What Are the Limits of Evolutionary Induction of Decision Trees?

Krzysztof Jurczuk(✉), Daniel Reska, and Marek Kretowski

Faculty of Computer Science, Bialystok University of Technology,
Wiejska 45a, 15-351 Bialystok, Poland
{k.jurczuk,d.reska,m.kretowski}@pb.edu.pl

Abstract. For typical assessment of applying machine learning or data mining techniques, accuracy and interpretability are usually the most important elements. However, when the analyst is faced with real contemporary big data problems, scalability and efficiency become crucial factors. Parallel and distributed processing support is often an indispensable component of operational solutions.

In the paper, we investigate the applicability of evolutionary induction of decision trees to large-scale data. We focus on the existing Global Decision Tree system, which searches the tree structure and tests in one run of an evolutionary algorithm. Evolved individuals are not encoded, so the specialized genetic operators and their application schemes are used. As in most evolutionary data mining systems, every fitness evaluation needs processing the whole training dataset. For high-dimensional datasets, this operation is very time consuming and to overcome this deficiency, two acceleration solutions, based on the most promising, latest approaches (NVIDIA CUDA and Apache Spark) are presented. The fitness calculations are delegated, while the core evolution is unchanged. In the experimental part, among others, we identify what are dataset dimensions which can be efficiently processed in the fixed time interval.

Keywords: Evolutionary data mining · Decision trees
Parallel and distributed computing · Spark · GPU · CUDA

1 Introduction

Decision trees [13] are one of the most popular forms of knowledge, which can be automatically discovered from the learning dataset. Typical induction algorithm is based on the well-known top-down approach [20]. Such a greedy heuristics, based on the classical divide and conquer schema, proved to be really fast and accurate. On the other hand, it can be easily shown that the resulting trees, even after post-pruning, are very often overgrown and not stable [17].

More global induction methods, especially based on the evolutionary approaches [2], have emerged recently as interesting alternatives. In this type of algorithms, a tree structure, all tests in non-terminal nodes and all predictions in leaves are searched simultaneously. Global methods are clearly more

A. Auger et al. (Eds.): PPSN 2018, LNCS 11102, pp. 461–473, 2018.
https://doi.org/10.1007/978-3-319-99259-4_37

computationally complex, but the generated decision trees can be significantly simpler, without sacrificing the prediction quality. For large-scale data, however, the potential gains from applying the evolutionary approach may be unachievable, as the population-based and iterative induction can be simply too slow. Moreover, as in many evolutionary data mining algorithms, the whole training dataset should reside in memory, since it is extensively reexamined. As a result, memory constraints may influence the applicability of such methods for larger datasets.

It is clear that the possible success of evolutionary induction of decision trees for large-scale data depends on the availability, easiness of use and costs of use of (parallel or distributed) acceleration solutions. In this paper, we discuss a case study of boosting one existing evolutionary data mining system: Global Decision Tree (GDT) [6,15]. It enables induction of various variants of decision trees, but here only univariate classification trees are considered. The investigations are focussed on economically reasonable approaches, thus, we restrict the hardware used to a small computing cluster or a single graphics processor unit (GPU) accelerator. Two novel parallel and distributed processing solutions are analyzed. The first solution applies general-purpose computation on GPUs (GPGPU) and it is based on NVIDIA CUDA [21] framework. The second one is devoted to computing clusters and it is based on the Apache Spark [24] engine. The limits and constraints imposed by the studied acceleration techniques are identified.

The rest of the paper is organized as follows. First, two acceleration techniques are briefly introduced and their recent applications are mentioned. Then, the original GDT system is presented. In Sect. 3 two considered extensions are detailed. Experimental results are described and discussed in Sect. 4. The paper is concluded and possible future works are sketched in the last section.

