ASYNCHRONOUS PARALLEL GRADIENT BOOSTING USING A PARAMETER SERVER APPROACH

Guanfu Liu, Dimitris Konstantinidis

Andrew id: guanful, dkonstan

1 SUMMARY

We implemented a gradient boosting decision tree (gbdt) algorithm in an asynchronous framework using parameter server. The algorithm is implemented in a shared memory environment, while the idea can be readily adapted to message passing programming model. Our analysis of the run time and convergence rate shows the async-gbdt implementation significantly speeds up gbdt training while at the same time preserves good accuracy. On the 16-core GHC machine, async-gdbt achieves 25-30x speedup over sequential implementation and 2-4x speedup over the fork-join implementation using OpenMP on average, depending on the sparsity of the dataset.

2 BACKGROUND

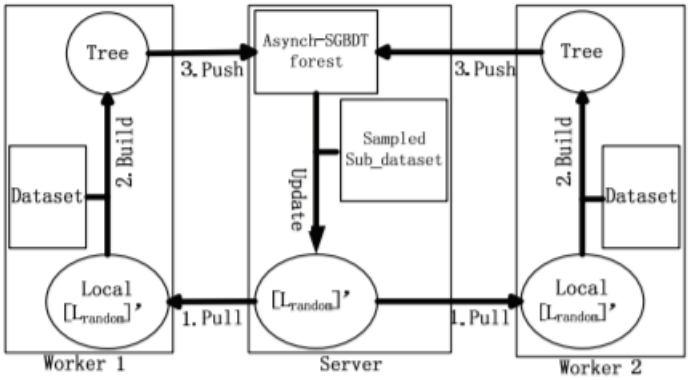


Figure 1: The Parameter Server Diagram

Gradient boosting is a machine learning algorithm that ensembles weak learners like decision stumps and improves accuracy. The basic idea is to incrementally train a model to minimize the empirical loss function over the function space by fitting a weak learner that points in the negative gradient direction. Many researchers have proposed various ways to parallelize GBDT algorithm by generating good subsample of the original dataset and have worker nodes train weak learners, usually decision trees, on each subset. Explicit synchronization will be done at the end of every iteration to aggregate all trees built. However, this fork-join paradigm fails to scale as

a small number of slow worker nodes can significantly slow down the training. For example, LightGBM, the state-of-art parallel GBDT framework, usually only achieve 5x to 7x speedup on a 32-core machines.

That's where the asynchronous parallel GBDT with parameter server framework comes to rescue: the server receives trees from workers, and workers build trees on subsamples of dataset asynchronously, which allows overlapping of communication and computation time and therefore hiding latency.

The major data structure we deal with is CART tree object composed of a collection of nodes. There are two key operations associated to it: fitting and prediction. To fit a CART tree, we need to loop over the feature universe, if without sampling, and split on each feature to pick the best one in terms of reducing the node impurity. When a predictions needs to be made, each tree goes down all of its nodes deciding on which children node to follow based on the characteristics of the feature until it reaches a leaf, at which point it returns a prediction based on the information stored on the leaf node. Fitting the CART tree is the most expensive computation in our problem, especially when the dataset is sparse with thousands of features. Additionally, in each iteration of the code, we have to build a large number of independent decision trees from a random subset of the data set in order to bolster our forest. Since building a tree object in itself is computationally costly and randomly creating a sample of a huge data set takes both time and memory resources, the learning algorithm takes the longest in the Fit function of the GBDT class which includes both these functions.

The inherent serial dependency between fitting and prediction plus the iterative characteristics of the algorithm is the challenging part of the implementation. Data parallelism is critical to speeding up the fitting and prediction procedures, while async communications play a vital role to mitigate the synchronization costs.

2.1 Data set Selection

In order to properly analyze the properties of each algorithm, we first want to select a data set with varying properties based on its features. We have done so by selecting two types of set: sparse and dense. Dense refers to the fact that most features in the data set have a value and thus contribute to the prediction. As such, a tree will have to consider all features when reaching a conclusion. However, due to the denseness of the features, their number is actually smaller, in order to only contain the ones relevant to the outcome. If a dense set did have a large number of features that impacted the result, their relationship would not be distinguishable from random noise due to the minuscule effect every single feature would have, and so training a set to make accurate predictions would be impossible.

