Computer Science • 17 (1) 2016 http://dx.doi.org/10.7494/csci.2016.17.1.69

Włodzimierz Funika
Paweł Koperek

SCALING EVOLUTIONARY PROGRAMMING WITH THE USE OF APACHE SPARK

Abstract *Organizations across the globe gather more and more data, encouraged by easy-to-use and cheap cloud storage services. Large datasets require new approaches to analysis and processing, which include methods based on machine learning. In particular,* symbolic regression *can provide many useful insights. Unfortunately, due to high resource requirements, use of this method for large-scale dataset analysis might be unfeasible. In this paper, we analyze a bottleneck in the open-source implementation of this method we call* hubert*. We identify that the evaluation of individuals is the most costly operation. As a solution to this problem, we propose a new evaluation service based on the Apache Spark framework, which attempts to speed up computations by executing them in a distributed manner on a cluster of machines. We analyze the performance of the service by comparing the evaluation execution time of a number of samples with the use of both implementations. Finally, we draw conclusions and outline plans for further research.*

Keywords distributed systems, evolutionary programming, symbolic regression, scaling, Apache Spark

Citation Computer Science 17 (1) 2016: 69–82

1. Introduction

Nowadays, many organizations around the world gather more and more data. Nearly unlimited storage and computing resources delivered as easily accessible cloud services only encourage the collection of all available data. Unfortunately, mining these ever-growing datasets is a very challenging task. It requires excellent domain knowledge and understanding which data is being analyzed. Furthermore, scalable algorithms and technologies need to be used to provide results in a reasonable amount of time. The problem gets more complicated when real-time analysis needs to be taken into account.

Until recently, such types of work were primarily limited to the human domain due to their complexity. Today, as computer techniques become more and more advanced, automated analysis continues to gain more and more attention. According to [4], computers are now used at multiple stages of the research process, from gathering knowledge about related work and similar experiments through automatic data analysis [17] and up to complete automatic systems capable of creating and verifying new hypotheses on their own [12]. Completely autonomous data mining isn't yet possible; however, there are many ways to help scientists and business intelligence specialists in their daily jobs.

Eureqa [16] and hubert [8, 11] are examples of a new kind of tool that is designed to help identify meaningful relationships in the available data. They both use a symbolic regression method to automatically search for relationships between variables. The results of their analyses is provided as a mathematical formula describing the discovered connections. Symbolic regression is based on evolutionary programming; it attempts to solve problems defined by the user through generating and improving a population of possible solutions. At the beginning, the population consists of randomly generated individuals. They are all evaluated with a defined fitness function. The best units are recombined and mutated in order to create better solutions in the next iterations. The whole process is repeated until an individual that meets a required fitness level is found. Unfortunately, implementing such an approach immediately imposes limits on the amount of analyzed information. Evaluation usually comprises of applying a solution to the whole data set, which requires reading it from storage. Because of the very high number of evaluations, the time required to obtain a good-enough solution is unacceptably high in the case of large, multimillion-data series.

The problem of scalability can be addressed with the use of Big Data technologies: Apache Hadoop MapReduce [6] or Apache Spark [20]. Both tools were designed to enable efficient distributed processing of large datasets. Their major advantage is the ability to horizontally scale to a large numbers of nodes; there are successful deployments of production clusters with thousands of nodes and petabytes of storage ([2, 14]). In such environments, hardware failures are very common. To provide reliable results, the frameworks under discussion have built-in mechanisms of graceful failure handling. Although Hadoop MapReduce and Spark have a lot in common, they are significantly different in regards to the basic concepts on which they are based.

The former provides an abstraction for a two-step processing algorithm. Each execution of a MapReduce job is independent. If the designer wishes to pass information between those executions, the output needs to be persisted (e.g., in HDFS). On the other hand, Apache Spark exploits in-memory processing techniques to speed up processing. This enables the implementation of iterative algorithms, applying different computations to the same data. It is also possible to create tools that allow for the execution of low-latency queries against large datasets.

