Improving parallel performance of ensemble learners for streaming data through data locality with mini-batching

Guilherme Cassales
gwcassales@estudante.ufscar.br
Federal University of São Carlos
Brazil

Heitor Gomes
Albert Bifet
Bernhard Pfahringer
{*heitor.gomes, abifet, bernhard*}*@waikato.ac.nz*
University of Waikato
New Zealand

Hermes Senger
hermes@ufscar.br
Federal University of São Carlos
Brazil

***Abstract*—Machine Learning techniques have been employed in virtually all domains in the past few years. New applications demand the ability to cope with dynamic environments like data streams with transient behavior. Such environments present new requirements like incrementally process incoming data instances in a single pass, under both memory and time constraints. Furthermore, prediction models often need to adapt to concept drifts observed in non-stationary data streams. Ensemble learning comprises a class of stream mining algorithms that achieved remarkable prediction performance in this scenario. Implemented as a set of (several) individual component classifiers whose predictions are combined to predict new incoming instances, ensembles are naturally amendable for task parallelism. Despite its relevance, an efficient implementation of ensemble algorithms is still challenging. For example, dynamic data structures used to model non-stationary data behavior and detect concept drifts cause inefficient memory usage patterns and poor cache memory performance in multi-core environments. In this paper, we propose a mini-batching strategy which can significantly reduce cache misses and improve the performance of several ensemble algorithms for stream mining in multi-core environments. We assess our strategy on four different state-of-art ensemble algorithms applying four widely used machine learning benchmark datasets with varied characteristics. Results from two different hardware show speedups of up to 5X on 8-core processors with ensembles of 100 and 150 learners. The benefits come at the cost of changes in predictive performances.**

***Index Terms*—Multicore task-parallelism, data-stream learning, ensemble learners, bagging algorithms.**

1. Introduction

Machine Learning (ML) models became fundamental for many applications in different domains. This omnipresence of ML models accompanied by domain requirements motivated the development of novel algorithms adhering to specific problem constraints. In many applications, learning algorithms have to cope with dynamic environments that collect potentially unlimited streams of data. Compared to traditional (static) data mining, stream processing algorithms have additional requirements. For instance, with a potentially infinite data stream, storing data for late processing is not a choice due to memory constraints. Algorithms need to incrementally process incoming data instances in a single pass while operating under memory and response time constraints. Furthermore, as data streams present transient behavior, prediction models often need to adapt to concept drifts observed in data [1] [2].

Ensemble learning comprises a class of machine learning algorithms that achieved remarkable prediction performance. However, they create a higher demand for computational resources and may break time constraints. Although parallelism can be a good strategy, many algorithms typically require collective communications with high overhead [3]. Algorithms derived from the Bootstrap Aggregating (Bagging) [4] ensemble algorithm do not depend on collective communications. They are typically composed of several homogeneous, weak and unstable, learners trained independently of each other. Without dependencies, minimum or no communication overhead will be present when parallelizing the underlying classifiers of the ensemble. However, since each model is different and relies on other operations than linear algebra, ensemble-based models designed to process streaming data [5] are not amendable for data parallelism. Ensembles usually operate on dynamic data structures used to model concept drift and non-stationary data behavior. Therefore, task parallelism has to be applied, which brings new challenges in the form of memory usage patterns and cache memory performance in multi-core environments.

In order to asses our strategy, we use the popular Stream Learning framework Massive Online Analysis (MOA) [6], which provides multiple implementations of Streaming algorithms, including a vast collection of ensemble methods. By using already implemented baselines and implementing our solution on the same environment, we remove potential biases on the evaluation. The optimizing of specific ensemble algorithms is out of the scope of this work.

The main contribution of this paper can be summarized as follows:

1) We show that task parallelism does not necessarily lead to acceptable performance of ensemble algorithms due to inefficient memory access patterns and non-stationary behavior observed in data streams;
2) We propose a parallel implementation strategy based on mini-batches, that can significantly reduce cache misses and improve performance of non-dependent ensembles, and asses it in four state-of-art ensemble algorithms;
3) We analyze the trade-off between accuracy and computational efficiency when using mini-batches. We observe that it is possible to considerably speed up the execution while having only a small impact on the predictive performance in most cases;
4) With further tests and improvements, our strategy can be implemented as a meta-classifier in MOA, to evenly support the execution of ensemble methods composed of independent classifiers.

This paper is organized as follows. Section 2 presents some works that tried similar things and how our work differs from those. Section 3 shows a brief explanation of the algorithms, frameworks and how our solution works from a high-level perspective. On Section 4 we go into detail about the setup of experiments, its results and discuss their implications. Finally, we finish with our concluding thoughts on Section 5.

2. Related work

Ensemble learning is a straightforward approach to improve the performance of ML models by combining several single models [5]. This approach is also popular in a streaming context, even though the limited resources and time constraints make it challenging to obtain models that are both efficient and yield high predictive performance. Algorithms such as the Adaptive Random Forest (ARF) [7] are able to produce highly accurate models, however it demands much more processing time and memory than a single incremental decision tree (Hoeffding Trees) [8].