As such, we are also using another type of data set called sparse. This data set although containing a large number of features has a large number of them be empty or zero for a specific row. In that case finding an accurate predictor revolves around finding what specific combination of features leads to a desired outcome by being able to compare rows where a feature is either empty or not and observe the actual result.

Lastly, we opted to use linear regression data sets, where a variety of features either return a positive or negative result. We chose this type due to its simplicity which results in less time needed to make an accurate predictor. Since the model type does not affect relative training time between approaches, we know that the speed up achieved by each implementation would not change based on the model

and so to be able to run more tests within a time frame, we opted for the quickest model. As such we only implemented one loss function type in the form Square error. All data sets used in this project were taken from the libSVM library due to their formatting.

3 The OpenMP Implementation

We referred to the algorithm and convergence analysis in the paper "Asynch-SGBDT: Asynchronous Parallel Stochastic Gradient Boosting Decision Tree based on Parameters Server" by Cheng Daning et al. in synchronous OpenMP as well as an asynchronous parameter server approach. We also started with an existing project on GBDT by Yiping Qi (qiyiping@gmail.com) called "Gradient Boosting Regression Tree" found on github.

The OpenMP implementation opted to use shared memory in order to update all independent gradients and build all the trees in an iteration in parallel. It does so by utilizing a dynamic omp loop for both by first updating all features independently and then building the resulting trees independently. However, the problem with that solution is synchronizing the for loops. Since the tree construction has to start after all the gradients have been updated and similarly the next iteration has to begin when all the trees have finished being built, the program stalls for a while as it waits for all threads to reach the synchronization point.

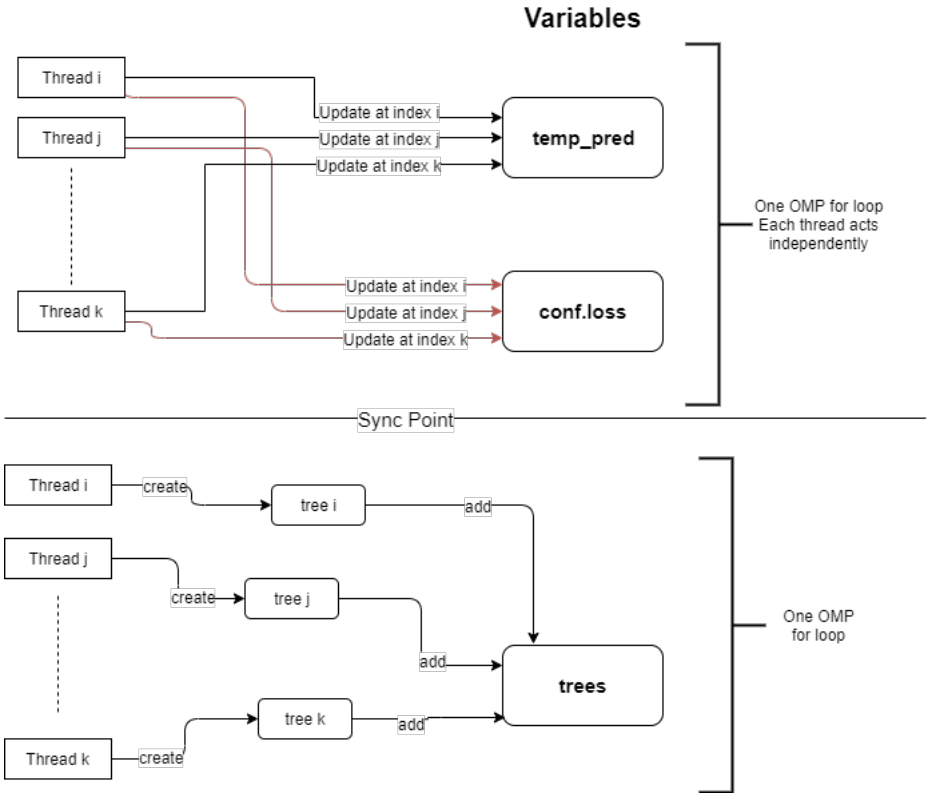


Figure 2: The basic control flow for one fitting iteration of OMP

We do achieve some speedup by sharing the predictions and loss function amongst all tasks. Then each task works on a specific feature of the data set by first finding the prediction and then updating the loss function for that feature. Since all features are independent from each other, no two tasks attempt to access the same point in any of the two collections. Additionally, we use the same logic for building

the trees. Each parallel task accesses a random sample of the data set and independently builds a tree to match it. Since accessing the values from the data set requires read only access, there is no contention between tasks when they all try to read that variable. Thus, each tree is built and then fitted independently, and so we can take advantage of the shared memory available by having each task responsible for only one part of it.