In this paper, we outline an implementation of fitness function evaluation based on the Apache Spark framework. First, we discuss the technical details of some related tools and frameworks as well as the symbolic regression itself. Later, we describe the architecture of the complete system and discuss a comparison of sample dataset processing with the use of Apache Spark and hubert. Finally, we draw conclusions from the conducted experiments and discuss further research directions.

2. Background and related work

In this section, we present information about the tools and concepts used in our research.

2.1. Symbolic regression

Polynomial regression [13] aims to obtain a function that describes a finite set of data points based on changing numerical coefficients of a polynomial. In other words, it describes a principle ruling the observed system (which explains the observed behavior). To use this method, the researcher needs to decide whether to use a linear, quadratic, or higher-order polynomial form of a fitted mathematical expression. Unfortunately, this is not an easy task. If the degree of polynomial is too low, it won't be able to fit into the given input; if it is too high – it will fit the data set perfectly but will be useless beyond it.

Symbolic regression [13] is a method that attempts to solve this problem. It focuses on identifying a mathematical expression (in its symbolic form) that would be a very good fit within a given data set. The parameter space and functional form of equations are being searched at the same time. This method relies on genetic programming. At first, a set of individuals (mathematical expressions) is randomly generated. Each expression is built from specified primitive elements such as algebraic operations ($+$, $-$, $*$, $/$), variables ($x, y, \dots$), constants ($3.1415, 2.71\dots$), etc. Although initially they don't fit the input at all, they gradually improve with an evolutionary process.

Owing to such a general definition, this method can be applied to solve a magnitude of different problems. Unfortunately, achieving meaningful results is challenging. The key issue lies in choosing the proper learning parameters, problem description, and (most importantly) the correct cost function. Very good results can be obtained with the one proposed in [16], which forces the algorithm to discover *implicit relation-*

ships. An implicit relationship is a function of form $f(x, y) = 0$ whereas the explicit function is represented as $y = f(x)$.

2.2. Parallel implementations of evolutionary algorithms

Improving the processing time of evolutionary algorithms (and genetic programming in particular) receives much attention from researchers. The major work in this area includes creating implementations utilizing parallel computing platforms: PVM [5], MPI [15], or MapReduce [3]. These tools focus on parallelizing all steps of the algorithm at once. The solution space is divided to many small populations, and all steps of an algorithm (population generation, fitness evaluation, mutation, and crossing-over) for a particular set of individuals are processed by a separate physical CPU. This so-called *island* model can be tuned in various ways. The first, straight-forward approach is to start the evolution with different parameters or starting conditions for each population. This broadens the searched solution space, but it might lead to creating many local solutions (*niching*). Usually obtaining a single, best global individual is preferred. In such a case, migrating specimens between populations can be conducted as an additional executed step before starting a new evolution iteration. Evolutionary computations can be parallelized with a focus on specific steps of the algorithm (e.g., parallelization of individual evaluation or mutation only). Another approach is to reduce computation time by limiting evaluation with the size of the used data (e.g., by splitting the original dataset between populations).

2.3. Apache Hadoop MapReduce

Apache Hadoop MapReduce was designed to provide a means of processing the vast amounts of data in an efficient way. It aims at delivering systems whose performance can scale linearly with the number of physical machines added. It applies a divide-and-conquer technique, splitting the data located on a distributed filesystem between CPUs. This splitting process takes into account information about which machine contains which data subset, so only the nodes actually storing the relevant subsets will be used. Finally, the job definition and JAR file with the executable code are sent to the identified nodes, and computations start.

The MapReduce computing model assumes that the whole process will be split into two phases: map and reduce. The map phase processes raw input, which is split into key-value pairs. As a result, it similarly emits a set of intermediate key-value pairs that can potentially be of a different type. Before reduction, the intermediate output is grouped by the key and sorted. Each reducer processes all data.

