Colorado State University

Parallel Programming

CS-475

Final Project

Parallel Decision Tree Classifier With OpenMP

Co-Author:
James Shaddix

Co-Author:
Tanveer Hussain

December 15, 2018

Abstract

For our project, we designed a Decision Tree Classifier. The decision tree algorithm, is a popular algorithm used in machine learning for classification experiments. We developed our first pass of the algorithm in C++ by using some reference code that we found online that was written in python [1]. We than parallized our code using openMP, and we display our results for various different file sizes in this paper. We than re-wrote the algorithm in a more efficient manner, in order to decrease the run-time of the program, and to make our program less data dependent. By decreasing our data dependence, we hoped to increase the speedup of the program when we parallized it.

1 Algorithm Description

1.1 Machine Learning Basics

Before we explain how the Decision Tree Classifier works, it is important to first understand some basic concepts of machine learning. Machine learning algorithms use large repositories of data, in order to construct an algorithm. In particular, machine learning algorithms generally work by looking at old data sets, in order to make predictions as to what will happen in the future. However, this means that your algorithm will vary with the data that you use for your experiment (this is important to note, as it is a result of this that we struggled with doing portions of our analysis).

Machine learning algorithms work in two different phases. The first phase is the **training phase**. During the training phase, the algorithm that will be used to make predictions is constructed. During this phase, your program analyzes all of the data that you would like your model to be based on, and a model is created. The second phase is the **testing phase**. During this phase, you use your model on some sample data that you already have results for, in order to see how often your model reproduces the correct results. This provides you with an idea as to how well your algorithm is performing.

In general, machine learning algorithms are either used for **classification experiments**, or **regression experiments**. Classification experiments our used to make predictions on discrete categories. while regression experiments are used to make predictions on continuous sets of data. Most machine learning algorithms can only perform one of these two types of experiments, but the decision tree algorithm can be designed for either of these experiments. However, the decision tree algorithm tends to see its best results when designed for classification experiments, and that is why we decided to build a decision tree classifier for out project.

Most machine learning algorithms fall into one of three different categories: **supervised learning**, **unsupervised learning**, **reinforcement learning**. Supervised learning algorithms are used for labeled data sets. In particular, this means that the data that you are using to train your algorithm has labels associated with

the features that you are trying to predict on. You can than use this the labels in your testing phase to determine what percentage of your data you are classifying correctly. Unsupervised learning algorithms are used for data with unlabelled data sets. These algorithms tend to revolve around using statistical techniques to make there predictions. Reinforcement learning algorithms build there model through system of trial and error. You generally specify how many iterations you would like to use to build your algorithm, and than you have your algorithm modify itself when it performs an incorrect prediction.

1.2 Decision Tree Algorithm

The Decision Tree algorithm is a **supervised learning** algorithm that is commonly used in machine learning for classification experiments. A decision tree is generally a binary tree that is composed of a **root node** at the top, followed by layers of **decision nodes**, with **leaf nodes** at the bottom of each branch. All of the nodes in a decision tree, except for the leaf nodes, contain a question with a binary output that is used to parse the data. All of the leaf nodes have an associated classification.

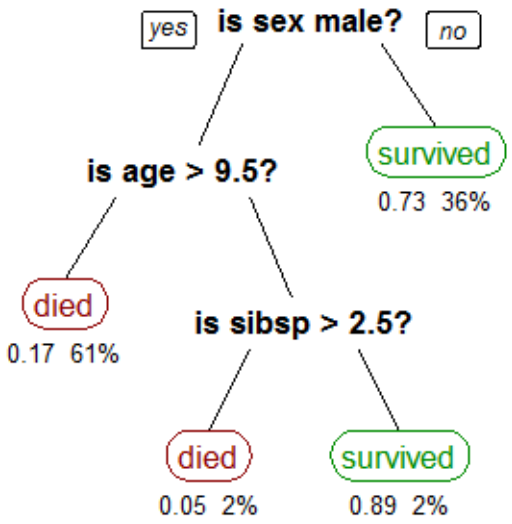

Figure 1: This is an example of a decision tree [2]. This is a decision tree that was trained on a data set associated with the passengers of the titanic to determine which passengers survived. The question "is sex male", is associated with the root node, and all of the other questions are associated with decision nodes. All of the leaf nodes contain a particular classification which in this case is either "survived" or "died".