4 The Parameter Server Implementation

For the baseline idea of the Parameter Server implementation, we adapted the algorithm outlined in the paper by Cheng Daning et al.[1] We essentially want a master task and a collection of worker tasks as in the MPI task, but instead of actually communicating through the tasks by transmitting the data through a message, we instead utilize a shared memory approach. Figure 2 outlines the basic structure of the algorithm in Figure 2(a) and the base pseudocode for the parameter server in Figure 2(b) as shown in the paper.

Algorithm 3 Asynch-SGBDT

Input: $\{x_i, y_i\}^N$: The training set; v: The step length;
Output:the Additive Tree Model $F = F(x)$, i.e. asynch-SGBDT forest.
For Server:
Produce the tree whose output is $\frac{1}{\sum_{i=0}^{N} m_i} \sum_{i=0}^{N} m_i y_i$.
Calculate L'^0_{random} and Maintain L'^0_{random}.
for $j = 1 ... forever$ **do**
  1.Recv a $Tree_{k(j)}$ from any worker, this tree is built based on $L'^{k(j)}_{random}$.
  2.Add $Tree_{k(j)}$ times v to whole asynch-SGBDT forest. $(F^j(x) = F^{j-1}(x) + vTree_{k(j)})$
  3.Generate an observed value vector of **Q** and produce sampling sub-dataset.
  4.Calculate current GBDT forest's L'^j_{random} based on sampling sub-dataset.
  5.Remove L'^{j-1}_{random} and Maintain L'^j_{random} (L'^j_{random} can be pulled by workers.).
end for
For Worker:
for $t = 1$ to forever **do**
  1.Pull the L'^t_{random} from Server (L'^t_{random} is current L'_{random} the Server holds.).
  2.Build $Tree_t$ based on L'^t_{random}.
  3.Send $Tree_t$ to Server.
end for

Figure 3: Pseudocode for the Parameter Server

As outlined in the paper, the master task initially creates a loss function and "publishes" it to all workers. Each worker then "pulls" that loss function and uses it to create a tree from a random sample of the data set and sends that tree to the master task. The task then picks any of the sent tree, with a preference to older ones and updates the forest and the loss function. Lastly, the master task then "publishes" the updated loss function again for the workers to "pull". A significant advantage of this algorithm is the fact that all tasks can act asynchronously. The master task only "pulls" one tree at a time, so as long as there are available trees to work on, it does not stall and continuously updates the loss function. Additionally all workers do not have to wait for the loss function to get updated since they can

[1] "Asynch-SGBDT: Asynchronous Parallel Stochastic Gradient Boosting Decision Tree based on Parameters Server" by Cheng Daning, Xia Fen, Li Shigang, Zhang Yunquan

create a completely new tree through the random sample, and do not stall when "publishing" the tree, since they can make a new one instantaneously. As such, we have removed one of the major set backs of both the previous approaches, by enabling all tasks to work without stopping.

However, we can't create an authentic implementation of the algorithm outlined in the paper. This is due to the fact that the paper describes the algorithm as a parameter server, which would imply that we needed to connect and communicate over multiple machines to truly follow the steps outlined in the paper. However, if we did so, we would be creating an unfair advantage for our server algorithm. Since, it is run in multiple machines, the hardware specification is different from the other implementations. With said differences, it would be impossible for us to conduct a fair analysis, since the OMP implementation relies on the resources of one machine which is obviously going to hinder its computing resources. As such, we adapted the base algorithm to fit in one machine. We did so by creating the idea of a server-worker relationship through shared variables in memory.

We know that there are essentially two pieces of information that have to be shared from the server to the workers and from the worker to the server. Firstly, we need to create a queue for the workers to submit their completed trees for the server to access and append to its forest. We do so by implementing a vector of the trees. When a worker finishes constructing the tree, it locks the vector, pushes its tree and unlocks it. Since, the vector is globally accessible to all workers, we had to modify the vector itself to fit with our lock implementation. Thus, as a worker is pushing back its tree unto the vector, it is essentially "publishing" it to the server in the form of a task queue. The server is then able to access that tree and append it to its forest before moving the head of the queue to the next available tree.