2.4. Apache Spark

Apache Spark is a framework for the parallel processing of big data sets in a fault-tolerant manner. It is based on a new concept of distributed-memory abstraction – Resilient Distributed Datasets (RDD). RDDs are motivated by the limitation of current computing frameworks: poor support for iterative algorithms and interactive

data mining tools. They provide a shared memory model that prefers coarse-grained transformations like *map*, *filter*, or *join* instead of fine-grained updates. Such operations can be applied at once to many data items. Fault-tolerance is achieved by logging all transformations used to create a dataset. In case of any error, only the required operations need to be computed again.

RDDs are immutable and can only be created by reading from a data source or as an effect of transformations of an existing dataset. The processing is lazy; actual computations are only triggered by actions that require access to the output. RDDs can be cached in memory or persisted on a hard-drive for further reuse.

Such features make Spark very useful for machine-learning algorithms, as they usually consist of many iterations of similar operations over the same dataset.

2.5. Hubert

Hubert is a result of our prior work in the area of applying symbolic regression to the monitoring of computer systems. It provides an open-source implementation of the ideas described in [16] and [18]. The goal of this project is to discover hidden relationships in the monitoring data streams in order to help gain deeper insight into complex computer systems. Such relationships are described with the use of precise mathematical expressions that are individuals from the genetic algorithm perspective. Each individual is represented as an expression tree built from primitive blocks ($+$, $-$, $\times$, $/$, *sin*, *cos*, variables defined in an input data set, constants). The number of nodes of the tree can be interpreted as a measure of how complicated the particular solution is. We call this parameter *complexity* and use as an indicator of the individual's generality.

As the fitness function, we used the formula proposed in [16]. To evaluate a particular candidate expression f, we compute numerically partial derivatives of a pair of variables x, y – $\frac{dx}{dt}$ and $\frac{dy}{dt}$. Then, we find symbolically partial derivatives of the candidate expression, $\frac{\delta f}{\delta x}$ and $\frac{\delta f}{\delta y}$. To compute the actual fitness value, we combine these elements in a formula 1. Its value can be interpreted as the error rate of a solution. Therefore, the algorithm attempts to minimize it (i.e., individuals with lower values are considered better).

$$\text{Fitness}(f) = \frac{1}{N} \sum_{i=1}^{N} \log \left(1 + \text{abs} \left(\frac{\Delta x_i}{\Delta y_i} - \frac{\delta x_i}{\delta y_i} \right) \right) \tag{1}$$

where: $\frac{\delta x}{\delta y} = \frac{\delta f}{\delta y} \Big/ \frac{\delta f}{\delta x}$ and N is the number of data points.

To tackle the problem of stagnating computations, we used the evolution method proposed in [18]. The population is evolved using the following list of steps:

1. Randomly initialize a population of a given size.
2. Randomly group individuals in pairs of *parent* individuals.
3. Create *children* individuals by crossing-over of created pairs.

4. Conduct mutations on children individuals.
5. Add mutated individuals to the population.
6. Repeat until the population size returns to the initial size:
   (a) select randomly two individuals,
   (b) form an age-fitness Pareto front from these individuals,
   (c) discard dominated individual,
   (d) if there are no more dominated individuals – break the loop.
7. Verify stop criteria – if they are met, return current population, otherwise go to step 2.

It is theoretically possible that the Pareto front is larger than the initial population. In this case, all non-dominated individuals should be stored and used in a following algorithm iteration. This case is handled by a additional test from point 6(d).

The discussed process can be used to analyze any series of numerical data. In hubert's case, we focused on the data coming from the monitoring of computer systems. In this case, the best discovered equations can describe the dependencies between the components and model complex behavior of the system. Such information can be used to improve the architecture or tune the parameters to make it more efficient. The iterative nature of such an analysis makes the model evolve over time, thus keeping it up to date. The amount of time spent on computations can also be easily adjusted by using one of the supported stop criteria:

- *time* – the computations are stopped after a fixed amount of time;
- *target fitness function value* – the computations are stopped once the best solution's fitness value is lower than the target value;
- *number of iterations* – the computations are stopped after a fixed number of iterations.