Multi-core parallelism has been widely used to improve the performance and reduce latency of real-time applications [9], [10]. For instance, the work in [11] proposes an optimized method for incremental Hoeffding Trees and Random Forest. It was implemented in C++ and benchmarked against MOA and StreamDM using two real and eleven synthetic datasets. It is noteworthy that the authors compare the performance of a single tree and the ensemble using different hardware architectures. MapReduce/Hadoop frameworks have been used to implement parallel and distributed ensembles based on SVM [12], extreme learning machine (ELM) [13], and Random Prism Ensemble [14]. Although MapReduce can process huge amounts of data with high scalability [15], it is not suitable for applications with low response times. Shen et al. [16] implement the Empirical Mode Decomposition (EMD) model for Earth Sciences data analysis using OpenMP and MPI, which support fine tuning mechanisms for parallelism. In contrast, similar approach could not be applied to the present work which uses, the language in which MOA is implemented. In [3], Ekanayake et al. implemented a parallel K-means and Multidimensional Scaling using task parallelism on Java. Spark, Flink, and MPI frameworks are used to compare the Fork-Join thread model to an optimized BSP implementation under different thread affinity models. Although this work uses Java as base programming language, it does not focus on ensembles for data streams.

Another interesting strategy which has proven to benefit real-time and data streams applications is *micro-batching*. In summary, it consists in processing small chunks containing several data instances to be processed at once, instead of processing a single instance at a time. Variation of micro-batching have been employed in many different scenarios, such as to improve performance or fault-tolerance of streams processing systems such as Spark and Flink [17], [18]. Wang et al. [19] proposed a scheduling strategy to find energy-optimal batching periods for real-time tasks with deadline constraints to execute on heterogeneous sensors. He et al. [20] proposed Comet, a stream processing system which identifies optimal sizes of batches of data items to be processed for large-scale data streams. In [21], Zhang et al. proposed two scheduling strategies to reduce both the delay and energy consumption of executing small batches of DNN tasks on edge nodes such as IoT devices. Although the strategies can be applied to CPUS, the proposal focus on executing DNN applications on GPU devices in the edge. Similar techniques that group work units into small batches have been used in other applications areas, such as in web search engines [22], [23] or content delivery applications [24].

Although several ensemble algorithms were proposed for streaming data, methods focusing on efficient implementations are still rare. For instance, the ARF [7] includes a parallel implementation, however the strategy used parallelized the training task for each individual instance, which added an overhead in terms of thread allocation and waiting periods, even though it guaranteed that it would not affect the predictive performance in comparison to a sequential processing. To avoid this type of bottleneck, in this work we exploit general strategies that process the incoming instances using mini-batches, thus substantially improving parallel performance without compromising too much the predictive performance.

3. Proposed method

We propose general strategies that exploit mini-batches to improve the efficiency of the parallel training of ensemble learning algorithms for data streams. To demonstrate the impact of such strategies, we implement them on four well-known ensemble algorithms. Our implementations are on top of the Massive Online Analysis (MOA) framework [6], using the Java parallel framework ExecutorService. In the following sections we introduce the ensemble algorithms and the proposed strategies.

3.1. Ensemble algorithms

Bagging is one of the most used ML techniques to improve the accuracy of several weak models. Although it was proposed over 20 years ago [4], Bagging and its variants (e.g. Random Forest) are still used to this day as an effective method to reduce error without resorting to intricate models, such as deep neural networks, that are not trivial to train and fine-tune. In contrast to Boosting [25], Bagging does not create dependency among the base models, which facilitates parallelizing its execution, and also processing incoming data in an online fashion. Besides that, Bagging variants yield higher predictive performance in the streaming setting in comparison to Boosting or other ensemble methods that impose dependencies among its base models. This phenomenon was observed in several empirical experiments [7], [26], [27], [28], and it can be hypothesized that it is due to the difficulty to effectively model the dependencies in a streaming scenario as noted in [5]. The four parallelized ensemble algorithms are based on the adaptation of Bagging to an online setting by Oza and Russeal [26]. We describe each algorithm in the following paragraphs.

Online Bagging (OzaBag) [26] is an incremental adaptation of the original Bagging algorithm. The authors demonstrate how the process of bootstrapping can be adapted to an online setting using a Poisson($\lambda = 1$) distribution. In essence, instead of sampling with reposition from a static dataset, in Online Bagging, the Poisson($\lambda = 1$) is used to assign weights to each incoming instance, such that these weights represent the number of times an instance will 'repeated' to simulate bootstrapping. One concern with using $\lambda = 1$ is that about 37% of the instances will receive weight 0, thus will not be used to train, which is desired to approximate it to the offline version of Bagging, but may be detrimental to an online learning setting [5]. Therefore, other works [7], [28] increase the number of times an instance is used for training by increasing the λ parameter.

Online Bagging ADWIN (OBADWIN) [27] combines OzaBag with the ADAptive WINdow (ADWIN) [29] change detection algorithm. When a change is detected, the worst (in terms of predictive performance) classifier of the ensemble is replaced by a new classifier. ADWIN keeps a variable-length window of recently seen items, with the property that the window has the maximal length statistically consistent with the hypothesis that there has been no change in the average value inside the window. This implies that at any time the average over the existing window can be reliably taken as an estimation of the current average in the stream, with the exception of a very small or very recent change that is still not statistically visible.

