and communication analysis has been given in section 3.

2.4 CA SGD

CA-SGD was proposed in [2] that reduces communication by reorganizing the SGD computation in such a way that it only requires communication every s iteration instead of every iteration. To derive the re-organized form of SGD we will use the following form where a fixed learning rate η has been used.

$$x_i = x_{i-1} + \frac{\eta}{m} (\mathbb{I}_i \tilde{A})^T sig(\mathbb{I}_i \tilde{A} x_{i-1}) \tag{12}$$

We can unroll the recurrence for the next iteration so that x_{i+1} can be written in terms of x_{i-1} as

$$x_{i+1} = x_{i-1} + \frac{\eta}{m} [(\mathbb{I}_i \tilde{A})^T sig(\mathbb{I}_i \tilde{A} x_{i-1}) + (\mathbb{I}_{i+1} \tilde{A})^T sig(\mathbb{I}_{i+1} \tilde{A} x_{i-1} + \frac{\eta}{m} \mathbb{I}_{i+1} \tilde{A} (\mathbb{I}_i \tilde{A})^T sig(\mathbb{I}_i \tilde{A} x_{i-1}))] \tag{13}$$

In Eq (14) we can see the term $sig(\mathbb{I}_i \tilde{A} x_{i-1})$ which is actually computed in Eq(12) and thus can be resused. So the re-organized SGD update requires only one $sig(.)$ update per iteration as we can reuse the other one from the previous calculation. We can generalize this calculation using a sequence of s solution vector by changing the iteration counter from i to $sh + j$ as

$$x_{sh+j} = x_{sh} + \sum_{i=1}^{j-1} \frac{\eta}{m} [(\mathbb{I}_{sh+i} \tilde{A})^T sig(\mathbb{I}_{sh+i} \tilde{A} x_{sh+i}) + (\mathbb{I}_{sh+j} \tilde{A})^T sig(\mathbb{I}_{sh+j} \tilde{A} x_{sh} + \sum_{i=1}^{j-1} \frac{\eta}{m} \mathbb{I}_{sh+j} \tilde{A} (\mathbb{I}_{sh+j} \tilde{A})^T sig(\mathbb{I}_{sh+i} \tilde{A} x_{sh+i}))] \tag{14}$$

Because of the nonlinear $sig(.)$ operation Eq (14) can not be simplified and requires matrix-matrix multiplication as well as matrix-vector multiplication. The overall CA-SGD technique is presented in Algorithm 3. Further analysis of CA-SGD has been provided in the next section.

Algorithm 3 CA-SGD

Input: Data $A \in \mathbb{R}^{m \times n}$, label $y \in \mathbb{R}^m$, batch size b , max no of iteration T , s-step s , learning rate η
$x_0 = \vec{0}, \tilde{A} = A \circ y$
for $t = 1, 2, \ldots \frac{T}{s}$ **do**
  for $j = 1, 2, \ldots s$ **do** $\{i_k \in [m] | k = 1, 2, ..., b\}$ uniformly at random without replacement
    $\mathbb{I}_{sh+j} = [e_{i_1}, e_{i_2}, ..., e_{i_b}]^T$ where e_{i_k} is the k-th standard basis vector.
  end for
  Let $Y = \begin{bmatrix} \mathbb{I}_{sh+1} \\ \mathbb{I}_{sh+2} \\ .... \\ \mathbb{I}_{sh+s} \end{bmatrix}$
  $G = YY^T$
  $r = Yx_{sh}$
  for $j = 1, 2, \ldots s$ **do**
    Update x_{sh+j} using Eq.14.
  end for
end for
Output: x_T

3 COMMUNICATION AND COMPUTATION ANALYSIS USING ALPHA-BETA-GAMMA MODEL

For analysis purposes, we are going to use the Alpha-Beta-Gamma model which is a way for us to analyze the computation and communication costs of a parellel SGD variant. The computation cost is the standard Big-Oh analysis and calculates the FLOPs(Floating