The tool was written in Java language; thus, it can be easily integrated with other technologies that use Java Virtual Machine. It is an open-source project – we encourage the reader to use and extend it, adapting to your specific needs.

3. Bottleneck analysis

While developing *hubert*, we noticed that the biggest part of the execution time is spent on individual evaluation. According to the cost function, this process requires computing symbolically partial derivative values over the input data set and comparing them with the numerically computed partial derivative values. Both elements involve reading all of the input values. When working with hubert, we noticed that evaluation often takes over 99% of the whole processing time. In case of more complex individuals and more data, this value gets even closer to 100%.

To improve the time of processing, the whole dataset is being preprocessed and kept in memory. Caching includes not only the raw data but also the numerical derivative values. Minimizing the formula 1 means that all of the combination pairs of candidate variables need to be taken into account. This creates $C_n^k = \binom{n}{k}$ data

series, about the same size as the input data set. All of them need to be stored at the same time in memory – they are reused each time an individual evaluation occurs. For a sample dataset of currency quotations containing 4 variables (EUR/USD, EUR/GBP, CHF/PLN, USD/CHF), each holding a 2,5 GB data series (data since 05.2005 till 06.2014), this means that $\binom{4}{2} = \frac{4!}{2!2!} = 6$ series of 2,5 GB each need to be cached. This number rapidly grows with the growth of input dimensions. Unfortunately, this imposes limits on the tool's capabilities – processing of data sets that cannot fit into the memory of a single machine would require reading the input data and recalculating partial numerical derivatives of each iteration. Such an approach would not be fast enough to provide meaningful results in the assumed time window. Such a limitation would render the discussed method unfeasible for use in a dynamically changing environment of software and hardware monitoring.

4. Overview of the evaluation service concept

The evaluation of individuals is the most resource-consuming part of evolutionary algorithms. It requires reading the input data set and evaluating the value of the assessed specimen over all of its data points. Since the same set is used in every iteration multiple times, the best way to speed up the computations is to cache it in memory. Unfortunately, in case of data sets whose sizes exceed the memory of a single machine, this solution is not possible to use. In such a case, the time of evaluation increases dramatically because of heavy I/O usage. This renders the symbolic regression algorithms unfeasible for use on larger amounts of information.

To address this limitation on the input size (both in hubert and in symbolic regression in general), we made use of the Apache Spark framework to implement a new fitness evaluation service. First, the user needs to register datasets. They can be stored in HDFS or on local hard-drives of computing nodes. We prefer using the former solution. In this case, the Spark framework can easily split computations in such a way that each node processes only the part of data that is stored on its local hard-drive. Later, when the client (e.g., hubert) needs to evaluate a specific individual, it sends a query to the service instead of performing the computations itself. The evaluation query contains the individual that is a mathematical expression, information about which dataset to use and specifies which formula should be used for numerical differentiation (forward, backward and central finite differences).

When an evaluation query for a specific individual is sent, the data from the registered datasets is read and processed according to the following steps:

1. Generate all pairs of variables in the dataset.
2. For each pair:
   (a) compute numerically derivatives for selected variables,
   (b) compute symbolic derivatives of the examined individual,
   (c) evaluate symbolic derivatives over the dataset,
   (d) calculate the cost function value.

Implementing the evaluation as a service has several advantages. First, it can be used in tools different from hubert. It also abstracts the Spark API. In case a better processing solution can be applied to speed up computations, such a change won't require any changes on the service users' side. Using a service allows for the handling of multiple requests at the same time, thus improving cluster utilization. Furthermore, it is a starting point for migrating hubert as a whole to a microservices architecture, which would enable further improvements (e.g., using populations with a greater number of individuals or evolving solutions for multiple datasets in parallel). The internal structure of the service is presented in Figure 1. The service is composed of three major elements:

- *Symbolic Differentiation* – performs symbolic differentiation on the passed expression and evaluates it over the dataset.
- *Numeric Differentiation* – numerically differentiates the given dataset.
- *Function Error Evaluator* – compares results of above evaluations and returns the error value according to a chosen error metric.

