Investigating batch execution of random forests using TL2cgen

Alexander Fourie

University of Twente

Abstract—Random forests are one of the most widely used machine learning methods that allow for high interoperability and explainability. They make up for where the standard decision tree falls short, namely in the cases of over fitting, poor generalization, and better handling of outliers and noise. In addition, they are powerful tools that can be used to solve a variety of non-linear classification and regression problems. Their applications extend to the domains of healthcare, finance, marketing, and data mining. In an effort to enable C++ applications to store and exchange tree based models, researchers developed Treelite and its sub-module TL2cgen. These libraries allow for minimal code duplication and for models to be stored in a platform independent format. In this project the focus has been placed on investigating and analysing the runtime for generating predictions for random forests using TL2cgen and Treelite, with an emphasis on the former. The results show that the thread pool implementation of Treelite is not the governing factor in its superior runtime to TL2cgen.

1. INTRODUCTION

TL2cgen (Treelite 2 C GENerator) is a model compiler for decision tree models. Using this software, developers are equipped with the ability to convert any decision tree model such as random forests and gradient boosting models into an optimized and platform independent if-else tree in C which can be distributed as a native binary [2]. TL2cgen integrates a submodule called Treelite, which is a lightweight library that enables C++ applications to exchange and store decision tree forests either on the disk or on a network. It supports conversion from tree models built in the following libraries, XGBoost, Scikit-learn, and LightGBM to a common specification that can be exchanged and used by C++ applications [2]. It may be the case that these applications wish to exchange trees or share them with a central server that performs read and write operations on these trees. Such a server using Treelite or TL2cgen, can take the tree models built in applications using any one of the above tree model libraries into a common specification that it can use to perform operations such as making predictions and making modifications. In addition, since no external libraries are used in the compiled C code generated by TL2cgen, devices with memory constraints can execute raw C code thus not needing to install any of the tree model libraries. To make predictions for a given batch of inputs, Treelite uses custom thread pools where parallelism is utilized to feed these inputs into a tree ensemble [1]. TL2cgen however uses OpenMP instead of custom thread pools to parallelize the prediction process. The justification for using OpenMP comes from a claim from the original developer stating that thread pools are hard to maintain and might add unwanted complexity for future developers. This research provides an extensive analysis on the impact of the aforementioned parallelization techniques on Treelite and TL2cgen .

2. PROBLEM STATEMENT

A GitHub issue [1] shows that TL2cgen is slower than Treelite. It was found that Treelite spawns more threads and utilizes a greater percentage of the CPUs logical processors as opposed to TL2cgen which uses about a factor of two less. The conversation on the GitHub issue asserted that the runtime of TL2cgen might be due to its parallelization scheme. This paper investigates this issue with a more detailed analysis to determine whether TL2cgen is indeed slower than Treelite.

2.1. Research Questions

The problem defined above leads to the following research question: Given a pre-trained random forest, to what extent do custom thread pools influence batch execution of if-else trees generated by TL2cgen? Which can be answered by answering the following sub questions:

1) How does the execution efficiency of custom thread pools in Treelite compare to that of OpenMP in TL2cgen?

2) How does the implementation of custom thread pools affect the runtime and CPU usage of TL2cgen?

3. BACKGROUND

3.1. Decision Trees and Random Forests

It is important to recognize that machine learning extends beyond the space of deep learning. While deep learning models provide solutions to speech recognition, object detection, natural language processing, and fraud detection [4], their architectures tend to be large and complex, making them hard to explain and interpret [9]. Another type of machine learning model is the decision tree, which addresses some of the pit falls of deep learning. Specifically, decision trees can accept categorical and continuous data as an input, are less computationally intensive, require less data for training, and typically do not require data normalization such as one-hot encoding or feature scaling [13]. Despite these advantages, decision trees suffer from over-fitting causing the model to generalize poorly [13]. To counter over-fitting, random forests were developed to capture the underlying patterns in the data, rather than memorizing it [12]. In essence, a random forest is a collection of decision trees that were built from the original data set [12] .

3.2. Treelite and TL2cgen Implementations

The problem statement and following research questions ponder the influence of thread pools on the runtime of TL2cgen. This paper will present an algorithm to potentially improve the runtime of TL2cgen and possibly even Treelite. The nature of the algorithm relies on how the Treelite thread pool implementation works, therefore when presenting the algorithm, an understanding of Treelite is required. The OpenMP implementation for TL2cgen is a central part of this research and since it is compared to Treelite, its only logical to explain how it works, what it is and what it looks like in code. Understanding both these implementations will aid in explaining the resulting runtimes and CPU usages. TL2cgen uses OpenMP which is an application programming interface (API) that supports shared memory multiprocessing programming by specifying compiler directives that will spawn threads before the target code block executes. Treelite makes use of a custom thread pool implementation where a fixed number of threads are spawned and execute tasks whenever they become available.

3.2.1. *TL2cgen OpenMP Implementation*

Before doing any parallel inference, TL2cgen first determines how many threads need to be spawned. This number is the minimum between the number of rows in the input for which predictions need to be made and the number of logical processors that the host device has. So twenty logical processors and one-hundred and fifty rows will result in twenty threads being spawned, and any number of rows less than twenty will result in that many threads being spawned. Once this number is determined, TL2cgen splits the input data into batches of at most equal size and assigns each thread a start and end index for which it will make predictions. Listing 1 shows how TL2cgen calls the ParallelFor function which will parallelise the lambda function containing the prediction function.

Listing 1: Code snippet for ParallelFor function call

```
tl2cgen::detail::threading_utils::ParallelFor(std::
    size_t(0), nthread, thread_config_,
  tl2cgen::detail::threading_utils::ParallelSchedule
      ::Static(),
  [&](std::size_t thread_id, int) {
    std::size_t rbegin = row_ptr[thread_id];
    std::size_t rend = row_ptr[thread_id + 1];
    result_size[thread_id]
        = pred_func_->PredictBatch(dmat, rbegin,
            rend, pred_margin, out_result);
  });
```

Listing 2 shows a snippet from the ParallelFor function that parallelises the lambda function above. It supports static, dynamic, guided, and auto scheduling, but static is the default. If a chunk size is not specified then each thread will be assigned one iteration to execute of the for loop, otherwise each thread will execute several iterations equal to the specified chunk [10] .