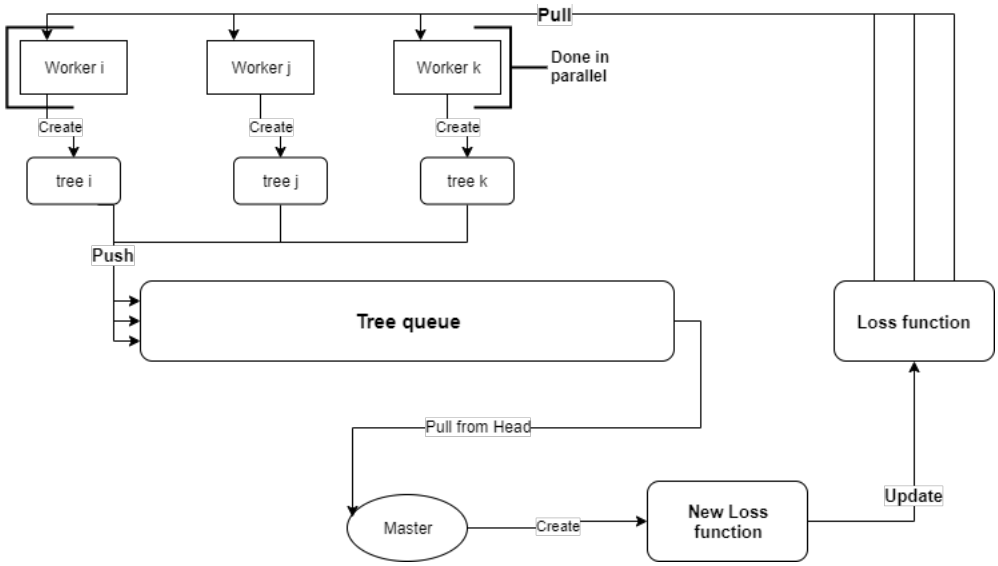


Figure 4: The basic control flow in an asynchronous setting

Secondly, we had to "publish" the updated loss function from the master and "pull" it in the worker side. To do so we created a pointer to the loss function represented as a collection of gradients. The only one who has access to write on that collection is the server. The workers only need to read it and copy specific indices to build their tree. As such we wanted a way to prioritize writing on that collection

without causing problems for the workers reading that collection. We did so by giving the server write priority. Essentially, the server would wait for all workers reading the collection to finish before writing on it, but any new worker would wait for the server to finish changing the function, even if the server has only placed a write request. Therefore, as soon as the server opts to change the loss function, no new workers can start reading it but the server will wait for all the workers currently reading it to finish. We found that the stall created by this lock was relatively minimal compared to the potential time lost in any other way of transferring the loss function from the server to the workers.

5 Analysis

In order to compare the implementations we have created, we are going to measure their performance based on multiple factors. Based on how we constructed the fitting algorithm, those factors are: the total number of trees constructed by the algorithm before stopping in the asynchronous case which corresponds to the number of iterations in the OMP case, the number of iterations or trees built effect on the final loss of the algorithm and lastly the number of processors used. While varying one of these variables, we are going to keep the other two constant. Additionally, we are going to run all the performance tests on both a sparse and a dense set to observe the extent that the type of data set affects performance. Lastly, we will keep the hardware we use to run the tests constant to make the comparisons more accurate. We also changed one hyper parameter between the sparse and dense data sets, that being the feature ratio used. Feature ratio defines the subset of features split when considering a node. Since our sparse data set has a significantly larger amount of features to split on, we aimed to reduce the overall time taken to train that set. Lastly, the relationship between the number of iterations on the OMP implementation and the number of trees in the Async one is defined as: $N_{trees} = N_{iter} * N_{processors}$ since in one iteration the OMP code will construct as many trees as there a parallel tasks which are represented by the number of processors.

5.1 Dense Data Set

We used the data set cadata found at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression.html#cadata` with 20,640 data rows and 8 features

5.1.1 Effects of the number of processors on training time

We first analyse how the amount of parallelism affects the overall training time. Figure 4 was produced for 5000 trees with a sample ratio of 0.3.

