Concurrent Learning of Decision Trees

Dmytro Lysak
dl389@kent.ac.uk
MSc Advanced Computer Science
University of Kent, 2018

Total Word Count: 11,675

Contents

Abstract

This project aims to create a concurrent version of J48 whilst retaining the classification accuracy of the original. The concurrent implementation was built upon WEKA in order to allow users to fully utilise the benefits of WEKA's options and features.

Various kinds of datasets were used to test the concurrent implementation and the results show a very good improvement for large datasets, however not much of an improvement for very small datasets.

Very large datasets that took ~2400 seconds to run on the original only took ~85 seconds to run on the concurrent version with a poolsize of 64 on a machine with 80 logical cores.

In the future with continues work, the results could further be improved and yield much more favourable measures for both smaller and larger datasets.

Figure List

Chapter 1 – Introduction

I have undertaken this project for my MSc in Computer Science at the University of Kent in 2018.

WEKA (a data mining tool made in Java) was used as the platform for this project and its J48 implementation was used as well; however I have made some small, but necessary, changes to J48 in order to make it run concurrently well and safe.

This project aims to take a common classification algorithm, such as J48, make it run concurrently, and analyse the benefits (or drawbacks) of making it concurrent whilst retaining the learning results of the original.

Some liberties have also been taken to explore other methods of enhancing the speed of J48, such as GPU acceleration.

1.1 Overview of Problem

Artificial Intelligence is becoming more prominent and is using bigger sets of data to learn from. These same datasets are also starting to become too large to reasonably process them in good time on a single CPU core. More speed is not necessarily needed for AI to learn, but the difference between learning for 1 hour instead of 10, is indeed, an enticing incentive.

J48 is a classification algorithm based on C4.5, and is used in machine learning to create decision trees and forests. CPUs are moving away from single, highly-clocked cores and into many, low-clocked cores. Algorithms such as J48 run only on one CPU core, so even if there are 10 of these cores, if each of their clock speeds is low, then J48 will be slow.

Machine learning is very important now, and is still becoming even more prominent. Large companies are investing a lot into machine learning in order to tailor their services to customers more accurately. However, because datasets are becoming larger and larger, the time it takes to use machine learning algorithms increases, so the incentive to speed up J48 becomes obvious now: speed up J48, speed up machine learning process, increase time for a company to react to its customers, and therefore, increase customer satisfaction.

J48 is an already existing algorithm in an already existing platform (WEKA). So another problem arises, trying to meddle with an already large code base, in an already implemented algorithm. Certain problems can arise, such as not being able to understand the train of thought the developers had when implementing certain features, or whether they made a mistake in a section of code.

1.2 Incentive

The incentive for me is the technical challenge of this project. I am very interested in concurrency and parallelism, and so this project presented a great opportunity to further hone my skills whilst also creating something that has some weight to it: being able to run a reasonably popular algorithm concurrently on a popular data mining platform such as WEKA.

Another enticing appeal of this is that no one has managed to create a concurrent J48 implementation yet; I will be the first, at least in academia.

1.3 Overview of Contents

Here I will just do a brief overview of the contents for you to get a clearer picture of what each chapter is going to talk about:

Chapter 2 deals with essential reading for this project. I will talk there about the tools, and libraries I have used, as well as some basic concepts on Java's concurrency and what decision trees are.

In **chapter 3** I will talk about work outside of this project, but is still related. I will give summaries of the papers I have read that have tried to implement concurrent implementations of decision trees, critique them, and then give some thoughts as to any future work I think their project will apply to.

In **chapter 4** I will explain the various methodologies and practices I've applied to the project, as well as any outside tools I have used to help me along. I will also talk here about the software and hardware setup of my machine (and other machines that the project has been tested on), as well as explain how I conducted my benchmarks.

In **chapter 5** I will briefly explain how I implemented my concurrency model, as well as talk about the difficulties I have had implementing them. I won't go into too much detail; however source code will be presented.

Chapter 6 will be a discussion of my results, the datasets I have used, differences from the original (whether good or bad), and comparing them against other projects.

Chapter 7 will be me evaluating my project and trying to discern any problems that could have been fixed, or whether a different approach would have been better.

Chapter 8 will be me discussing any future work that I did not have enough time (or skill) to perform, suggest improvements for any of the tools/libraries I have used and briefly discuss any interesting concepts such as GPU acceleration.

Chapter 2 – Prerequisite Reading

2.1 Java and Concurrency

Here I will briefly go over some general concepts on Java's concurrency model that will hopefully help in reading the later parts of this dissertation. I will mainly go over the concepts/functions I have used, and I will use the "*Java Concurrency In Practice*"[A] book as a reference.

Threads: Threads in Java are heavy-weight, which means they use a high amount of resources in comparison to more lightweight implementations from languages such as: Erlang, Google Go, or other languages designed with concurrency from the get go. Some performance may also be lost if you use threads because of the overhead of the operating system (and JVM in this case) having to schedule and maintain them[A]. Later in this paper there will be an instance where injecting threads into a function did not speed it up; you cannot simply throw threads and expect a speedup, it is more complicated than that.

Locks: To put it simply, if a thread wants to modify a variable it needs to lock it first, then modify it, then unlock it to allow other threads to be able to lock then modify[A]. There are different types of locks, some are just primitive implementations, others have a bias to a thread, others have a priority bias, etc...[A]
The problem with locks is that they create contention times, e.g. Thread A wants to lock object X, but Thread B is doing something with object X, so Thread A has to wait, this then would result in a speed-loss; with enough contention it may even run slower than the sequential version[A].

Atomics: I would say atomics are the bread and butter of my implementation, I use them a lot and they work pretty darn well. Atomics essentially are functions in the Java library that allow one to tell the Java compiler to treat the code encapsulated within an atomic function in a thread safe manner. One of these *thread safety* actions involves the Java compiler using special CPU instructions (CAS operations) to add, replace, increment, or decrement the desired variable within the atomic function[A]. An atomic function is considered thread safe (if used correctly) because it swaps the original value with a new value atomically (fast enough for there not to be a thread safety issue because it took so few cycles)[A]. Atomics are not situated at the CPU's L1 cache, they are situated in L2 (or whichever one acts as a shared buffer between all the cores); because of this they will never be as fast as a regular variable modification (e.g. i++), however this will make the variables visible to all the threads all the time[A]. Atomic functions are fragile because they will not tell you whether you have used them correctly, so one must be careful in using them as without enough knowledge, one might think his code is safe, however it might not be.

Atomics are faster than locks because there is no contention period, but even if there wasn't a contention period for locks, would atomics still be faster? The answer is yes because locks have to use a CAS operation (just like atomics) to lock an object, then unlock it, whereas atomics perform the CAS operation once[A]. However, there are instances where locks would be more beneficial, such as having to modify many complicated fields within an object, or trying to perform a chunk of code instead of modifying a variable because atomic functions are limited to primitive variables.

CAS stands for "*Compare and Swap*" and they do what the name implies, compare a given variable to another given variable, if they match a condition then the first variable's value is swapped with the second variable's value. The "*swap*" is not necessary, so you may also refer to CAS as "*Compare and Set*".

Concurrent Linked Queue: This function is a concurrently linked Queue (list) that is perfectly thread safe (unless you use it in strange ways). What this allows a thread to do, is to be able to use the list at the same time as other threads; this makes it invaluable in operations such as creating queues of tasks for threads to go through. In terms of speed, it cannot beat atomics; however it is easier and safer to use.

Executor Service: The Executor Service library is a predefined thread scheduler[A]. It works well if you just want to get something running concurrently effectively and quickly; however if you are doing something complicated, this may present a few unexpected hiccups. This service does not allow you to modify *its* task queue, so if you want to add to the task queue, you have to go through some hoops, it might be better to just create a custom scheduler instead, which will also give you more freedom to tailor it to your needs.

2.2 Decision Trees

Here I will briefly describe what decision trees are and how they look like. I will use the "*Machine Learning*"[B] book by Tom M. Mitchell; however I won't go into too much detail in how they work.

The decision tree algorithm that this project is based upon is J48, a Java implementation of the C4.5 algorithm. The J48 algorithm is freeware; however it is mostly used in WEKA. C4.5 and J48 both use the .arff dataset format to construct their trees, and that is what was used in this project as well.

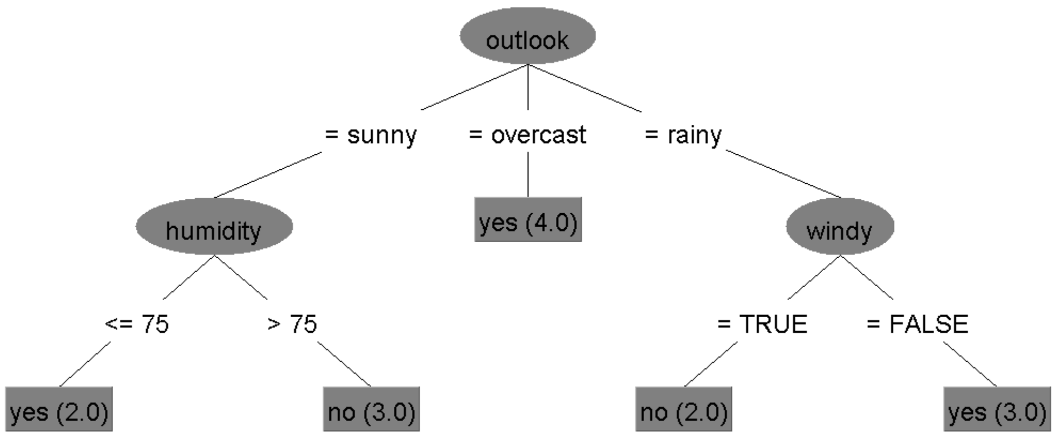

Figure 2.21 – This is a simple decision tree (taken from J48) representing a person who plays tennis and predicting what he will do if the weather is in a certain state.