1.3 Testing Phase

In order to use a decision tree, all of the data that needs to be classified is passed to the root node. The data is than parsed based on the binary categorization of the question, and, based on the results, the parsed information is passed on to the next two nodes. This process is continued until all the data has arrived at a leaf node. Once a data element has arrivied at a leaf node, we use the class associated with the leaf node as the prediction for what class that particular data element belongs to.

1.4 Training Phase

In this section, we will discuss how to build a decision tree. In order to build a decision tree, you need to come up with a method for determining what question should be asked at each node. The goal of the question is to split the data into two sets, such that you get the purest possible mixings of dependent variables in each set. There are a handful of of different tree building algorithms that are commonly used for determining the questions that should be asked. Each of these algorithms vary slightly in the way that they decide on what question should be asked at a particular node. A few of the more common algorithms include: CART (Classification and Regression Tree), ID3 (Iterative Dichotomiser 3), and CHAID (Chi-Squared Adaptive Regression Spines). All of these algorithms are greedy algorithms that use a metric referred to as **information gain**, that quantifies how good a question is at parsing data into more homogeneous sets. For our project, we used the CART algorithm to build our decision tree.

1.4.1 CART Algorithm:

The CART algorithm considers a set of candidate questions at a particular node, and chooses the question that provides the largest information gain. The CART algorithm considers every value in a data-set at a particular node as being a boundary value, and each of these boundary values represents a candidate question. There is an example below that explains how exactly this works.

	Color	Size	Type
0	Blue	2	Friendly
1	Blue	5	Friendly
2	Green	7	TeamMate
3	Red	10	Enemy

Table 1: For out example, we will consider this data set as being all of the data at a particular node. We will consider a decision tree whose dependet variable we are trying to classify on is "Type". We now need to find all of the candidate questions associated with this data by considering each value in the table associated with our independent variables as being a boundary value.

Color	Size
is color == Blue	is size >= 2
is color == Blue	is size >= 5
is color == Green	is size >= 7
is color == Red	is size >= 10

Table 2: These are the eight questions that would be generated from the data-set given above. It is important to note here that there is a duplicate question, but this degeneracy will not harm the correctness of the algorithm.

In order to determine the best question, The CART algorithm uses **Gini Impurity** in order to calculate the information gain. Gini Impurity quantifies the amount of impurity at a particular node. In particular, gini impurity calculates your chance of drawing two elements with replacement, and having their "Types" (or whatever variable you are using as your dependent variable) not match. The goal of the decision tree algorithm is to come up with questions that will minimize the Gini Impurity in the child nodes. We can calculate the Gini Impurity by summing over the probabilities of picking each class, multiplied by the probability of miss-classifying each class.

- J = number of classes
- p_i = probability of picking a particular class i
- $p_k = (1 - p_i)$ = probability of miss-classifying a random data point, with class i
- $G(p_i)$ = Gini Impurity

$$G(p_i) = \sum_{i=1}^{J} p_i p_k = \sum_{i=1}^{J} p_i(1 - p_i) = \sum_{i=1}^{J} p_i - \sum_{i=1}^{J} p_i^2 = 1 - \sum_{i=1}^{J} p_i^2$$

$$\boxed{G(p_i) = 1 - \sum_{i=1}^{J} p_i^2} \tag{1}$$

Gini Impurity

We can than use the Gini Impurity to calculate the information gain. The information gain of a particular question is the Gini Impurity of node containing the question, minus the weighted average Gini Impurity of the child nodes.