Point Operations - γ) While the communication costs are proportional to Latency (α) and Bandwitdh (β). Therefore, our overall model looks like this:

$$T(n) = \gamma \cdot (FloatingPointOps) + \alpha \cdot (NumberOfMessages) + \beta \cdot (MessageSize) \quad (15)$$

Where γ is the Seconds per flop, α is the seconds per message and β is the seconds per byte moved (inverse of link bandwidth)

3.1 Row Major SGD

Communication costs:
First, we will analyze the latency (α). In the implementation of row SGD a single MPI All_Reduce call is used to communicate the vector. MPI creates a binomial tree when communicating, and to traverse that tree it takes O(log_2P) time. This happens over 'k' iterations and therefore the latency for row SGD is $O(log_2P * k)$. Next, we look into the bandwidth cost. For this, we need to evaluate how much data is transferred overall. Since at each iteration, we have an MPI All_Reduce call to send an updated x. This vector is of length n and happens k times. Therefore, the bandwidth cost is $O(n * k)$
Computation costs:
Over each iteration, gradient is calculated followed by an update to the weight vector. This costs $O(\frac{bn}{P} * k)$ FLOPs.

Finally, by combining these 3 metrics we get the cost according to Alpha-Beta-Gamma model as:

$$T(n) = \alpha \cdot O(log_2P * k) + \beta \cdot O(n * k) + \gamma \cdot O(\frac{bn}{P} * k)$$

3.2 Local SGD

Communication costs:
First, we will analyze the latency (α). In the implementation of local SGD a single MPI All_Reduce call is used to communicate the vector. This happens every τ iterations and therefore the latency for local SGD is $O(\frac{k}{\tau} * log_2P)$. Next, we look into the bandwidth cost. For this, we need to evaluate how much data is transferred overall. Since at every τ iterations, we have an MPI All_Reduce call to send an updated x. This vector is of length n and happens $\frac{k}{\tau}$ times. Therefore, the bandwidth cost is $O(\frac{k}{\tau} * n)$.
Computation costs:
Over each iteration, the gradient is calculated followed by an update to the weight vector. There is also the summing of weight vectors locally from each factor. Therefore, the cost is very similar to row major and costs $O(\frac{k}{\tau} * (\frac{\tau bn}{P} + \tau n))$.

Finally, by combining these 3 metrics we get the cost according to Alpha-Beta-Gamma model as:

$$T(n) = \alpha \cdot O(\frac{k}{\tau} * log_2P) + \beta \cdot O(\frac{k}{\tau} * n) + \gamma \cdot O(\frac{k}{\tau} * (\frac{\tau bn}{P} + \tau n))$$

3.3 CA-SGD

Communication costs:
For the communication-avoiding variant, we have lesser communication in terms of latency. The overall latency goes to $O(\frac{k}{s} * log_2P)$. Here 's' is the tunable parameter that we set. Bandwidth is now a little more and goes to $O(kbn + kb + k\frac{n}{s})$. This is because we now communicate $O(sbn + sb + n)$ words per outer iteration. We have an MPI All_Gather to compute the Gram matrix, G, and also an All_Reduce to communicate the weights finally.
Computation costs:
Lastly, we move onto the computation costs for CA-SGD. This is quite expensive as there is a lot of math taking place in order to save communication costs (which is why the latency is low). Calculating 'Y' takes $O(\frac{sbn}{P})$ FLOPs. Computing the Gram Matrix takes $O(\frac{s^2b^2n}{P})$ FLOPs. We consider ω to be the cost of computing the sigmoid of a single element. The Sigmoid function takes $O(\frac{s^2b^2+\omega sb}{P})$ FLOPs. Our gradient computation costs $O(\frac{sbn}{P})$, summing these gradients takes $O(sn)$. Our overall cost adds up to be, $O(\frac{ksb^2n}{P} + kn + \frac{ksb^2}{P} + \frac{k\omega b}{P})$
Finally, by combining these 3 metrics we get the cost according to Alpha-Beta-Gamma model as:

$$T(n) = \alpha \cdot O(\frac{k}{s} * log_2P) + \beta \cdot O(kbn + kb + k\frac{n}{s}) + \gamma \cdot O(\frac{ksb^2n}{P} + kn + \frac{ksb^2}{P} + \frac{k\omega b}{P})$$