Listing 2: OpenMP usage with static scheduling

```
  case ParallelSchedule::kStatic: {
    if (sched.chunk == 0) {
#pragma omp parallel for num_threads(thread_config.
    nthread) schedule(static)
      for (OmpInd i = begin; i < end; ++i) {
        exc.Run(func, static_cast<IndexType>(i),
            omp_get_thread_num());
      }
    } else {
#pragma omp parallel for num_threads(thread_config.
    nthread) schedule(static, sched.chunk)
      for (OmpInd i = begin; i < end; ++i) {
        exc.Run(func, static_cast<IndexType>(i),
            omp_get_thread_num());
      }
    }
    break;
  }
```

3.2.2. *Treelite thread pool implementation*

As mentioned before, Treelite implements a custom thread pool to make parallelised predictions. This requires an understanding of the nature of thread pools themselves. A thread pool is a group of pre-spawned threads that are continuously waiting for tasks to arrive through some data structure, typically a queue [15]. The main idea behind thread pools is that instead of incurring overhead by creating and destroying a thread every time a task comes in, a fixed number of threads are spawned and are only destroyed when all the tasks are completed [15]. Treelite defines a vector of threads each executing its own lambda function. Unlike TL2cgen who uses OpenMP to spawn threads whenever the prediction function is called (through the C++ API) in a Python program, Treelite creates the threads when the predictor object is instantiated. This means that whenever a Python program calls the predict function, it will always use the same threads created by the predictor object. According to the original developer of Treelite and TL2cgen, this reduces the potential overhead of creating and destroying threads [1].

The lambda function contains a loop that will continue trying to pop a task from a local single producer-single-consumer (SPSC) queue. Once the main thread submits a task to the queue, the waiting thread will pop the task, execute it, and push the result to another SPSC queue that stores the output of the task. Once the main thread has submitted all the tasks it will start trying to pop results off from the output queue that get used later in the program. The synchronisation between the pushing and popping from queues is not trivial, and for more efficiency, is made to be lock free and atomic. While locks are generally straightforward to use and understand, they introduce overhead due to the CPU needing to allocate memory for storing the lock, manage the creation and destruction of locks, and handle the acquisition and release processes [11]. What makes atomic operations complicated is the notion of program order. When executing an arbitrary program, it is expected that statements and operations happen in the order they are written. However, each line is susceptible to the compiler reordering it for improved program execution efficiency while still ensuring the program has the intended output [15]. Furthermore, there may be several orderings that still give the same output, for instance Listing 3 and Listing 4 both give the same output despite their program order being different.

Listing 3: Program order example 1

```
int x = 5;
int y = 3;
int z = x + y;
```

Listing 4: Program order example 2

```
int y = 3;
int x = 5;
int z = x + y;
```

When dealing with multiple threads who are each running their own process, the goal is to ensure that the orderings of a program executed by one thread does not invalidate the output of another process. For example “flag” and “message” being swapped in thread 1 wont effect the output, however it will influence the output of thread 2 who will either see “ ” (assumed initial value of “message”) or “Hello world” depending on where ”flag” is in the program execution.

Listing 5: Code fragment for thread 1

```
string message = "Hello world";
int flag = 1;
print(message);
```

Listing 6: Code fragment for thread 2

```
string text = "Hi";
while(flag!=1);
text = message;
print(text);
```

With regards to how this relates to the implementation of Treelite, there are two central functions that allow for synchronised popping and pushing, namely “enqueue” and “pop”. Before discussing these functions its important to note that pushing happens at the tail of the queue, and popping happens at the head. When an item is pushed, the tail is incremented and points to the next available slot, the same happens for the head. The queue implementation is not specific to Treelite and may be used in other applications. Therefore, for purpose of correct implementation, it may happen that a many pop or enqueue operations are executed, and since the size of the buffer is finite, the head may catch up to the tail (or vice versa). When the head and tail indices are equal, then the queue is said to be empty. If the next tail index (modulo the size) is equal to the head index, then the queue is said to be full.

Listing 7: SPSC Queue enqueue function

```
bool enqueue(T const& item) {
  size_t const current_tail = tail_.load(std::
      memory_order_relaxed);
  size_t const next_tail = incr(current_tail);

  if (next_tail != head_.load(std::
      memory_order_acquire)) {
    buffer_[current_tail] = item;
    tail_.store(next_tail, std::memory_order_release);
    return true;
  }
  return false;
}
```

Starting with the enqueue function, its clear that this function pushes an item to the queue. When an item is pushed, the function first needs to obtain the current tail. When dealing with shared atomic variables, the notion of memory order as mentioned before, becomes important. There are four main memory tags available, relaxed, acquire, release, and sequential consistency (default) [6]. The relaxed tag implies that the operation is simply atomic, there is no synchronisation between threads [6]. In the case where the main thread pushes and a worker thread pops, then the main thread uses the relaxed tag to get the tail since no other thread is modifying it, hence no synchronisation is needed. The same can be said for the case where a worker thread is pushing, and the main thread is popping.

The memory orders acquire, and release are used in pairs, where acquire is used on loads and release on stores. So, when one thread performs a store and another thread performs a load on the same variable, the compiler will ensure that the store happens before the load [6]. Hence using these together provides thread synchronisation. In addition, release ensures that all operations before are guaranteed to happen before, preventing the compiler from reordering the code. The same can be said for acquire except everything after is guaranteed to happen after [6]. It should thus be clear that the acquire when loading the head is done so that when another thread is done popping (modifying the head), then the load will receive the latest value. The same is true for the release when storing the tail. It ensures that a thread that wants to load the tail in the pop function will see the latest store performed in the enqueue function.