Decision Trees can essentially be referred to as giant *if-then* statements[B]. For example, in *Figure 2.21*, a simple decision tree is displayed that was based on the pattern of whether a tennis player will play tennis on a specified day. Now we can use this tree to help us make a prediction if our tennis player will play on a certain day. For example, if today is sunny, and the humidity is above 75, our tennis player will probably not play today.

This is a fairly simple tree, so the results it gives may not be very accurate. More complicated trees can be generated with more *if-then* branches that would allow for a more accurate prediction; a bigger dataset of the days the tennis player plays would also be helpful. *Pruning* can also be performed in order to cut off unneeded branches of a tree; this might make the predictions more accurate.

Decision trees can be good at solving some problems; *what are the best problems decision trees are suited for* you say? Well, that depends on the types of characteristics that the problem has[B]. An example would be "*Instances are represented by attribute-value pairs*"[B] which means values such as humidity (*Figure 2.21*) are represented by numbers, or states such as sunny, overcast, or rainy.
Another example would be if the last output value was a Boolean value, a clear number, or a clear state[B]; in our *Figure 2.21* example, we can see that all of the outputs are either yes or no.

2.3 Decision Tree Construction

Here I will just briefly go over how decision trees are constructed; for more detail go over the "*Machine Learning*"[B] book by Tom M. Mitchell. I will use the example tree in *Figure 2.21* as it is nice and simple.

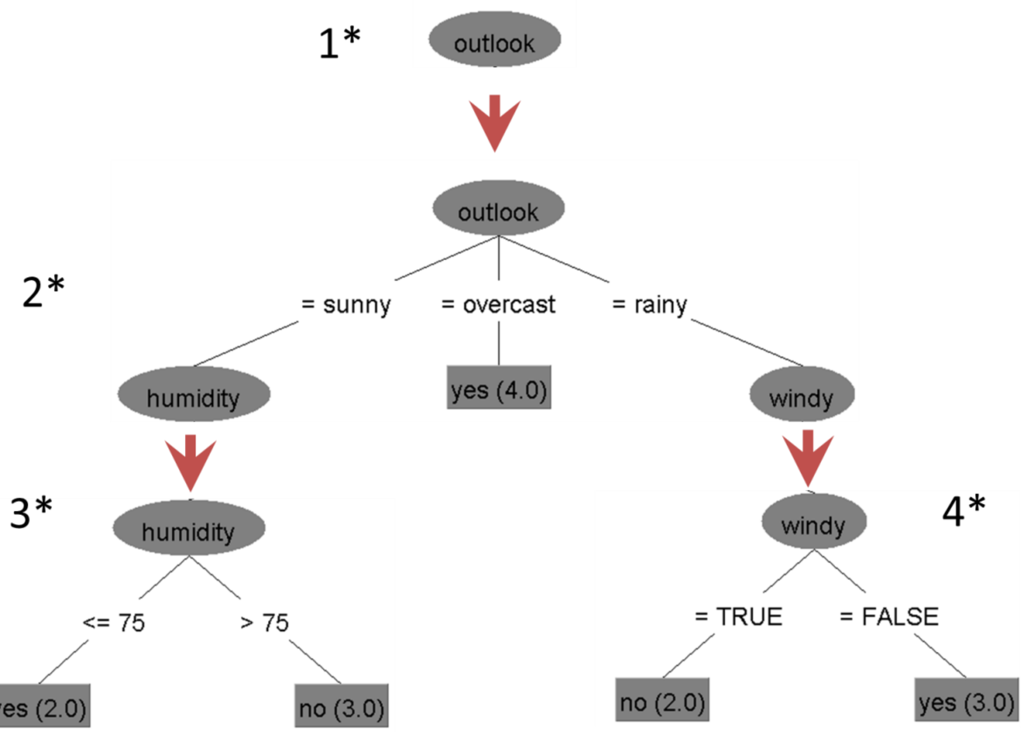

Figure 2.31 – This is an abstracted representation of how a decision tree is constructed. Start from phase 1, and keep following the numbers to get a clear picture. Note: This is a sequential representation.*

This might be a bit simple, but no more knowledge is really required, at least for this paper. Looking at *Figure 2.31*; first a root node is located from the ".arff" dataset given, in our case the root node is going to be "Outlook". After locating the root, the best split distribution is calculated; essentially J48 tries to find the best next tree branches, in our case, J48 decides that having "Outlook" branch towards Humidity, Windy and terminate if overcast is given, is the base model. Afterwards, the same method is applied to "Humidity" and "Windy" as was with "Outlook".
The original J48 is a sequential implementation, which means that, from *Figure 2.31*, phase 1* must be done first, then 2*, then 3*, and then 4*. 3* and 4* are not performed concurrently.

As you can see, it seems obvious how to parallelise this. From *Figure 2.31*, phases 3* and 4* are not performed in parallel, however with some modification, you can make them execute in parallel.

2.4 WEKA

Here I will just give some brief information on this tool/library called **WEKA**[I], and mention any relevant information concerning it.
The version of WEKA used is version 3.6.1; it is not the latest version, however it should suffice and not much has been changed for J48 specifically over the years.

WEKA is a data mining tool -written in Java- that allows one to use different types of data mining algorithms, one of which being J48. WEKA also allows one to view a representation of a decision tree (if the appropriate algorithm has been picked), just like in *Figure 2.21*.

Unless it has changed in more recent versions, WEKA's algorithms are all sequential; however some parallelism can be exploited; e.g. splicing a dataset into different parts, then classify them on different computers and compare which one is more accurate. As you can see, it may not be the most enticing way of doing things, especially if you have a large dataset, and only one computer.

2.5 JCSP

Here I will briefly talk about what JCSP is, and how it relates to this project.

JCSP[F] is a Java implementation of the CSP (Communicating sequential processes) model, it allows one to mathematically (logically) describe how a program should behave in a concurrent environment; theoretically making it easier to write concurrent systems and/or software. Another benefit this holds is making Threads in Java more lightweight, which means less resources and memory will be used for functions such as context switching (wherein a CPU core switches from one thread to another); in other words, JCSP's processes are similar to Google Go's goroutines or Erlang's processes. From this project's conclusions, the "*lightweight threads"* claim may be over-exaggerated.

2.6 SLIQ & SPRINT

SLIQ[J] and SPRINT[K] are similar algorithms. SLIQ is a decision tree algorithm that is meant to be able to handle very large datasets without putting them into memory by making the datasets go through the hard disk; SLIQ also removes the computational cost of sorting the dataset every time a split is performed. However, SLIQ suffers from still having some memory usage requirements that increase in size the larger the dataset is used. SPRINT is a better implementation wherein the authors fix the memory requirement limitation, and they also implement a parallel solution as well.

Chapter 3 – Related Work

In this chapter I will go over some papers that have tried to implement parallel/concurrent implementations of decision tree algorithms and see what they can bring to the table for my project. I will not go over the papers in great detail, so if you want the full technical details of how their work was performed, please read the papers.
Furthermore, I am sad to inform that I could not manage to find any parallel implementations of J48 specifically; I seem to be the only one so far who has attempted this, at least on an academic level.

3.1 Parallel Implementation of Decision Tree Learning Algorithms [C]

Summary: This paper's goal is to parallelise a decision tree algorithm based upon C4.5 with inspiration from SLIQ whilst retaining the same classification accuracy as C4.5. The methodologies used in the building of this parallelised decision tree algorithm are: Task parallelism, Data parallelism, and Hybrid parallelism. The authors did not attempt to parallelise pruning as they deemed it insignificant for performance improvement. The parallel decision tree algorithm's performance is measured using the *Synthetic* dataset from *Agrawal et. al*. The Synthetic dataset has some malleable qualities to it, e.g. the values of the attributes are randomly generated, and it provides different classification functions based on the complexity and value of attributes. Each test provided has been run 3 times.

The incentive for the parallelisation of the algorithm for the authors was because datasets were becoming larger and larger, classifying them was becoming too slow. The machine that the tests were conducted on possesses the following properties: four Pentium Pro CPUs, each running at 200MHz and containing 256Kb of cache memory; 256MB of RAM running on Linux 2.2.12 from the standard RedHat Linux release 6.0 Distribution.

In the end, this paper praises itself on being able to preserve the functions of C4.5 such as the same classification accuracy and the ability to deal with unknown attributes values, whilst being able to make it run in parallel. The results show good speedup and good scalability potential, however the authors do acknowledge that more tests should be done, and the performance could be bottlenecked by the communication overhead they have created in their implementation.

Critique: Let's start off with the most problematic issue, performance comparison. I am not sure if it is me, or if the authors simply assumed, but I could not find the performance measurement for the original C4.5 algorithm in their paper. This is problematic as the reader cannot tell whether they indeed achieved a speedup, or if this was just a hoax. I want to also mention that that the amount of tests that the experiments were conducted was too small for my liking, in addition to only using one dataset; however the authors do acknowledge this.

Suggestions: Other than the obvious suggestion of increasing the dataset pool, doing more tests, and comparing the results to the original, I would say perhaps the thread communication model they went with might not have been the best, I don't think languages such as C, or even Java, work that well with a shared memory model in terms of performance. I cannot really say more as their implementation is based upon C4.5, whereas my implementation is based upon J48. It would also have been nice to see some relevant code snippets or pseudo code from them in order to get a better idea of their implementation.