1.1 GPU, CUDA

GPUs of modern graphics cards are equipped with hundreds/thousands of small, energy-efficient computing units (GPU cores). Each GPU core, though smaller, simpler and slower than a CPU core, is tuned to be especially efficient at the basic mathematical operations. This simplicity allows many more GPU cores to be crammed into a single chip. Moreover, current GPU architectures are approaching terabytes per second memory bandwidth that, coupled with many computational units, creates an ideal device for handling multiple tasks in parallel and managing workloads efficiently. Thus, not only graphics applications but also GPGPU have gained in popularity [23].

Compute Unified Device Architecture (CUDA) [21] is a programming interface and parallel platform that has revolutionized GPGPU. Although there are some alternatives (like OpenCL), CUDA is the most widespread platform. In CUDA, a GPU is considered as a co-processor to a CPU. It means that a part of CPU's tasks can be delegated to the GPU and be processed by thousands of threads in parallel concurrently to the CPU operations. From a programming perspective, CPU calls a kernel that is a function run on the GPU. Then, many threads are created to run the function. The threads are hierarchically

grouped into thread blocks, which are in turn arranged on a grid. The CUDA GPU memory also has a hierarchical structure [21].

GPGPU is recently widely applied in many computational intelligence methods [4,9,11]. Application of GPUs in evolutionary data mining usually focuses on boosting the performance of the evolutionary process which is relatively slow due to high computational complexity, especially for the large-scale data [5,12].

1.2 Apache Spark

Apache Spark [24] is an open-source distributed computing engine for large-scale data processing and one of the most widely used tools in the ever-growing Big Data ecosystem. Spark architecture is based on a concept of Resilient Distributed Dataset (RDD) - an immutable distributed data structure that provides fault tolerance and can be processed in parallel using high-level APIs.

The main advantages of Spark are its in-memory computing capabilities for iterative algorithms and interactive data exploration. Spark processes data in distributed shared memory model, preferably in the RAM of the cluster nodes. Furthermore, Spark offers a much broader set of high-level functional-style data operators that simplify the implementation of distributed applications.

One of the earliest applications of Spark to evolutionary algorithms was proposed by Deng et al. [7], where the population in a differential evolution method is treated as an RDD and only the fitness evaluation is distributed to workers. Teijeiro et al. [22] also described a parallel differential evolution, focusing on individual's mutation, in both master-slave and island models. As for evolutionary data mining approaches using Spark, fuzzy rule-based classifiers were proposed in [8,18]. In [10] the authors tried to scale a genetic programming solution for symbolic regression and proposed a fitness evaluation service based on Spark.

2 Global Decision Tree System

The GDT system [6,15] enables induction of several types of decision trees, depending among others on the type of a predictive task to be solved (classification or regression), the permitted test types in nodes (univariate or oblique) and the prediction types in leaves (single value or model), etc. All variants of the algorithm share the same typical evolutionary process [16] with an unstructured, fixed size population (default population size: 64 individuals) and a generational selection (ranking linear selection and elitist strategy are applied). In this study, to facilitate understanding and to eliminate less important details, the authors focus only on the simplest classification binary trees with tests based on continuous-valued features and without missing data.

2.1 Representation, Initialisation and Termination

Tree-based representation is well-known in genetic programming, where first attempts at evolving a decision tree were presented by Koza [14]. Following these

ideas, in the GDT system, decision trees are not specially encoded and they are processed in their actual form. In non-terminal nodes, typical inequality tests with two outcomes are used, but only precalculated candidate thresholds are considered as potential splits.[1]

An initial individual is created by applying a simple top-down algorithm to randomly chosen small sub-samples of the original training data (default: 10% of the training dataset, but not more than 500 objects), which provides a high degree of heterogeneity of the initial population and is not computationally complex. Among objects located in the considered node, two objects from different classes (so-called *mixed dipole*) are randomly chosen. An effective test that separates these two objects into subtrees is randomly created, taking into account only attributes with different feature values. The recursive partitioning is finished when all training objects in a node are characterized by the same class or the number of objects in a node is lower than the predefined value (default value: 5) or the maximum tree depth is reached (default value: 10). Finally, the resulting tree is post-pruned based on the fitness function.