Listing 8: SPSC Queue pop function

```
bool pop(T* item) {

  for (uint32_t i = 0; i < 300000 && pending_.load()
       == 0; ++i) {
    std::this_thread::yield();
  }
  if (pending_.fetch_sub(1) == 0) {
    std::unique_lock<std::mutex> lock(mutex_);
    condition_.wait(lock, [this] { return pending_.
        load() >= 0 || stop_.load(); });
  }
  if (stop_.load(std::memory_order_relaxed)) {
    return false;
  }
  size_t const current_head = head_.load(std::
      memory_order_relaxed);
  if (current_head == tail_.load(std::
      memory_order_acquire)) {
    return false;
  }
  *item = buffer_[current_head];
  head_.store(incr(current_head), std::
      memory_order_release);
  return true;
}
```

When entering the pop function, the thread will enter a for loop, which simulates a spinlock, to prevent it from going to sleep, since waking up a thread may incur overhead. The idea behind is spinlock is busy-waiting, where a thread will check if the lock is available or not instead of going to sleep and waiting for a notify. In this case, the thread will repeatedly check if there are any pending tasks to complete. If there is a task, the thread will atomically decrement the task count and wait for a kill signal or continue with popping if the number of pending tasks is zero or has gone up due to a push operation. The thread will first check if it should terminate, if not then it will proceed with the popping operation where it increments the head only if the current head index is less than the current tail index.

4. METHOD

The approach to answering the presented research questions will involve several systematic steps. First the problem statement will be verified, followed by defining test cases for assessing performance and providing an overview of the tools used.

4.1. Replicating GitHub Issue

Before collecting any data, it is important to verify the problem and reaffirm that it indeed is a problem to be explored and not a result of the hardware the initial test case was run on.

Table 1: Replicated issue and original issue runtimes

Repl.TL2cgen	TL2cgen	Repl.Treelite	Treelite
0.152	0.021	0.024	0.012

While the initial GitHub issue shows that Treelite is about twice as fast as TL2cgen, the replicated issue shows that Treelite is about six times as fast as TL2cgen. Its evident that this is not enough to conclude that Treelite is faster than TL2cgen in all cases. From this small sample alone, it is impossible to know whether Treelite is superior to TL2cgen in all circumstances, and it will be shown that this is indeed not the case.

4.2. Data Collection

Assessing the influence of thread pools on the runtime of TL2cgen and evaluating its current performance using OpenMP requires several test cases to be defined. A test case is defined as a 3-tuple (C, D, T) where C is a configuration from either TL2cgen or Treelite, and D is a set of datasets for which the configuration must perform task T. Table 2 shows the breakdown of these test cases.

Table 2: Test case breakdown, C = Classification, R = Regression

C	D	T
TL2cgen OpenMP	{Iris, Wine, Digits, Cl.Custom1, Cl.Custom2, Cl.Custom3, Cl.Custom4}	C
	{Diabetes, California Housing, Re.Custom1, Re.Custom2, Re.Custom3, Re.Custom4}	R
Treelite thread pool	{Iris, Wine, Digits, Cl.Custom1, Cl.Custom2, Cl.Custom3, Cl.Custom4}	C
	{Diabetes, California Housing, Re.Custom1, Re.Custom2, Re.Custom3, Re.Custom4}	R
TL2cgen copied thread pool	{Iris, Wine, Digits, Cl.Custom1, Cl.Custom2, Cl.Custom3, Cl.Custom4}	C
	{Diabetes, California Housing, Re.Custom1, Re.Custom2, Re.Custom3, Re.Custom4}	R
Treelite copied OpenMP	{Iris, Wine, Digits, Cl.Custom1, Cl.Custom2, Cl.Custom3, Cl.Custom4}	C
	{Diabetes, California Housing, Re.Custom1, Re.Custom2, Re.Custom3, Re.Custom4}	R

For each data set D_i from D in (C, D, T), configuration C will be executed for each of the random forest sizes taken from the following set:

$$\{1, 20, 40, 60, 80, 100, 200, 500, 1000, 1200, 1500, 2000\}$$

Where the depth of each decision tree in the random forest is capped at five. For each tree size, the program will make 700 predictions instead of just one. The reason for this is because the time for a single prediction is shown as zero but is some value of the order 10e-5. Values this small would make comparison between Treelite and TL2cgen more difficult. Note that the GitHub issue did the same except with 1000 predictions. But since this research works with much larger test cases, a reduced number of iterations was chosen.

Table 3: Dataset properties, C = Classification, R = Regression

Dataset	Task	#Samples	#Classes	#Attributes
Iris	C	150	3	4
Digits	C	1797	10	64
Wine	C	178	3	13
Cl.Custom1	C	750	10	100
Cl.Custom2	C	1500	10	100
Cl.Custom3	C	3000	10	100
Cl.Custom4	C	6000	10	100
Diabetes	R	442	-	10
California Housing	R	20640	-	8
Re.Custom1	R	750	-	100
Re.Custom2	R	1500	-	100
Re.Custom3	R	3000	-	100
Re.Custom4	R	6000	-	100

4.3. Tools

This project will require several pieces of hardware and software. Below is a detailed deconstruction of the necessary components.

4.3.1. Versions

Given that the versions of TL2cgen and Treelite in the original GitHub issue were 0.3.1 and 3.9.0 respectively, the same will be used for this project. For loading data sets scikit-learn 1.4.2 will be used. The data sets to be loaded are Iris, Wine, Diabetes, Digits, California Housing, four custom classification tasks, and four custom regression tasks. The custom tasks will be made using sklearn.datasets for building the custom dataset and sklearn.ensemble to build a RandomForestClassifier and a RandomForestRegressor.

4.3.2. Hardware

All the testing and programming will be done on an 13th Gen Intel i7 with 16GB of RAM and with 14 cores and 20 logical processors. This means that there can be 20 threads running concurrently.

4.3.3. Software

On top of using Treelite and TL2cgen, the Microsoft Visual Studio Integrated Development Environment (IDE) version 17.9.6 will be used for all the programming related tasks. For collecting CPU usage data, the built in Microsoft Visual Studio Diagnostic Tool will be used. This tool can monitor the CPU usage, memory usage and provide a detailed function call stack overview.