- I = information gain
- $G(p_i)_p$ = Gini Impurity of parent node
- $G(p_i)_{c1}$ = Gini Impurity of child node 1
- $G(p_i)_{c2}$ = Gini Impurity of child node 2
- $w_1 = \frac{samples\ in\ parent\ node}{samples\ in\ child\ node\ 1}$
- $w_2 = \frac{samples\ in\ parent\ node}{samples\ in\ child\ node\ 2}$

$$\boxed{I = G(p_i)_p - [w_1 * G(p_i)_{c1} + w_2 * G(p_i)_{c2}]} \tag{2}$$

Information Gain

We than use the information gain as the cost function to determine which question to use. We pick the question with the largest information gain. We can than recurse down in our program until we arrive at a node that is totally pure.

2 Experiments Planned

2.1 Basic Decision Tree Classifier

We built are Decision Tree Classifier based off of Josh Gordons python approach [1]. We used similar techniques to him to build our Decision Tree but we built are decision tree so that it would only use classification data, as opposed to both classification and regression data. We chose to make this simplification after we finished writing our sequential program more generally, because we realized that if we were going to

write an improved version of our code, we would have to treat both of these types of data separately, and this was not going to feasible under the time constraints of this project.

Josh Gordon's algorithm is displayed on his website as a means for teaching people how decision trees work. In his algorithm (and in ours) he considers every value in his set of independent variables as being a particular boundary value (similar to the example I proposed in the Chart Algorithm section). As a result, this means that there are $n * (m - 1)$ questions at every depth of the tree where no leaf nodes have been found yet. Here n represents the number of elements in the data-set, and (m-1) represents the number of attributes in the data-set that aren't the attributes that we are classifying on. With every question, we than need to split up the data based on the question to determine the information gain of each question. The time complexity associated with iterating through all of our elements and splitting them based on a question is β, where β represents the number of elements at a given node. The problem with going any further with determining the complexity of this program, is that β is some fraction of n, that we cannot know until we have built algorithm. Here lies the essential problem with determining the complexity of Machine Learning programs. Since the algorithm is developed based on the set of the data that is given to it, it is difficult to determine the exact complexity of the algorithm, until it has been built. The number of questions would also change as traverse into the tree and leaf nodes are encountered.

2.2 Parallel Decision Tree Classifier

For our parallel implementation of the decision tree, we decided that we would parallize the training phase of the algorithm. We decided to parallize the training phase as opposed to the classification phase, because, as is generally the case with machine learning algorithms, the training phase takes substantially more time.

During the training phase, there are two essential operations that are being done at every decision node in our program.

1. They determine what the best question is to ask from a set of candidate questions.

2. They split the data up based on what the best candidate question is.

For our parallel implementation, we parallized the portion of the code that determined what the best candidate question is. We did this because we found that this portion of code comprised the majority of the run-time of our program, and because the separation of data is an inherently sequential operation. We were able to do this by by using a `pragma omp for` command on the outer loop that iterated through our matrix questions. We made sure to situate the loops so that the inner loop iterated across a single array of elements, rather than across all of the arrays in our

matrix, in order to make better use of locality. We also added `pragma omp critical sections` in the areas where we modified shared variables (such as the variable that stored the current best question), so that we could eliminate any race conditions that may otherwise occur.

We also timed the sequential and parallel portions of our code, in order to determine what possible speed-up would be obtainable by our program. We found that the parallel portion of our code comprised 99.9% of the run-time of our program, while the sequential portion of our program only comprised the other 0.01% of the run-time. We can than use Amdahl's Law in order to figure the potential speed up of our program.

- ψ = speed up
- f = sequential fraction of the code = 0.001
- (1-f) = parallelizable fraction of the code = 0.999
- p = number of the threads run on the program (we used up to eight threads for our experiments)

$$\boxed{\psi \leq \frac{1}{f + \frac{(1-f)}{p}}} \tag{3}$$

Amdahl's Law

For our experiment, the best speed-up we may obtain is:

$$\psi \leq \frac{1}{0.001 + \frac{0.999}{8}} = 7.94$$

This value represents the theoretical speed-up that may obtain, but this expression does not take into account the sequential nature of the critical statement in our program, the time to spawn the threads at every node, or the fact that our code may have an upper bound on the speedup that is associated with data dependence on our program.