All of them transparently distribute the required computations over the cluster using the Spark API.

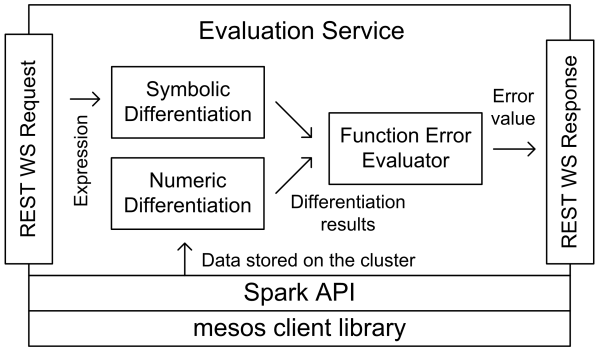

Figure 1. Evaluation service internal structure.

5. Back-end architecture

The whole system is meant to be deployed in a cloud environment or on a dedicated hardware cluster. To deploy Spark and balance resource allocation, we used Apache Mesos [10] – a tool based on the concept of Google's Omega system [19]. Mesos handled automatic deployment of executors, gracefully dismissed the unused ones, and recreated the broken units. Such a facility greatly improved the speed of work and enabled using computational resources only when they were actually necessary.

The architecture of the solution under discussion is shown in Figure 2.

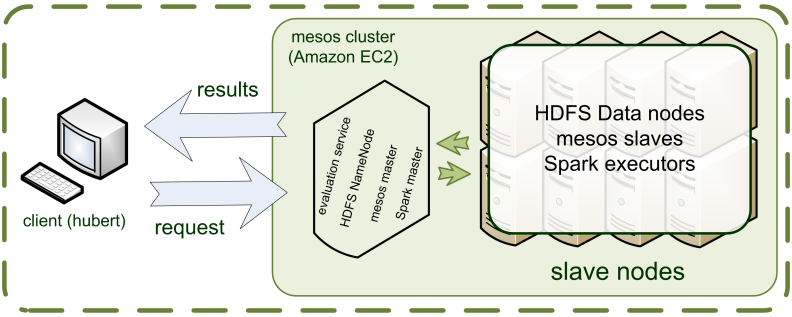

Figure 2. General architecture diagram.

The system comprises a *front-end node*, which contains an instance of the evaluation service and infrastructure services: *mesos-master*, *HDFS Name Node* and *Spark Master*. Back-end nodes are running *mesos-slave*s and *HDFS Data Node*s.

Upon receiving the first request, the evaluation service notifies *mesos-master* about the required resources (CPUs and memory). *mesos-master* and *mesos-slave*s negotiate which machine should execute which task, then download and install the software wherever necessary. Deployment happens only once before actual computations start. When a Spark cluster is ready, the evaluation service submits a new job to *Spark Master*. *Spark Master* splits the job and sends it as a serialized Java bytecode to slaves. Input data is delivered by HDFS services. They maintain the distribution of information across the cluster. Owing to this, CPUs process the data that is actually stored nearest to them – on their hard-drives. The progress is constantly monitored. If an executor fails, it is automatically restarted, and missing computations are rescheduled on other machines. Finally, when all of the stages of processing finish, the evaluation service returns the result. The related RDDs are automatically cached in memory.

The life-cycle of the service begins before the actual evolutionary computations start. It needs to be initialized: mesos executors have to be deployed, and required datasets need to be registered. After a satisfactory solution is found, the service can be shut down. It can also be left waiting for requests if further processing of the same data is planned. The service will retain cached data structures.