Applicability: In terms of applicability to my project, there isn't much I can discern from this paper because it has mostly been written in C. The methodologies they have used were certainly interesting as some of them were similar to what I have created for J48. However, one of the more interesting findings is that pruning is indeed not relevant for the overall performance of the classification operation.

3.2 A Parallel, Multithreaded Decision Tree Builder [D]

Summary: This paper attempts to parallelise C4.5 via the use of Posix Threads (Pthreads). These Pthreads are lightweight threads that allow one to change the stacksize of the threads, and have a much lower overhead of creating/destroying threads. The Pthreads' stacksize has been set to 8KB by the authors. This paper did not apply pruning to its implementation as the authors deemed it not worthy enough; the authors claimed that pruning was responsible for only 1% (or less) of the computation time; because of this, the classification accuracy may not be the same as the original's. The main dataset used for this implementation is the *Synthetic* dataset by *Agrawal et. Al*. It seems that each dataset has roughly 8 attributes; however it is not made entirely clear. They used a varied amount of instances in their Synthetic dataset, ranging from 20k to 1600k.

The authors' tree building implementation is similar to mine. They have made a thread scheduler that schedules the Pthreads, and the tree is built from top to bottom recursively. It also seems that each node is allocated its own thread, again, similar to my implementation. The difference however is that this paper's implementations switches to a serialised version if the Instance size is less than 4000 in order to avoid any contention periods being too long.

The machine they have used is a 8-processor Sun Enterprise 5000 running Solaris, with 2GB of RAM. Each processor is a UltraSPARC, running at 167MHz, and with a L2 cache size of 512KB. It is clear that the CPUs they are using only have 1 processing core, therefore their experiments would suffer from having a greater overhead constraint than a CPU with 8 cores because the CPUs are just further away from each other.

The authors have created two versions of the scheduler. The first version is a simple scheduler that partitions the work equally between the processors. The second version is called a "space-efficient scheduler", it conserves more memory by *prioritizing threads in their serial, depth-first execution order*. Also, threads that use a larger amount of memory have a lower priority, so a processor spends less time running them.

In the end, the speedup is roughly 6 times that of the original at 8 processors using the space-efficient scheduler.

Critique: A more varied dataset would have been better, so as to add more diversity into the benchmarks; however the authors do acknowledge that, and specify that they could not find (or had difficulty) in locating large enough datasets. This indeed seems a valid criticism as back in the late 90s, data was not as readily available as it is now.

The authors also used an 8 processor machine instead of a CPU with 8 cores, which I think would have given them a greater speedup; however it might not have as I am not sure as to how their parallel implementation was implemented. I am not sure, but I believe atomic functions were not created back then, or perhaps were too new to be noticed, so they probably used locks, which introduce contention; so a CPU with many cores might not be as beneficial.

The authors mention that Pthreads are very good at creating/destroying threads; however it might have been interesting to see if they could have created a version wherein the Pthreads aren't destroyed, instead they are reused. However I do not know how Pthreads work specifically, so perhaps that is what Pthreads do, reuse threads, whereas it looks like threads are being destroyed from an outsider.

This is a minor complained perhaps, but I would have liked to see the speedup in milliseconds instead of factors, as this would allow someone like me to be able to measure it against my implementation easier.

Suggestions: It would be hard for me to make some of the more interesting suggestions because this paper was published in 1998. For example, a suggestion on using a CPU with 8 cores instead of 8 processors would have made an interesting comparison, however I do not believe a single CPU with 8 cores was even available to purchase at the time.
A reasonable suggestion perhaps would be to have not destroyed threads when they finished tasks, but instead reuse them. But as I mentioned before, this may be how Pthreads work, or I suppose in this paper's case, how they worked at the time.

Applicability: This paper comes in with the same conclusion about pruning as the last one, the fact that pruning is deemed irrelevant for the overall performance. In this paper Pthreads were used, which were lightweight in comparison to what was available at the time. I am not exactly sure how heavy these Pthreads are, or how they run under the hood to be able to compare them to JCSP or Java's threads, so I cannot entirely make a comment on this. However, I do not think Pthreads would yield much of an improvement in comparison to Java's or JCSP's threads.

3.3 Parallel Formulations of Decision-Tree Classification Algorithms [E]

Summary: In this paper, the authors parallelised a modified C4.5 algorithm, with inspiration from the SLIQ, and SPRINT decision tree algorithms. The authors have also managed to retain the same classification results as the serialised version. The authors have tested three different approaches to parallelising the C4.5 algorithm: *Synchronous Tree Construction Approach*, *Partitioned Tree Construction Approach*, and a hybrid of the two. Pruning has not been parallelised as they authors mentioned that pruning was responsible for less than 1% of the computation time of the algorithm. The tests have been performed using different variations of the Synthetic dataset by *Agrawal et. Al*. The dataset contains 9 attributes in total, 3 categorical and 6 continuous.

Synchronous Tree Construction Approach: "*In this approach, all processors construct a decision tree synchronously by sending and receiving class distribution information of local data.*"
Partitioned Tree Construction Approach: "*In this approach, whenever feasible, different processors work on different parts of the classification tree. In particular, if more than one processors cooperate to expand a node, then these processors are partitioned to expand the successors of this node.*"
Hybrid Parallel Formulation: "*The hybrid scheme keeps continuing with the* [Synchronous Tree Construction] *first approach as long as the communication cost incurred by the first formulation is not too high. Once this cost becomes high, the processors as well as the current frontier of the classification tree are partitioned into two parts.*"

As mentioned previously, the authors took inspiration from the SLIQ and SPRINT algorithms; in particular, the authors implemented a pre-sorting approach, similar to that of the SLIQ and SPRINT algorithms. This way, there should be a performance increase when dealing with continuous attributes, as sorting them every time a processor moves onto a new node will not be required.

The authors used a processor communication model rather than a shared memory model or other models.

The hardware specification for this paper's tests is as follows: IBM SP2; 16 processors each with 66.7MHz, and 256 RAM; however they do mention that they will be going up to 126 processors. The operating system is the AIX version 4; the processors communicate via a high performance switch (hps). The authors also mention that keep the "attribute lists" on the hard disk and use the memory only for storing program specific data structures, class histograms and the clustering structures.

The results in this paper say that the Synchronous approach performs well with 2 processors; however it suffers at 4 or more. The Partitioned approach performs well until 8 processors wherein it decreases in performance thereafter. The Partitioned approach suffers from load imbalance and high data movement for each partitioning phase. However, the Hybrid approach seems to perform well all around and shows good scalability.
In the end, they achieved a speed-up factor of 66 (looking at their graph) with 126 processors. At 64 processors they managed a speedup factor of about 48.

Critique: This paper uses a similar hardware setup as the previous **[D]** one; however the processor count is much larger, although the clock speed for each one is lower. I would have liked to see a comparison of a CPU with multiple cores to that of a computer with multiple CPUs; however this paper was published in 1998, just like the previous **[D]** one, so I should not hold it against them.

It is not explicitly stated, or explained in enough detail I feel, but I believe the authors used a communication model instead of shared memory model. It would have been interesting to see another model, however with their setup, any other model would probably not have worked well.

I would have liked to see different types of datasets instead of this Synthetic one they have used, although it does seem fairly popular, at least back then.

Suggestions: Again, it is hard for me to make any reasonable suggestions because of the time gap; but I suppose a CPU with multiple cores instead of different CPUs would have been interesting to see.
They do mention that they have used a "high performance switch" in order for the processors to communicate with each other, perhaps different types of these "switches" would have been interesting to benchmark.

Applicability: Again, similar conclusion in regards to pruning as the previous papers. There were also some interesting methodologies applied here that could be utilised in my implementation, such as the Hybrid method because I do think that maybe one of the bottlenecks for my implementation is the overhead switching incurred by having to switch from sometimes completely different tasks.

Chapter 4 – Methodologies, Tools and Practices

Here I will simply list the different methodologies, tools and practices I have used in my project. I will also list the hardware specifications of each of the devices the tests have been run on, and the operating systems they have used; as well as how I have conducted the tests.

4.1 Methodology

Truthfully told, I have not used any sophisticated methodologies as described in the **[C]**, **[D]**, or **[E]** papers; I have simply used my knowledge to approach the problems I was faced with and it yielded some interesting solutions that seem similar to the papers I have looked at.

However, I have applied some practices from the "*Java Concurrency in Practice*"**[A]** book, such as:
-First make it thread safe, and then make it fast.
-Double check whether you really do need a lock.
-It may seem like it works fine, however CPUs, software, and Operating systems are fairly complicated these days, make sure to test it thoroughly.

The way I measured the decision tree construction time was to simply take the time that was displayed on WEKA's output. Cross-validation was a bit more complicated as I had to write a new piece of code that takes the time before the cross-validation operation has started, then take it again after it has finished, and then find the difference.

4.2 Systems Operated On

I have officially tested my implementation on three different systems. Hopefully three will be enough to test for the thread safety (and efficiency) of my implementation. The three different systems are:

Windows 7 Ultimate 64bit Service Pack 1: This system is my home system, so I have the most control over how things operate on this system. The hardware specification for this system is: an i5-4690k@3.5GHz, 4 DDR3 Memory sticks each at 4GB and 1600MHz, an SSD that the system is installed on, and a Z97 PC Mate motherboard.