Leveraging Bagging (LBag) [28] extends OBADWIN by increasing the λ parameter of the Poisson distribution to 6, effectively causing each instance to have higher weights and be used for training more often. In contrast to OBADWIN, LBag maintains one ADWIN detector per model in the ensemble and independently reset the models. This approach leverages the predictive performance of OBADWIN by merely training each model more often (higher weight) and resetting them individually. One drawback of LBag in comparison to OB and OBADWIN is that it requires more memory and processing time, since the base models are trained more often, and there are more instances of ADWIN. In [28], authors also attempted to further increase the diversity of LBag by randomizing the output of the ensemble via random output codes. However, this approach was not very successful in comparison to maintaining a deterministic combination of the models' outputs.

Adaptive Random Forest (ARF) is an adaptation of the original Random Forest algorithm [30] to the streaming setting. Random Forest can be seen as an extension of Bagging, where further diversity among the base models (decision trees) is obtained by randomly choosing a subset of features to be used for further splitting leaf nodes. ARF uses the incremental decision tree algorithm Hoeffding tree [8] and simulates resampling as in LBag, i.e., Poisson($\lambda = 6$). The Adaptive part of ARF stems from a change detection and recovery strategy based on detecting warnings and drifts per tree in the ensemble. After a warning is signaled, another model is created (namely a 'background tree') and trained without affecting the ensemble predictions. If the warning escalates to a drift signal, then the associated tree is replaced by its background tree. Notice that in the worst case, the number of tree models in ARF can be at most double the total number of trees due to the background trees, however, as noted in [7] the co-existence of a tree and its background tree is often short-lived.

Every ensemble used a Hoeffding Tree (HT) [8] as a base model. A HT is an incremental tree designed to cope with massive stationary data streams. Thus, it can make splits with reasonable confidence on the data distribution while having very few instances available. This is possible because of the Hoeffding bound, which quantifies the number of observations required to estimate some statistics. The HT also has guarantees that its model becomes more similar to non-incremental trees as more instances are seen.

3.2. Parallel framework

Threads are a well-known mechanism used to implement parallel tasks in programs. Fork-Join is an abstraction for expressing parallel implementations of algorithms without knowing in advance how much parallelism the target system will offer. In essence, Fork-Join is composed of three steps: fork, computation and join.

In the fork step, new threads are created on-demand. Then, in the computation step, each thread executes one or more tasks. Finally, in the join step, the parallel threads synchronize and finish before continuing the sequential region of the program. This fork-compute-join process can be repeated many times during the execution of a program. Fork-Join implementations usually employ thread pools which support forked tasks management to reduce the overhead of thread creation/destruction. These pooled threads are not destroyed when the task finishes but instead release resources and become idle [3].

In this study, we use **Executor Service (ES)**, a framework that implements the Fork-Join thread model. The framework is available since Java 7. It has methods to track the progress of a task and manage their termination. The main goal of this framework is to facilitate thread management through the creation of a Service with a fixed thread pool size, reserving and reusing these threads. Once a service has been created, tasks can be invoked by passing Runnables for it.

3.3. Parallel implementation

As the items in a data stream can arrive at irregular intervals and variable rates, it is not amendable for data parallelism. On the other hand, task parallelism can be applied as the underlying classifiers in bagging ensembles can execute as independent tasks which do not communicate with each other. Algorithm 1 describes a task parallel-based implementation. This version improves the performance of the current parallel implementation of the ARF algorithm [7], in the latest version in MOA [6], by reusing the data structures and avoiding the costs of allocating new ones every time the runnables would be called.

Algorithm 1 High level parallel algorithm

```
1:  Input: an ensemble E, num_threads, a data stream S
2:  P ← Create_service_thread_pool(num_threads)
3:  T ← Create_trainers_collection(E)
4:  for each arriving instance I in stream S do
5:      E.classify(I)
6:      for each trainer T_i in trainers T do
7:          k ← poisson(λ)
8:          T_i.update(I, k)
9:      end for
10:     for all trainers T do in parallel
11:         W_inst ← I * k
12:         Train_on_instance(W_inst)
13:     end for
14:     if change detected then
15:         reset_classifier
16:     end if
17: end for
```

In lines 2-3, a thread pool is started and one Trainer (runnable) is created for each classifier of the ensemble. For each arriving data instance (lines 4-17), votes from all the classifiers are obtained (line 5). Then, the *Poisson* weights are computed and trainers are updated for training in lines 6-9. The prediction phase has low computational cost because the algorithm uses Hoeffding trees [8], and thus, it is carried out sequentially. On the other hand, the training phase is more expensive as training involves updating many statistics along the trees, the calculation of new splits, and detection of changes on data distribution (for three of the methods). As the training phase dominates the computational cost, parallelism is implemented (in lines 10-13) by training many classifiers simultaneously. Finally, lines 14-16 appear only on OBAdwin, which implements a global change detector where the worst classifier of the ensemble will be reseted.

3.4. Introducing the *mini-batching*

Although parallelization of learners in an ensemble can be straightforward through task parallelism, poor memory usage can severely hinder performance. For instance, high-frequency accesses to data structures larger than cache memories can raise a performance bottleneck. Also, algorithms that continuously perform memory allocation/release operations to discard old models and create new ones during the learning/training process may create pressure on the garbage collection. To circumvent such problems, we introduced a mini-batching strategy which in essence groups several data instances of a stream in a *mini-batch* to be operated by each learner of the ensemble. Each learner is implemented by a task that processes training by iterating on each instance of a mini-batch, instead of processing a single instance at a time. When a learner is invoked, its data structures are loaded into the upper levels of the memory hierarchy (upper-level caches), and quickly accessed to process the next instances in the same mini-batch, reducing cache misses and improving performance.