As can be seen, the asynchronous implementation is consistently faster than the OMP one. Although in the larger number of processors of 16 the difference seems small, the asynchronous code runs 4 times faster than the OMP one. However, waiting for either the loss function to be updated as a worker or for the tree queue to not be empty as the server are a constant overhead for the asynchronous version and although the code becomes faster as the processors increase the stalling always results in a plateau and so a larger number of

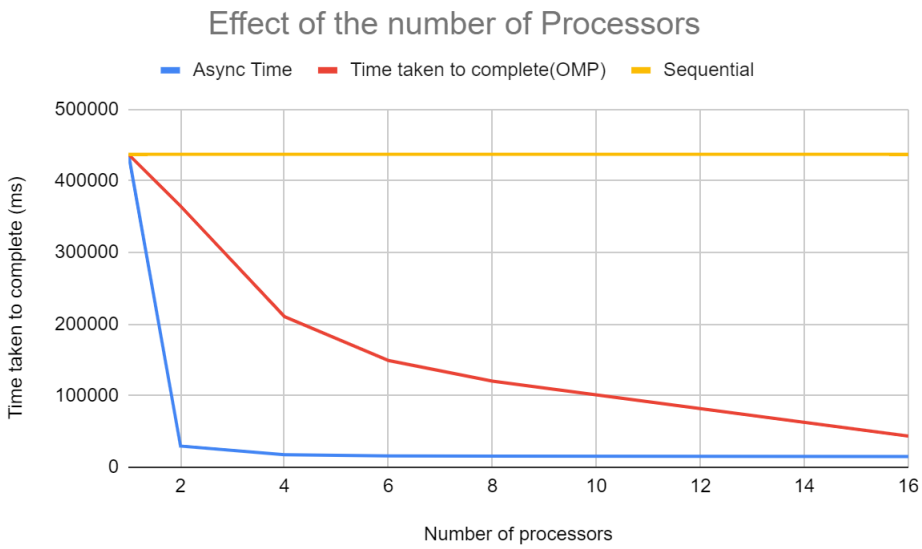
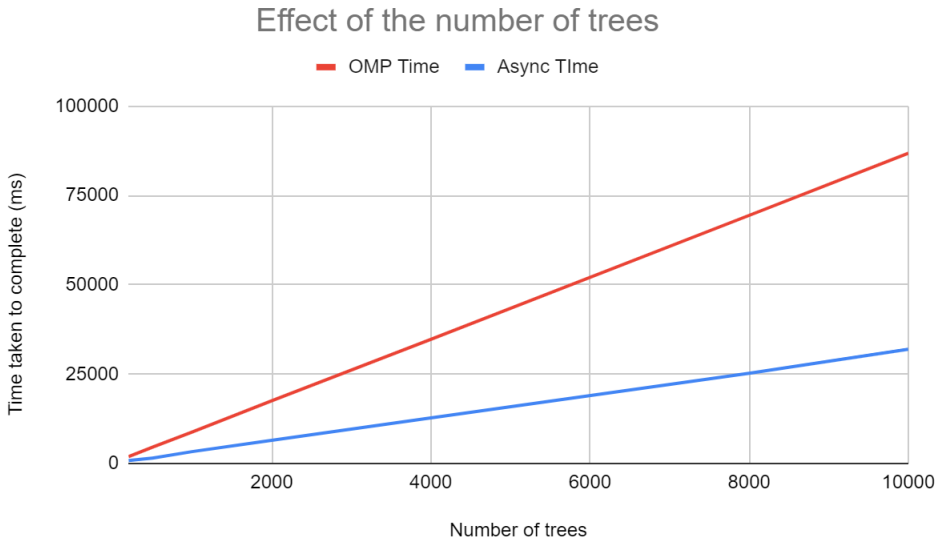


Figure 5: Relationship of the number of processors to the time taken to complete

processors will not increase performance. As such, when considering the optimal number of processors, 16 seems to be the best choice given the hardware and thus will be used in the following tests.