Windows 10 version 1803: I have less control on this system compared to the first one, so sometimes background processes might have interfered with the runtime of my project. The hardware specifications are as follows: an i7-4790k@3.6GHz, 4 DDR3 Memory sticks each at 4GB and 1600MHz, the system was installed on a HDD however it was connected to a server which may have hampered the performance, also I am not sure what motherboard was used.

Raptor - Ubuntu 16.04: I have even less control on this system than the previous one, so I am not sure how much the background processes (or outside elements) could have interfered with me running my project on it. The hardware specification is not going to be as clear because of security; however what I managed to find out is the following: there are four CPU sockets, each holding an E7-4830@2.2GHz; it is unknown how many RAM sticks there are, however the total memory seems to be at around 208GB, with each stick clocking at 1066MHz; it is also unknown what motherboard was used, or whether the system was installed on a HDD or SSD.

4.3 Tests and Benchmarks

The time it took to run a classification was taken via calculating the time it took to run it in the program itself. For example, the current time was taken before the classifier was built, then after execution, another timer was taken and was compared to the first to attain the execution time; this then was displayed on either the terminal, or the Explorer details window. The tests times were recorded on Google Spread sheet so as to have access to it on different systems online.

Different types of tests were conducted, some with cross validation, and some with just using the training set. Cross-validation works by constructing the tree again depending on how many folds were specified, e.g. folds of 1000 would construct the tree 1000 times. However, this could have skewed the results as I have also parallelised cross-validation, meaning that whilst a thread works on constructing one tree, another could be constructing a different tree in parallel.

The three tables I have created that feature the results of execution times can be found in Appendixes: **[AC]**, **[AD]**, and **[AE]**.

Chapter 5 – Design and Implementation

In this chapter I will describe how and why I implemented the modifications that I did in order to make it run concurrently, though they might be too simplistic for some. I will go through some parts of the source code; however, only the important parts, I will not be explaining things like why I used an ArrayDeque instead of an ArrayList etc…

5.1 Tree Construction

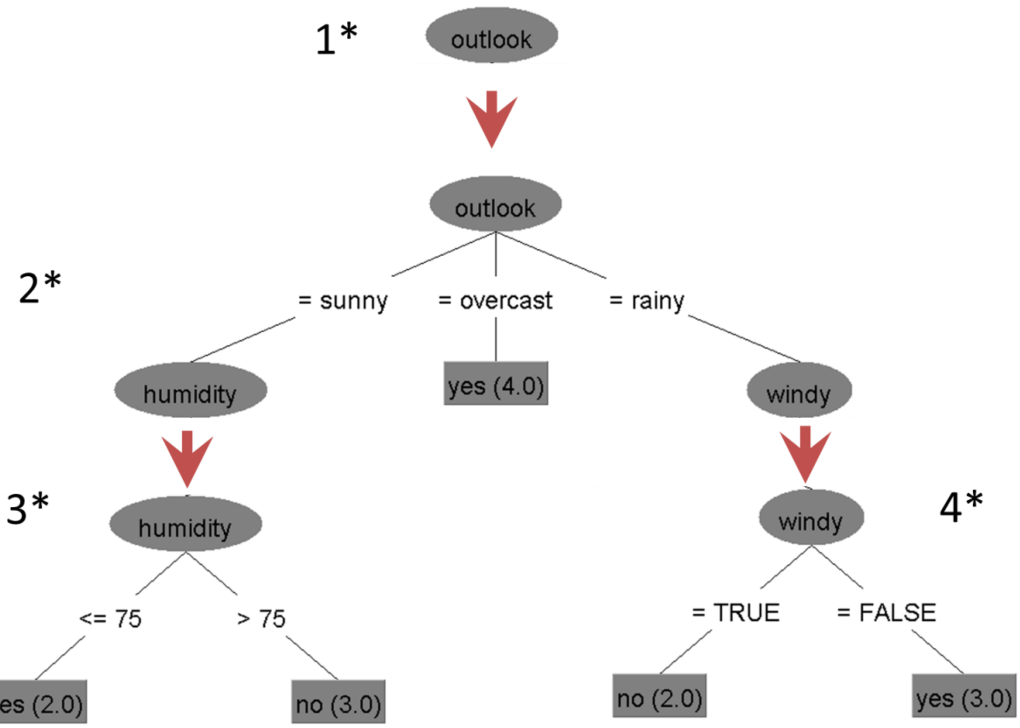

Figure 5.11 – This is an abstracted representation of how a decision tree is constructed. It lists the various phases of how a classifier such as J48 goes about constructing this tree.

Figure 5.11 may seem familiar, that is because it is; it’s the abstracted tree construction figure last seen in Chapter 2. The tree construction process is essentially the same as the sequential one, except that J48 can now run phases in parallel. Looking at *Figure 5.11*’s scenario, my implementation of J48 would start off at phase 1*, proceed sequentially (relatively) to phase 2*, and once there are more phases to work with, it would then run in parallel; so in our case phase 3* and 4* would run in parallel.

```
157     public void buildTree(Instances data, boolean keepData,
158             ConcurrentLinkedQueue<BuildTreeTask> taskQueue)
159             throws Exception
160     {
161         final AtomicReferenceArray<Instances> localInstances;
162         final ConcurrentLinkedQueue<Integer> indexQueue =
163                 new ConcurrentLinkedQueue<Integer>();
164 
165         int i;
166 
167         if(keepData)
168         {
169             m_train = data;
170         }
171         m_test = null;
172         m_isLeaf = false;
173         m_isEmpty = false;
174         m_sons = null;
175         m_toSelectModel = new C45ModelSelection(
176                 m_toSelectModel.getMinObj(),
177                 m_toSelectModel.getData()
178                 );
179 
180         m_localModel = m_toSelectModel.selectModel(data);
181         if( m_localModel.numSubsets() > 1 )
182         {
183             localInstances = new AtomicReferenceArray<Instances>( m_localModel.split(data) );
                data = null;
185             m_sons = new AtomicReferenceArray<ClassifierTree>( m_localModel.numSubsets() );
186             for( i = 0; i < m_sons.length(); i++ )
187             {
188                 indexQueue.add(i);
189             }
190             for( i = 0; i < POOLSIZE; i++ )
191             {
192                 BuildTreeTask task = new BuildTreeTask( indexQueue, taskQueue,
193                         localInstances, this);
194                 taskQueue.offer(task);
195                 MAINTASKQUEUE.put(task);
196             }
197         }
198         else
199         {
200             m_isLeaf = true;
201             if( Utils.eq(data.sumOfWeights(), 0) )
202                 m_isEmpty = true;
                data = null;
204         }
205     }
```

Figure 5.12 – A part of the source code concerning the construction of a decision tree. This is the parallel implementation.

Figure 5.12 shows a more detailed look at how a decision tree is constructed via code. Ignoring some irrelevant details, line 180 in *Figure 5.12* invokes the method which calculates the best fit model for the tree with the current dataset, the same as described previously. Next we create an *AtomicReferenceArray*, this is an atomic variable, it works in the same way I mentioned in the pre-requisite reading; however to expand on detail for the current context, the *AtomicReferenceArray* is an atomic array that holds an array of *pointers* (references) to other objects. These pointers themselves are thread safe (if the appropriate atomic functions are used), however if you were to access the objects that these pointers point to, then they will not be thread safe. Essentially, you can swap what pointers reside in this atomic array safely, however if you were to modify the objects themselves, it would still result in a thread safety violation.

m_sons is essentially the branches of the node; so if the root node was "*Outlook*", the *m_sons* would be the lines that will point to whatever the *localModel* has decided when it ran it's calculation, looking at *Figure 5.11*, they will point to Humidity, Windy, and a Boolean. "*localInstances*" is the list of available data attributes left to be put into the tree.

Later on in the code, lines 192 and below, a task object is created and sent to two different queues. The "*taskQueue*" is used by a thread to discern whether it can move on, or whether it needs to keep waiting for the tasks to be completed. The "*MAINTASKQUEUE*" holds tasks that threads can grab and work on.

Once the task(s) has been put into the queues, it terminates from this method, and waits for the task(s) to finish before it can move on in the previous method. However, while it waits, it will grab a task from the main task queue and compute it.

```
46          @Override
            public void begin()
48          {
49              try
50              {
51                  Integer index = indexQueue.poll();
52                  while( index != null )
53                  {
54                      ClassifierTree newTree = tree.getNewTree( instances.get(index), taskQueue );
55                      tree.m_sons.set( index, newTree );
56
57                      index = indexQueue.poll();
58                  }
59              }
60              catch(Exception ex){}
61
62              finished = true;
63          }
64      }
```

Figure 5.13 – This is the main execution method from the BuildTreeTask class. It is a task object that is executed by a thread if it sits in the MAINTASKQUEUE variable.

Figure 5.13 shows us the source code for the *BuildTreeTask* class, one of the task classes. This is a continuation of the *Figure 5.12*; basically the thread that will run this task will execute the code. As you can see from lines 54, we essentially perform an atomic get function from our instances variable, which then we construct a branch from via the *getNewTree* method.

Just a line below we use another atomic function to swap whatever pointer is located at the specified index with a pointer that points to our newly constructed branch.

The last line (62) simply flips the Boolean, telling the thread that is waiting for these tasks to be done that it is finished and it can move on to its next stage.

This is essentially how a tree is constructed via a basic overview, there are obviously other parts that haven't been covered, however I do not want to spent pages explaining every minute detail, this was just the overall concept.