Evolution terminates when the fitness of the best individual in the population does not improve during the fixed number of generations (default: 1 000) or the maximum number of generations is reached (default value: 1 000).

2.2 Genetic Operators

In the GDT system, there are two specialized genetic operators corresponding to classical mutation applied to a single individual (default probability: 0.8), or to crossover that recombines two individuals (default probability: 0.2).

A mutation operator begins by randomly choosing the node type (equal probability of selecting a leaf node or an internal node), but if the mutation of one type is not possible, the other type is chosen. The ranked list of nodes of the selected type is created, and a mechanism analogous to the ranking linear selection is applied to decide which node will be affected. In case of internal nodes, the ranking takes into account both location (level) of the node in the tree and the reclassification accuracy of each node, whereas for leaves only the second factor is considered. It should be noticed that a modification of a test in a root node affects the whole tree and can have a large impact. On the other hand, mutating an internal node in the lower parts of the tree has only a local impact. As a result, nodes on higher levels of the tree are mutated with lower probability and among nodes on the same level, the reclassification quality is used to sort them. Less accurate leaves are mutated with higher probability and homogenous leaves (all training instances from the same class) are not mutated at all.

There are a few possible mutation variants, which can be performed on internal nodes:

– a test can be modified by shifting a threshold value;

[1] A candidate threshold for the given attribute is defined as the midpoint between such a successive pair of objects in the sequence sorted by the increasing value of the attribute, in which the objects are characterized by different classes.

- a test can be replaced by another test existing in a tree or by a new one. New tests can be created based on randomly chosen dipoles (like in initial population) or locally searched according to some optimality criteria (this can be called memetic extension);
- one subtree can be replaced by another subtree from that node;
- a node can be pruned into a leaf.

Considering a mutation of a leaf node, the range of variants is more modest: a leaf can be just transformed into a subtree.

A crossover operator begins by randomly selecting two trees (and nodes in each of them) that will be affected. There are a few variants of recombination:

- exchange subtrees, branches or only tests associated with nodes (if possible); such an exchange can be purely random or can use a mixed dipole as a guide;
- transfer subtrees asymmetrically where the subtree of the first/second individual is replaced by a new one that was duplicated from the second/first individual. The replaced subtree starts in the node denoted as a receiver, and the duplicated subtree starts in the node denoted as a donor. It is worth to note that different preferences could be used for choosing donor and receiver sites: the receiver node should have a high classification error because it is replaced by the donor node that should have a small value of classification error as it is duplicated.

For both genetic operators each time a choice of operator variant is random, but only valid variants are considered. The default probability distribution of variants is uniform.

2.3 Fitness Function

In most of the data mining system, the first and the most important objective is to find the predictor with the highest classification quality. The main problem with such an objective is that there is no possibility to measure classifier performance in advance. We could only estimate the quality on a given dataset and typically one can only estimate the classifier performance on the training dataset. However, it is well known, that due to the over-fitting problem, a classifier which perfectly reclassifies the training dataset usually performs much worse on unseen objects. The second objective, which is often indicated, is devoted to the classifier simplicity and it can be expressed by the number of nodes. And hopefully, putting emphasis also on a classifier simplicity could be a good way to prevent the over-fitting.

In the GDT system, many forms of the single-objective or multi-objective fitness function are available. As, in this paper, only univariate classification trees are considered, the simplest weighted form of the fitness function is used:

$$Fitness(T) = Accuracy(T) - \alpha * Size(T), \tag{1}$$

where $Accuracy(T)$ represents the classification quality of the tree T estimated on the training dataset, $Size(T)$ is the number of nodes in T and α is the user-supplied parameter (default value: 0.001). The second part of the equation works

