Low-latency Multi-threaded Ensemble Learning for Dynamic Big Data Streams

Diego Marrón*†, Eduard Ayguadé*†, José R. Herrero†, Jesse Read‡, Albert Bifet§
*Computer Sciences Department, Barcelona Supercomputing Center
Barcelona, Spain
Email: diego.marron,eduard.ayguade@bsc.es
† Computer Architecture Department, Universitat Politécnica de Catalunya,
Barcelona, Spain
Email: dmarron,eduard,josepr@ac.upc.edu
‡ LIX, École Polytechnique
Palaiseau, France
Email: jesse.read@polytechnique.edu
§ LTCI, Télécom ParisTech, Université Paris-Saclay,
75013 Paris, France
Email: albert.bifet@telecom-paristech.fr

***Abstract*—Real–time mining of evolving data streams involves new challenges when targeting today's application domains such as the Internet of the Things: increasing volume, velocity and volatility requires data to be processed on–the–fly with fast reaction and adaptation to changes. This paper presents a high performance scalable design for decision trees and ensemble combinations that makes use of the vector SIMD and multicore capabilities available in modern processors to provide the required throughput and accuracy. The proposed design offers very low latency and good scalability with the number of cores on commodity hardware when compared to other state–of–the art implementations. On an Intel i7-based system, processing a single decision tree is 6x faster than MOA (Java), and 7x faster than StreamDM (C++), two well-known reference implementations. On the same system, the use of the 6 cores (and 12 hardware threads) available allow to process an ensemble of 100 learners 85x faster that MOA while providing the same accuracy. Furthermore, our solution is highly scalable: on an Intel Xeon socket with large core counts, the proposed ensemble design achieves up to 16x speedup when employing 24 cores with respect to a single threaded execution.**

***Keywords*-Data Streams, Random Forests, Hoeffding Tree, Low-latency, High performance**

I. Introduction

Modern daily life generates an unprecedented amount of dynamic big data streams (Volume), at high ratio (Velocity), in different forms of data such as text, images or structured data (Variety), with new data rapidly superseding old data (Volatility). This increase in volume, velocity and volatility requires data to be processed on–the–fly in real–time, with fast reaction and adaptation to changes, sometimes in the order of few milliseconds. Some scenarios and applications where real-time data streams classification is required are TCP/IP packet monitoring [1]; sensor network security [2]; or credit card fraud detection [3], just to name a few.

Real–time classification imposes the following constraints: the classifier must be ready to predict at any time, has to be able to deal with potentially infinite data streams, and has to use each sample in the data stream only once (with limited amount of CPU cycles and memory). In addition, in order to meet the throughput and accuracy requirements imposed by current and future applications, real–time classification algorithms have to be implemented making an efficient use of modern CPUs capabilities.

Incremental decision trees have been proposed for learning in data streams, making a single pass on data and using a fixed amount of memory. The *Hoeffding Tree* (HT [4]) and its variations are the most effective and widely used incremental decision trees. They work out-of-the-box (no hyper-parameters to tune), and are able to build very complex trees with acceptable computational cost. To improve single HT predictive performance, multiple HTs are combined with ensemble methods. Random Forests (RF [5]) and Leveraging Bagging (LB [6]) are two examples of ensemble methods, making use of randomization in different ways. Changes on the stream, which can can cause less accurate predictions as time passes, are detected by using *Drifting Detectors* [7].

This paper presents the design of a high–performance low–latency incremental HT and multi-threaded RF ensemble. Modularity, scalability and adaptivity to a variety of hardware platforms, from edge to server devices, are the main requirements that have driven the proposed design. The paper shows the opportunities the proposed design offers in terms of optimised cache memory layout, use of vector SIMD capabilities available in functional units and use of multiple cores inside the processor. Although the parallelisation of decision trees and ensembles for batch classification has been considered in the last years, the solutions proposed do not meet the requirements of real-time streaming.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

The paper also contributes an extensive evaluation of the proposed designs, in terms of accuracy and performance, and comparison against two state–of–the–art reference implementations: MOA (Massive Online Analysis [8]) and StreamDM [9]. For the evaluation, the paper considers two widely used real datasets and a number of synthetic datasets generated using some of the available stream generators in MOA. The proposed designs are evaluated on a variety of hardware platforms, including Intel i7 and Xeon processors and ARM–based SoC from Nvidia and Applied Micro. The paper also shows how the proposed single decision tree behaves in low–end devices such as the Raspberry RPi3.

The rest of the paper is organised as follows: the necessary background on HT and RF are described in Section II. The proposed designs are then presented in Sections III and IV for a single HT and the RF ensemble, respectively. The main results coming out of the experimental evaluation, in terms of accuracy, throughput and scalability, are reported in Section V. Related work and research efforts are identified in Section VI. Finally, some implementation notes are outlined in Section VII, finishing the paper with some conclusions and future work in Section VIII.

II. Background

A. Hoeffding Tree

The Hoeffding Tree (HT) is an incrementally induced decision-tree data structure in which each internal node tests a single attribute and the leaves contain classification predictors; internal nodes are used to route a sample to the appropriate leaf where the sample is labelled. The HT grows incrementally, splitting a node as soon as there is sufficient statistical evidence. The induction of the HT mainly differs from batch decision trees in that it processes each instance once at time of arrival (instead of iterating over the entire data). The HT makes use of the Hoeffding Bound [10] to decide when and where to grow the tree with theoretical guarantees on producing a nearly-identical tree to that which would be built by a conventional batch inducer.

Algorithm 1 shows the HT induction algorithm. The starting point is an HT with a single node (the root). Then, for each arriving instance X the induction algorithm is invoked, which routes through HT the instance X to leaf l (line 1). For each attribute X_i in X with value j and label k, the algorithm updates the statistics in leaf l (line 2) and the number of instances n_l seen at leaf l (line 3).