4 EXPERIMENT

4.1 Implementation

We implemented row SGD, local SGD and CA-SGD using C-programming language. We used the open-source OpenMPI library for parallel computing. Implementation of all the algorithms and their experimental data can be found at [source code and result link].

4.2 Dataset Description

In order to conduct a comparative analysis of the algorithms we used the dataset from LibSVM tools[1] in our experiments. This dataset contains data from detection of non-coding RNAs on the basis of predicted secondary structure formation free energy change. It contains 59,535 rows, but we omit a few rows to make it 59,532. This is done so that it is equally divisible by 2,4 and 6 processors. This helps us when we analyze algorithms amongst different processors. It also contains 8 features (2 integers and 6 doubles), we remove the integer features and keep the doubles only. Therefore, we have 6 features and 59,532 rows in total.

4.3 Comparative Objective Value Analysis

In our first experiment, we try to figure out which algorithm converges faster over the iterations by analyzing the corresponding objective values. During the experiment, we fixed the hyperparameters as shown in Table 1. Furthermore, for local SGD, we set the local iteration $\tau = 10$. For CA-SGD, we run it with 3 values of s. The experimental result is shown in fig.1.

Table 1: Hyper-parameters

Iterations	10000
Learning Rate	0.001
Batch Size	200
No of parallel processor	4

From Fig. 1, we can see that when s=1, CA-SGD behaves similarly to row-major SGD. However, for s=2, we notice that CA-SGD converges much faster than row-major SGD and local SGD. Finally, for

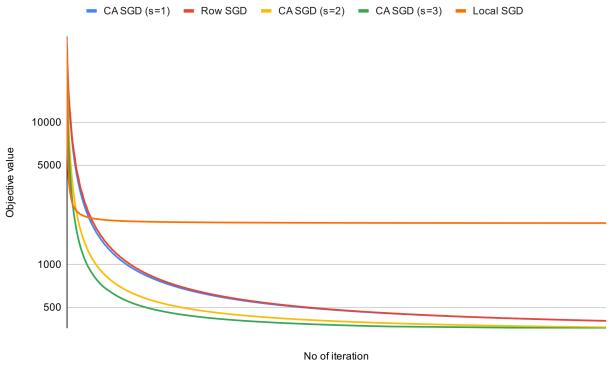

Figure 1: Comparison of objective values.