5.2 Abstract view of Queue

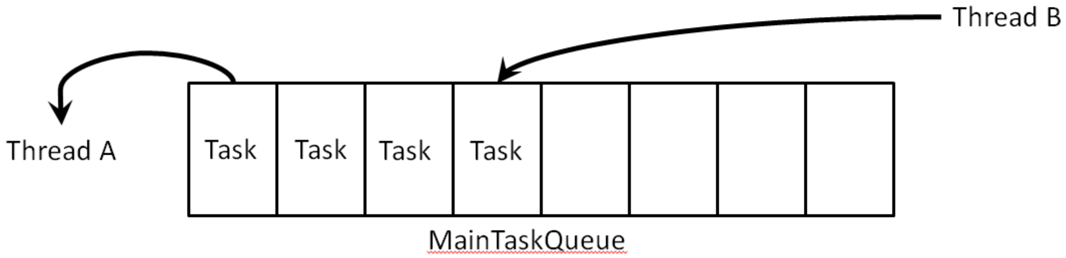

Figure 5.21 – This is an abstracted representation of the MAINTASKQUEUE variable queue which can be found in the Explorer.Java class.

To better illustrate how the main Queue works, *Figure 5.15* shows an example. In our example, let us assume that Thread B is currently doing some processing, e.g. constructing a tree; let us also assume that Thread A is waiting for some tasks to appear in the Queue. Eventually, Thread B will come across a parallelised solution, which would tell it to create various tasks and put them into the *MainTaskQueue*. When there aren't any tasks in the *MainTaskQueue*, the threads will wait for a task in a blocking manner; this simply means that the threads will not be running, they will be sleeping until they are interrupted by the *MainTaskQueue* with a task.

Once there is a task in the *MainTaskQueue*, in our example, Thread A will take the task and remove it from the queue in a thread safe manner so as other threads do not perform the same task at the same time.

Figure 5.21 shows a simple example, however in reality it is more complicated, Figure 5.22 shows a more accurate depiction of how it is performed in realtime.

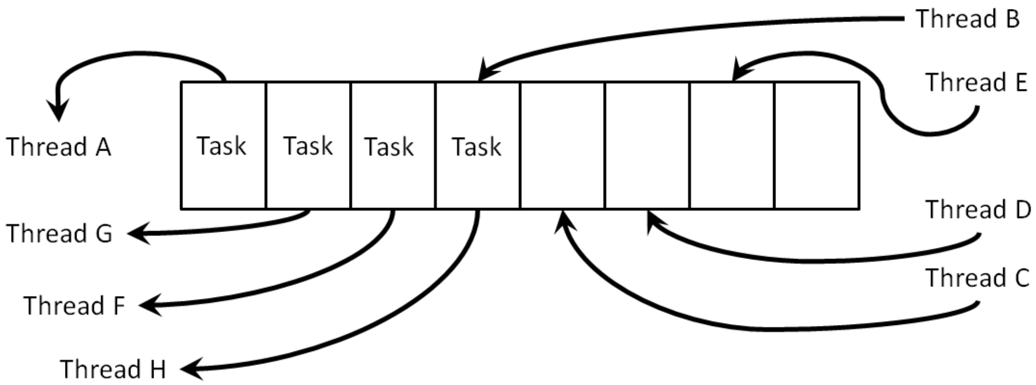

Figure 5.22 – This is a more complicated abstracted representation of the MAINTASKQUEUE variable queue which can be found in the Explorer.Java class.

5.3 Scheduler

Here I will just describe briefly how the scheduler I made works, as well as an abstracted view of how it handles the threads.

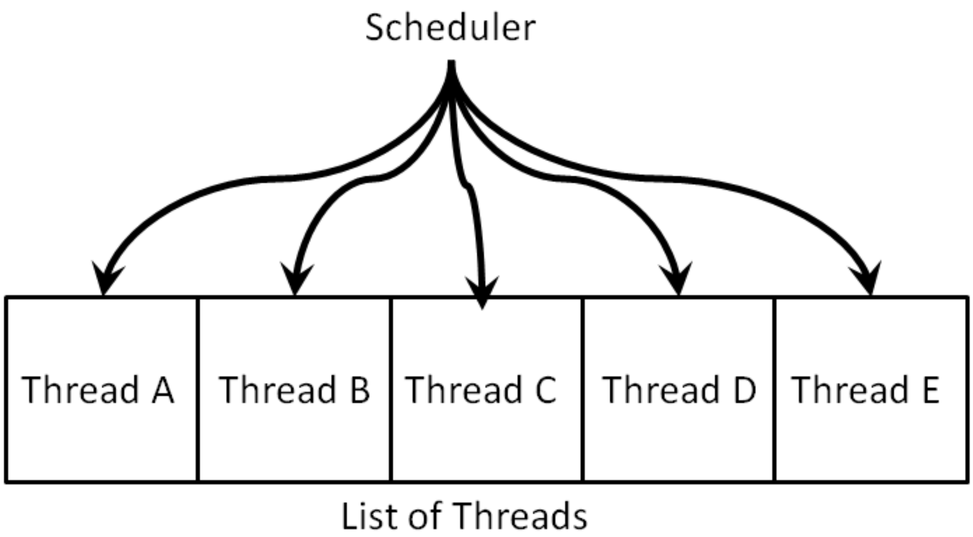

Figure 5.31 – An abstracted representation of the scheduler creating threads and putting them into the list of threads.

```
        //initialise threads
        for( int i = 0; i < POOLSIZE; i++ )
        {
            CommonThread thread = new CommonThread();
            THREADQUEUE.add( thread );
            thread.start();
        }
```

Figure 5.32 – The source code that generates threads, puts them into a list, and then starts them.

Figure 5.32 shows the source code of how the scheduler creates threads. In the example, a *CommonThread* class is created, added into a *ThreadQueue* and then it tells the class to start running the thread. When the thread starts running, it immediately goes to the *MAINTASKQUEUE* queue and checks whether there are any tasks for it to do, if there aren't, then it goes to sleep until it is interrupted by a task (blocking queue).
Figure 5.31 illustrates a more abstracted view of how the scheduler generates threads and puts them into a list of threads to keep.
There is not real reason to keep the threads in a list as currently I do not do anything with the threads themselves, perhaps in the future one could implement a *shutdown* feature to eliminate threads safely; however in the current implementation, the threads are killed off when the process is terminated.

5.4 Split Model Distributions

Here I will briefly describe how I implemented the split model distribution and go over the source code for it; it basically is the *selectModel()* method from line 180 in *Figure 5.12*. However, unlike the previous section, I will not go over how the distribution is picked as it is more complicated this time around; if you want to know how it works in more detail, refer to the "*Machine Learning*"[B] book by Tom M. Mitchell.

Just as a side note: I originally just modified the tree construction method; it yielded fairly reasonable results when running J48 with small datasets, however when running larger datasets (yeats.arff) it took longer (an extra two or three seconds) than with the sequential version; this length would probably be longer with larger datasets.

This time the source code is too large to put on this page, so I have put it in the Appendix. Appendix **[AA]** shows the source code for the *selectModel* method, which is responsible for picking the right distribution model. It is not the entire method, I have only kept the relevant parts. If you have read the tree construction section previously, there is not much more I could add. I essentially approached this in a similar way to the tree construction method; the idea being that I relied on atomics if I needed a variable to be accessed/written by multiple threads. Just like last time, at lines 145 we create tasks and put them onto two different queues, one into the thread queue so the task can be computed by other threads, and the other queue acts as a placeholder to allow the current thread to check whether the tasks have finished before it continues.

Appendix **[AB]** shows us the source code for the model selection task. As you can see, it goes through some algorithm with which it discerns the *infoGain* and builds a classifier model to test later on.

There is not much going on here in terms of multi-threading; however as you might have noticed from previous examples, I created a thread safe list of index's called the *indexQueue*. This list is thread safe and the idea behind it is that a thread takes an index number from the list (which removes it from the list), then does some processing using the parameters of the index, and then it just simply repeats this until the list is empty, which it finally finishes. This way I only spawn tasks equal to the amount of the size of the *POOLSIZE* variable. The *poolsize* variable holds the size of the number of threads. The alternative would be to simply create as many tasks as I need to, however objects are much more expensive than simple integers, and some of these datasets would require hundreds of thousands of task objects to be spawned. With this method I can avoid bloating the memory.

At the end of the source code for *C45ModelSelection* class, it discerns the correct model via some formula that the original WEKA developers have picked. This last part of the code has not been modified by me as the computational impact of it is fairly small, so there was no need.

5.5 Cross-Validation

I will not be presenting the source code for my cross-validation parallelisation because it is fairly simple and no new features have been used in respect to the previous section; instead I will simply detail the overall execution of it.

The cross-validation method is pretty simple; it generates tasks according to the *poolsize* variable. The cross-validation tasks simply run the *buildclassifier* method, which starts the construction of a classifier, and once the thread has finished constructing the classifier, it puts it into a list of finished classifiers that will later be assessed.
Once all of the classifiers have been built, they are then evaluated. I am not entirely sure how the evaluation process is done, so I cannot describe it here. I also did not modify the evaluation methods as the computational impact was insignificant.

5.6 JCSP

The way I implemented JCSP[F] is simple, perhaps too simple. There is no real need to show source code, so I will simply talk in an abstract sense.
The way I had originally done it, is by creating my own scheduler that creates and maintains threads. This scheduler does not destroy threads when they have finished, instead it reuses them. My JCSP implementation essentially performs the same operation; however the difference being that my custom scheduler now uses JCSP's processes instead of Java's threads.
I did have to modify some parts of my source code to get it to run, however I have not modified the entirety of my code to work in the CSP language that came with JCSP; this, I realise, might have not played to the strengths of JCSP's processes.
The strength of JCSP's processes comes from being able to interact with the CSP model, which I have not used. I also believe that JCSP's processes were created to be able to have minimal performance impact on the destruction or creation of processes, however my implementation does not create or destroy any processes, aside from the initial start-up phase of WEKA.