2.3 Improved Decision Tree Classifier

In order to improve the run-time and data dependence of our classifier, rather than searching through every at a given node, I instead stored a count associated the number of times a unique classification existed at a node inside of an array. I than updated that array by subtracting elements from it whenever we separated the data. This meant that we could separate our data much faster, and because I was only storing counts associated with unique value, instead of the entire matrices of values, I

had much fewer questions to consider at every node, because there would be a question associated with every unique value that we are trying to classify on, as opposed to an entire matrix of values.

2.4 Experimental Setup

We performed all of our experiments on a machine in the Colorado State University Computer Science Building. Specific information about the machine we used is given below.

- Host-Name: tuna
- Number of Cores: 12
- Cache Sizes: (L1d: 32K) (L1i: 32K) (L2: 256K) (L3: 15360K)

2.5 Code

We generated a single executable for each of the three different programs that we created. In total, we wrote roughly 3-4 thousand line of C++ code creating this project, as well as several hundred lines of the python code to compute all of our results. Details about the executables are given below.

- decTree_SEQ = This is file corresponds to our original sequential version of the code. This file was compiled from the DecisionTree1.cpp file.
- decTree_PAR = This is file corresponds to the parallel version of our original sequential program. This file was compiled from the DecisionTree1p.cpp file.
- decTree_SEQ_B = This is file corresponds to our improved sequential version of the code. This file was compiled from the DecisionTree2.cpp file.

All of these programs can be run without any arguments. However, all of these programs allow for a single command line argument that specifies how many elements will be read from the file when the program executes.

3 Experimental Results

3.1 Basic Decision Tree Classifier

	rows	time
1	2000	10.56460
2	4000	54.86040
3	6000	135.02100
4	8000	221.66200

Table 3: These are the results that we found for the sequential program.