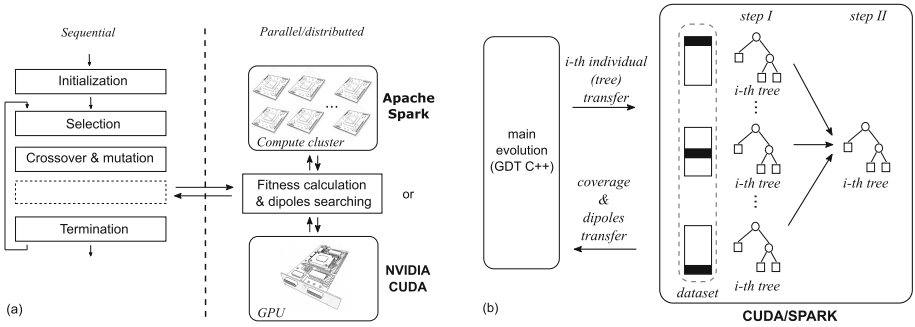

Fig. 1. Evolutionary induction accelerations: (a) general idea, (b) more details concerning processesing an individual in parallel by blocks/workers (in case of CUDA/SPARK acceleration) (step I) as well as reducing/merging results (step II).

as a penalty term and helps to mitigate the over-fitting problem. A similar solution can be identified in the well-known cost-complexity pruning from the CART system [3]. It should be at least mentioned that the value of the α parameter can be usually tuned up for a given dataset, especially if the perfect reclassification cannot be expected, but the parameter tuning is far outside the paper's scope.

3 Boosted GDT Versions

The general idea of GDT accelerations is illustrated in Fig. 1(a). The following operations: initialization of the population as well as selection of the individuals remain unchanged compared to original GDT system. The reason why these initial steps are not accelerated is that the initial population is created only once on a small fraction of the dataset. In the evolutionary loop, also other relatively fast operations like genetic operations (without individual evaluation) are run in a sequential manner. After successful application of crossover or mutation, there is a need to evaluate the individuals. It is the most time-consuming operation since all objects in the training dataset need to be passed through the tree starting from the root node to an appropriate leaf. Thus, this operation is isolated and accelerated by one of the two solutions: CUDA- or Spark-based.

3.1 CUDA Based Acceleration

A GPU-based solution begins by sending the whole dataset to the GPU [12]. This CPU to GPU transfer is done only once and the data is saved in the allocated space in the global memory. Thus, all objects of the dataset are accessible for all threads at any time.

The CPU controls the evolutionary induction. The GPU is called to perform calculations when there is a need to evaluate an individual after successful crossover and/or mutation. At first, the affected individual is sent to the GPU

(Fig. 1(b)). Then, the CPU asks the GPU to take on some of its work. Two kernel functions are called. The first kernel is called to propagate objects from the tree root to the leaves. Next, the second kernel function merges information about the objects' location in the leaves, calculates class distributions and classification errors and finally propagates them from the leaves toward the tree root. The obtained tree statistics (like coverage, errors) as well as dipoles are sent back to the CPU that uses them to update the affected individual.

The first kernel function uses the data decomposition strategy (step I in Fig. 1(b)). At first, the whole dataset is spread into smaller parts that are processed by different GPU blocks. Next, in each block, the assigned objects are further spread over the threads. Each GPU block makes a copy of the evaluated individual that is loaded into the shared memory. This way the threads process the same individual in parallel but handle different chunks of the data.

At the end of the first kernel function, in each tree leaf the number of the objects of each class that reach that particular leaf is stored. However, these values are spread over GPU blocks. Thus, the second kernel function is called (step II in Fig. 1(b)). It merges information from multiple copies of the individual allocated in each GPU block. This operation sums the counters from copies of the individual, and the total number of objects of each class in each tree leaf is obtained. Finally, in the second kernel function reclassification errors in each leaf are calculated. Then, all gathered information: class distribution and errors are propagated from the leaves towards the root node.