5.7 Pruning

I have also modified pruning in order to see if it would yield much of a speedup. I have attempted two versions. The first version works by creating a task at each branch. This then would execute all of the branches in parallel. However the issue, as some of you might have guessed, is that this could cause a lot of overhead, especially a context switching overhead. Indeed this was true as my results yielded very bad execution times.
The second version essentially creates the pruning tasks only from the root branch. This yielded much better times, however they were not better than the original's; though fairly similar.

Chapter 6 – Results and Analysis

6.1 Terminal

There is something I must mention about WEKA specifically that may have had an impact on my results. There are two different versions of how WEKA will approach the handling of classifiers: the terminal version, and the explorer version. The explorer version is what I mainly worked on, and aside from some differences, the terminal version is similar. The main difference I would like to mention is that the terminal version seems to have an extra step in the construction of a classifier. Once the terminal version has finished classifying, it will give some extra information, that being: "*Time taken to test model on training data:*"; the explorer version does not have this.

I have taken some time to look over the source code of WEKA in order to figure out why this is, in my conclusion, I believe the explorer version does indeed have the same method being processed as in the terminal version; however it does not seem to show it in the results tab; furthermore, the time taken to perform it seems shorter in the explorer version than in the terminal version.

It is strange indeed, however the impact on performance should not be great as this operation seems to be performed only once, and does not have too great of a computational cost. If cross-validation is performed, the operation is still only performed once.

6.2 Datasets

contact-lenses.arff	25/01/2018 14:29	ARFF File	3 KB
cpu.arff	25/01/2018 14:29	ARFF File	6 KB
cpu.with.vendor.arff	25/01/2018 14:29	ARFF File	7 KB
iris.arff	25/01/2018 14:29	ARFF File	8 KB
kdd_ipums_la_99-small.arff	01/02/2018 15:16	ARFF File	3,774 KB
labor.arff	25/01/2018 14:29	ARFF File	9 KB
segment-challenge.arff	25/01/2018 14:29	ARFF File	196 KB
segment-test.arff	25/01/2018 14:29	ARFF File	108 KB
sick.arff	01/02/2018 15:16	ARFF File	299 KB
soybean.arff	25/01/2018 14:29	ARFF File	199 KB
test.arff	05/02/2013 22:12	ARFF File	107,530 KB
train.arff	05/02/2013 22:11	ARFF File	536,427 KB
waveform-5000.arff	01/02/2018 15:17	ARFF File	1,055 KB
weather.arff	25/01/2018 14:29	ARFF File	1 KB
weather.nominal.arff	25/01/2018 14:29	ARFF File	1 KB
yeast.arff	22/07/2018 17:10	ARFF File	35,298 KB

Figure 6.21 – This is the list of all the datasets I have used. Some may have appeared in my tests, whilst others I have ignored.

Figure 6.21 shows my collection of datasets. I have not used all of them in this project; however the largest ones are my favourite, and have been used the most: train.arff, test.arff, and yeast.arff.

The test.arff and train.arff are the same datasets except that the test.arff is much smaller. The train.arff is meant to be used as the training dataset, whilst the test.arff is meant to be used as the testing data; however I have ignored this fact and simply used cross-validation instead, or just the standard "train upon itself" options in WEKA, wherein it reuses the same training dataset to test the tree.

6.3 Results

There are various things I want to talk about in regards to my results; however, I think it would be best to first start off with a simple graph to demonstrate the speedup.

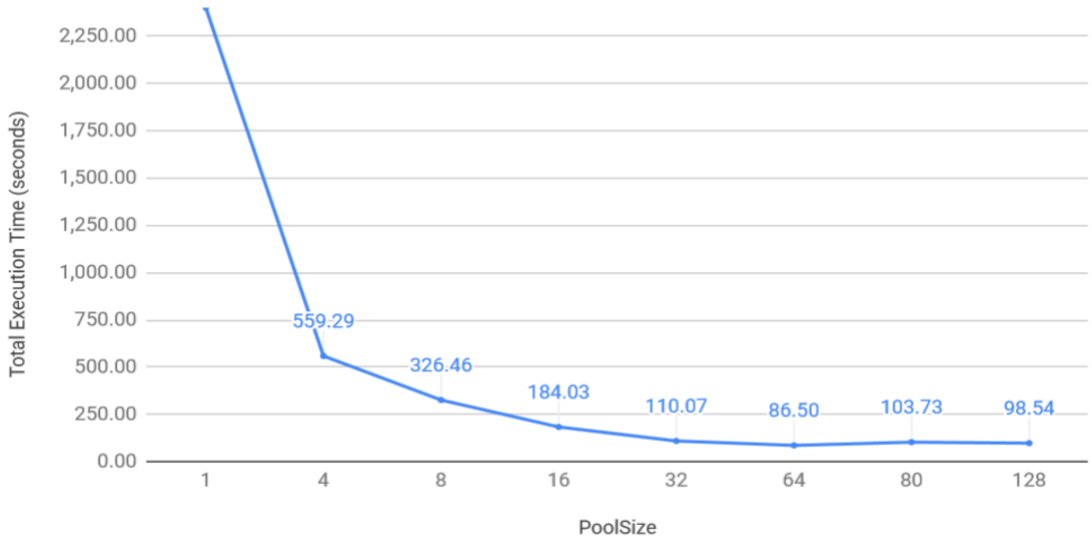

*Figure 6.31 – This is a graph representing the average time it took to build a classifier. This was executed on the raptor machine via the terminal, and is not the JCSP version. The result from poolsize of 1 belongs to the original non-parallelised version of WEKA. The results were extracted from Appendix **[AD]**.*

87.14	88.55	86.84	89.88	124.25	129.65	130.57	130.06	130.49	120.60	121.48	120.21

Figure 6.32 – This is a more detailed look at what went wrong in Figure 6.31 with a poolsize of 80 being slower, these values are in seconds. This was taken from the non-JCSP parallelised table.

Figure 6.31 shows us the average execution times for running the concurrent version of J48 via the Terminal. There is some good speed-up here; for example, the original took quite a lot longer than the parallelised version with a poolsize of 4, that is roughly a speed-up factor of 4 times. Sadly, this ridiculous improvement is not continued, as the speed-up factor from a poolsize of 4 to 8 is only about 1.7. This keeps decreasing up until a poolsize of 80, where something strange happens.

You may be wondering why the poolsize of 80 is slower than 64. Some of that is the fault of the raptor system itself; as I mentioned in the methodology chapter, I do not have much control over raptor. If we look at *Figure 6.32*, we can see that at first, the runtime is just a bit slower than the poolsize of 64, however then there is some sort of spike. I am not certain what that spike could have been, however it is most likely a different user running some of their own computations, perhaps it was just regular maintenance, or perhaps I was limited on purpose by the system because I was hogging up too much of the system's resources…

The experiment for *Figure 6.31* was executed dozens of times and the results can be found at Appendix **[AD]**. A similar execution setup can also be found in the JCSP version in Appendix **[AE]**.

Dataset	Method	Total Execution time	Subsequent runs	Poolsize	CPU cores	GUI
yeast.arff	100 folds:cross-val	301.76	No	4	4	Explorer
yeast.arff	100 folds:cross-val	339.09	No	4	4	Explorer
yeast.arff	100 folds:cross-val	330.53	Yes	4	4	Explorer
yeast.arff	100 folds:cross-val	315.34	Yes	4	4	Explorer
yeast.arff	100 folds:cross-val	317.51	Yes	4	4	Explorer
yeast.arff	100 folds:cross-val	317.78	Yes	4	4	Explorer
yeast.arff	100 folds:cross-val	307.26	Yes	4	4	Explorer
yeast.arff	100 folds:cross-val	305.33	Yes	4	4	Explorer

Figure 6.33 – Table entries taken from the parallelised non JCSP version table.

Subsequent runs are when an execution was executed straight after the same prior execution, it was part of the same process. For example the yeast.arff dataset was executed, this is not a subsequent run, however if another execution was to follow straight after, then this would be a subsequent run. Originally I thought that subsequent runs would execute quicker, however that is not the case. *Figure 6.33* shows us that there does not appear to be much of a difference between the executions that took place. My original assumption was that the first execution would prepare everything up, as in, Java's JVM would perform some optimisations on some loops to get them to run quicker, since the JVM is a JIT type of compiler, which optimises code while it is in runtime, in addition to the normal compilation optimisations; however that appears not to be the case, or at least not noticeable.

Dataset	Method	Total Execution time	Subsequent runs	Poolsize	CPU cores	Logical Processors	GUI
waveform-5000.arff	use training set	0.40	No	1	4	3	Explorer
waveform-5000.arff	use training set	0.55	No	1	4	3	Explorer
waveform-5000.arff	use training set	0.38	Yes	1	4	3	Explorer
waveform-5000.arff	use training set	0.37	Yes	1	4	3	Explorer
waveform-5000.arff	use training set	0.47	No	8	4	3	Explorer
waveform-5000.arff	use training set	0.18	Yes	8	4	3	Explorer
waveform-5000.arff	use training set	0.15	Yes	8	4	3	Explorer
waveform-5000.arff	use training set	0.11	Yes	8	4	3	Explorer

Figure 6.34 – Table entries taken from the parallelised non JCSP version table. This shows a smaller dataset.