Splitting a leaf is considered every certain number of instances (*grace* parameter in line 4, since it is unlikely that an split is needed for every new instance) and only if the instances observed at that leaf belong to different labels (line 5). In order to make the decision on which attribute to split, the algorithm evaluates the split criterion function G for each attribute (line 6). Usually this function is based on the computation of the Information Gain, which is defined as:

$$G(X_i) = \sum_{j}^{L} \sum_{k}^{V_i} \frac{a_{ijk}}{T_{ij}} \log(\frac{a_{ijk}}{T_{ij}}) \quad \forall \ i \in N \qquad (1)$$

being N is the number of attributes, L the number of labels and V_i the number of different values that attribute i can take. In this expression T_{ij} is the total number of values observed for attribute i with label j, and a_{ijk} is the number of observed values for which attribute i with label j has value k. The Information Gain is based on the computation of the *entropy* which is the sum of the probabilities of each label times the logarithmic probability of that same label. All the information required to compute the Information Gain is obtained from the counters at the HT leaves.

The algorithm computes G for each attribute X_i in leaf l independently and chooses the two best attributes X_a and X_b (lines 7–8). An split on attribute X_a occurs only if X_a and X_b are not equal, and $X_a - X_b > \epsilon$, where ϵ is the Hoeffding Bound which is computed (line 9) as:

$$\epsilon = \sqrt{\frac{R^2 ln(\frac{1}{\delta})}{2n_l}} \qquad (2)$$

being $R = log(L)$ and δ the confidence that X_a is the best attribute to split with probability $1 - \delta$. If the two best attributes are very similar (i.e. $X_a - X_b$ tends to 0) then the algorithm uses a tie threshold (τ) to decide splitting (lines 10–11).

Once splitting is decided, the leaf is converted to an internal node testing on X_a and a new leaf is created for each possible value X_a can take; each leaf is initialised using the class distribution observed at attribute X_a counters (lines 12–15).

Although it is not part of the induction algorithm shown in Algorithm 1, predictions are made at the leaves using leaf classifiers applied to the statistics collected in them. Different options are possible, being Naive Bayes (NB) one of the most commonly used, a relatively simple classifier that applies Bayes' theorem under the *naive* assumption that all attributes are independent.

B. Random Forest

Random Forest (RF) is an ensemble method that combines the predictions of several individual *learners*, each with its own HT, in order to improve accuracy. Randomisation is applied during the induction process that forms the HT ensemble: on one side adding randomisation to the input training set that each HT observes (sampling with replacement); and on the other side randomising the particular set of attributes that are used when a new leaf is created (i.e. when splitting is applied).

The streaming RF design proposed in this paper makes use of Leveraging Bagging [6]: to randomise the input training set and simulate sampling with replacement, each input in

Algorithm 1 Hoeffding Tree Induction

```
Require:
    $X$: labeled training instance
    HT: current decision tree
    $G(.)$: splitting criterion function
    $\tau$: tie threshold
    $grace$ : splitting-check frequency (defaults to 200)
 1: Sort X to a leaf $l$ using HT
 2: Update attribute counters in $l$ based on X
 3: Update number of instances $n_l$ seen at $l$
 4: if ($n_l$ mod $grace$=0)
 5: and (instances seen at $l$ belong to different classes) then
 6:     For each attribute $X_i$ in $l$ compute $G(X_i)$
 7:     Let $X_a$ be the attribute with highest $G$ in $l$
 8:     Let $X_b$ be the attribute with second highest $G$ in $l$
 9:     Compute Hoeffding Bound $\epsilon$
10:     if $X_a \neq X_b$ and $G_l(X_a) - G_l(X_b) > \epsilon$
11:                                         or $\epsilon < \tau$ then
12:         Replace $l$ with an internal node testing on $X_a$
13:         for each possible value of $X_a$ do
14:             Add new leaf with derived statistics from $X_a$
15:         end for
16:     end if
17: end if
```

the training set receives a random weight w that indicates how many times this input would be repeated; this weight is generated using a Poisson distribution $P(\lambda)$ with $\lambda = 6$. When the input is routed to the appropriate leaf node during the induction process, the statistics (lines 2 and 3 in Algorithm 1) are updated based on the value of w.

In order to add randomisation when splitting a node, for each different leaf to be created , *RF* randomly selects $\lfloor\sqrt{N}\rfloor$ attributes (N is the total number of attributes) out of those that are not in the path from the root of the tree to the node being split. This variation of the HT induction algorithm affects lines 13-15 in Algorithm 1 and it is called *randomHT*.

As soon as drifting [7] is detected in any of the *learners*, one complete *randomHT* is pruned (substituted with a new one that only contains the root with $\lfloor\sqrt{N}\rfloor$ attributes). Several drift detectors have been proposed in the literature, being ADWIN [11] one of the most commonly used.

Finally, the output of each learner is combined to form the final ensemble prediction. In this paper we combine the classifier outputs by adding them, and selecting the label with the highest value.

III. LMHT Design Overview

This section presents the design of *LMHT*, a binary Low-latency Multi-threaded Hoeffding Tree aiming at providing portability to current processor architectures, from mobile SoC to high–end multicore processors. In addition, LMHT has been designed to be fully modular so that it can be reused as a standalone tree or as a building block for other algorithms, including other types of decision trees and ensembles.

A. Tree Structure

The core of the LMHT binary tree data structure is completely agnostic with regard to the implementation of leaves and counters. It has been designed to be cache friendly, compacting in a single L1 CPU cache line an elementary binary sub-tree with a certain depth. When the processor requests a node, it fetches a cache line into L1 that contains an entire sub-tree; further accesses to the sub-tree nodes result in cache hits, minimising the accesses to main memory. For example, Figure 1 shows how a binary tree is split into 2 sub-trees, each one stored in a different cache line. In this example, each sub-tree has a maximum height of 3, thus, a maximum of 8 leaves and 7 internal nodes; leaves can point to root nodes of other sub-trees.