Although mini-batching can hopefully improve performance, in the extreme its misuse can break the idea of stream processing. For instance, a mini-batch with length as large as the size of the entire data stream would prevent the algorithms from training along the stream processing. On the opposite side, a mini-batch size of one instance boils down to an instance-based operation.

We have implemented a mini-batching strategy that maximizes the parallel portion of the code. This strategy impacts the predictive outcome in two ways: (i) the closer to the end of the batch an instance is, the more outdated is the model classifying it; (ii) instead of using a global random number sequence, each trainer has its own random generator. The Algorithm 2 shows the mini-batching strategy.

The first difference between the parallel mini-batching and parallel single instance approach appears in lines 5-6 of algorithm 2, where the ensemble will only accumulate the instances until the desired mini-batch size is met or the stream ends. When this condition is fulfilled, classification (line 7) and training (lines 8-21) will take place. In line 9 the whole mini-batch is copied to each trainer, the weight is left to be calculated inside each trainer. Line 11 starts the parallel section, each trainer will iterate over all mini-batch instances while calculating the weight, creating the weighted instance and training the classifier with this instance (lines 12-16). ARF and LBag, exclusively, will execute lines 17-19 as a local change detector for each classifier in the ensemble. Finally, in line 20 the mini-batch is emptied to begin accumulating again. In OBAdwin lines 17-19 would be outside the parallel section, as the change detection is a global operation.

Thanks to our strategy, the amount of times the models have to be fetched from memory was reduced. Each model is several times larger than a single instance, thus, fetching the

Algorithm 2 mini-batching algorithm

Input: an ensemble E, $num\_threads$, a data stream S, mini-batch size L_{mb}
$P \leftarrow Create\_service\_thread\_pool(num\_threads)$
$T \leftarrow Create\_trainers\_collection(E)$
for each arriving instance I in stream S **do**
    $B.append(I)$
    if $B.size() == L_{mb}$ **then**
        E.classify(B)
        for each trainer T_i in trainers T **do**
            $T_i.instances.append(B)$
        end for
        for all trainers T **do in parallel**
            for each instance I in B **do**
                $k \leftarrow poisson(\lambda)$
                $W\_inst \leftarrow I * k$
                $Train\_on\_instance(W\_inst)$
            end for
            if change detected **then**
                $reset\_classifier$
            end if
            $B.clear()$
        end for
    end if
end for

whole model to the upper caches to process a single instance was highly inefficient. The amount of times a model has to be fetched can be roughly estimated by $S * T$ on the single instance parallel version and $(S/B) * T$ on the mini-batch version. S represents the total amount of instances in the Stream, T the ensemble size, and B the batch size.

4. Experiments and results

First of all, we need to choose the implementation platform for the experiments. We chose MOA[1] because it implements several ensemble learners for data streams processing, while having been used for many studies in ML techniques [6]. This choice allows us to assess the efficiency of the *mini-batching* strategy for several ensemble algorithms. Although MOA is implemented in Java, it supports efficient task parallelism through native APIs. Despite environments such as OpenMP or MPI provide remarkable support for the parallel implementation of *ad hoc* algorithms, this approach is out of the scope of this study. The hardware used for experiments is described in Table 1.

We executed the experiments in a dedicated environment. Execution runtime is measured as wall clock time including both prediction and training steps. Each configuration was executed three times and results were averaged. Sequential executions used the original code without any parallel framework, thus avoiding overhead. Parallel executions, varied the following parameters: amount of cores [2, 4, 8], ensemble size [10, 25, 50, 100, 150]. We have chose to present only results from ensemble size 100 and 150, as well as only the eight core parallel data. Initially only the experiments with bigger ensembles showed benefits, smaller ensembles were not tested with the mini-batch technique, thus they were left out of the report. Mini-batch experiments used mini-batch sizes [50, 500, 2000], ensemble sizes [100, 150] and eight cores.

TABLE 1. HARDWARE SPECIFICATIONS.

Processor	Silver 4208	Intel 2600
Cores/socket	8	4
Threads/core	2	2
Clock frequency (GHz)	2.1	3.4
L1 cache (core)	32 KB	32 KB
L2 cache (core)	1024 KB	256 KB
L3 cache (shared)	11264 KB	8192 KB
Memory	128 GB	16 GB
Memory frequency	2400 MHz	1333 MHz
Memory channels	6	2
Maximum bandwidth	107.3 GiB/s	21 GB/s

We have chosen to use the speed up to evaluate our solution regarding the execution performance. Graphs are presented on Subsection 4.2. Regarding the predictive performance we have analysed the impact of different mini-batch sizes on the accuracy of the algorithms. Also, we analyse the model node count and memory footprint of each ensemble in order to further explain some behaviors.

4.1. The datasets used

We have used four datasets[2] to perform the experiments, their summary statistics are shown in Table 2. A quick description of the problem each one of them was used to solve is provided below.

TABLE 2. SUMMARY OF DATASET STATISTICS

Datasets	Airlines	GMSC	Elec	CovType
# Instances	540k	150k	45k	581k
# Features	7	10	8	54
# Nominal feat	4	0	1	45
Normalized	No	No	Yes	Yes

The Airlines data set was inspired by the regression data set from Ikonomovska. The task is to predict whether a given flight will be delayed given information on the scheduled departure. Thus, it has 2 possible classes: delayed or not delayed.

