Using Machine Learning to Optimize Parallelism in Big Data Applications

Álvaro Brandón Hernández[a], María S. Perez[a], Smrati Gupta[b], Victor Muntés-Mulero[b]

[a] *Ontology Engineering Group, Universidad Politecnica de Madrid, Calle de los Ciruelos, 28660 Boadilla del Monte, Madrid*
[b] *CA Technologies, Pl. de la Pau, WTC Almeda Park edif. 2 planta 4, 08940 Cornellà de Llobregat, Barcelona*

Abstract

In-memory cluster computing platforms have gained momentum in the last years, due to their ability to analyse big amounts of data in parallel. These platforms are complex and difficult-to-manage environments. In addition, there is a lack of tools to better understand and optimize such platforms that consequently form backbone of big data infrastructure and technologies. This directly leads to underutilization of available resources and application failures in such environment. One of the key aspects that can address this problem is optimization of the task parallelism of application in such environments. In this paper, we propose a machine learning based method that recommends optimal parameters for task parallelization in big data workloads. By monitoring and gathering metrics at system and application level, we are able to find statistical correlations that allow us to characterize and predict the effect of different parallelism settings on performance. These predictions are used to recommend an optimal configuration to users before launching their workloads in the cluster, avoiding possible failures, performance degradation and wastage of resources. We evaluate our method with a benchmark of 15 Spark applications on the Grid5000 testbed. We observe up to a 51% gain on performance when using the recommended parallelism settings. The model is also interpretable and can give insights to the user into how different metrics and parameters affect the performance.

Keywords: machine learning, spark, parallelism, big data

1. Introduction

Big data technology and services market is estimated to grow at a CAGR[1] of 22.6% from 2015 to 2020 and reach $58.9 billion in 2020 [1]. Highly visible early adopters such as Yahoo, eBay and Facebook have demonstrated the value of mining complex information sets, and now many companies are eager to unlock the value in their own data. In order to address big data challenges, many different parallel programming frameworks, like Map Reduce, Apache Spark or Flink have been developed [2, 3, 4].

Planning big data processes effectively on these platforms can become problematic. They involve complex ecosystems where developers need to discover the main causes of performance degradation in terms of time, cost or energy. However, processing collected logs and metrics can be a tedious and difficult task. In addition, there are several parameters that can be adjusted and have an important impact on application performance.

While users have to deal with the challenge of controlling this complex environment, there is a fundamental lack of tools to simplify big data infrastructure and platform management. Some tools like YARN or Mesos [5, 6] help in decoupling the programming platform from the resource management. Still, they don't tackle the problem of optimizing application and cluster performance.

One of the most important challenges is finding the best parallelization strategy for a particular application running on a parallel computing framework. Big data platforms like Spark[2] or Flink[3] use JVM's distributed along the cluster to perform computations. Parallelization policies can be controlled programmatically through APIs or by tun-

Email address: abrandon@fi.upm.es (Álvaro Brandón Hernández)

[1] Compound Annual Growth Rate

[2] http://spark.apache.org/ (last accessed Jan 2017)

[3] https://flink.apache.org/ (last accessed Jan 2017)

ing parameters. These policies are normally left as their default values by the users. However, they control the number of tasks running concurrently on each machine, which constitutes the foundation of big data platforms. This factor can affect both correct execution and application execution time. Nonetheless, we lack concrete methods to optimize the parallelism of an application before its execution.

In this paper, we propose a method to recommend optimal parallelization settings to users depending on the type of application. Our method includes the development of a model that can tune the parameters controlling these settings. We solve this optimization problem through machine learning, based on system and application metrics collected from previous executions. This way, we can detect and explain the correlation between an application, its level of parallelism and the observed performance. The model keeps learning from the executions in the cluster, becoming more accurate and providing several benefits to the user, without any considerable overhead. In addition, we also consider executions that failed and provide new configurations to avoid these kinds of errors. The main two contributions of this paper are:

- A novel method to characterize the effect of parallelism in the performance of big data Workloads. This characterization is further leveraged to optimize in-memory big data executions by effective modelling of the performance correlation with application, system and parallelism metrics.
- A novel algorithm to optimize parallelism of applications using machine learning. Furthermore, a flavour of our proposed algorithm addresses the problem of accurate settings in the execution of applications due to its ability to draw predictions and learn from the comparison with actual results dynamically.