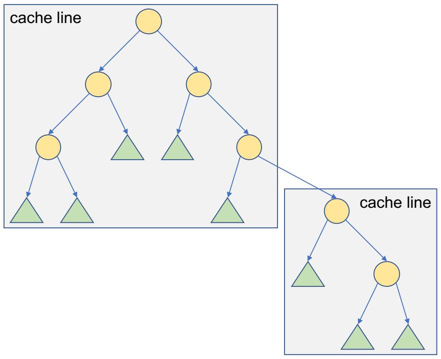

Figure 1. Splitting a binary tree into smaller binary trees that fit in cache lines

In the scope of this paper we assume 64-bit architectures and cache line lengths of 64 bytes (the usual in Intel x86_64 and ARMv8 architectures today). Although 64 bits are available only 48 bits are used to address memory, leaving 16 bits for arbitrary data. Based on that we propose the cache line layout shown in Figure 2: 8 consecutive rows, each 64 bits wide storing a leaf flag (1 bit), an attribute index (15 bits) and a leaf pointer address (48 bits).

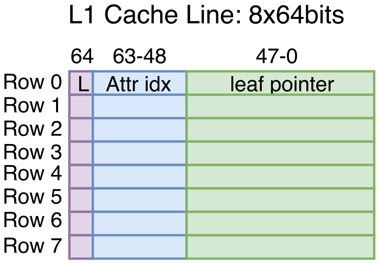

Figure 2. Sub-tree L1 cache line layout

With this layout a cache line can host a sub-tree with a maximum height of 3 (8 leaves and 7 internal nodes, as the example shown in Figure 2). The 1-bit leaf flag informs if the 48-bit leaf pointer points to the actual leaf node data structure (where all the information related with the leaf is stored) or points to the root node of the next sub-tree. The 15-bit attribute index field indexes the attribute that is used in each one of the 7 possible internal nodes. This imposes a maximum of 2^{15} (32,768) combinations (i.e. attributes per instance), one of them reserved to indicate that a sub-tree internal node is the last node in the tree traversal. For current problem sizes we do not expect this number of attributes to be a limiting factor. Having an invalid attribute index allows sub-trees to be allocated entirely and internally grow in an incremental way as needed.

The specific mapping (encoding) of the sub-tree into this 64-byte cache line layout is shown in Figure 3. Regarding attributes, the root node attribute index is stored in row 4, level 1 attributes are stored in rows 2 and 6, and level 2 attributes are stored in rows 1, 3, 5 and 7; the attribute index in row 0 is left unused. Regarding leaf pointers, they are mapped (and accessed) using a 3-bit lookup value in which each bit represents the path taken at each sub-tree level: the most significant bit is root node, next bit is the attribute in level 1, and the least significant bit represents the attribute at level 2. The bit is true if at that level the traverse took the left child, and false otherwise. The resulting value is used as the row index (offset) to access to the leaf pointer column.

B. Leaves and Counters

Each leaf node in the HT points to an instance of the data structure that encapsulates all the information that is required to compute its own split criterion function (G in Algorithm 1) and apply a leaf classifier; the design for these two functionalities is based on templates and polymorphism in order to provide the required portability and modularity. The key component in the proposed design are the leaf counters, which have been arranged to take benefit of the SIMD capabilities of nowadays core architectures.

For each label j ($0 \leq j < L$) one needs to count how many times each attribute i in the leaf ($0 \leq i < N$) occurred with each one of its possible values k ($0 \leq k < V_i$). This requires $L \times \sum_{i=0}^{N-1} V_i$ counters in total. For simplicity, in this paper we use binary attribute counters (though there is no reason why other attribute counters could not be implemented) and no missing attributes in the input instances. Therefore, for each label j one only needs to count how many times attribute i had value 1 and the total number of attributes seen for that label (in order to determine how many times each attribute i had value 0). With these simplifications $L \times (N + 1)$ counters are needed in total.

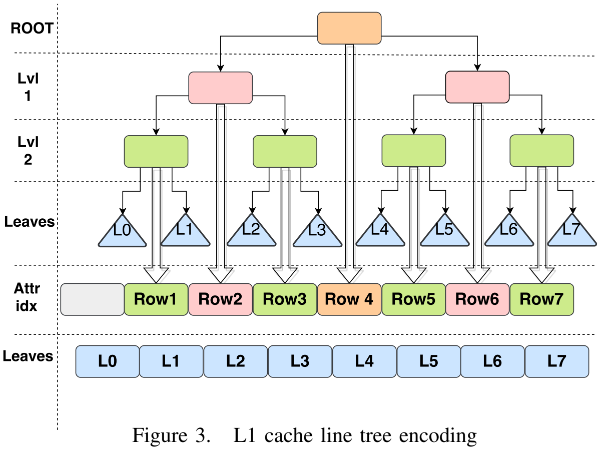

Figure 3. L1 cache line tree encoding

Attribute counters are stored consecutively in memory for each label, each one occupying a certain number of bits (32 bits in the implementation in this paper). This layout in memory allows the use of SIMD registers and instructions available in current processors. For example Intel AVX2 [12] can accommodate 8 32-bit counters in each SIMD register and operate (sum for example) them in parallel. The proposed layout allows the use of both vertical (between two SIMD registers, e.g. the same attribute for different labels) and horizontal (inside one SIMD register, e.g. different attributes or values for the same attribute for the same label) SIMD instructions. These are the operations needed to perform the additions, multiplications and divisions in expression 1 (the logarithm that is needed to compute the entropy is not available in current SIMD instruction sets). The computation of the related Naive Bayes classifier is also very similar in terms of operations required, so it also benefits from SIMD in the same way. We need to investigate how new extensions recently proposed in ARM SVE [13] and Intel AVX512 [14], which include predicate registers to define lane masks for memory and arithmetic instructions, could also be used in data structures such as the LMHT.

IV. Multithreaded Ensemble Learning

This section presents the design of a multithreaded ensemble based on *Random Forest* for data streams. The ensemble is composed of L learners, each one making use of the *randomHT* described in the previous section. The overall design aims to low-latency response time and good scalability on current multi-core processors, also used in commodity low-end hardware.