The Electricity data set was collected from the Australian New South Wales Electricity Market, where prices are not fixed. These prices are affected by demand and supply of the market itself and set every 5 min. The Electricity data set tries to identify the changes of the price (2 possible classes: up or down) relative to a moving average of the last 24h. An important aspect of this data set is that it exhibits temporal dependencies.

The give me some credit (GMSC) data set is a credit scoring data set where the objective is to decide whether a

1. Available at: https://github.com/Waikato/moa

2. Available at: https://github.com/hmgomes/AdaptiveRandomForest

loan should be allowed. This decision is crucial for banks since erroneous loans lead to the risk of default and unnecessary expenses on future lawsuits. The data set contains historical data on borrowers.

The forest covertype data set represents forest cover type for 30 x 30 m cells obtained from the US Forest Service Region 2 resource information system (RIS) data. Each class corresponds to a different cover type. The numeric attributes are all binary. and there are 7 imbalanced class labels.

4.2. Results

For the sake of organization, we divided this section in three parts according to the evaluation measures: speedup, memory usage and accuracy.

4.3. Speedup analysis

We present the Speedup measure for both architectures in Figure 1. As general remark, performance is improved by combining parallelism with mini-batching strategies, which improves memory access patterns. Also, Speedups are closely related to model complexity, which varies according to algorithm and dataset used. Simpler algorithms (OzaBag and OzaBagAdwin) show smaller Speedups, due to smaller amount of work for each thread.

In Fig. 1, Speedup for Xeon architecture is presented by the blue and orange bars, while green and red bars are used for i7 architecture. The parallel implementation alone does not guarantee performance improvement on Xeon. The deterioration in performance indicates the lack of efficiency when processing one instance per time (incrementally), because each execution thread has to bring large data structures to the cache memory to process a single instance (much smaller than the model). In addition, increasing the size of the mini-batch improves performance, since bigger mini-batches further reduce the amount of times each model has to be brought from memory to cache.

On i7 architecture, the parallel implementation with single instance improves performance on all datasets, however, the usage of mini-batches presents small gains on most cases. In addition, most cases shows small performance differences regarding mini-batch size. This behavior indicates that I/O is probably the limiting factor on this architecture, given the smaller amount of memory channels coupled with smaller memory bandwidth and higher clock frequency.

Speedups optimally should approach the number of cores used, however, LBag presented speedup of 12X for the Airlines dataset, indicating the benefit of combining the two techniques.

4.4. Analysis of memory usage

Next, we analyze memory footprint and cache use during the execution of ensembles of 100 learners. Table 3 shows the maximum memory footprint for each algorithm dataset. Values represent the average of three executions. Airlines has the largest memory footprint among the datasets, while LBag is the most memory consuming algorithm. This is the case with both the largest memory footprint, and higher speedups, suggesting that its super linear speedup is due to memory use and data locality.

TABLE 3. MEMORY FOOTPRINT FOR ENSEMBLES W/ 100 LEARNERS

Dataset	ARF	LBag	OBAdwin	OB
Airlines	6.5GB	7.9GB	5.1GB	6.2GB
GMSC	3.4GB	1.6GB	0.52GB	0.53GB
Electrical	2.1GB	1.2GB	0.24GB	0.34GB
Covertype	2.2GB	3.4GB	5.8GB	3.3GB

Table 4 shows measures of cache use collected with the perf tools[3] **Cache-misses** accounts for data requests missed in all three cache levels, therefore having to fetch data from the main memory. **Cache-references** accounts for data requests missed in the L1 and L2, whether they miss the L3 is irrelevant in this case. The difference between the two is the amount of L3-hits.

Although the two measures change according to the dataset size, both measures tend to decrease when we introduce the mini-batch technique and increase the batch size. In some cases the cache_refer starts to rise with mini-batches of 2000 instances for the Electrical and Covertype datasets, suggesting that there is an optimal size for the mini-batch. The reduction in cache misses can be explained by a better locality of memory accesses due to the introduction of the mini-batch. In the original way, the data structures of each ensemble learner were accessed once for each data instance. With the mini-batch, the algorithm tends to refer to the data structures of the same learner repeatedly for each data instance of the mini-batch (as shown in lines 1-4 of Algorithm 2). This confirms that the benefit of mini-batching to profit from multithread parallelism.

4.5. Impact of mini-batch size on accuracy

After demonstrating the performance benefits, we wonder if to evaluate the impact of mini-batching on the predictive performance of the algorithms. A comparison between accuracy of the sequential and mini-batch implementations is presented in Tables 5, 6, 7 and 8, which show the accuracy for the datasets Airlines, GMSC, Electrical and CoverType, respectively. They are organized as follows. E size shows the ensemble size, B size is the number of instances in the mini-batch, the two rows with SEQ in the mini-batch size shows the accuracy for the sequential algorithm, while the remaining rows show the accuracy. Columns ending with the suffix MB+ show accuracy for the mini-batch implementation. The best accuracy of the algorithm is highlighted in bold font.

We also present Table 9 with the number of nodes across all learners in each ensemble. As such number can vary along with the execution, the number of current tree nodes was sampled several times during the execution and then averaged.