We choose Spark running on YARN as the framework to test our model because of its wide adoption for big data processing. The same principles can be applied to other frameworks, like Flink, since they also parallelize applications in the same manner.

This paper is structured as it follows. Section 2 provides some background about Spark and YARN. In Section 3, we motivate the problem at hand through some examples. Section 4 details the proposed model and learning process. In Section 5, we evaluate the model using Grid 5000 testbed [7]. Section 6 discusses related work. Finally, Section 7 presents future work lines and conclusions.

2. Overview of YARN and Spark

2.1. YARN: A cluster resource manager

YARN was born from the necessity of decoupling the programming model from the resource management in the cluster. The execution logic is left to the framework (e.g, Spark) and YARN controls the CPU and memory resources in the nodes. This tracking is done through the Resource Manager (RM), a daemon that runs in a machine of the cluster.

An application in YARN is coordinated by a unit called the Application Master (AM). AM is responsible for allocating resources, taking care of task faults and managing the execution flow. We can consider the AM as the agent that negotiates with the manager to get the resources needed by the application. In response to this request, resources are allocated for the application in the form of containers in each machine. The last entity involved is the Node Manager (NM). It is responsible for setting up the container's environment, copying any dependencies to the node and evicting any containers if needed. A graphical explanation of the architecture is depicted in Figure 1.

The number of available resources can be seen as a two dimensional space formed by memory and cores that are allocated to containers. Both the MB and cores available in each node are configured through the node manager properties: `yarn.nodemanager.resource.memory-mb` and `yarn.nodemanager.resource.cpu-vcores` respectively. An important aspect to take into account is how the scheduler considers these resources through the ResourceCalculator. The default option, *DefaultResourceCalculator*, only considers memory when allocating containers while *DominantResourceCalculator*, takes into account both dimensions and make decisions based on what is the dominant or bottleneck resource [8]. Since CPU is an elastic resource and we do not want to have a limit on it, we will use the *DefaultResourceCalculator* for YARN in our experiments. By doing this, we will be able to scale up or down the number of tasks running without reaching a CPU usage limit.

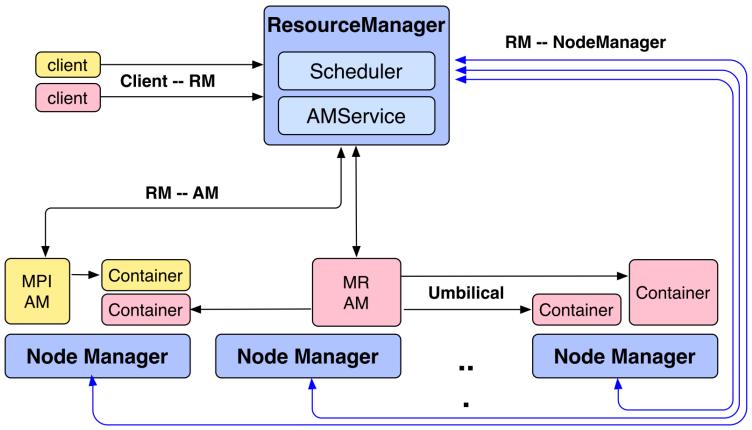

Figure 1: YARN Architecture. In the example a MPI and MapReduce application have their containers running in different nodes. Each node manager is going to have some memory and cores assigned to allocate containers [5]

2.2. Spark: A large data processing engine

Spark [9] is a fault-tolerant, in-memory data analytics engine. Spark is based on a concept called resilient distributed dataset (RDDs). RDDs are immutable resilient distributed collections of data structures partitioned across nodes that can reside in memory or disk. These records of data can be manipulated through transformations (e.g map or filter). A RDD is lazy in the way that will only be computed when an action is invoked (e.g count number of lines, save as a text file). A Spark application is implemented as a series of actions on these RDDs. When we execute an action over a RDD, a job triggers. Spark then formulates an execution Directed-Acylic-Graph (DAG) whose nodes will represent stages. These stages are composed of a number of tasks that will run in parallel over chunks of our input data, similar to the MapReduce platform. In Figure 2 we can see a sample DAG that represents a WordCount application.