To improve the algorithm's performance, the CPU does not have a direct access to the objects that fall into particular nodes of the tree. The propagation of the instances is performed only on the GPU (in contrast to the sequential version). However, some variants of the mutation operator require (object) dipoles to construct a new test in an internal node. This is why the GPU also provides the CPU with two objects of each class in each tree node. In the first kernel, such objects are randomly selected from the objects that reach particular leaves. In the second kernel function, when the multiple copies of the tree are merged, among the available objects again two objected are randomly selected. The CPU using these two objects (of each class in each tree nodes provided by the GPU) can quickly and easily constitute the desired dipoles.

3.2 Spark Based Acceleration

The proposed Apache Spark-based acceleration relies on the distribution of the dataset over a Spark cluster and parallelization of its processing, while the rest of the evolution is unaffected in principle and is realized sequentially.

The Spark-based approach uses a multi-process architecture: while the original GDT system is a native C++ application, Spark is written in the Scala language and its processes run on the Java Virtual Machine (JVM). The Spark processes consist of a single Driver that dispatches the work to a multiple Workers that run on the cluster worker nodes. Both the Spark Driver and GDT applications are running on the same machine and utilise named pipes mechanism for inter-process communication. As a result, the core evolution is performed in

GDT process (C++), whereas the distributed object propagation and dipoles searching procedures, re-implemented in Java, are realized by Spark (JVM).

The proposed approach is based on an implementation described in [19], which was modified and optimized to accommodate big datasets processing. The method starts with the loading of the training dataset, which is processed line-by-line and transformed into an RDD of objects representing packages of observations (1 000 obs. in a package by default). This "packing" operation is highly beneficial from the memory usage standpoint, as it reduces the number of objects in RDD for the given dataset, minimizing the overhead of RDD data structures. Next, the observation RDD is split into a number of partitions that are then cached in the cluster memory.

To prevent data skew, the dataset should be split into partitions of even size and uniformly distributed over the nodes. In our solution, each observation package is randomly assigned to a group with a numeric ID, where the group number equals the number of partitions. The partitions number depends on the data size, with the usual range of 1 to 4 partitions per single worker CPU core.

During the evolution, the GDT process sends a request with a single tree data to the Spark Driver. During the induction, all observations are passed through the transferred decision tree and distributions of classes and dipoles in its leaves are obtained. The parallel processing is realized by typical pair of `map-reduce` operations evoked on the grouped RDD (see Fig. 1(b)). Each dataset partition group emits a locally processed copy of the tree (`map(group)` $\rightarrow$ `tree`) and the local trees are then reduced into a final result (`reduce(tree1, tree2)` $\rightarrow$ `tree3`). During the reduction, the class distributions are simply merged, while the dipoles are reduced implicitly by selecting the dipoles from one of the trees. Finally, the error calculations and propagation of classes and dipoles in the final tree are performed and the results are sent back to the GDT process, where the overall accuracy is estimated. The process ends when the last tree is processed.

4 Experiments

Experimental validation was performed on an artificially generated dataset called *chess* with two 2 real-values attributes and objects arranged on a 3 $\times$ 3 chessboard (Fig. 2). It is a dataset for which moderate sized decision trees are induced. We used the synthetic dataset to scale it freely, unlike real-life datasets. We examined various numbers of objects, from hundreds of thousands to a few billions. All presented results correspond to averages of 5–10 runs and were obtained with a default set of parameters from the sequential version of the GDT system [6]. As we are focused in this paper only on size and time performance of the GDT system, results for the classification accuracy are not included. However, for the tested dataset, the GDT system managed to induce trees with optimal structures and accuracy about 99% [15].

GPU experiments were performed on a workstation equipped with Intel Xeon E5-2620 v4 (20 MB Cache, 2.10 GHz), 256 GB RAM, and running Ubuntu 16.04. The sequential algorithm was implemented in C++ and compiled with gcc 5.4.0.