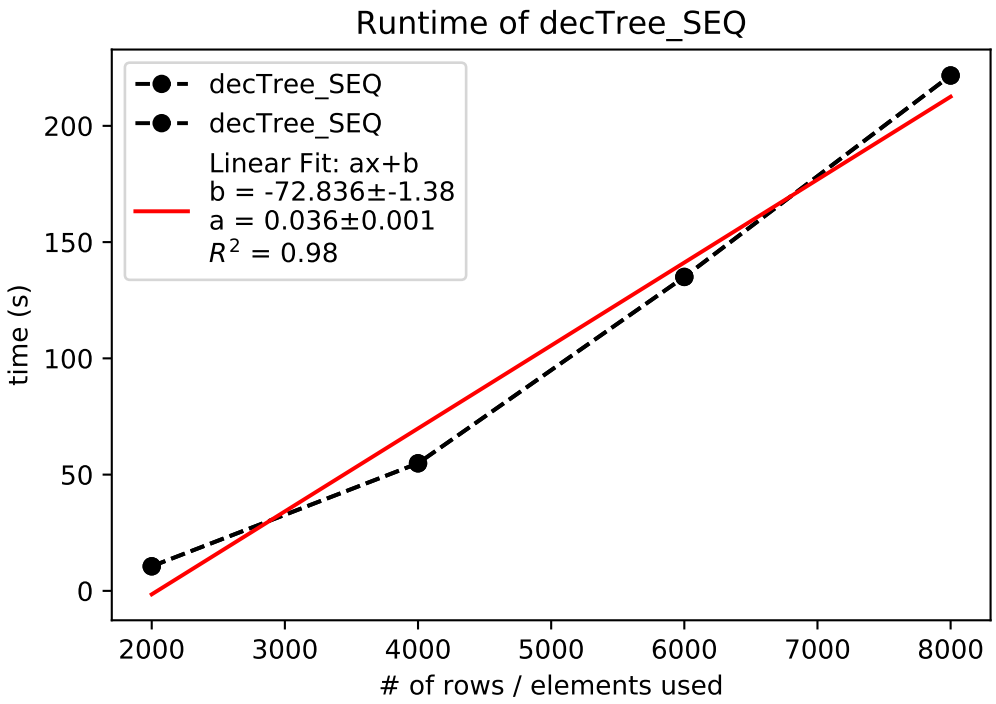

Figure 2: This is a graph of the table represented above. Although we would have a difficult time trying to compute the complexity of our program, we can use the coeficient in our linear fit to find the complexity of our program here. From our results we can see that the complexity of our program is 0.036n.

3.2 Parallel Decision Tree Classifier

	threads	rows	time	speed-up
0	1	8000	221.7540	0.999585
1	2	8000	146.4120	1.513961
2	3	8000	109.7810	2.019129
3	4	8000	89.9062	2.465481
4	5	8000	85.4729	2.593360
5	6	8000	79.4276	2.790743
6	7	8000	80.9437	2.738471
7	8	8000	79.8857	2.774739

Table 4: We ran are parallel version of our program with row counts of 8000 across many different threads. The results are represented above.

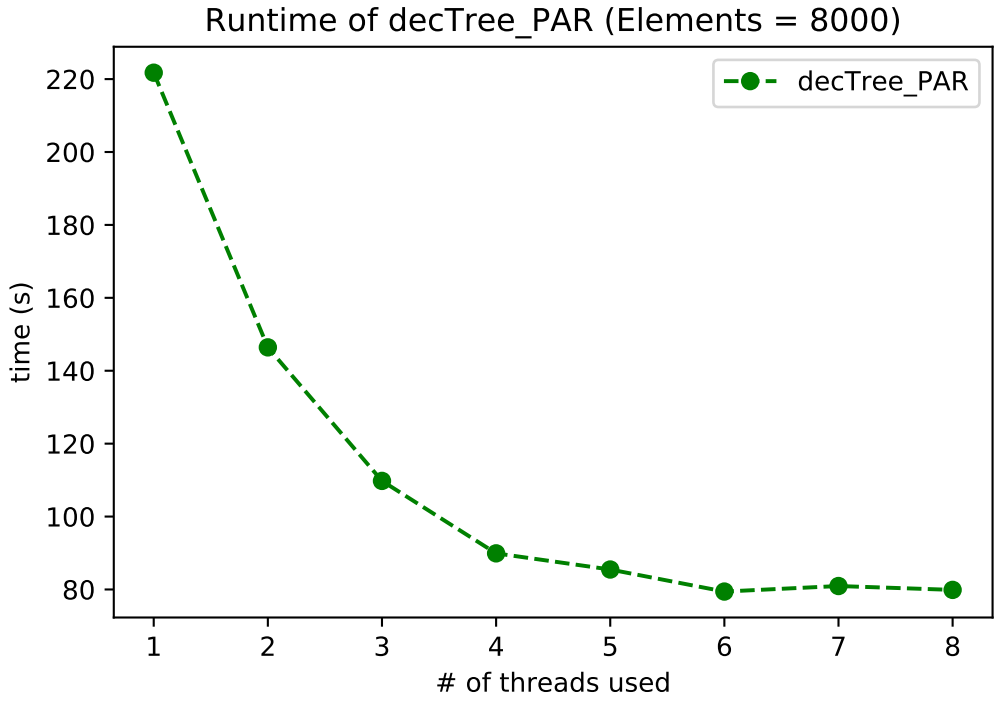

Figure 3: From this graph we can that our program seems to be getting monatomically faster as we increase the thread count.