6. Conducted experiments

6.1. Experiment characteristics

To evaluate the new implementation and examine whether it processes the data faster than the initial one, we compared the execution time of processing for three sample individuals generated by hubert:

- A: $\sin(x + y)$,
- B: $(x - y + \cos(x) - 4.906 + 5.8 + x - y)/(\cos(4.56575) + \cos(x) + \sin(x) * x/y)$,

- C: $(((x-y)*x*1.0951405*\sin(x*y)*\cos(\cos(y)+1.0951405/3.01411))*(\sin((x-y)+2.377/2.817)*\cos(x)*(x+y)-(\sin(x)-\sin(y))))-(\cos(\cos(x/y)/((x+y)-(x-2.3776817))/\sin((7.305318+x)+(x-y))))$.

We present them in their non-simplified form – as such, they are processed by both tools.

Each specimen was being evaluated against four datasets of different sizes: 1 MB, 10 MB, 100 MB, and 1024 MB. Each of them contained a different time window of financial data series: price quotes for the euro and American dollar currency pair.

The computations were carried out in Amazon Elastic Computing Cloud [1]. The Spark cluster consisted of nine instances of *m1.medium* type virtual machines, each using the following resources:

- 1 VCPU,
- 3,75 GB RAM,
- 410 GB local hard drive storage.

The hubert-bound evaluation was carried out on a single *m1.medium* instance.

To rule out differences in the execution time coming from the dynamic nature of the cloud environment, each run was repeated 8 times. Minimal and maximal results were removed, and an average was computed from the remaining values.

6.2. Results discussion

A computation times comparison is presented in Figure 3.

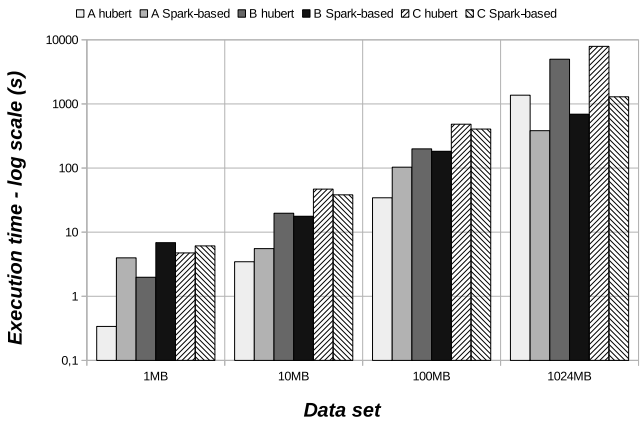

Figure 3. Execution times for single- and multi-processor implementations.

These results show that both algorithm implementations have their own ranges of use. For small inputs, the clear winner is the single processor version. However, when increasing the amount of data and complexity of processing, we notice that it doesn't scale well. In the case of bigger input files, the speedup due to parallelization exceeds the overhead introduced by Spark – the second implementation is significantly

faster. More CPU-intensive processing for specimen B and C only strengthen this effect. The actual speedup depends on multiple factors (e.g., whether splitting the computations into smaller chunks has optimal granularity). Table 1 shows how the complexity of computations influences actual times of execution.

Table 1

Execution times for single- (*hubert*) and multi-processor (*Apache Spark*) implementations for sample expressions of different complexity (dataset: 1024 MB).

Expression	hubert(s)	Apache Spark (s)	Speedup
A	1369	383	3.57
B	4981	693	7.19
C	7907	1298	6.09

The best results in terms of speedup were achieved for expression B. The multi processor-bound processing took 693 seconds (i.e., when compared with the single processor time, 4982 seconds gives us more than a seven-time speedup.

One of the factors that enables such improvements is Spark's ability to cache data in memory. RDDs related to current computations are being automatically cached whenever possible. Thanks to this, the speedup can be also observed when executing subsequent evaluations over the same dataset. The execution times observed in such a scenario are depicted in Figure 4.