Figure 6.34 shows us another dataset, though much smaller. What is interesting here is that Subsequent runs seem to make a relatively large difference. We can see that there is not much of a difference between the original WEKA execution (poolsize: 1), and the parallelised execution (poolsize: 8). However, there is a relatively significant difference between the subsequent runs of the original WEKA against the parallelised version.
Now that I have a clearer picture of it, I believe the JVM's JIT compiler didn't have enough time to optimise the non-subsequent runs because they were just so short. However, in *Figure 6.33*, because the dataset took so much longer to run, the JIT had a lot more time to optimise the various loops and other parts of the code.

6.4 Comparing JCSP against Concurrent Version

Here I will just compare and analyse the results I have received from the JCSP version to that of my concurrent version.

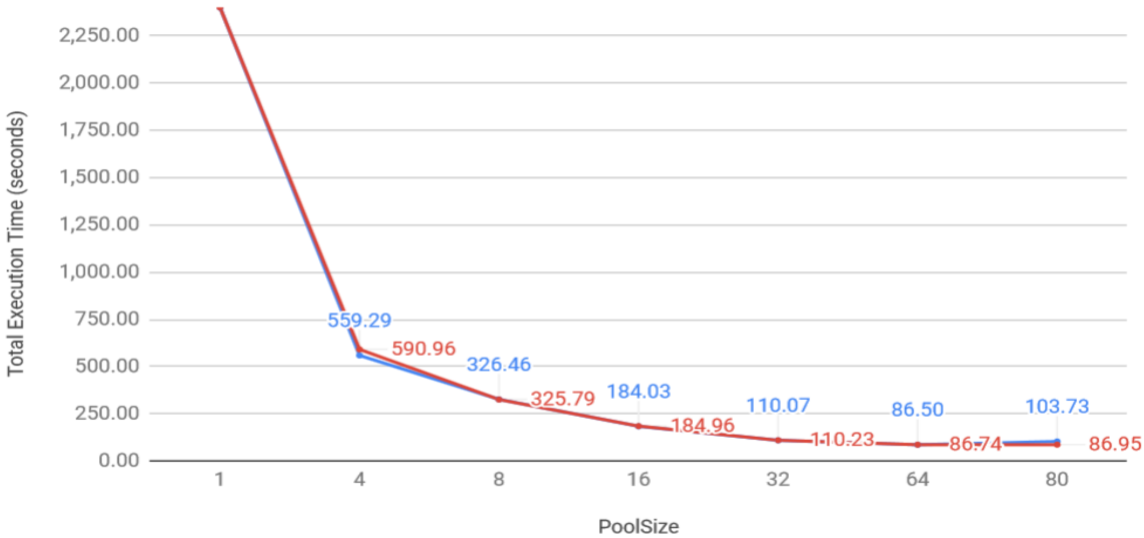

Figure 6.41 – Graph comparing the average execution times between the concurrent (blue) version and the JCSP (red) version. Both have been executed on the raptor system.

Looking at *Figure 6.41*, we can see that there is barely a difference between the two versions. There are minor differences I suppose, such as for the poolsize of 4, we can see that the concurrent (blue) version is quicker by 30 seconds, though perhaps it is more likely that there might have been another spike on raptor again. At the end, poolsize of 80, we can see the JCSP version is better, however that is because there was a spike for the concurrent version that pushed the results up higher.

Whether you are surprised (or not), there appears to be no noticeable difference between the concurrent version and JCSP version. As mentioned previously, I have not used the JCSP library to its fullest as I have only used the JCSP processes, not the CSP library that might have been specifically optimised for JCSP processes.

6.5 Pruning Time

Dataset	Total execution time(seconds)	Prune time(ms)	GUI
yeast.arff	11.830	62	Explorer
yeast.arff	10.840	16	Explorer
yeast.arff	11.140	16	Explorer
yeast.arff	11.170	16	Explorer
yeast.arff	11.280	16	Explorer
yeast.arff	11.300	42	Explorer
yeast.arff	11.340	16	Explorer
yeast.arff	11.490	15	Explorer

Figure 6.51 – Entries taken from the pruning table. This shows the how long pruning specifically took to compute for the yeast dataset. This has been run on the i5:4690k on the original WEKA version.

As can be seen from *Figure 6.41*, the computational time dedicated to pruning is very small in comparison to the whole. In the worst case scenario, the one that took 62 milliseconds to run, the computation time of pruning is worth less than 1% of the total execution time, for the yeast dataset. It seems that according to my conclusions, and that of papers **[C]**, **[D]**, and **[E]**, pruning is indeed not worth parallelising.
However, out of curiosity, I have attempted to parallelise pruning and have been somewhat successful; though the results show no improvement, so it would be best to leave pruning out of the equation. The results can be found on Appendix **[AC]**.

6.6 Comparison Against Other Concurrent Methods

To be blunt, I cannot perform good comparisons to most papers (even the ones I picked) because most of them simply do not state the execution time, they only specify the speed-up factor; and to make matters worse, they usually don't mention the entire properties of their datasets either. For example, some will state that they are executing a dataset of 100k instances; however they would fail to mention the attribute size, or would mention that their dataset has somewhere in-between 5-10 attributes.
Another problem is that none have attempted to parallelise J48, a Java implementation of a decision tree classifier; they have mostly attempted C4.5, a C implementation of a decision tree classifier. The problem here would be the difference between the two languages; one could perform parallelisation better than on the other (sometimes vastly better).

If we decide to ignore the (hard) comparison problems, then my one does not perform all that well. Going back to the Related Work chapter, for the "*Parallel Formulations of Decision-Tree Classification Algorithms*"**[E]** paper, they managed to achieve a speed-up factor of about 48 times with roughly 64 processors (judging from their graph). In comparison to my one, I only managed to achieve a speed-up factor of about 27 times with a poolsize of 64, on an 80 core CPU. However my dataset is very large (train.arff), the speed-up factor would be likely much lower if I used a much smaller dataset. So in conclusion, my one in comparison is very slow.

6.7 Classification Accuracy

I have executed my modified version as many times as I could in order to find out if there are any thread safety issues, as well as to make sure the classification accuracy is as accurate as the originals'. In conclusion, it appears to be the exact same in terms of classification accuracy; however I cannot guarantee a 100% certainty.

Chapter 7 – Evaluation

In this chapter I will evaluate my project: what worked, what didn't work, what I could have done instead, etc…

The biggest problem right off the bat would be WEKA itself, or more specifically, J48. As I have mentioned earlier, I could not manage to find any parallel/concurrent implementations of J48. This made it harder for me to figure out how well my implementation is performing, or even how well things in general were going.
There were also some general problems with WEKA, such as having two somewhat similar, but still different implementations of how WEKA works: the terminal/explorer versions. Figuring out how WEKA works was not particularly hard, it just took a lot of time, especially for me since I did not have any knowledge of classifications, or data mining in general.
Parallelising J48 was hard as I had to keep on thinking of different kinds of implementations for the methods, as well as make sure it was all thread safe. There were also some difficulties in trying to modify some parts of WEKA, such as how it handled datasets. The way it handled datasets was through a custom class called "*FastVector*", I'm still not entirely sure how it works, but modifying it too much would cause a noticeable slow down within the entire algorithm. My guess is that the creator(s) spent a lot of time working and optimising a lot of it, which means changing bits and pieces wouldn't work all too well.

Perhaps it would have been a better idea to implement the entirety of JCSP; however that would have caused me to have to redesign my concurrent implementation, and probably even more than that so as to take full advantage of CSP. From experience and reading through the JCSP documentation, I don't think re-designing my whole project to use CSP would have made much of an impact; it would have made it easier to detect thread safety errors, but in terms of performance speed, I don't think it would have yielded much.

Towards the end of the project, I attempted to try to implement GPU acceleration, however this proofed fairly difficult as not only was the OpenCL[G] extension complicated, I would have also had to have re-designed some core parts of WEKA in order for OpenCL to accept them.

In the beginning I did not look up any methods or ideas for how to implement a concurrent version of J48. This was due to me having first wanting to become familiar with WEKA's source code; and I simply *forgot* to stop working on the project. By the time I finished (or was very close to finishing) I started reading other papers and realised some models were similar to my implemented model. I realise now that, that might have been a bad idea and it perhaps would have been better to first look over what others have done.

Acquiring datasets was relatively easy; however finding large datasets was not. It took me a while to find a dataset that was over 500MB in size, and I would have loved to have found one that was over 1GB; not sure if I could run it though, at least with my RAM specification.

Modifying J48 so heavily had its toll, the WEKA version I have used is essentially broken when you try to run a different decision tree algorithm. For my project I do not think this matters at all as my sole goal was modifying J48, however it might be somewhat problematic for applicability.

As some of the more astute readers might have realised, I have not used locks in my concurrent implementation of J48, this is indeed true. I have described atomic functions fairly highly back in Chapter 2 and have dismissed locks, and I believe this still. I do not think any performance increase would be gained with replacing locks with atomics, however perhaps in the future if I decided to re-implement some of the more complex parts of WEKA locks might be useful.

Amdahl's law[H] states that there is a limit to how much one can parallelise code; this limit is the necessary serialised portion of the code. I would have liked to make a detailed analysis on whether Amdahl's law is true or not, and especially whether it applies to my work, however I simply do not have enough time to look into it in greater detail. In order for me to be able to make a valid comparison, I would first need to identify the code that cannot be made parallel, obtain it's execution times, and then make an evaluation on whether that law is upheld.

There are indeed certain parts that cannot be made parallel in my implementation, such as a thread having to wait for the various different parts of a tree to finish constructing in order to move on to the finalization of the code; however, while threads wait for certain tasks to finish, they go and pick up other tasks in the meantime.