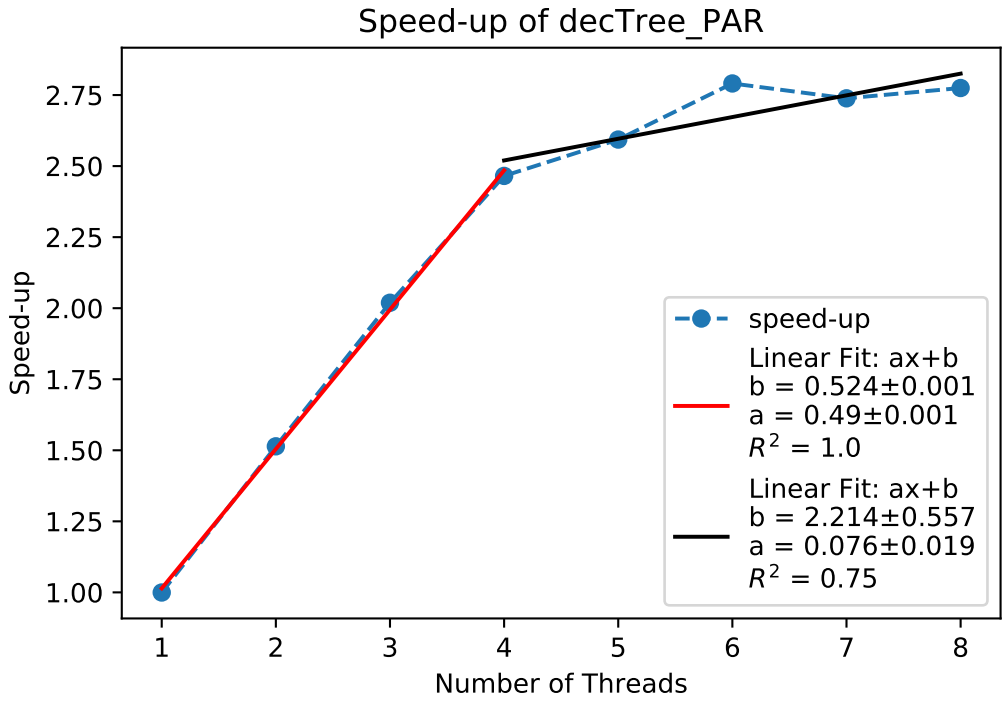

Figure 4: This speed up plot of our results is very interesting. In the plot, I have fit two sepperate linear lines. From the R^2 value for the first fit, you can see that the speed up is increasing linearly. However, after four threads, we see very little speed up. This is a results of the data dependence of the program!

3.3 Roofline analysis

- Maximum memory bandwidth: 59.7 GB/s
- number of bytes read/written in the find_best_question function
- total number of loop iterations done by find_best_question throughout the execution of the program = 2*n*m (where n is number of row of the data we are reading which is 8124 in our case and m is number of attributes-label of our data which is equal to 23-1=22)
- Reads = n*m for each question asked and there will be writes = 1 for each iteration. moreover, we are reading and writing strings of one character and the number of bytes a string takes up is equal to the number of characters in the string plus 1 (the terminator), times the number of bytes per character. So, 2 bytes in total. Total Giga Bytes:

with n=8124 and m= 22 our bytes will be= 63 G Bytes.