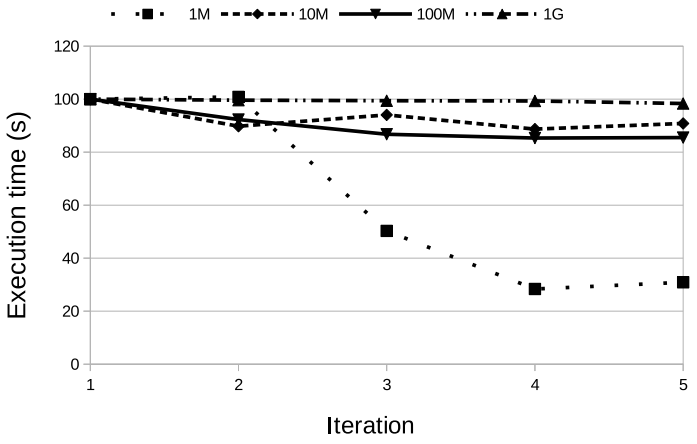

Figure 4. Comparison of iteration execution time. Time of computations is shorter due to caching of partial results in memory.

Although the way the execution time changes is different for each dataset size, it is clearly visible that later requests are processed faster.

7. Conclusions and further work

Symbolic regression is a useful analysis method. It can be used to create mathematical models based on a numerical data series. In particular, such a model can be used to describe the behavior of a computer system in a given time range. Unfortunately, the amount of data acquired by monitoring facilities is large in many cases. To make the idea of evolutionary computations feasible for use in this scenario, we demonstrated how to speed them up by applying new concepts from the distributed computing area.

In this paper, we presented a new implementation of the fitness evaluation service that improves processing time for large datasets. We examined the available technologies and explained why we chose the Resilient Distributed Datasets concept as a basis for our work. Further, we discussed the architecture and behavior of the system. The comparison between single-processor and multi-processor implementation showed that the first version cannot be replaced in all cases. It is still the best choice for small amounts of data; however, the bigger the datasets, the more benefits parallelization gives.

The results obtained show that Apache Spark is a viable solution to execute machine-learning algorithms. It significantly sped up the fitness analysis of big datasets and allowed for the processing of data sets that did not fit into a single machine's memory.

We had an opportunity to observe how fault-tolerance mechanisms handle failures occurring during processing. When *Spark Master* noticed that a virtual machine was not responding, it automatically rescheduled the tasks that were running on that node. Similarly, in case a part of the RDDs was purged from the cache, it got recomputed at the time it was needed again.

The RDD memory model was simple to use. Unfortunately, we noticed that not all types of computation can be easily represented in such a way. If the algorithm requires the combination of subsequent values of data series, it is necessary to copy the whole input, change the data indices, and perform a costly *join* operation over the original and new RDD. Spark applications can be easily tested in a single-machine configuration. We encourage the reader to use this facility to assess the usefulness of the framework before deciding whether to use it in production. Otherwise, the limitations of that model might induce an unnecessary overhead that will outweigh the other benefits.

hubert proved that it is a robust and flexible implementation of symbolic regression. The tool's architecture is also very flexible. We were able to easily refactor the code to a form of service. According to the results, the single processor implementation of fitness evaluation is still very useful. The use of a distributed computing framework in the case of small datasets induces too much overhead. Instead of completely migrating to a new evaluation implementation, we plan to allow switching between both of them.

The work on the open-source implementation of symbolic regression is ongoing. We plan to further optimize the execution time of the evaluation service. We believe

that enabling low-latency several-second responses is possible. Furthermore, we see another scaling opportunity in migrating hubert's architecture from the monolithic code base to a microservices architecture, which enables independent scaling of different parts of the system and potentially hosting them on separate clusters. From the functional point of view, we aim to combine the ideas developed in hubert with a semantic-oriented approach [7, 9].

Acknowledgements

We would like to thank Dr. Maciej Malawski for his valuable help with Amazon EC2 experiments. This research is supported by AGH grant no. 11.11.230.124 as well as by the EU ICT-269978 VPH-Share project.

References

[1] Amazon.com, Inc.: AWS Amazon Elastic Compute Cloud (EC2) – Scalable Cloud Hosting. `http://aws.amazon.com/ec2`, 2014, accessed 2.12.2014.

[2] Apache Software Foundation: Welcome to Apache Hadoop! `http://hadoop.apache.org/`, 2014, accessed 11.11.2014.