5.1.2 Effects of the number of iterations on training time

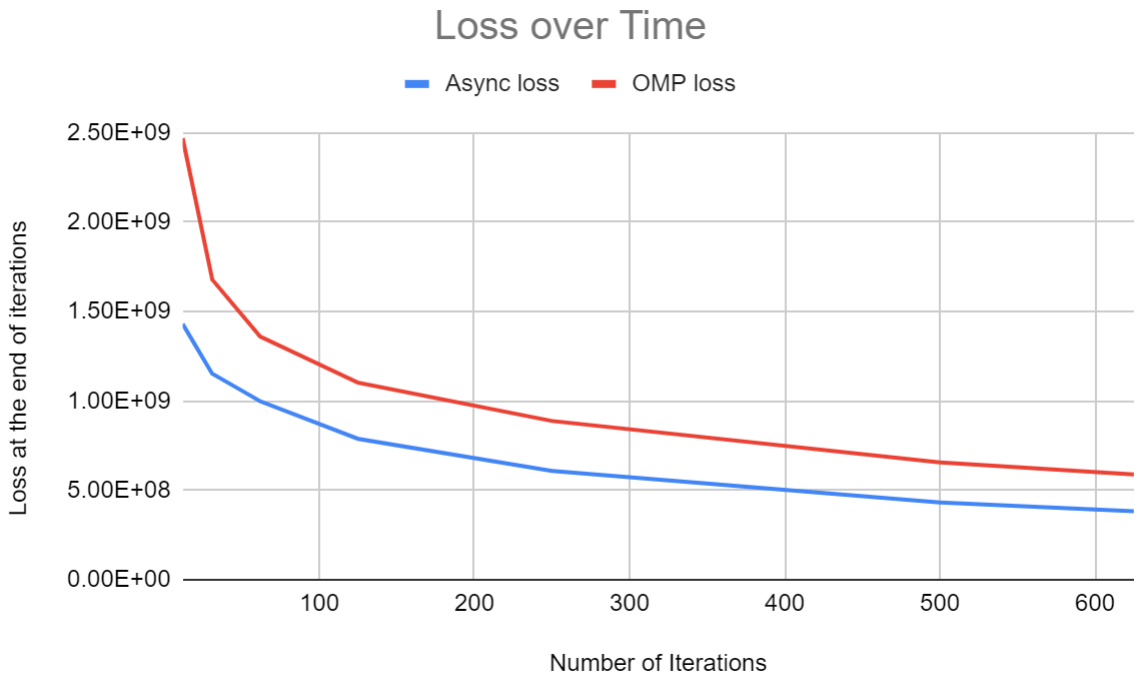
We then analyze the time of training related to the number of iterations or trees completed. We do expect the relationship between the two to be linear as each iteration or tree building step takes roughly the same amount of time. From running the tests on 16 processors with a sample ratio of 0.3 we get the graph shown in Figure 5.



As we have predicted, both relationships are linear. However, the asynchronous code has a significantly lower gradient in terms of increasing its trees. Therefore, we an conclude that as the data sets become larger the asynchronous implementation becomes increasingly preferable as more iterations would be needed to reach an acceptable level of accuracy.

5.1.3 Effects of the number of iterations on final loss value

Based on the previous trend of the asynchronous implementation completing significantly faster than the OMP one, we want to compare the loss reached at the very end of all iterations in order to see which implementation is quicker to reach a converging point. Since loss is calculated as the distance of a residual from the correct result, the lower the value, the more accurate the result. Thus, the graph shown in figure 6 is using 16 processors, a sample ratio of 0.3 and ran for the same number of iterations as the graph above, in this case represented as completed iterations rather than total built trees.



As per the graph, both implementations show the same type of trend line, implying that both of them are able to finally converge on a value. However, the asynchronous implementation is consistently more accurate, having a significantly lower value at all iterations. This is likely due to all trees in the OMP case being built using the same loss function. Thus most trees created in an iteration will be outdated. In contrast, the asynchronous threads are able to pull the most recently updated loss function, making the created trees more accurate.

5.1.4 Overall Dense Set Results

For dense data sets we have analyzed the extent to which each implementation utilizes parallelization, the speed of each implementation and their accuracy. In all three tests, the asynchronous code is shown to be preferable. Not only can it perform better given the

same resources, but is also faster when training and results in more accurate answers in the same number of iterations as the OMP version for this small number of features. As such, for a dense data set, we have concluded that it is preferable to use an asynchronous implementation, falling in line with the expectations set in our source paper.

5.2 Sparse Data Set

We used the data set found at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression.html#E2006-tfidf` containing 16,087 data rows and 150,360 features. For a sparse data set, due to the number of features, both implementations took significantly longer to run. However, this is understandable due to the amount of actual data points in the set being more than 14 thousand times the size of the dense set.