5. RESULTS

5.1. Runtime Data

Table 4: Runtime table for Iris dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.069	0.016	0.134	0.091
20	0.07	0.023	0.07	0.086
40	0.072	0.023	0.141	0.106
60	0.061	0.025	0.137	0.119
80	0.058	0.022	0.151	0.096
100	0.067	0.029	0.151	0.093
200	0.068	0.035	0.148	0.108
500	0.083	0.061	0.178	0.111
1000	0.161	0.123	0.238	0.171
1200	0.197	0.168	0.265	0.237
1500	0.287	0.212	0.311	0.284
2000	0.385	0.330	0.419	0.394

Table 5: Runtime table for Wine dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.058	0.016	0.143	0.096
20	0.066	0.021	0.152	0.098
40	0.07	0.02	0.145	0.100
60	0.065	0.022	0.146	0.107
80	0.070	0.026	0.145	0.103
100	0.072	0.033	0.151	0.103
200	0.077	0.045	0.179	0.112
500	0.133	0.118	0.226	0.213
1000	0.355	0.284	0.496	0.409
1200	0.418	0.370	0.498	0.472
1500	0.558	0.493	0.562	0.566
2000	0.677	0.790	0.733	0.823

Table 6: Runtime table for Diabetes dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.068	0.016	0.140	0.099
20	0.073	0.022	0.147	0.108
40	0.072	0.031	0.151	0.110
60	0.094	0.053	0.172	0.121
80	0.109	0.076	0.202	0.145
100	0.116	0.087	0.216	0.166
200	0.285	0.229	0.345	0.319
500	0.715	0.733	0.770	0.768
1000	1.670	1.927	1.441	2.004
1200	2.139	2.529	1.967	2.593
1500	2.884	3.473	2.509	3.489
2000	4.043	4.736	3.318	4.745

Table 7: Runtime table for Digits dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.193	0.050	0.254	0.192
20	0.225	0.098	0.29	0.251
40	0.273	0.182	0.455	0.355
60	0.448	0.335	0.548	0.459
80	0.511	0.431	0.659	0.579
100	0.572	0.520	0.629	0.695
200	1.132	1.128	1.270	1.219
500	3.799	3.836	3.015	4.07
1000	8.273	11.367	7.84	11.117
1200	11.088	13.924	10.052	13.689
1500	12.358	15.189	12.091	12.542
2000	16.413	16.914	15.533	17.224

Table 8: Runtime table for Housing dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.232	0.081	0.356	0.288
20	0.414	0.277	0.559	0.485
40	0.625	0.513	0.779	0.706
60	0.835	0.818	1.154	1.014
80	1.155	1.111	1.408	1.35
100	1.423	1.497	1.675	1.751
200	3.764	3.697	3.796	4.012
500	12.486	12.316	12.089	12.254
1000	33.282	31.293	29.668	30.704
1200	46.338	41.868	39.969	40.405
1500	63.663	61.475	59.969	58.911
2000	98.518	92.948	89.769	90.154

Table 9: Runtime table for Re.Custom3 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.315	0.089	0.324	0.343
20	0.395	0.134	0.397	0.412
40	0.443	0.226	0.455	0.518
60	0.650	0.354	0.629	0.673
80	0.754	0.484	0.663	0.804
100	0.774	0.664	0.870	0.965
200	1.615	1.504	1.619	1.846
500	4.064	4.714	4.182	4.448
1000	8.866	9.580	8.510	8.766
1200	13.257	12.782	11.504	11.839
1500	15.911	16.565	15.021	15.294
2000	21.13	22.998	20.39	20.483

Table 10: Runtime table for Re.Custom4 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.456	0.156	0.485	0.567
20	0.612	0.248	0.611	0.697
40	0.740	0.445	0.768	0.875
60	0.987	0.780	0.913	1.189
80	1.250	1.026	1.194	1.485
100	1.546	1.367	1.469	1.793
200	2.989	2.953	2.998	3.605
500	7.755	8.430	7.624	8.061
1000	19.720	19.466	20.617	19.243
1200	24.690	25.406	24.411	25.466
1500	31.439	32.259	32.261	30.856
2000	42.024	42.925	42.470	42.996

Table 11: Runtime table for Re.Custom2 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.259	0.061	0.280	0.222
20	0.308	0.092	0.296	0.265
40	0.282	0.154	0.328	0.325
60	0.436	0.226	0.408	0.403
80	0.438	0.321	0.608	0.504
100	0.572	0.386	0.584	0.571
200	1.038	0.821	1.200	1.034
500	2.268	2.448	3.081	2.509
1000	5.703	6.846	6.307	8.160
1200	7.190	8.860	8.202	9.444
1500	12.309	12.370	10.068	13.602
2000	11.731	13.397	13.509	18.841

Table 12: Runtime table for Re.Custom1 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.175	0.041	0.265	0.386
20	0.200	0.063	0.264	0.168
40	0.250	0.086	0.297	0.212
60	0.285	0.128	0.352	0.242
80	0.320	0.160	0.343	0.273
100	0.396	0.208	0.408	0.332
200	0.567	0.426	0.639	0.561
500	1.320	1.278	1.496	1.414
1000	2.789	2.942	3.097	3.178
1200	3.381	3.531	4.050	4.079
1500	4.305	4.677	5.002	5.324
2000	7.044	6.542	7.163	8.535

Table 13: Runtime table for Cl.Custom1 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.187	0.041	0.224	0.164
20	0.206	0.063	0.232	0.195
40	0.279	0.086	0.280	0.269
60	0.324	0.128	0.408	0.347
80	0.410	0.160	0.384	0.411
100	0.344	0.208	0.464	0.416
200	0.623	0.426	0.744	0.676
500	1.391	1.278	1.616	1.716
1000	3.636	2.942	4.698	4.992
1200	4.665	3.531	5.858	6.515
1500	5.622	4.677	7.402	8.456
2000	7.813	6.542	10.140	11.423

Table 14: Runtime table for Cl.Custom2 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.266	0.071	0.304	0.240
20	0.338	0.120	0.400	0.296
40	0.450	0.209	0.520	0.420
60	0.397	0.334	0.616	0.512
80	0.498	0.396	0.616	0.622
100	0.595	0.501	0.840	0.651
200	1.053	1.049	1.248	1.212
500	2.820	3.362	3.553	3.670
1000	7.313	8.838	9.836	10.252
1200	8.815	11.453	11.685	13.050
1500	11.325	14.161	14.861	16.031
2000	15.787	17.583	20.129	22.632