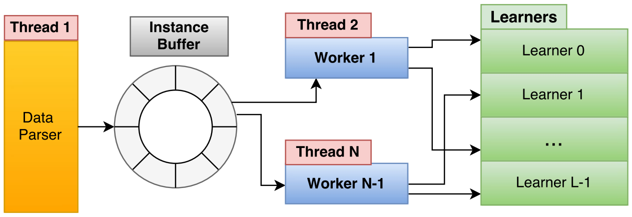

Figure 4. Multithreaded ensemble design

The proposed multithreaded implementation makes use of N threads, as shown in Figure 4: thread 1, the *Data Parser* thread, is in charge of parsing the attributes for each input sample and enqueuing into the *Instance Buffer*; threads 2 to N, the so-called *Worker* threads, execute the learners in parallel to process each of the instances in the *Instance Buffer*. The number of threads N is either the number of cores available in the processor, or the number of hardware threads the processor supports in case hyper-threading is available and enabled.

A. *Instance Buffer*

The key component in the design of the multithreaded ensamble is the *Instance Buffer*. Its design has been based on a simplified version of the LMAX disruptor [15], a highly scalable low-latency *ring buffer* designed to share data among threads.

In LMAX each thread has a *sequence number* that it uses to access the ring buffer. LMAX is based on the *single writer principle* to avoid writing contention: each thread only writes to its own sequence number, which can be read by other threads. Sequence numbers are accessed using atomic operations to ensure atomicity in the access to them, enabling the *at least one makes progress* semantics typically present on lock-less data structures.

Figure 5 shows the implementation of the Instance Buffer as an LMAX *Ring Buffer*. The *Head* points to the last element inserted in the ring and it is only written by the data parser thread, adding a new element in the ring if and only if $Head - Tail < \#slots$. Each worker thread i owns its $LastProcessed_i$ sequence number, indicating the last instance processed by worker i. The parser thread determines the overall buffer *Tail* using the circular lowest $LastProcessed_i$ for all workers i.

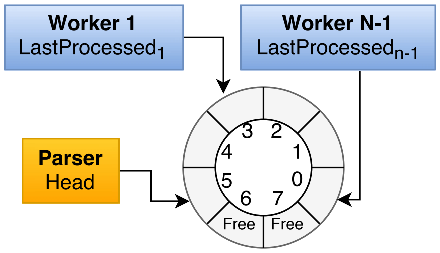

Figure 5. Instance buffer design

Atomic operations have an overhead: require fences to publish a value written (order non-atomic memory accesses). In order to minimise the overhead introduced, our proposed design allows workers to obtain instances from the *Ring Buffer* in batches. The batch size is variable, depending on the values of each worker $LastProcessed_i$ and *Head*.

B. *Random Forest Workers and Learners*

Random Forest learners are in charge of sampling the instances in the *Instance Buffer* with repetition, doing the *randomHT* inference and, if required, reseting a learner when drift is detected. Each worker thread has a number of learners ($\frac{|L|}{|N-1|}$ approximately) assigned in a static way (all learners l such that $l\%(N-1) = i$, being i the worker thread identifier). This static task distribution may introduce certain load unbalance but avoids the synchronisation that would be required by a dynamic assignment of learners to threads. In practice, we do no expect this unbalance to be a big problem due to the randomisation present in both the sampling and in the construction of the *randomHT*.

Each entry in the *Ring Buffer* stores the input instance and a buffer where each worker stores the output of the classifiers. To minimise the accesses to this buffer, each worker locally combines the output of its assigned learners for each instance; once all learners assigned to the worker are finished, the worker writes the combined result into the aforementioned buffer. Finally the data parser thread is responsible of combining the outputs produced by the workers and generating the final output.

V. Experimental Evaluation

This section evaluates the proposed design and implementation (based on templates and C++14) for both LMHT and the multithreaded RF ensemble.

Performance and accuracy are compared against two state–of–the–art reference implementations: MOA (Massive Online Analysis [8]) and StreamDM [9]. StreamDM does not provide a RF implementation, but we considered it in the single HT evaluation since it is also written in C++.

Two widely used datasets that have been used in several papers on data stream classification are used to conduct the evaluation in this section: Forest Covertype [16] and Electricity [17]. In addition, larger datasets have also been generated using some of the available synthetic stream generators in MOA. Table I summarises the resulting datasets used in this section after binarising them.

Table I
Datasets used in the experimental evaluation, including both real world and synthetic datasets

Dataset	Samples	Attributes	Labels	Generator
CoverType	581,012	134	7	Real world
Electricity	45,312	103	2	Real world
r1-6	1,000,000	91	5	RandomRBF Drift
h1-2	1,000,000	91	5	Hyperplane
l1	1,000,000	25	10	LED Drift
s1-2	1,000,000	25	2	SEA

The hardware platforms that have been used to conduct the accuracy and performance analysis in this section are summarised in Table II. A desktop-class Intel I7 platform is used to compare accuracy and to evaluate the performance of the two reference platforms, StreamDM and MOA (running on Oracle JAVA JDK 1.8.0_73). In order to perform a more complete evaluation of the throughput and scalability achieved by LMHT, additional platforms have been used, including three ARM-based systems, from low-end Raspberry Pi3 to NVIDIA Jetson TX1 embedded system and Applied

Micro X-Gene 2 server board. Finally, a system based on the latest Intel Xeon generation sockets has been used to explore the scalability limits of the multithreaded RF.

A. Hoeffding Tree Accuracy

Table III compares the accuracy achieved by MOA, StreamDM and LMHT. The main conclusion is that the three implementations behave similarly, with less than one percent difference in accuracy in all datasets but r2 and r5 for which LMHT improves almost two percent. On the real world data sets (CoverType and Electricity) learning curves are almost identically for all three implementations (not included for page limit reasons).

The minor differences that are observed are due to the fact that in few cases LMHT obtains different values for the Hoeffding Bound (eq. 2) at the same time step when compared to MOA, and this may cause node splits at different time steps (and in few cases using different attributes). MOA uses dynamic vectors to store the attribute counters. These counters are forwarded to child nodes as the previous class distribution in the presence of a split. LMHT uses a preallocated vector that can only grow. In some cases these vectors can have different sizes at the same time step, affecting the class range used to compute the bound.