5.2.1 Effect of the number of processors on training time

Figure 5 shows the training time for each implementation based on the number of processors. We suspect that the stall amount for the workers and the server is much more pronounced due to the increased number of features. At every server iteration the task locks the loss function and makes it impossible for any worker to gain access to it. However, due to the increased number of features, it takes longer for the server to update the loss and so more workers stall waiting to access it. As such, the asynchronous implementation plateaus on a significantly smaller number of processors. However, the number of iterations relating to time spent and the overall end accuracy of the

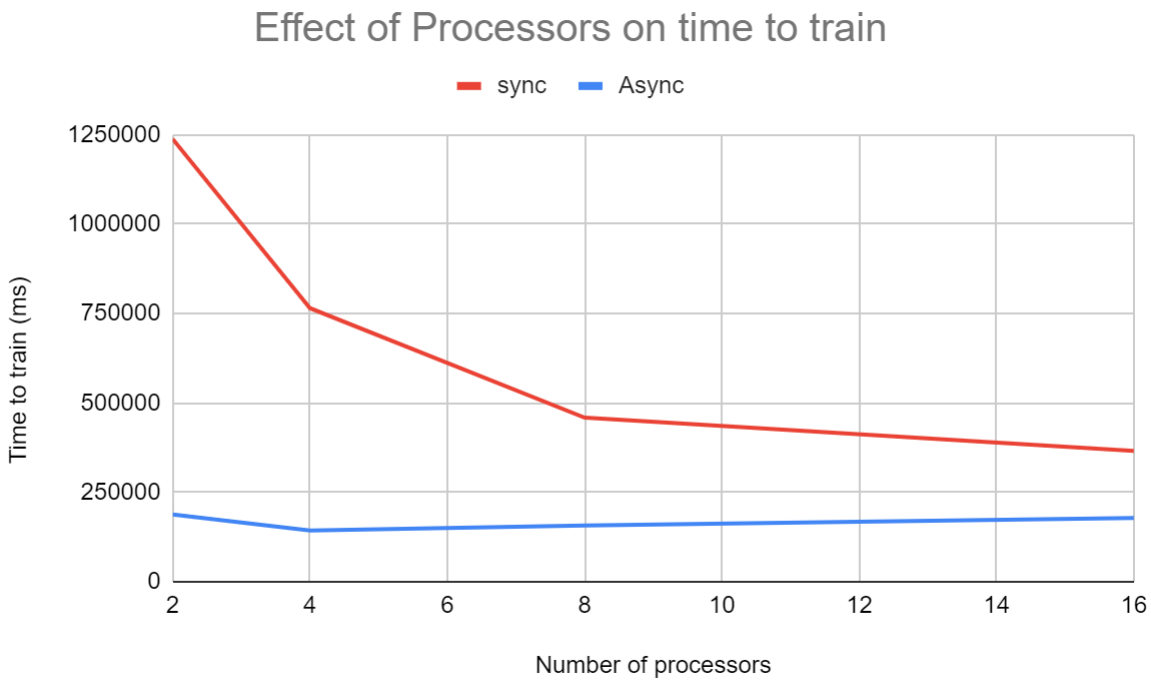


Figure 6: Relationship between the number of processors and overall training time in a sparse setting

asynchronous version are similar to the ones for the sparse set, once again showing that having an asynchronous implementation does provide a significant advantage.

6 Conclusion and Improvements

Through our analysis we were able to show the advantages of an asynchronous implementation over a synchronous one based on its speed and accuracy.

In terms of improvements, as mentioned in the section above, there still exists a synchronization point in terms of the workers waiting for the server to update the loss function which becomes apparent as the data set increases in size. To combat this, we could implement an asynchronous update of having two loss functions and pointers to them, one being worked on privately by the server and one being publicly available to all workers. Thus, when the server needs to update the loss function, it does so by swapping the pointers to the structure, a process which is significantly faster than manually updating all features while workers are unable to gain access to the structure.

7 Credit Distribution

We worked in equal amounts with Guanfu writing the majority of the algorithms and Dimitris running the tests and creating the main body of the write up. This included multiple meetings to create the main algorithm to be implemented and the overall design of the structure of the solution.

8 References

[1] Yiping Qi, Gradient Boosting Regression Tree

[2] J. H. Friedman, "Stochastic gradient boosting," Computational Statistics Data Analysis, vol. 38, no. 4, pp. 367–378, 2002.

[3] J. Ye, J. H. Chow, J. Chen, and Z. Zheng, "Stochastic gradient boosted distributed decision trees," in Acm Conference on Information Knowl- edge Management, 2009, pp. 2061–2064.

We mainly refer to the following paper for our implementations of GBDT followed the parameter server framework outlined there.

[4] Daning, Fen, et al. "Asynch-SGBDT: Train a Stochastic Gradient Boosting Decision Tree in an Asynchronous Parallel Manner"