Table 15: Runtime table for Cl.Custom3 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.329	0.107	0.432	0.393
20	0.431	0.263	0.632	0.506
40	0.606	0.436	0.744	0.798
60	1.017	0.636	1.024	0.953
80	0.958	0.839	1.176	1.178
100	1.100	1.045	1.296	1.380
200	2.236	1.994	2.352	2.470
500	5.593	7.151	6.970	8.082
1000	15.202	19.050	20.662	21.790
1200	18.629	23.644	24.332	27.084
1500	23.509	30.063	30.802	31.308
2000	33.063	41.571	43.488	45.564

Table 16: Runtime table for Cl.Custom4 dataset

#Trees	TL2cgen OpenMP	Treelite thread pool	TL2cgen copied thread pool	Treelite copied OpenMP
1	0.521	0.169	0.656	0.659
20	0.717	0.368	0.944	0.946
40	1.148	0.752	1.488	1.307
60	1.563	1.104	2.072	1.795
80	1.706	1.472	2.160	2.170
100	2.004	1.701	2.456	2.510
200	3.546	3.778	4.233	4.579
500	10.976	11.984	11.819	14.357
1000	30.589	36.659	31.367	32.658
1200	36.011	42.658	37.786	37.784
1500	46.250	51.924	51.314	48.086
2000	64.884	71.037	72.738	68.085

5.2. CPU Usage Data

Table 17: CPU usage table for Iris dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	9.62	10.75	36.28	7.54
20	10.39	14.24	40.11	8.70
40	9.44	23.81	40.97	9.36
60	9.79	22.47	34.00	7.91
80	9.93	21.56	36.36	8.16
100	10.08	19.09	41.04	7.82
200	10.72	20.21	35.38	7.26
500	13.06	19.80	36.52	8.57
1000	17.43	24.76	37.81	10.73
1200	17.28	24.73	44.04	13.79
1500	18.04	28.04	41.01	14.62
2000	20.80	29.95	44.67	17.28

Table 18: CPU usage table for Wine dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	9.19	24.43	41.61	9.59
20	10.76	21.33	37.29	7.60
40	10.28	24.40	37.55	7.22
60	9.74	27.72	43.23	10.62
80	9.73	23.45	40.11	8.05
100	11.00	24.38	40.09	7.82
200	12.76	18.16	51.54	5.90
500	13.47	31.75	37.22	13.14
1000	25.21	36.28	49.18	15.98
1200	26.31	36.95	44.49	21.24
1500	26.90	35.17	46.16	23.39
2000	28.48	39.51	46.80	22.11

Table 19: CPU usage table for Diabetes dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	9.11	22.75	42.43	9.37
20	10.82	29.29	41.00	7.50
40	8.41	37.38	41.68	10.51
60	12.07	28.08	49.35	10.94
80	15.33	30.05	53.84	7.72
100	12.38	31.45	43.23	10.34
200	22.12	34.54	45.49	20.61
500	31.14	39.36	44.25	26.36
1000	36.96	38.78	41.08	30.13
1200	37.63	41.55	44.95	29.60
1500	33.24	49.67	48.47	29.13
2000	43.28	52.29	62.13	19.73

Table 20: CPU usage table for Digits dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	20.50	17.19	41.98	15.84
20	21.15	25.76	50.62	19.96
40	28.02	31.43	45.24	22.23
60	27.57	35.06	51.85	23.60
80	29.38	34.10	51.05	24.53
100	31.93	34.86	48.20	26.55
200	36.11	39.40	47.95	29.69
500	36.83	46.88	46.51	34.12
1000	49.00	33.19	48.51	38.39
1200	40.48	43.32	46.39	34.34
1500	43.78	47.38	47.50	39.74
2000	55.06	33.14	53.65	33.68

Table 21: CPU usage table for California Housing dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	27.91	21.12	54.36	19.65
20	29.85	29.28	46.09	26.26
40	31.00	34.79	45.58	24.94
60	30.45	36.59	42.94	27.56
80	31.43	36.72	41.58	27.15
100	31.18	38.49	44.34	29.13
200	33.67	42.34	50.20	28.04
500	34.31	45.82	49.33	31.78
1000	41.03	42.56	49.42	34.61
1200	41.51	45.14	56.26	31.34
1500	57.88	24.88	66.52	25.27
2000	54.96	29.75	60.86	29.75

Table 22: CPU usage table for Re.Cusomt4 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	41.17	20.05	43.70	28.13
20	30.28	18.05	38.79	25.32
40	25.84	21.34	36.97	20.56
60	28.64	25.93	37.34	21.26
80	24.09	23.37	34.40	22.01
100	24.32	24.75	32.50	20.63
200	24.28	26.03	33.95	22.55
500				
1000				
1200				
1500				
2000				

Table 23: CPU usage table for Re.Cusomt3 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	34.91	10.70	46.07	23.66
20	27.03	14.56	39.65	20.26
40	22.48	16.70	38.83	19.03
60	25.27	21.96	37.29	19.29
80	25.51	26.40	34.83	20.86
100	24.71	27.06	33.04	19.81
200	25.45	30.07	34.68	22.10
500	27.14	35.34	35.04	22.97
1000				
1200				
1500				
2000				

Table 24: CPU usage table for Re.Cusomt2 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	28.80	21.61	48.00	19.31
20	21.25	22.28	38.19	20.06
40	19.67	23.21	36.13	19.36
60	21.65	24.53	35.19	16.69
80	22.78	24.59	37.37	20.29
100	22.46	27.24	35.97	19.44
200	24.56	30.92	37.31	20.87
500	26.63	35.28	35.00	24.59
1000	30.46	38.49	43.57	22.81
1200	31.15	38.87	39.46	28.61
1500	29.86	42.51	40.89	25.74
2000				