B. Hoeffding Tree Throughput Evaluation

Table IV shows the throughput (instances per millisecond) achieved by the two reference implementations and LMHT on the different platforms, for each dataset and the average of all datasets. For each implementation/platform, the speedup with respect to MOA is also shown.

On the Intel i7-based platform LMHT outperforms the two reference implementations by a 6.7x factor, achieving on the average a throughput above the 500 instances per millisecond. The worst case (CoverType) has a throughput close to 250 instances (of 134 attributes) per millisecond. StreamDM performs the worst in almost all datasets, with an average slowdown compared to MOA of 0.9.

On the Jetson and X-Gene2 ARM-based platforms, LMHT performs quite similarly, achieving 3x lower throughput, on the average, compared to the Intel i7-based system. However, on the average LMHT is able to process 168 and 149 instances per millisecond on these two ARM platforms, which is better than the two reference implementations on the Intel i7, and in particular 2x better than MOA. The last column in Table IV corresponds to the RPi3 throughput, which is similar to MOA on the Intel i7, showing how the implementation of LMHT is portable to low-end devices doing real-time classification on the edge.

Figure 6 summarises the results in terms of performance. Up to this point, the main factor limiting the performance of LMHT on a single HT is the CSV parser, which is in charge of reading from a file the attributes for each input sample. In order to disect the influence of this parser, Table V shows the overhead introduced by the parser when data is read from a file or when data is already parsed and directly streamed from memory, resulting in an average 3x improvement.

C. Random Forest Accuracy and Throughput

In this section we compare the performance and accuracy of MOA and the proposed RF ensemble design with 100 learners. Table VI compares the accuracy of the two implementations, with less than one percentage points of difference in the average accuracy. The comparison with StreamDM is not possible since it does not provide an implementation for RF. The same table also shows the numerical stability of LMHT, with a small standard deviation (on 12 runs). These variations in LMHT are due to the random number generator used at the sampling and random attributes selection. LMHT uses a different seed at each execution, while MOA uses a default seed (unless a custom one is specified by the user; we used the default seed in MOA).

As with the single HT, learning curves for the real world datasets CoverType and Electricity have a similar pattern, as shown in Figure 7: at early stages LMHT is slightly better, but soon they become very similar.

Table VII summarises the results in terms of throughput, comparing again with the performance that the MOA reference implementation provides. On the same hardware platform (Intel i7) we observe an average throughput improvement of 85x compared to MOA when 11 threads are used as workers, resulting on an average throughput very close to 100 instances per millisecond; MOA throughput is less than 2 instances per millisecond in all tests. The Intel Xeon platform results in almost the same throughput than the Intel i7 which uses a much modest core count (6 instead of 24). Two main reasons for this behaviour: 1) the parser thread reads data from a CSV file stored in GPFS on a large cluster with several thousand nodes; if the parser thread directly streams data from memory, the throughput that is obtained raises to 175 instances per millisecond (143x

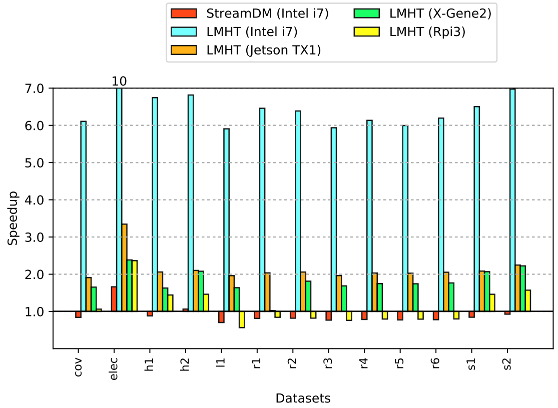

Figure 6. LMHT and StreamDM speedup over MOA using a single HT

Table II
Platforms used in the experimental evaluation

Platform	Cores	Procesor	RAM	Storage	SO	Kernel	Compiler
Intel Xeon	24	Intel Xeon Platinum 8160 @ 2.1GHz	96GB	Network (gpfs)	SUSE 12 server	4.4.59-92.20-default	GCC 7.1.0
Intel i7	6	Intel i7-5930K @ 3.7GHz	64GB	SSD	Debian 8.7	4.7.8-1	GCC 6.3.0
X-Gene2	8	ARM ARMv8-A @ 2.4 GHz	128GB	Network (gpfs)	Debian 8	4.3.0-apm_arm64_sw_3.06.25	GCC 6.1.0
Jetson TX1	4	ARM Cortex A57 @ 1.9 GHz	4GB	eMMC	L4T	3.10.96	GCC 6.1.0
Raspberry RPi3	4	ARM Cortex A53 @ 1.2GHz	1GB	Class 10 SD Card	Fedora 26 64bits	4.11.8-300.fc26.aarch64	GCC 7.1.0

Table III
Single Hoeffding Tree accuracy comparison

Dataset	MOA	StreamDM	LMHT
CoverType	**73.18**	**73.18**	73.16
Electricity	**79.14**	**79.14**	**79.14**
h1	**84.67**	**84.67**	**84.67**
h2	**78.03**	**78.03**	**78.03**
l1	**68.58**	**68.58**	68.40
r1	82.04	82.04	**82.98**
r2	43.71	43.71	**45.86**
r3	31.58	31.58	**32.24**
r4	82.04	82.04	**82.98**
r5	75.88	75.88	**77.40**
r6	73.71	73.71	**74.61**
s1	85.81	85.81	**85.85**
s2	85.75	85.75	**85.76**

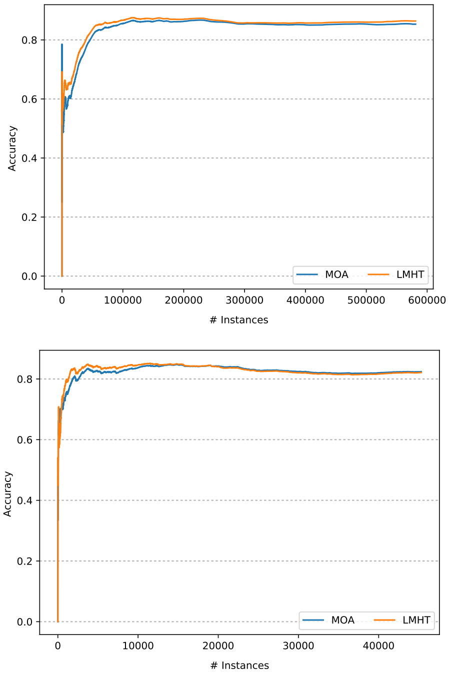

Figure 7. Random Forest learning curve: CoverType (top) and Electricity (bottom)

faster than MOA). And 2) the different clock frequencies at which the i7 and Xeon sockets operate (3.7 and 2.1 GHz, respectively, as shown in Table II); in any case, the Xeon–based platform allows us to do an scalability analysis up to a larger number of cores.