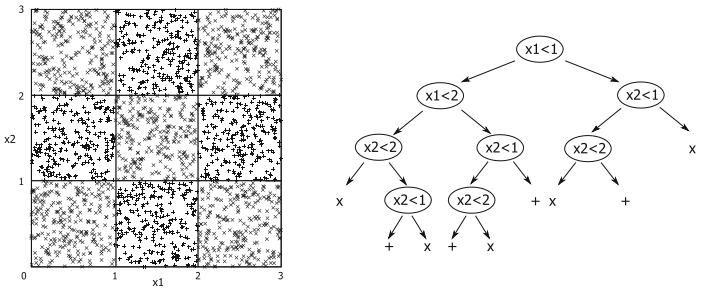

Fig. 2. Examples of analyzed *chess3x3* dataset variant and the corresponding ideal structure classification tree.

The GPU-parallelization was implemented in CUDA-C and compiled by nvcc CUDA 7.5/8.0 [1] (single-precision arithmetic). We tested two NVIDIA GPU cards: *(i)* GeForce GTX 780 (3 GB memory, 2 304 cores) and *(ii)* Pascal P100 (12 GB memory, 3 584 cores). The first GPU card is the consumer line GeForce GPU, while the second one is the professional-level GPU accelerator that currently costs about 5 000 $ (almost 10 times more than the first one).

Apache Spark (version 2.2.0) was deployed on a cluster of 18 workstations with a quad-core Intel Xeon E3-1270 3.4 GHz CPU, 16 GB RAM, running Ubuntu 16.04 and connected by a Gigabit Ethernet network. 16 worker nodes were used by Spark executors, one node was dedicated to Spark Master and HDFS NameNode (Hadoop 2.7.3) and the last node was running Spark Driver and GDT C++ processes. The experiments were performed on 4, 8 and 16 workers, which corresponds, respectively, to 16, 32 and 64 CPU cores in total.

We are interested in estimating the size of the dataset which can be processed on the given platform/hardware in fixed amount of time: 1 min, 1 h and 1 day (Tables 1 and 2). Table 1 concerns the GPU-accelerated solution. We see that both GPU cards provide a significant boost in training dataset processing. Pascal P100 is able to handle 1 million of objects in 1 min. In 1 h, the size of the processed dataset increases to nearly 100 millions of objects.

Comparing GPUs, we see that a cheaper one (GTX 780) gives about twice worse results than Pascal P100. However, this performance difference decreases with the increase of the dataset size. In all cases, for the maximum datasets that can be stored in each GPU memory, the induction does not last longer than 1 day. The time of processing those maximum datasets is included in Table 3.

The scale of the performance improvement is more visible when comparing the sequential and GPU-accelerated versions of the GDT system. For example, as regards 1 h, results show that the GPU-supported version is able to handle the dataset greater by two orders of magnitude (200 000 objects by the sequential version vs 84 000 000 objects with a support of Pascal P100). Comparing results of the GPU-based acceleration and OpenMP parallelization using eight CPU cores, similarly, we see that the first solution wins.

Table 1. The maximum size of the *chess3x3* dataset variant which can be completely processed in the given period of time by GPU-accelerated GDT system. In addition, results for a sequential CPU version as well as an OpenMP parallelization using eight CPU cores are provided.

GPU card/period	1 min	1 h	1 day
GTX 780	530 000	57 000 000	256 000 000*
Pascal P100	966 000	84 000 000	1 033 000 000*
Sequential CPU	1 200	275 000	3 500 000
OpenMP (8 CPU cores)	45 000	970 000	14 000 000

*The processing time of the maximum datasets (that can be stored in each GPU memory) was shorter than 1 day, see Table 3.

Table 2. The maximum size of the *chess3x3* dataset variant which can be completely processed in the given period of time by Spark-accelerated GDT system.

Period number of workers	1 h	1 day
4	13 000 000	500 000 000
8	20 000 000	1 250 000 000
16	35 000 000	2 500 000 000

Table 3. The maximum size of the *chess3x3* dataset which can be completely processed by the given platform/hardware. Processing time is also included.