Table 25: CPU usage table for Re.Cusomt1 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	17.11	22.78	40.33	13.61
20	18.30	18.74	38.37	15.39
40	15.06	24.63	38.81	14.40
60	17.40	13.19	43.25	11.97
80	15.70	24.37	36.82	16.32
100	16.74	25.28	40.27	19.39
200	21.58	27.97	37.20	21.18
500	24.68	33.26	36.27	24.18
1000	27.82	37.92	40.02	25.07
1200	28.43	39.15	39.07	27.18
1500	29.38	39.37	35.67	27.42
2000	29.02	39.92	40.61	27.56

Table 26: CPU usage table for Cl.Cusomt1 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	44.46	22.10	49.97	25.44
20	38.62	28.77	46.23	29.41
40	35.76	37.83	46.72	29.53
60	36.64	35.91	45.29	29.86
80	37.25	38.72	44.29	31.44
100	36.64	40.08	47.61	30.28
200	37.11	42.25	47.58	32.45
500	37.54	47.50	55.93	29.70
1000	37.70	50.92	58.62	28.80
1200	36.80	53.34	54.80	33.54
1500	42.79	47.35	60.92	29.19
2000	40.68	49.50	55.01	34.98

Table 27: CPU usage table for Cl.Cusomt2 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	30.32	16.85	53.91	19.57
20	32.90	20.35	47.52	24.67
40	34.36	28.58	52.47	25.78
60	33.71	32.56	52.78	26.78
80	34.85	38.93	47.66	28.95
100	34.51	38.90	47.96	29.02
200	34.62	44.20	46.13	31.47
500	38.16	46.52	48.23	33.10
1000	37.49	51.31	48.37	35.86
1200	37.69	51.66	49.62	37.14
1500	38.35	50.27	60.68	27.38
2000	39.90	49.72		

Table 28: CPU usage table for Cl.Cusomt3 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	19.64	13.43	36.82	13.01
20	17.63	25.35	36.23	13.91
40	22.25	23.49	37.24	16.49
60	24.06	31.91	42.75	16.69
80	27.51	32.37	42.61	21.33
100	25.15	31.13	42.88	23.05
200	29.73	35.30	44.81	24.81
500	33.42	43.69	45.10	28.51
1000	35.35	49.06	47.64	31.21
1200	36.16	47.83	47.65	31.71
1500	38.48	47.49	42.70	35.17
2000	38.42	48.42	44.02	34.22

Table 29: CPU usage table for Cl.Cusom4 dataset

#Trees	TL2cgen OpenMP	Treelite Thread Pool	TL2cgen Copied Thread Pool	Treelite Copied OpenMP
1	24.57	13.08	43.63	16.43
20	23.51	25.49	40.52	19.95
40	23.97	33.29	45.76	20.65
60	26.27	37.13	44.35	25.72
80	30.45	31.98	39.54	25.71
100	29.17	37.20	48.53	23.62
200	32.61	39.35	47.19	27.44
500	35.06	45.91	47.11	30.83
1000	36.37	48.21	48.57	31.86
1200	38.65	48.69	48.75	30.75
1500	37.59	49.85	50.72	31.06
2000	35.36	53.40	55.06	29.58

6. DISCUSSION

6.1. Runtimes

Considering the performance of the test cases with regards to the Iris dataset shown in Table 4, the thread pool implementation of Treelite had the best runtime for all entries. While the initial GitHub issue showed that Treelite was twice as fast as TL2cgen, it is evident that this relationship is inconsistent with the data. With a random forest with only one decision tree, Treelite showed to be about 4.31 times faster than TL2cgen. However, as the number of decision trees starts to increase, the performance difference begins to blur, showing that Treelite and TL2cgen do not vary much. When looking at whether the thread pool from Treelite improves the performance of TL2cgen, it appears that it performs marginally worse. It is expected that more decision trees will once again cause this performance difference to become negligible. On the flip side, Treelite with the OpenMP implementation also performs slightly worse but still better than TL2cgen with a thread pool. Table 5 shows that all configurations are once again similar in runtime for the largest number of decision trees, and that Treelite is a few times faster than the rest for a small number of decision trees (< 200).

Looking at the slightly larger datasets, Tables 6 and 7 show that TL2cgen with a thread pool slightly outperformed the other cases for the larger random forests, even though both Treelite and the original TL2cgen implementation were many times faster for a single decision tree. The pattern where Treelite starts off being noticeably faster than all the other cases is a recurring one (and aligns with the GitHub issue), but as the size of the data sets grow, as shown in Table 8 (largest dataset), all the cases show negligible differences in performance. Another good example of this is shown by the custom datasets whose runtimes are shown in Tables 9 – 16, where the number of samples are 750, 1500, 3000, and 6000 where the number of attributes remained at 100. Already at a random forest size of 100 do incrementing sample sizes show a linear relationship, suggesting that by scaling the size of the dataset, the runtime will scale by a similar factor independent of how the parallelization scheme of TL2cgen and Treelite are configured. These observations remain consistent for Tables.

6.2. CPU Usage

When the program is executed with the Microsoft Visual Studio Diagnostic Tool (DT) enabled, it will take snapshots of a functions execution to provide an estimate of how much time the CPU spent executing it. The number of snapshots per unit time is called the sample rate and was set to a value of 4000 samples per second. For each dataset and for each tree count N the program was executed for a pair of configurations (C_i, C_{i+1}):

1) $C_1 = (TL2Cgen\ OpenMP, D_i, N)$
2) $C_2 = (Treelite\ Thread\ Pool, D_i, N)$
3) $C_3 = (TL2Cgen\ Copied\ Thread\ Pool, D_i, N)$
4) $C_4 = (Treelite\ Copied\ OpenMP, D_i, N)$

Its important to note that configuration pair (C_1, C_2) were executed in sequence and in a separate execution as the other pair, the same was done for (C_3, C_4). The reason for this is so

then the DT could provide the percentage of time the CPU spent executing the Treelite and TL2cgen configurations and not mixed configurations.