On the ARM-based platforms we observe improvements of 10x and 20x on the Jetson TX1 and X-Gene2 platforms, respectively.

D. Random Forest Scalability

Finally, this subsection analyses the scalability of the proposed ensemble implementation with the number of worker threads, always limiting the analysis to the number of cores (hardware threads) available in a socket. On the commodity Intel i7 platform, LMHT achieves a relative speedup with respect to single threaded execution, between 5-7x when using 11 workers (12 threads), as shown in Figure 8. It is interesting to observe the drop in performance observed when going from 5 to 6 worker threads. Since the i7-5930K processor has 6 cores and 12 threads (two threads mapped on the same physical core), when 6 workers are used they start competing for the same physical cores introducing some work imbalance. However, the hyperthreading capabilities of the i7-5930K are able to mitigate this as the number of threads tends to the maximum hardware threads.

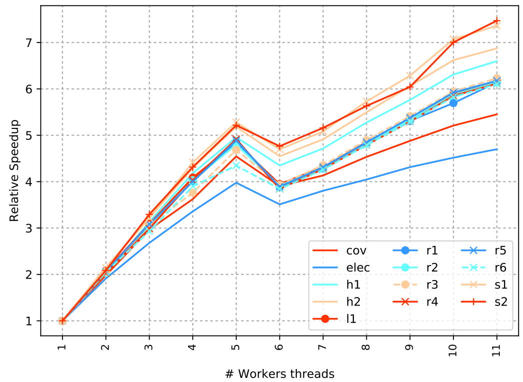

Figure 8. Random Forest speedup on Intel i7

X-Gene2 scalability has some variability for the different datasets and speed-ups in the range 5-6.5x when using 7 worker threads (Figure 9). On the other side, Jetson achieves an almost linear speedup very close to 3x when using 3 threads as workers. (Jetson scalability figure is omitted due to page limits).

In order to better study the scalability limits of the LMHT ensemble, we have extended our evaluation to one of the latest Intel Xeon Scalable Processor, the Platinum 8160 socket which include 24 cores. To better analyse if the limits are in the producer parser thread or in the implementation of the instance buffer and worker threads, we consider two scenarios: parser thread reading instances from storage and directly streaming them from memory.

Table IV
SINGLE HOEFFDING TREE THROUGHPUT COMPARED TO MOA. ↓ INDICATES SPEED-DOWN (MOA IS FASTER)

	MOA(Intel i7)	StreamDM (Intel i7)		LMHT (Intel i7)		LMHT (Jetson TX1)		LMHT (X-Gene2)		LMHT (RPi3)	
Dataset	Throughput	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup
Covertype	41.20	34.65	0.84 ↓	251.63	6.11	78.67	1.91	68.13	1.65	43.78	1.06
Electricity	36.70	60.98	1.66	415.71	11.33	122.80	3.35	87.47	2.38	86.80	2.37
h1	62.11	54.79	0.88 ↓	418.94	6.74	127.96	2.06	101.18	1.63	89.57	1.44
h2	61.14	65.04	1.06	416.67	6.81	128.49	2.10	127.16	2.08	89.39	1.46
l1	141.20	99.18	0.70 ↓	834.03	5.91	277.01	1.96	231.27	1.64	79.81	0.57 ↓
r1	51.56	41.96	0.81 ↓	333.00	6.46	104.98	2.04	52.74	1.02	43.43	0.84 ↓
r2	52.23	42.76	0.82 ↓	333.56	6.39	107.54	2.06	94.71	1.81	42.84	0.82 ↓
r3	56.06	42.98	0.77 ↓	332.78	5.94	110.06	1.96	94.55	1.69	42.60	0.76 ↓
r4	54.54	42.79	0.78 ↓	334.56	6.13	110.84	2.03	95.30	1.75	43.43	0.80 ↓
r5	54.52	42.17	0.77 ↓	326.90	6.00	110.56	2.03	95.05	1.74	43.24	0.79 ↓
r6	53.69	41.83	0.78 ↓	332.56	6.19	110.24	2.05	94.81	1.77	42.95	0.80 ↓
s1	192.68	162.81	0.84 ↓	1253.13	6.50	401.61	2.08	398.57	2.07	281.53	1.46
s2	179.22	166.09	0.93 ↓	1250.00	6.97	402.58	2.25	398.57	2.22	281.69	1.57
Average	79.76	69.08	0.90 ↓	525.65	6.73	168.72	2.14	149.19	1.80	93.16	1.13

Table V
LMHT PARSER OVERHEAD (INSTANCES PER MS)

Dataset	With Parser	In Memory	speedup
Covertype	251.63	859.49	3.42
Electricity	415.71	1618.29	3.89
h1	418.94	1647.45	3.93
h2	416.67	1636.66	3.93
l1	834.03	1550.39	1.86
r1	333.00	890.47	2.67
r2	333.56	878.73	2.63
r3	332.78	881.83	2.65
r4	334.56	889.68	2.66
r5	326.90	884.96	2.71
r6	332.56	875.66	2.63
s1	1253.13	4000.00	3.19
s2	1250.00	4000.00	3.20
Average	525.65	1585.66	3.03

Table VI
RANDOM FOREST ACCURACCY

	MOA	LMHT	
Dataset		Avg.	Std. dev.
CoverType	85.34	**86.37**	0.23
Electricity	**82.41**	82.12	0.22
h1	87.97	**88.41**	0.24
h2	**83.18**	82.48	0.24
l1	**68.68**	68.18	0.20
r1	86.35	**87.39**	0.13
r2	65.96	**68.04**	0.14
r3	40.61	**45.05**	0.12
r4	86.35	**87.41**	0.13
r5	81.42	**82.79**	0.12
r6	79.10	**79.81**	0.12
s1	86.49	**86.54**	0.24
s2	86.48	**86.53**	0.24

The two plots in Figure 10 show the speed-up, with respect to a single worker, that is achieved when using up to 24 cores (23 worker threads). With the parser from storage (top plot), an speed-up between 6-11x is obtained when using 23 worker threads, which corresponds to a parallel efficiency below 45%. Both figures raise to 10-16x and 70% when the parser streams directly from memory (bottom plot).