Chapter 8 – Future Work

Here I will just talk about any future plans I had or am thinking about for the project.

8.1 JCSP

There's not much too say about future work for JCSP, I personally would not think it would have yielded much of an increase in performance if my project was redone with the CSP model, however it would be interesting to try, and I could be wrong.

I'm not entirely sure, but I think parts of the original WEKA code might have to be redone via the CSP model in order to make it work cleanly, this would probably make it harder than my current project, but then again you wouldn't need to worry much about thread safety.

8.2 Pruning

As I've mentioned in chapter 6, pruning is computationally not important. However for extremely large datasets (over 1GB) it could prove beneficial. The parallel version 2 implementation is a good starting point; I think in the future I could add a level threshold feature. For example, a threshold of 2 would mean that tasks are created for the root node, and the nodes following the root node, however the nodes following the nodes after the root node will not have tasks created for them.

8.3 GPU Acceleration

As mentioned in the previous chapter, I have tried to attempt some GPU acceleration using the OpenCL library from the LWJGL package[G], however without much time left, and the complexity, I had to give it up. I would, however, be fairly interested in a parallel-GPU-accelerated version. This would also prove to be the hardest improvement I think, as this would require you to redesign some core parts of WEKA; it might be better to just start from scratch.

8.4 Other Implementations

I think my implementation model is the fastest for Java; I don't think a message passing model would be faster, though probably easier to understand/modify. I have not parallelised every single part of the J48 algorithm, and there are some parts that have to wait for other tasks to finish in order to continue; that part could be improved upon.

The papers I have analysed in the related works chapter have mentioned various kinds of implementations for how a tree could be constructed, one of them even mentioned constructing a tree sideways instead of how my one does it from top to bottom. This certainly sounds interesting, though whether it would yield a noticeable improvement is sceptical.

Bibliography:

[A] Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. and Lea, D. (2006). Java Concurrency In Practice. 1st ed. Addison-Wesley Professional, p.424.

[B] M. Mitchell, T. (1991). Machine Learning. McGraw-Hill Science/Engineering/Math, p.432.

[C] Amado, N., Gama, J. and Silva, F. (2018). Parallel Implementation of Decision Tree Learning Algorithms. Portuguese Conference on Artificial Intelligence, [online] p.8. Available at: https://www.researchgate.net/publication/220773622 [Accessed 19 Aug. 2018].

[D] Narlikar, G. (1998). A parallel, multithreaded decision tree builder. Pittsburgh, Pa.: School of Computer Science, Carnegie Mellon University.

[E] A., S., E., H., V., K. and V., S. (1998). Parallel Formulations of Decision-Tree Classification Algorithms. Proceedings. 1998 International Conference on Parallel Processing, [online] p.24. Available at: https://ieeexplore.ieee.org/document/708491/ [Accessed 24 Aug. 2018].

[F] Welch, P. and Brown, N. (2014). Communicating Sequential Processes for Java. [online] cs.kent.ac.uk. Available at: https://www.cs.kent.ac.uk/projects/ofa/jcsp/ [Accessed 7 Aug. 2018].

[G] lwjgl.org. (n.d.). LWJGL. [online] Available at: https://github.com/LWJGL/lwjgl3 [Accessed 7 Aug. 2018].

[H] Argentini, G. (2002). A generalization of Amdahl's law and relative conditions of parallelism. New Technologies and Models, Riello Group, Legnago (VR), Italy., [online] p.11. Available at: https://arxiv.org/ftp/cs/papers/0209/0209029.pdf [Accessed 7 Sep. 2018].

[I] Frank, E., Hall, M. and Witten, I. (2016). Weka 3: Data Mining Software in Java. [online] cs.waikato.ac.nz. Available at: https://www.cs.waikato.ac.nz/ml/weka/index.html [Accessed 7 Aug. 2018].

[J] Mehta, M., Agrawal, R. and Rissanen, J. (1996). SLIQ: A Fast Scalable Classifier for Data Mining. IBM Almaden Research Center, [online] p.15. Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=631E7260D2E9C36FADB4607135094643?doi=10.1.1.42.5335&rep=rep1&type=pdf [Accessed 10 Aug. 2018].

[K] Shafer, J., Agrawal, R. and Mehta, M. (1996). SPRINT: A Scalable Parallel Classifier for Data Mining. IBM Almaden Research Center, [online] p.12. Available at: http://www.vldb.org/conf/1996/P544.PDF [Accessed 10 Aug. 2018].

Appendix:

[AA] This is about half of the source code concerning the *selectModel* method. This method is located in the *C45ModelSelection.java* class.

```java
public final ClassifierSplitModel selectModel(Instances data)
{
  double minResult = 0;
  final AtomicReferenceArray<C45Split> currentModel;
  final AtomicReferenceArray<Enumeration> instancesArray;
  C45Split bestModel = null;
  NoSplit noSplitModel = null;
  DoubleAdder averageInfoGain = new DoubleAdder();
  AtomicInteger validModels = new AtomicInteger(0);
  boolean multiVal = true;
  Distribution checkDistribution;
  Attribute attribute;
  double sumOfWeights;
  int i;
  final ArrayDeque<ModelSelectionTask> taskQueue =
          new ArrayDeque<ModelSelectionTask>(POOLSIZE);
  final ConcurrentLinkedQueue<Integer> indexQueue =
          new ConcurrentLinkedQueue<Integer>();

  try
  {
      // Check if all Instances belong to one class or if not
      // enough Instances to split.
      checkDistribution = new Distribution(data);
      noSplitModel = new NoSplit(checkDistribution);
      if( Utils.sm(checkDistribution.total(), 2 * m_minNoObj) ||
          Utils.eq(checkDistribution.total(),
                  checkDistribution.perClass(checkDistribution.maxClass())) )
      {
          return noSplitModel;
      }

      // Check if all attributes are nominal and have a
      // lot of values.
      if( m_allData != null )
      {
          Enumeration enu = data.enumerateAttributes();
          while( enu.hasMoreElements() )
          {
              attribute = (Attribute) enu.nextElement();
              if ((attribute.isNumeric()) ||
                  (Utils.sm((double)attribute.numValues(),
                            (0.3*(double)m_allData.numInstances()))))
              {
                  multiVal = false;
                  break;
              }
          }
      }

      currentModel = new AtomicReferenceArray<C45Split>(data.numAttributes());
      sumOfWeights = data.sumOfWeights();


      for( i = 0; i < currentModel.length(); i++ )
      {
          indexQueue.add(i);
      }

      for( i = 0; i < POOLSIZE; i++ )
      {
          ModelSelectionTask task = new ModelSelectionTask(
                  indexQueue, currentModel, new Instances(data), m_minNoObj, multiVal, validModels,
                  m_allData, averageInfoGain, sumOfWeights
          );
          taskQueue.add(task);
          MAINTASKQUEUE.put(task);
      }

```

[AB] Source code for the main execution method for the ModelSelectionTask class.

```java
    @Override
    public void begin()
    {
        try
        {
            Integer index = indexQueue.poll();
            while( index != null )
            {
                // Apart from class attribute.
                if( index != (data).classIndex() )
                {
                    // Get models for current attribute.
                    currentModel.set(index, new C45Split( index, m_minNoObj, sumOfWeights ) );
                    currentModel.get(index).buildClassifier(data);


                    // Check if useful split for current attribute
                    // exists and check for enumerated attributes with
                    // a lot of values.
                    if( currentModel.get(index).checkModel() )
                    {
                        if( m_allData != null )
                        {
                            if( (data.attribute(index).isNumeric()) ||
                                (multiVal || Utils.sm((double)data.attribute(index).numValues(),
                                        (0.3*(double)m_allData.numInstances()))) )
                            {
                                averageInfoGain.add( currentModel.get(index).infoGain() );
                                validModels.incrementAndGet();
                            }
                        }
                        else
                        {
                            averageInfoGain.add( currentModel.get(index).infoGain() );
                            validModels.incrementAndGet();
                        }
                    }
                }
                else
                {
                    currentModel.set(index, null);
                }

                index = indexQueue.poll();
            }
        }
        catch (Exception ex){}

        finished = true;
    }
}
```

[AC] This table holds the results of the different pruning parallelisations. There are three different versions, the non-parallel one (which does not apply any parallel implementation of pruning), the parallel one (implements my first pruning version attempt), and the parallel version 2 (implements my 2nd version of pruning). This was taken from the *pruning cross-val comparisons* table.

To access the table, go to this link and pick the "*pruning cross-val comparisons"* tab: https://docs.google.com/spreadsheets/d/1jzmcN_xSdGnoUz2bkDQxRatBBUN5bEabRigjV_chelY/edit?usp=sharing

[AD] This table holds the entries for the concurrent execution times, non JCSP version, and non-parallel pruning version (original pruning). A poolsize of 1 means that the original WEKA version was executed, not the parallelised one.

To access the table, go to this link and pick the "*non jcsp, non pruning parallelised"* tab: https://docs.google.com/spreadsheets/d/1jzmcN_xSdGnoUz2bkDQxRatBBUN5bEabRigjV_chelY/edit?usp=sharing

[AE] This table holds the entries for the JCSP execution times, and non-parallel pruning version (original pruning). A poolsize of 1 means that the original WEKA version was executed, not the parallelised one.

To access the table, go to this link and pick the “*jcsp, non pruning parallelised”* tab: https://docs.google.com/spreadsheets/d/1jzmcN_xSdGnoUz2bkDQxRatBBUN5bEabRigjV_chelY/edit?usp=sharing