[3] Baldeschwieler E.: Yahoo! Launches Worlds Largest Hadoop Production Application. `https://goo.gl/wrOZ2v`, 2008, accessed 11.11.2014.

[4] Du X., Ni Y., Yao Z., Xiao R., Xie D.: High performance parallel evolutionary algorithm model based on MapReduce framework. *International Journal of Computer Applications in Technology*, vol. 46(3), pp. 290–295, 2013.

[5] Evans J., Rzhetsky A.: Machine Science. *Science*, vol. 329, pp. 399–400, 2010.

[6] Fernndez F., Snchez J.M., Tomassini M., Gmez J.A.: A Parallel Genetic Programming Tool Based on PVM. In: J. Dongarra, E. Luque, T. Margalef, eds., *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, *Lecture Notes in Computer Science*, vol. 1697, pp. 241–248, Springer, Berlin–Heidelberg, 1999.

[7] Funika W., Godowski P., Pegiel P., Król D.: Semantic-Oriented Performance Monitoring of Distributed Applications. *Computing and Informatics*, vol. 31(2), pp. 427–446, 2012.

[8] Funika W., Koperek P.: Genetic Programming in Automatic Discovery of Relationships in Computer System Monitoring Data. In: *Parallel Processing and Applied Mathematics*, *Lecture Notes in Computer Science*, vol. 8384, pp. 371–380, Springer, Berlin–Heidelberg, 2014.

[9] Funika W., Kupisz M., Koperek P.: Towards Autonomic Semantic-Based Management of Distributed Applications. *Computer Science*, vol. 11, pp. 51–64, 2010.

[10] Hindman B., Konwinski A., Zaharia M., Ghodsi A., Joseph A.D., Katz R., Shenker S., Stoica I.: Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In: *Proceedings of the 8th USENIX Conference on Networked Sys-*

tems Design and Implementation, NSDI'11, pp. 295–308, USENIX Association, Berkeley, CA, USA, 2011.

[11] hubert: project source code. `https://github.com/pkoperek/hubert`, 2015, accessed 15.02.2015.

[12] King R.D., Rowland J., Oliver S.G., et al.: The Automation of Science. *Science*, vol. 324, pp. 85–89, 2009.

[13] Koza J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[14] Ryan A.: Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode. `https://goo.gl/ifnqx`, 2012, accessed 11.11.2014.

[15] Salhi A., Glaser H., De Roure D.: Parallel Implementation of a Genetic-programming Based Tool for Symbolic Regression. *Inf. Process. Lett.*, vol. 66(6), pp. 299–307, 1998.

[16] Schmidt M., Lipson H.: Distilling free-form natural laws from experimental data. *Science*, vol. 324, pp. 81–85, 2009.

[17] Schmidt M.D., Lipson H.: Data-Mining Dynamical Systems: Automated Symbolic System Identification for Exploratory Analysis. *ASME Conference Proceedings*, vol. 2008(48364), pp. 643–649, 2008.

[18] Schmidt M.D., Lipson H.: Age-fitness pareto optimization. In: M. Pelikan, J. Branke, eds., *GECCO*, pp. 543–544, ACM, 2010.

[19] Schwarzkopf M., Konwinski A., Abd-El-Malek M., Wilkes J.: Omega: flexible, scalable schedulers for large compute clusters. In: *SIGOPS European Conference on Computer Systems (EuroSys)*, pp. 351–364, Prague, Czech Republic, 2013.

[20] Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauley M., Franklin M.J., Shenker S., Stoica I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 2–2, USENIX Association, Berkeley, CA, USA, 2012.

Affiliations

Włodzimierz Funika
AGH University of Science and Technology, ACC CYFRONET AGH, Krakow, Poland, `funika@agh.edu.pl`

Paweł Koperek
AGH University of Science and Technology, `pkoperek@gmail.com`

Received: 11.12.2014
Revised: 16.03.2015
Accepted: 20.03.2015