VI. RELATED WORK

Real-time high-throughput data stream classification has been a hot research topic in the last years. The unfeasibility to store potentially infinite data streams has led to the proposal of classifiers able to adapt to concept drifting only with a single pass through the data. Although the throughput of these proposals is clearly limited by the processing capacity of a single core, little work has been conducted to scale current data streams classification methods.

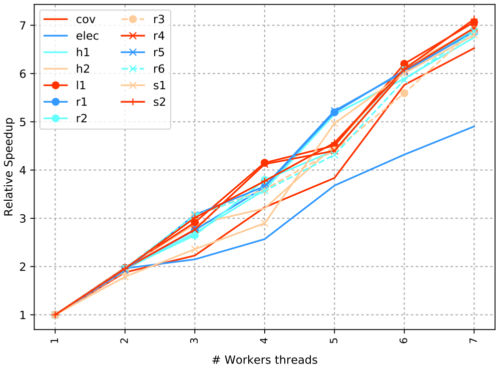

Figure 9. Random Forest speedup on X-Gene2

For example Vertical Hoeffding Trees (VHDT [18]) parallelize the induction of a single HT by partitioning the attributes in the input stream instances over a number of processors, being its scalability limited by the number of attributes. A new algorithm for building decision trees is presented in SPDT [19] based on Parallel binning instead of the Hoeffding Bound used by HTs. [20] propose MC-NN, based on the combination of Micro Clusters (MC) and nearest neighbour (NN), with less scalability than the design proposed in this paper.

Related to the parallelization of ensembles for real-time classification, [21] has been the only work proposing the porting of the entire Random Forest algorithm to the GPU, although limited to binary attributes.

VII. IMPLEMENTATION NOTES

This section includes some implementation notes that may be useful for someone willing to use the design proposed in this paper or extend its functionalities to implement other learners based on classification trees. Upon publication, the

Table VII
RANDOM FOREST THROUGHPUT COMPARISON (INSTANCES/MS)

	MOA (Intel i7)	LMHT (Intel i7, 11 workers)		LMHT (Intel Xeon, 23 workers)		LMHT (Jetson TX1, 3 workers)		LMHT (X-Gene2, 7 workers)	
Dataset	Throughput	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup
Covertype	1.30	96.48	74.22	90.85	69.89	14.68	11.29	31.31	24.09
Electricity	1.48	109.71	74.03	97.87	66.04	15.04	10.15	23.32	15.74
h1	1.15	103.42	90.25	100.63	87.82	12.51	10.92	28.52	24.89
h2	1.37	98.89	72.23	97.99	71.57	11.19	8.18	26.95	19.68
l1	1.54	100.95	65.56	142.11	92.29	12.07	7.84	26.99	17.53
r1	0.93	103.00	111.29	105.99	114.53	13.17	14.23	29.49	31.87
r2	1.15	101.54	87.97	104.55	90.57	13.05	11.30	29.21	25.31
r3	1.72	99.86	57.91	103.36	59.93	12.70	7.36	28.49	16.52
r4	0.91	102.70	113.07	103.64	114.10	13.17	14.50	29.76	32.77
r5	0.92	102.09	111.47	104.03	113.58	13.13	14.33	29.32	32.02
r6	0.94	103.08	109.72	106.76	113.63	13.14	13.98	29.49	31.39
s1	1.71	127.39	74.29	131.08	76.44	12.36	7.21	30.76	17.94
s2	1.74	124.64	71.67	127.00	73.03	11.54	6.64	30.20	17.37
Average	1.30	105.67	85.67	108.91	87.95	12.90	10.61	28.76	23.62

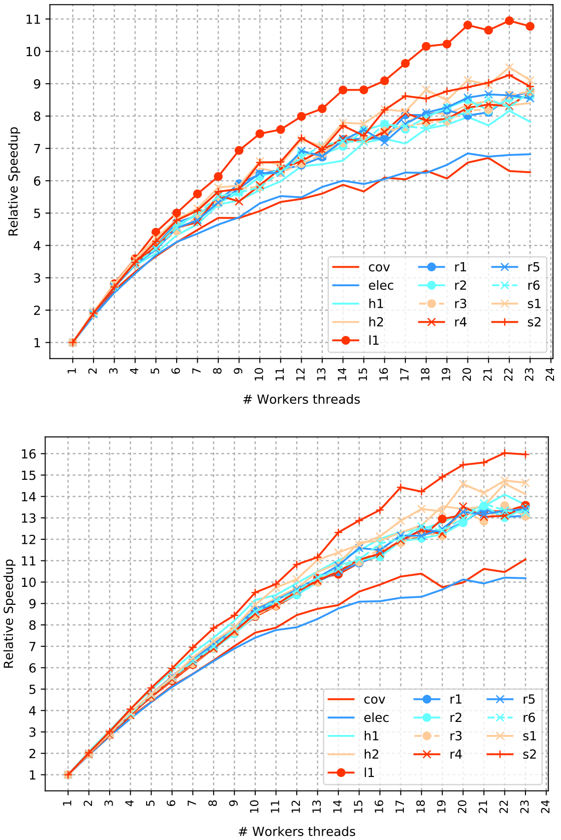

Figure 10. Intel Xeon Platinum 8160 scalability, with the parser thread streaming from storage (top) and memory (bottom).

implementation of both LMHT and the RF ensemble will be released as open source, including in this section the appropriate reference to github.