3. https://man7.org/linux/man-pages/man1/perf.1.html

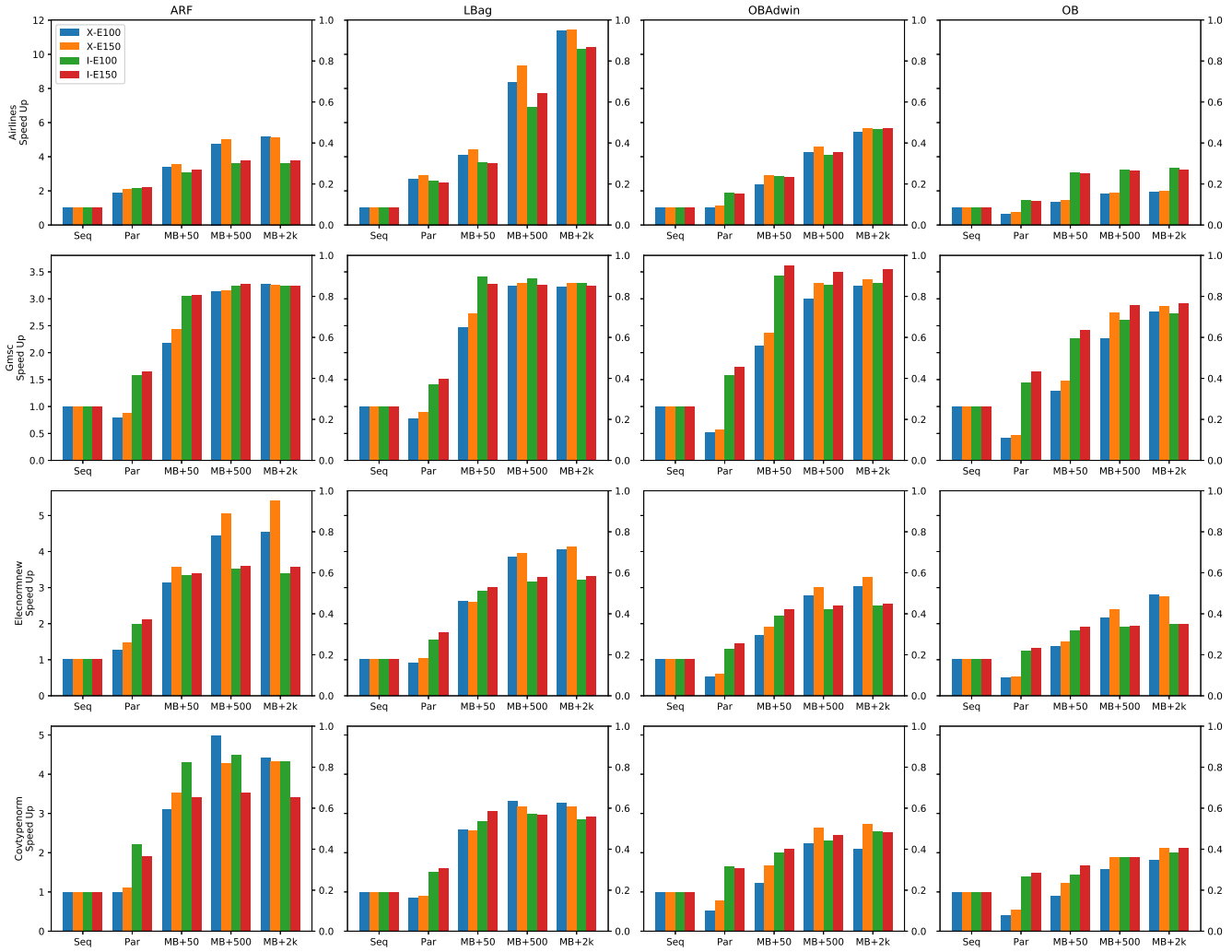

Figure 1. Speed up on both architectures. Algorithms are placed on columns, while datasets are placed in different rows of the grid. Graphs in the same row (dataset) share Y axis. E100 and E150 indicate ensembles with 100 and 150 learners, respectively. X and I indicate Xeon and i7 processor. Three algorithms described in Section 3 are depicted, one of them with 3 parameter configurations: (i) Baseline (sequential) implementation (**Seq**), (ii) Parallel implementation (**Par**), (iii) Improved Parallel with mini-batches of 50 instances (**MB+50**), (iv) Improved Parallel with mini-batches of 500 instances (**MB+500**), (v) Improved Parallel with mini-batches of 2,000 instances (**MB+2k**).

Analyzing the predictive performance impact, it seems to be more closely related to the dataset and mini-batch size than algorithm characteristics. This can be verified by the minimal variability on GMSC accuracy and the significant decreases in accuracy for all algorithms as the mini-batch size increases when classifying the Electrical dataset. ARF's predictive performance is especially affected, a behavior that can be attributed to its characteristic of building trees faster than other ensembles, i.e. given the same amount of instances, it will produce deeper trees faster as long as the algorithm is presented to instances as soon as possible [7]. The mini-batching approach hinders this characteristic, impacting the accuracy of the algorithm.

Analysing accuracy tables in conjunction with Table 9, we can offer explanations on why some datasets have significant variance in predictive performance while others show a minimal variance. In essence, the node count measure is a combination of how fast a model can grow and how often it resets. Both of these behaviors are related to resampling algorithm strategies described in Section 3. In summary: (i) ARF and LBag present a faster growth speed when compared to both OzaBag versions because of the λ parameter; (ii) Although OzaBagAdwin and OzaBag have the same growth speed, OzaBag will most likely present a higher node count because only OzaBagAdwin employs a change detector; (iii) Even having the same λ, ARF still grows faster than LBag thanks to the random subspaces technique, which reduces the amount of instances seen to perform splits on the leaf nodes.