Solution		Dataset size	Time
GPU	GTX 780	256 000 000	5 h
	Pascal P100	1 033 000 000	17 h
Spark	4 workers	900 000 000	33 h
	8 workers	1 850 000 000	35.5 h
	16 workers	3 900 000 000	38 h

Concerning the Spark-based acceleration, we see that Spark deployed on all 16 nodes can process a dataset of 35 millions objects in 1 h period (Table 2). It is less than the best Pascal P100 GPU result of 84 millions, but within the same order of magnitude. Furthermore, the 1 min execution time cannot be achieved due to the framework and networking overhead. The algorithm usually processes about 80 thousand trees during its execution and the overhead of Spark can be from 6 to 8 milliseconds for every tree in smaller datasets processed in 1 h. This gives about 9 min of total overhead for the entire algorithm. In the 24 h period, however, the datasets are significantly bigger and the framework impact is not as noticeable. In the end, Spark can process 2.5 billion observations in one day, outperforming the GPUs due to their memory limitations.

We are also interested in verifying how big datasets are able to be processed in the given platform/hardware, taking into account available memory restrictions (see Table 3). Concerning GPU cards, the dataset size is strictly limited by their global memory sizes. Since GTX 780 is equipped with 4 times less memory than Pascal P100, it is able to process about 4 times smaller datasets. Pascal P100 handles over than 1 billion objects in less than 1 day. Because of very long computation time, we did not even try to process these datasets either by the

sequential version or OpenMP-based parallelization. Each year, NVIDIA releases new GPUs with faster and larger memory, thus, it is only a matter of short time when GPU cards with 24, 48, etc. GB of memory appear.

Spark, given its Big Data processing capabilities, excels with bigger datasets. The maximum dataset processed on the entire cluster has 3.9 billions objects (see Table 2), which corresponds to 78 GB of raw text data and 96 GB in memory-cached RDD. The results show datasets that can be processed on the cluster in stable and efficient manner, without swapping, excessive JVM garbage collection and with the dataset fully cached in the memory. Technically it is possible to configure the RDD to "spill" into disk storage if it does not fit completely in the memory, but this mode of operation results in drastic performance degradation (reading from memory vs reading from disk). Objects in RDD can also be serialized for space efficiency, but this process also comes with a performance penalty due to necessary deserialization. We observed over ten-fold slowdown with serialized objects, but achieved up to 3:1 compression ratio.

The main advantage of Spark is its capability to easily scale with the size of the cluster. The results in Table 2 show that the size of the dataset can basically double with twice the number of nodes. The same application can also run without any modification on a cluster with potentially thousands of nodes.

5 Conclusions

In this paper, we investigate the applicability of evolutionary induction of decision trees for large-scale data. We show that boosted solutions are able to process really large-scale data, even up to billions of objects. It is clear that the CUDA-based acceleration is generally faster but limited by the size of the GPU memory. On the other hand, the Spark-based solution is preferable if a dataset becomes huge, in our case exceeds one billion of objects. Moreover, an unmodified Spark solution can be easily scaled up just by adding more hardware to the cluster.

In this work, we focus on the dataset dimension expressed as a number of objects. In future works, we would also like to investigate the influence of the number of features. This could be especially interesting for genomic data where the number of features is often large. We also plan to extend the GPU-based solution into a framework where one can easily add more GPUs to distribute dataset over them and push the data size limit.

Acknowledgments. This work was supported by the grant S/WI/2/18 from BUT founded by Polish Ministry of Science and Higher Education.

References

1. NVIDIA Developer Zone - CUDA Toolkit Documentation (2018). https://docs.nvidia.com/cuda/cuda-c-programming-guide/
2. Barros, R.C., Basgalupp, M.P., De Carvalho, A.C., Freitas, A.A.: A survey of evolutionary algorithms for decision-tree induction. IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.) **42**(3), 291–312 (2012)

3. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and Regression Trees. CRC Press, Boca Raton (1984)
4. Cano, A.: A survey on graphic processing unit computing for large-scale data mining. WIREs: Data Min. Knowl. Discov. **8**(1), e1232 (2018)
5. Chitty, D.: Improving the performance of GPU-based genetic programming through exploitation of on-chip memory. Soft Comput. **20**(2), 661–680 (2016)
6. Czajkowski, M., Kretowski, M.: Evolutionary induction of global model trees with specialized operators and memetic extensions. Inf. Sci. **288**, 153–173 (2014)
7. Deng, C., Tan, X., Dong, X., Tan, Y.: A parallel version of differential evolution based on resilient distributed datasets model. In: Gong, M., Pan, L., Song, T., Tang, K., Zhang, X. (eds.) BIC-TA 2015. CCIS, vol. 562, pp. 84–93. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49014-3_8
8. Ferranti, A., Marcelloni, F., Segatori, A., Antonelli, M., Ducange, P.: A distributed approach to multi-objective evolutionary generation of fuzzy rule-based classifiers from big data. Inf. Sci. **415–416**, 319–340 (2017)
9. Fonseca, A., Cabral, B.: Prototyping a GPGPU neural network for deep-learning big data analysis. Big Data Res. **8**, 50–56 (2017)
10. Funika, W., Koperek, P.: Towards a scalable distributed fitness evaluation service. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 493–502. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_46
11. Jinjing, L., Qingkui, C., Bocheng, L.: Classification and disease probability prediction via machine learning programming based on multi-gpu cluster mapreduce system. J. Supercomput. **73**(5), 1782–1809 (2017)
12. Jurczuk, K., Czajkowski, M., Kretowski, M.: Evolutionary induction of a decision tree for large-scale data: a GPU-based approach. Soft Comput. **21**(24), 7363–7379 (2017)
13. Kotsiantis, S.B.: Decision trees: a recent overview. Artif. Intell. Rev. **39**(4), 261–283 (2013)
14. Koza, J.R.: Concept formation and decision tree induction using the genetic programming paradigm. In: Schwefel, H.-P., Männer, R. (eds.) PPSN 1990. LNCS, vol. 496, pp. 124–128. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0029742
15. Kretowski, M., Grzes, M.: Evolutionary induction of mixed decision trees. Int. J. Data Warehous. Min. (IJDWM) **3**(4), 68–82 (2007)
16. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer, Heidelberg (1996). https://doi.org/10.1007/978-3-662-03315-9
17. Murthy, S.K.: Automatic construction of decision trees from data: a multi-disciplinary survey. Data Min. Knowl. Discov. **2**(4), 345–389 (1998)
18. Pulgar-Rubio, F.J., Rivera-Rivas, A.J., Pérez-Godoy, M.D., González, P., Carmona, C.J., del Jesus, M.J.: MEFASD-BD: multi-objective evolutionary fuzzy algorithm for subgroup discovery in big data environments - a MapReduce solutioon. Knowl.-Based Syst. **117**, 70–78 (2017)
19. Reska, D., Jurczuk, K., Kretowski, M.: Evolutionary induction of classification trees on spark. In: Rutkowski, L., Scherer, R., Korytkowski, M., Pedrycz, W., Tadeusiewicz, R., Zurada, J.M. (eds.) ICAISC 2018. LNCS (LNAI), vol. 10841, pp. 514–523. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91253-0_48
20. Rokach, L., Maimon, O.: Top-down induction of decision trees classifiers-a survey. IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.) **35**(4), 476–487 (2005)
21. Storti, D., Yurtoglu, M.: CUDA for Engineers : An Introduction to High-Performance Parallel Computing. Addison-Wesley, New York (2016)

22. Teijeiro, D., Pardo, X.C., González, P., Banga, J.R., Doallo, R.: Implementing parallel differential evolution on spark. In: Squillero, G., Burelli, P. (eds.) EvoApplications 2016. LNCS, vol. 9598, pp. 75–90. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31153-1_6
23. Yuen, D., Wang, L., Chi, X., Johnsson, L., Ge, W., Shi, Y.: GPU Solutions to Multi-scale Problems in Science and Engineering. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-16405-7
24. Zaharia, M.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)