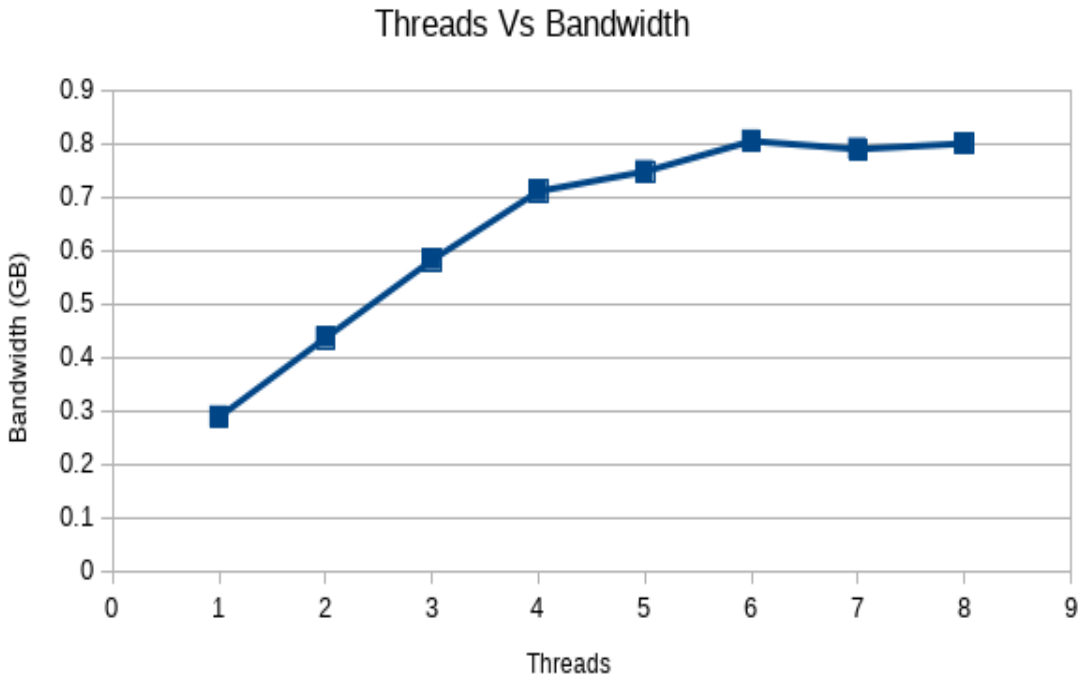

Figure 5: This is a plot of the roofline analysis having bandwidth in GB on y axis and Number of threads on x axis.

From the graph above we can see that 6 threads are required to saturate the bandwidth in this case. In our opinion program is memory bound and not compute bound because we can see that the program makes a lot of memory accesses than computation (only two operations).

3.4 Improved Decision Tree Classifier

	rows	time	speed-up
1	2000	0.000435	24293.417649
2	4000	0.000768	71437.835473
3	6000	0.001119	120672.982393
4	8000	0.001612	137511.709420

Table 5: These are the results for the improved sequential version of our code.