From accuracy tables we get that airlines and GMSC have fairly stable accuracy, showing minimal variability when using batches, while electrical and covertype datasets

TABLE 4. CACHE STATISTICS FOR ENSEMBLES WITH 100 LEARNERS (MILLIONS)

		Airlines		GMSC		Electrical		Covertype	
Algorithm	MB size	cache-miss	cache-refer	cache-miss	cache-refer	cache-miss	cache-refer	cache-miss	cache-refer
ARF	inc	40,171	94,910	2,518	11,366	882	4,490	12,652	65,321
ARF	50	41,634	63,303	2,499	4,825	821	2,323	13,325	23,201
ARF	500	42,321	62,185	2,162	4,394	742	2,040	12,315	21,246
ARF	2000	42,522	61,728	2,047	4,369	728	2,293	12,367	21,912
LBag	inc	45,337	99,010	2,600	8,962	508	2,870	17,809	104,735
LBag	50	49,425	74,854	1,680	3,746	516	1,706	16,560	47,024
LBag	500	26,659	37,783	1,645	3,497	473	1,457	19,315	45,322
LBag	2000	19,556	26,152	1,546	3,714	463	1,309	21,342	48,600
OBAdwin	inc	26,627	71,987	723	5,770	232	2,037	5,780	108,281
OBAdwin	50	20,338	30,542	439	1,539	183	910	4,687	37,948
OBAdwin	500	15,417	21,888	419	1,357	177	872	5,576	35,341
OBAdwin	2000	11,669	16,414	371	1,427	149	915	6,228	33,759
OB	inc	9,423	27,560	981	5,580	221	1,864	11,314	94,976
OB	50	9,810	13,606	635	1,853	180	735	9,683	36,822
OB	500	9,504	12,468	421	1,531	173	738	7,983	32,385
OB	2000	8,965	12,299	353	1,386	155	793	7,141	32,146

TABLE 5. ACCURACY FOR AIRLINES DATASET

E size	B size	ARFMB+	LBagMB+	OBAdwinMB+	OBMB+
100	SEQ	**66.41**	63.51	66.58	**65.07**
100	50	66.38	63.46	**67.02**	65.07
100	500	66.02	64.82	66.5	65.03
100	2000	65.14	**65.86**	66.25	64.91
150	SEQ	**66.48**	63.56	66.6	**65.08**
150	50	66.34	63.44	**66.82**	65.06
150	500	66.05	65.36	66.55	65.05
150	2000	65.17	**65.85**	66.08	64.89

TABLE 6. ACCURACY FOR GMSC DATASET

E size	B size	ARFMB+	LBagMB+	OBAdwinMB+	OBMB+
100	SEQ	**93.57**	93.52	93.46	93.46
100	50	93.53	93.43	93.4	**93.5**
100	500	93.53	**93.57**	**93.52**	93.48
100	2000	93.53	93.54	93.45	93.47
150	SEQ	93.54	**93.56**	**93.5**	**93.49**
150	50	93.52	93.49	93.44	93.47
150	500	**93.55**	93.54	93.5	93.47
150	2000	93.52	93.54	93.47	93.45

TABLE 7. ACCURACY FOR ELEC DATASET

E size	B size	ARFMB+	LBagMB+	OBAdwinMB+	OBMB+
100	SEQ	**89.05**	**89.16**	**84.94**	**82.83**
100	50	80.23	80.8	80.58	78.9
100	500	78.38	77.7	77.79	76.87
100	2000	76.19	75.27	75.54	75.35
150	SEQ	**88.97**	**88.71**	**84.73**	**82.67**
150	50	80.49	80.51	80.24	78.98
150	500	78.22	77.43	77.27	77.13
150	2000	76.21	75.17	75.29	75.13

TABLE 8. ACCURACY FOR COVERTYPE DATASET

E size	B size	ARFMB+	LBagMB+	OBAdwinMB+	OBMB+
100	SEQ	**92.65**	**93.36**	**84.29**	**84.42**
100	50	89.64	91.43	83.27	83.55
100	500	89.11	91.46	84.71	82.87
100	2000	84.82	85.97	82.1	79.36
150	SEQ	**92.59**	**93.49**	**84.32**	**84.32**
150	50	89.35	91.63	83.27	83.55
150	500	89.04	91.26	85.17	82.78
150	2000	84.59	85.46	81.58	79.33

show a bigger decrease. Besides, except in the case of constant resets, OzaBag should never have a bigger or similar node count to LBag, indicating that Covertype and Electrical datasets suffer from many resets. Since the batch reduces the frequency and possible amount of resets of the algorithm, it could explain why these datasets' predictive performance is more intensely affected and also establish a correlation to the erratic behavior found on cache data for these two datasets.

5. Conclusion

Ensemble learning is a straightforward approach to improve the performance of ML models by combining several single models. Examples of this class include the algorithms Adaptive Random Forest, Leveraging Bag, and OzaBag. This approach is also popular in a data streams processing context. Despite of their importance, many aspects of their efficient implementation still remain to be studied. In this paper we proposed an efficient strategy to improve locality of the memory access pattern and reduce processing time. We demonstrated that the performance achieved by multicore parallelism can be remarkably improved through the application of a mini-batching technique and showed a trade-off between mini-batching size and prediction performance.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Programa Institucional de Internacionalização – CAPES-PrInt UFSCar (Contract 88887.373234/2019-00). Authors also thank Stic AMSUD (project 20-STIC-09), and FAPESP (contract numbers 2018/22979-2, 2018/00452-2, and 2015/24461-2) for their support.