All the implementation relies on C++14 powerful template features. Templates replace dynamic allocations at runtime by static objects that can be optimized by the compiler. Attribute counters are designed using a relaxed version of the *Single Responsibility Principle* (SRP [22]). The counter class only provides the data, and any extra functionalities such as the split criterion (Information Gain in this study) or leaf classifier are implemented in a separate object. Information gain and leaf classifiers rely on the compiler for the automatic vectorisation of the computation in the HT leaves as a fast way to achieve SIMD portability.

For safe access to the instance buffer in the multithreaded implementation of Random Forest, the implementation makes use of the C++11 atomic API (`std::memory_order`), allowing to fine tune the order of memory accesses in a portable way. In particular, the use of the *memory_order_consume* for write operations and *memory_order_relaxed* for read operations. Regarding threads, although the C++11 `std::thread` offers a portable API across platforms, pinning threads to cores must be done using the native thread library (e.g. Pthreads on Linux); thread pinning has been required to improve scalability in some platforms.

VIII. CONCLUSIONS AND FUTURE WORK

This paper presented a novel design for real-time data stream classification, based on a Random Forest ensemble of randomised Hoeffding Trees. This work goes one big step further in fulfilling the low-latency requirements of today and future real-time analytics. Modularity and adaptivity to a variety of hardware platforms, from server to edge computing, has also been considered as a requirement driving the proposed design. The design favours an effective use of cache, SIMD units and multicores in nowadays processor sockets.

Accuracy of the proposed design has been validated with two reference implementations: MOA (for HT and Random Forest) and StreamDM (for HT). Throughput is evaluated on a variety of platforms. On Intel-based systems: i7 desktop (6 cores) and Xeon server (24 cores) class sockets. And on ARM-based systems: NVIDIA Jetson TX1 (4 cores), Applied Micro X-Gene2 (8 cores) and low-end Raspberry RPi3 (4 cores). For single HT the performance evaluation in terms of throughput reports 6.7x (i7), around 2x (Jetson TX1 and X-Gene2) and above 1x (RPi3) compared to MOA executed in i7. For Random Forest the evaluation reports throughput improvements of 85x (i7), 143x (Xeon parsing from memory), 10x (Jetson TX1) and 23x (X-Gene2) compared to single-threaded MOA on i7. The proposed multi-threaded implementation for the ensemble shows good scalability up to the largest core count socket that we have evaluated (75% parallel efficiency when using 24 cores on the Intel Xeon).

The evaluation also reports how the parser thread, in charge of feeding instances to the HT and ensemble, can easily limit throughput. The limits are mainly observed because of the media used to store the data (GPFS, solid–state disks, eMMC, SD, ...) that feeds the learners. For large core counts, we need to investigate if the proposed single–parser design limits scalability and find the appropriate number of learners per parser ratio. Improving the implemention in order to consider counters for attributes other than binary is also part of our near future work. Scaling the evaluation to multi-socket nodes in which NUMA may be critical for performance and extending the design to distribute the ensemble across several nodes in a cluster/distributed system are part of our future work.

Acknowledgments

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project, by the Generalitat de Catalunya (contract 2014-SGR-1051), by the Universitat Politècnica de Catalunya through an FPI/UPC scholarship and by NVIDIA through the UPC/BSC GPU Center of Excellence.

References

[1] T. Bujlow, T. Riaz, and J. M. Pedersen, "Classification of http traffic based on c5.0 machine learning algorithm," in *2012 IEEE Symposium on Computers and Communications (ISCC)*, July 2012.

[2] A. Jadhav, A. Jadhav, P. Jadhav, and P. Kulkarni, "A novel approach for the design of network intrusion detection system(nids)," in *International Conference on Sensor Network Security Technology and Privacy Communication System*, May 2013.

[3] A. Salazar, G. Safont, A. Soriano, and L. Vergara, "Automatic credit card fraud detection based on non-linear signal processing," in *2012 IEEE International Carnahan Conference on Security Technology (ICCST)*, Oct 2012.

[4] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 71–80.

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

[6] A. Bifet, G. Holmes, and B. Pfahringer, "Leveraging Bagging for Evolving Data Streams," in *Machine Learning and Knowledge Discovery in Databases*, 2010, pp. 135–150–150.

[7] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 44:1–44:37, Mar. 2014.

[8] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive online analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, Aug. 2010.

[9] *StreamDM-C++: C++ Stream Data Mining*. IEEE Computer Society, 2015. [Online]. Available: https://github.com/huawei-noah/streamDM-Cpp

[10] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

[11] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *In SIAM International Conference on Data Mining*, 2007.

[12] Intel, "Optimizing performance with intel advanced vector extensions. intel white paper," 2014.

[13] F. Petrogalli, "A sneak peek into sve and vla programming. arm white paper," 2016.

[14] I. Corporation, "Intel architecture instruction set extensions programming reference," 2016.

[15] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, *DISRUPTOR: High performance alternative to bounded queues for exchanging data between concurrent threads*, 2015.

[16] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," 1999.

[17] M. Harries, U. N. cse tr, and N. S. Wales, "Splice-2 comparative evaluation: Electricity pricing," Tech. Rep., 1999.

[18] N. Kourtellis, G. D. F. Morales, A. Bifet, and A. Murdopo, "Vht: Vertical hoeffding tree," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 915–922.

[19] Y. Ben-Haim and E. Tom-Tov, "A streaming parallel decision tree algorithm," *J. Mach. Learn. Res.*, vol. 11, pp. 849–872, Mar. 2010.

[20] M. Tennant, F. Stahl, O. Rana, and J. B. Gomes, "Scalable real-time classification of data streams with concept drift," *Future Generation Computer Systems*, April 2017.

[21] D. Marron, G. D. F. Morales, and A. Bifet, "Random forests of very fast decision trees on gpu for mining evolving big data streams," in *Proceedings of ECAI 2014*, 2014.

[22] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, ser. Alan Apt series. Pearson Education, 2003.