A logical expectation for the CPU usage results would be that Treelite spends a smaller percentage of CPU time and has a faster runtime compared to TL2cgen, however the data indicates that his is not always the case. At face value it appears that generally, as the size of the data set increases, then the CPU spends a larger percentage of time on execution for both TL2cgen and Treelite. Additionally, for a given data set, as the number of trees in a random forest increase, then once again, the CPU spends a larger percentage of time on execution. A deeper inspection of the data will show that the percentage values do not converge and stabilize as consistently as the runtimes do. For example, Table 19 in the final row, the Treelite Copied OpenMP implementation used between 2 and 3 times less CPU time compared to the others. A similar case is true for Tables 17,18, and 21. While its clear that the runtimes get larger for larger random forests, the same cannot be said for the CPU usages.

In Tables 17 – 29 there is at least one case where the next CPU usage percentage is smaller than the previous one. While it is not of much interest if they differ by a few percentage points, but decreasing by almost fifty percent is cause for investigation. Table 21 illustrates this in the Treelite Thread Pool column and at tree count 500 and 1500. On top of this, the runtimes that map to Table 21 are Table 8, showing that the CPU spent a smaller percentage of time on the case for tree count 1500 as opposed to tree count 500, despite the former having a much slower run time. Additionally for tree count 2000 in Table 21, the TL2cgen Copied Thread Pool was marginally faster but had a significantly higher CPU usage percentage than the Treelite Thread Pool and the Treelite Copied OpenMP implementations.

In Tables 17 – 29 it appears that when TL2cgen adopted the custom thread pool from Treelite, the CPU usage percentage in most cells tends to be significantly larger than the OpenMP implementation. Why this is the case might have to do with manually creating threads instead of having OpenMP manage them as it is designed for parallel process management and creation. The standard implementation for Treelite also tends to have larger CPU percentages than the OpenMP implementation from TL2cgen, however the significance is not as large as when TL2cgen uses the thread pool. Tables 22, 23, 24, and 27 all have missing data. The reason for this is because the DT adds overhead by taking samples of function execution and thus increases the total execution time. The overhead got so large that the program began to take too long to complete.

The reason why these results are interesting is because the test cases assume all else is constant, the only changes for a given configuration are the number of trees, so at face value, the CPU should dedicate a similar amount of time to all components of the program. However, it was observed during the data collection phase that an increasing amount of time was spent compiling the model into a .dll file (larger random forest). So it may be logical to assume that for a given configuration pair (C_i, C_{i+1}), that if the $\mathrm{T}(C_i) > T(C_{i+1})$ where T is some CPU usage function, and the actual prediction runtimes are approximately the same, then the CPU must have spent more time on other processes related to the operation of C_i.

6.3. Custom Thread Pool Approach

Despite the data showing that the thread pool from Treelite does not significantly influence the runtime of TL2cgen, it is still interesting to explore other possible parallelization schemes. The proposed scheme has not been thoroughly tested and will require further investigation and development, however if correctly implemented, it may improve the runtimes of the current implementations in both TL2cgen and Treelite. The algorithm does not deviate much from what Treelite already does, it uses the same SPSC queue and still has an input and output channel per thread for popping and pushing tasks and their prediction results. Where the proposed algorithm differs is in the structure of the tasks. Specifically, Treelite currently pushes for a single thread a single task containing a batch of inputs for which corresponding predictions are to be made. Instead, the algorithm will separate this batch into individual prediction tasks, so a single task consisting of a batch of five inputs, will now be stored as five individual tasks.

The motivation for this algorithm stems from the fact that if multiple threads are spawned, its expected that these threads will complete their tasks at different points in time. Since initially, each thread only has a single task, a thread who finishes its task quicker than the others will wait idly until it is destroyed. The new algorithm suggests that a thread who is finished with its tasks, may steal tasks from another thread to make productive use of its time alive. The first action a thread will take is to try and pop from the input queue. After a successful pop, the thread will make a prediction for the given input and store the result. The thread will keep track of the tasks that it has completed and a variable to store the accumulated prediction result.

Once a thread has completed all its tasks it then can continuously steal and execute the tasks of the next thread (thread id + 1). The steal function is the same as the pop function except with the condition that the steal will fail if the number of tasks in the victim thread queue is one. If the victim thread only has one task left, then the stealing thread will leave it. If there is more than one task, then the stealing thread will atomically pop it from the queue of the victim. The stealing thread will store the results from each prediction to a variable and keep track of the number of tasks it has stolen. Once the stealing thread can no longer steal, it will atomically share the number of stolen tasks and result value with the victim thread using an atomic store operation. Since the victim thread will have at most one task remaining, it will be able to pop this task and as usual increment its completed count and current result value. The difference now is that the number of stolen tasks may be larger than zero, in which case it can load the stolen result and task count and update its own values as if it were the one to have executed those tasks. At this point the victim thread is finished with its tasks and may begin stealing tasks.

Listing 9: Custom thread pool worker function

```
void do_work(size_t thread_id) {
  InputToken input;
  int num_complete = 0;
  size_t sum = 0;
  while (in_queue_[thread_id]->pop(&input, thread_id))
       {
    size_t result = input.pred_func->PredictBatch(
        input.dmat, input.rbegin, input.rend, input.
            pred_margin, input.out_result);
    if (num_complete < input.num_tasks) {
      sum += result;
      num_complete++;
      if (in_queue_[thread_id]->num_stolen.load() > 0)
           {
        while (in_queue_[thread_id]->stolen_result.
            load() == 0) {
        sum += in_queue_[thread_id]->stolen_result.
            load();
        num_complete += in_queue_[thread_id]->
            num_stolen.load();
        in_queue_[thread_id]->num_stolen.store(0);
        in_queue_[thread_id]->stolen_result.store(0);
      }
      if (num_complete >= input.num_tasks) {
        size_t steal_frm_idx = (thread_id + 1) %
            num_worker_;
        InputToken token;
        size_t num_stolen = 0;
        size_t stolen_result = 0;
        while (in_queue_[steal_frm_idx]->steal(&token)
            ) {
          stolen_result += token.pred_func->
              PredictBatch(
              token.dmat, token.rbegin, token.rend,
                  token.pred_margin, token.out_result
                  );
          num_stolen++;
        }
        in_queue_[steal_frm_idx]->num_stolen.store(
            num_stolen);
        in_queue_[steal_frm_idx]->stolen_result.store(
            stolen_result);
        out_queue_[thread_id]->push(OutputToken{sum});
        num_complete = 0;
        sum = 0;
      }
    }
  }
}
```