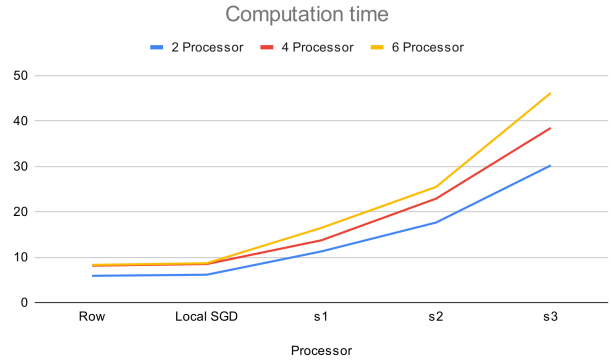

Figure 2: Comparison of computation time

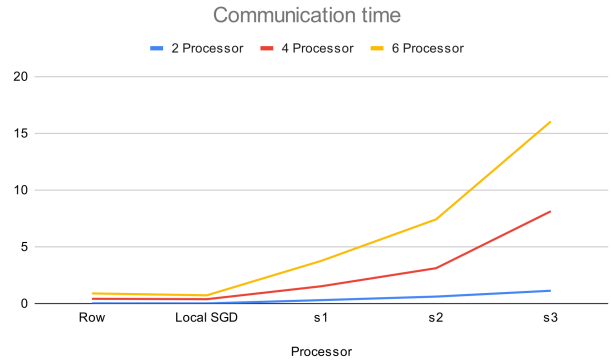

Figure 3: Comparison of communication time

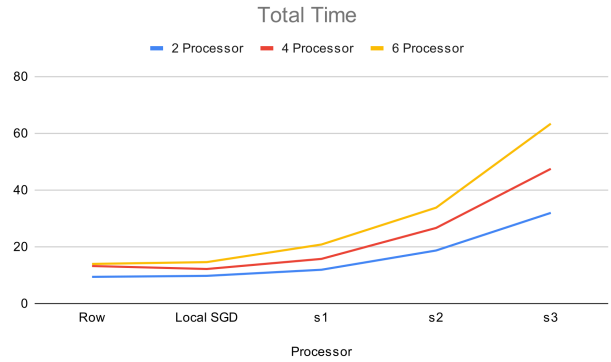

Figure 4: Comparison of total time

s=3, convergence is even faster. In addition, CA-SGD with s=2 and s=3 also attained the lowest objective function value. Meanwhile, we can see that though local SGD decreased quite quickly at first, it was not able to attain a lower objective value as it kept revolving around similar values.

4.4 Comparative Time Analysis

In this experiment, we analyze the computation and communication costs by measuring the time taken for running the algorithms. We used the OpenMPI function, WTime(), to evaluate the time by placing it between computation and communication function calls. The time cost is calculated as the difference between the start and end of WTime(). We measured three times for our experimental purpose: total_time, computation_time, and communication_time. Total_time is measured from the start of allocating memory until the last step of freeing it, while the other two variables are incremented between computation (e.g., sigmoid, gradient) and communication (e.g., all_reduce, all_gather) function calls. During this experiment, we kept the other hyperparameters of the algorithms the same as before, as shown in Table 1, except for the number of processors. We varied the number of processors to 2, 4 and 6 while taking time measurements. The comparative result for computation time, communication time, and total time has been shown in fig.2, fig.3, and fig.4 respectively.

As expected, we notice an increase in communication times as the number of processors increases, irrespective of algorithm variant. We observe that CA-SGD also shows higher communication time in comparison to row-major SGD and local SGD, due to the small number of features $n = 6$ in the dataset. However, with a large feature space, CA-SGD is expected to provide better communication costs. Meanwhile, we also notice that the CA-SGD variant has the longest computation time as well as total time, which increases with an increase in s. This can be attributed to additional calculations required in matrix-matrix multiplication to evaluate the gram matrix.

5 CONCLUSION

In this project, we have successfully implemented CA-SGD for logistic regression and compared it to two baseline algorithms, row-major SGD and local SGD. We analyzed the communication and computation costs of each algorithm and observed the expected

increase in communication time as the number of processors increased. We also found that CA-SGD had higher communication time than row-major SGD and local SGD due to the small feature dimension of our dataset, but we anticipate that CA-SGD will perform better on datasets with larger feature spaces. Our implementation accurately reflected our analyses, and we plan to conduct experiments on more powerful computing systems with larger datasets. One major challenge we faced was that our local systems struggled to handle large iterations combined with large values of 's'. With better processing power, we can overcome this issue and implement CA-SGD for a wider range of datasets, longer iterations, and varied values of 's'.

REFERENCES

[1] [n. d.]. LibSVM . https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/cod-rna. Accessed: 2023-05-01.

[2] Aditya Devarakonda and James Demmel. 2020. Avoiding Communication in Logistic Regression. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 91–100.

[3] Sebastian U Stich. 2018. Local SGD converges fast and communicates little. *arXiv preprint arXiv:1805.09767* (2018).

[4] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized stochastic gradient descent. *Advances in neural information processing systems* 23 (2010).