TABLE 9. AVERAGE NODE COUNT FOR ENSEMBLES WITH 100 LEARNERS

Dataset	ARF	LBag	OBAdwin	OB
Airlines	1,278,406	613,744	372,750	457,193
GMSC	187,246	33,227	5,200	3,724
Electrical	31,488	4,361	615	2,480
Covertype	19,134	3,835	389	9,313

References

[1] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, Jun 2018. [Online]. Available: https://doi.org/10.1186/s13174-018-0087-2

[2] X. Fei, N. Shah, N. Verba, K.-M. Chao, V. Sanchez-Anguix, J. Lewandowski, A. James, and Z. Usman, "Cps data streams analytics based on machine learning for cloud and fog computing: A survey," *Future Generation Computer Systems*, vol. 90, pp. 435 – 450, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17330613

[3] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox, "Java thread and process performance for parallel machine learning on multicore hpc clusters," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 347–354.

[4] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996. [Online]. Available: https://doi.org/10.1007/BF00058655

[5] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–36, 2017.

[6] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "Moa: Massive online analysis, a framework for stream classification and clustering," in *Proceedings of the First Workshop on Applications of Pattern Analysis*, 2010, pp. 44–50.

[7] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfharinger, G. Holmes, and T. Abdessalem, "Adaptive random forests for evolving data stream classification," *Machine Learning*, vol. 106, no. 9, pp. 1469–1495, 2017. [Online]. Available: https://doi.org/10.1007/s10994-017-5642-8

[8] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM SIGKDD, Sep. 2000, pp. 71–80.

[9] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, pp. 1–44, 2011.

[10] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[11] D. Marrón, E. Ayguadé, J. R. Herrero, J. Read, and A. Bifet, "Low-latency multi-threaded ensemble learning for dynamic big data streams," in *IEEE International Conference on Big Data (BIGDATA)*, 2017.

[12] N. K. Alham, M. Li, Y. Liu, and M. Qi, "A mapreduce-based distributed svm ensemble for scalable image classification and annotation," *Computers & Mathematics with Applications*, vol. 66, no. 10, pp. 1920 – 1934, 2013, iCNC-FSKD 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0898122113004434

[13] S. Huang, B. Wang, J. Qiu, J. Yao, G. Wang, and G. Yu, "Parallel ensemble of online sequential extreme learning machine based on mapreduce," *Neurocomputing*, vol. 174, pp. 352 – 367, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231215011662

[14] F. Stahl, D. May, and M. Bramer, "Parallel random prism: A computationally efficient ensemble learner for classification," in *Research and Development in Intelligent Systems XXIX*, M. Bramer and M. Petridis, Eds. London: Springer London, 2012, pp. 21–34.

[15] H. Senger, V. Gil-Costa, L. Arantes, C. A. Marcondes, M. Marín, L. M. Sato, and F. A. Da Silva, "Bsp cost and scalability analysis for mapreduce operations," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2503–2527, 2016.

[16] B. Shen, S. Cheung, Y. Wu, J. Li, and D. Kao, "Parallel implementation of the ensemble empirical mode decomposition (peemd) and its application for earth science data analysis," *Computing in Science Engineering*, pp. 1–1, 2017.

[17] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[19] D. Wang, T. Abdelzaher, B. Priyantha, J. Liu, and F. Zhao, "Energy-optimal batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mplatform case study," *Real-Time Systems*, vol. 48, no. 2, pp. 135–165, 2012.

[20] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 63–74.

[21] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang, "Edge-batch: Towards ai-empowered optimal task batching in intelligent edge systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 366–379.

[22] C. Bonacic, D. Bustos, V. Gil-Costa, M. Marin, and V. Sepulveda, "Multithreaded processing in dynamic inverted indexes for web search engines," in *Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval*, ser. Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval. ACM, 2015, pp. 15–20.

[23] R. Gaioso, V. Gil-Costa, H. Guardia, and H. Senger, "Performance evaluation of single vs. batch of queries on gpus," *Concurrency and Computation: Practice and Experience*, p. e5474, 2019.

[24] C. Jayasundara and V. Gopalakrishnan, "Facilitating multicast in vod systems by content pre-placement and multistage batching," in *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, 2013, pp. 1–10.

[25] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *ICML*, vol. 96, 1996, pp. 148–156.

[26] N. C. Oza and S. Russell, "Online bagging and boosting," in *Eighth International Workshop on Artificial Intelligence and Statistics*, T. Jaakkola and T. Richardson, Eds. Key West, Florida. USA: Morgan Kaufmann, January 2001, pp. 105–112.

[27] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà, "New ensemble methods for evolving data streams," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 139–148.

[28] A. Bifet, G. Holmes, and B. Pfahringer, "Leveraging bagging for evolving data streams," in *Machine Learning and Knowledge Discovery in Databases*, J. L. Balcázar, F. Bonchi, A. Gionis, and M. Sebag, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 135–150.

[29] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *Proceedings of the 7th SIAM International Conference on Data Mining*, vol. 7, 04 2007.

[30] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.