7. RELATED WORK

Given a random forest and an input vector for which a prediction must be made, it might be that there are hundreds or even thousands of decision trees in the random forest. It is therefore reasonable to assume that generating predictions for each tree sequentially is not as computationally and memory efficient as opposed to using some form of parallel computing. There has been ample research done in the domain of improving the inference times of tree ensembles. One paper investigated fast inference of tree ensembles on ARM devices where they investigate the effects of using fixed point quantization in random forests along with the architectural differences between ARM and Intel CPUs [7]. Another paper proposed solutions for making training of random forests faster by efficiently finding split points [14]. Specifically, they investigated accelerating the training process of Gradient Boosted Decision Trees in the case where the outputs are multidimensional. Here multidimensional output refers to multiclass classification, multilabel classification, or multioutput regression. A scoring function was developed to find the best split of a decision tree. A similar paper was written that implemented faster training using MABSplit, which is a subroutine used to efficiently find split points [8].

A more related paper did research into a CUDA based implementation of random forests. CUDA stands for Compute Unified Device Architecture which is a C API that gives direct access to the instruction set of the GPU and allows developers to utilize the parallel computing capabilities of said GPU [5]. What is interesting about this paper is that their implementation parallelized both training and classification, except it was directed towards GPUs. While the use of GPUs is essential in applications that make use of machine learning, not all such applications can make use of one. Specifically, GPUs are expensive, resource intensive, require a lot of computational power and memory as opposed to embedded devices that only have a CPU. In addition, TL2cgen compiles a model to an easy to work with format that can be distributed amongst C++ applications. This paper did not specify the model format or how it gets handled in their program. Furthermore, they assigned each thread to a tree for prediction, so if a random forest contains hundreds or even thousands of trees, then the GPU spawns an equivalent number of threads.

8. CONCLUSION

The problem addressed in this research was to find out if TL2cgen was indeed slower than Treelite, and if it was, how the influence of thread pools used in TL2cgen might affect the runtime. The reason why thread pools were of interest was because it was asserted in a conversation on a GitHub issue that Treelite using custom thread pools might be the reason for its superior performance. However, it was found that TL2cgen is a few times worse than Treelite for a small number of decision trees in a random forest and performs slightly better or approximately the same for anything larger than 1000 trees. In addition, it was also uncovered that the initial GitHub issue was misleading as it presented a single case for which TL2cgen was slower, and that the reason for this was due to not using thread pools, but turned out that the thread pool from Treelite has very little influence on performance.

9. FUTURE WORK

There is still room for investigation, as the reason why Treelite is superior to TL2cgen for small instances is still unknown. Furthermore, while the CPU data presented interesting results, a more detailed breakdown of the respective libraries per function would make a good exploration. Finally, reproducing the same or similar experiments as the ones conducted in this research on different architectures and operating systems would show if Treelite or TL2cgen are optimised for different pieces of hardware.

10. ACKNOWLEDGEMENTS

I would like to personally thank my supervisors, Dr.Kuan-Hsun Chen and Duncan Bart for guiding me throughout this research project and always being there if I needed any assistance. I would also like to thank Philip.H.Cho for taking the time to help me understand his work, answer my questions and giving me pointers.

REFERENCES

[1] Cho, P.H. (2023) *Why does it seem that TL2cgen is slower than treelite_runtime? · issue #18 · DMLC/TL2cgen*, *GitHub*. Available at: https://github.com/dmlc/TL2cgen/issues/18 (Accessed: 30 June 2024).

[2] Cho, P.H.. 2023. “Treelite”. https://github.com/dmlc/treelite/tree/release_3.9. (2024)

[3] Cho, P.H..2023. “TL2cgen ”. https://github.com/dmlc/TL2cgen /tree/release_0.3. (2024)

[4] Coursera Staff (2024) *10 examples of Deep Learning Applications*, *Coursera*. Available at: https://www.coursera.org/articles/deep-learning-applications (Accessed: 30 June 2024).

[5] H. Grahn, N. Lavesson, M. H. Lapajne, and D. Slat, “Cudarf: A cudabased implementation of random forests,” in 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2011, pp. 95–101.

[6] Humayun, Z. (2024) *Atomics and concurrency in C++*, *freeCodeCamp.org*. Available at: https://www.freecodecamp.org/news/atomics-and-concurrency-in-cpp. (Accessed: 30 June 2024).

[7] L. M. Koschel, Buschjager, “Fast inference of tree ensembles on arm devices,” arXiv:2305.08579, 2023.

[8] L. T. P. S. Z. Tiwari, Kang, “Mabsplit: Faster forest training using multiarmed bandits,” arXiv:2212.07473, 2022.

[9] Molnar, C. (2020) *Interpretable machine learning ; a guide for making Black Box models explainable*. Leanpub.

[10] OpenMP - scheduling(static, dynamic, guided, runtime, auto) (no date) OpenMP - Scheduling(static, dynamic, guided, runtime, auto) - Yiling’s Tech Zone | 风逝无殇的瞎逼逼基地. Available at: https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/ (Accessed: 30 June 2024).

[11] Ousterhout, J. and Mazières, D. (no date) *Locks and condition variables*, *Locks & Cond. Vars.* Available at: https://web.stanford.edu/~ouster/cs111-spring21/lectures/locks/ (Accessed: 30 June 2024).

[12] scikit-learn developers (no date b) *1.11. ensembles: Gradient boosting, random forests, bagging, voting, stacking*, *scikit*. Available at: https://scikit-learn.org/stable/modules/ensemble.html#forest (Accessed: 30 June 2024).

[13] scikit-learn developers (no date) *1.10. decision trees*, *scikit*. Available at: https://scikit-learn.org/stable/modules/tree.html (Accessed: 30 June 2024).

[14] V. Iosipoi, “Sketchboost: Fast gradient boosted decision tree for multioutput problems,” arXiv:2211.12858, 2022.

[15] Williams, A. (2019) *C++ concurrency in action, second edition by Anthony Williams*. S.l., Norwood, Mass.: Manning Publications : distributed by Skillsoft Books.