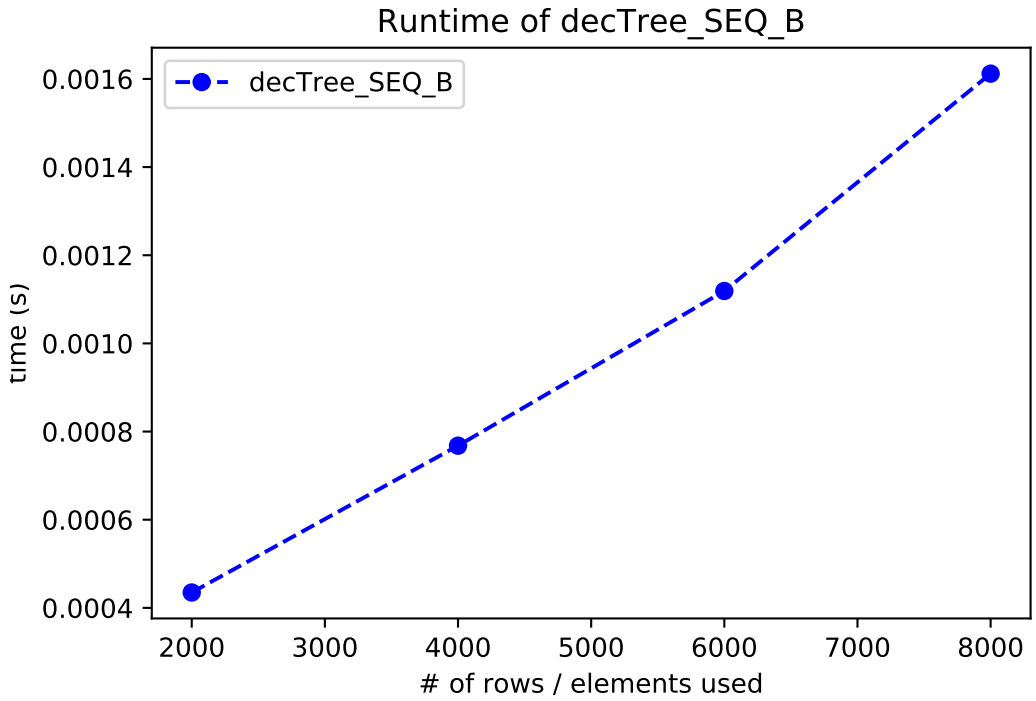

Figure 6: From the plot, you can see that the our program appears to be increasing fairly linearly. Originally we had planned on making this program run in parallel, however, since the program was already running so fast, there would not be any point in parallizing this program as the overhead of spawning the threads would take up all of the time.

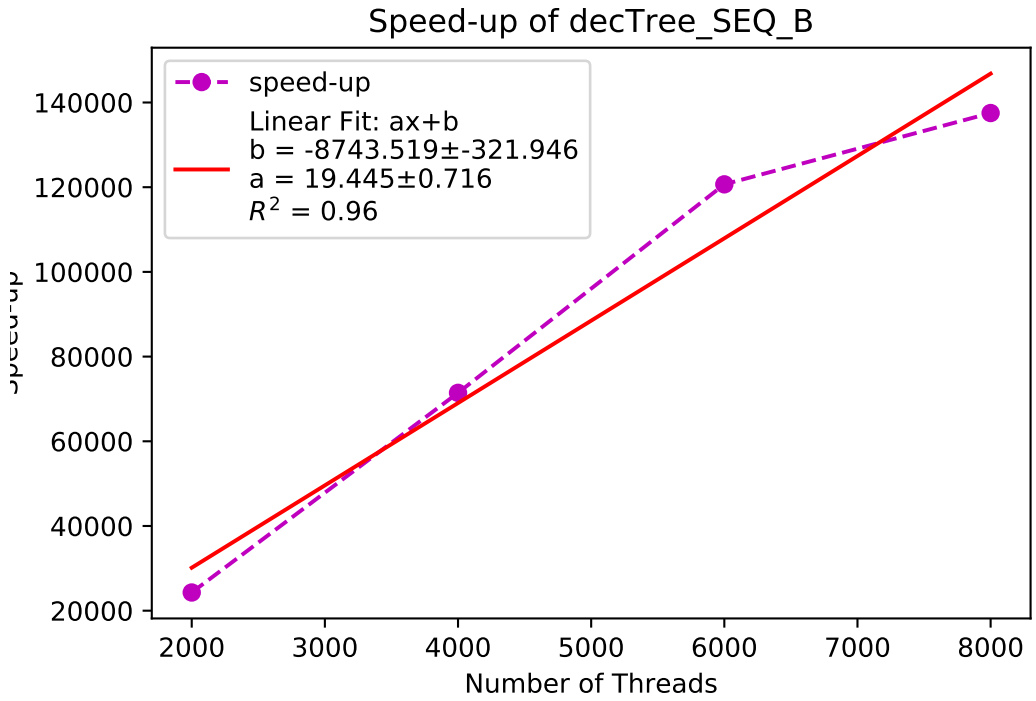

Figure 7: This is a plot of the speedup of our faster program across many different numbers of rows (the x-axis is labelled incorrectly, it should be row count).

4 Conclusion

For this project we performed three different experiments. For our first experiment, we created a sequentially run decision tree that was based off of a python script we found online. We were successfully able to re-create the script and we found the programs complexity was $O(n)$. For our second experiment, we parrallized our code, and we found there was an upper limit on the speedup as a results of our data dependence. For our final experiment, we found that we could re-write the algorithm in a much more efficient manner by mitigating the number of questions asked at each node.

References

[1] Random Forests Retrieved Dec 1, 2018, from https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb

[2] Machine-learning-Decision-tree Retrieved Dec 3, 2018, from https://intelligentjava.wordpress.com/2015/04/28/machine-learning-decision-tree/