When we launch a Spark application, an application master is created. This AM asks the resource manager, YARN in our case, to start the containers in the different machines of the cluster. These containers are also called executors in the Spark framework. The request consists of number of executors, number of cores per executor and memory per executor. We can provide these values through the `--num-executors` option together with the parameters `spark.executor.cores` and `spark.executor.memory`. One unit of `spark.executor.cores` translates into one task slot. Spark offers a way of dynamically asking for additional executors in the cluster as long as an application has tasks backlogged and resources available. Also executors can be released by an application if it does not have any running tasks. This feature is called dynamic allocation [10] and it turns into better resource utilization. We use this feature in all of our experiments. This means that we do not have to specify the number of executors for an application as Spark will launch the proper number, based on the available resources. The aim of this paper is to find a combination for `spark.executor.memory` and `spark.executor.cores` to launch an optimal number of parallel tasks in terms of performance.

3. Studying the impact of parallelism

The possibility of splitting data into chunks and processing each part in different machines in a divide and conquer manner makes it possible to analyse big amounts of data in seconds. In Spark, this parallelism is achieved through tasks that run inside executors across the cluster. Users need to choose the memory size and the number of tasks running inside each executor. By default, Spark considers that each executor size will be of 1GB with one task running inside. This rule of thumb is not optimal, since each application is going to have different requirements, leading to wastage of resources, long running times and possible failures in the execution among other problems.

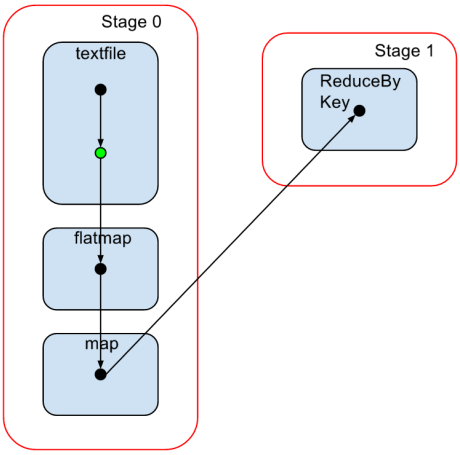

Figure 2: Example DAG for WordCount

spark.executor.memory	spark.executor.cores
512m	1
1g	1
2g	1
2g	2
2g	3
3g	1
3g	3
3g	4
4g	1
4g	3
4g	6
6g	1

Table 1: Combinations of Spark parameters used for the experiments

We explore these effects in the following experiments. We deployed Spark in four nodes of the Taurus cluster in the Grid 5000 testbed [7]. We choose this cluster because we can also evaluate the power consumption through watt-meters in each machine [11]. These machines have two processors Intel Xeon E5-2630 with six cores each, 32GB of memory, 10 Gigabit Ethernet connection and two hard disks of 300 GB. The operating system installed in the nodes is Debian 3.2.68, and the software versions are Hadoop 2.6.0 and Spark 1.6.2. We configure the Hadoop cluster with four nodes working as datanodes, one of them as master node and the other three as nodemanagers. Each nodemanager is going to have 28GB of memory available for containers. We set the value of vcores to the same number as physical cores. Note however that we are using all the default settings for YARN and by default the resource calculator is going to be DefaultResourceCalculator [12]. This means that only memory is going to be taken into account when deciding if there are enough resources for a given application. HDFS is used as the parallel filesystem where we read the files. We intend to illustrate the difference in level of parallelism for applications that have varied requirements in terms of I/O, CPU and memory. To this end, we utilize one commonly used application which is CPU intensive, like kMeans and another common one that is intensive in consumption of I/O, CPU and memory, like PageRank.

Our objective is to see what is the effect of the number of concurrently running tasks on the performance of an application. To do that, we are going to try different combinations of `spark.executor.memory` and `spark.executor.cores`, as seen in Table 1. We choose these values to consider different scenarios including many small JVM's with one task each, few JVM's with many tasks or big JVM's which only host one task.

In Figure 3, we can see the time and energy it takes to run a kMeans application in the cluster with the default, best and worst configuration. We can observe several effects in both energy and duration for each configuration. Firstly, we observe that the best configuration is 4GB and 6 `spark.executor.cores`. Since kMeans only shuffles a small amount of data, the memory pressure and I/O per task are low, allowing this configuration to run a higher number of tasks concurrently and reducing execution times. Additionally, we can see that the worst configuration is the one with 512MB and 1 `spark.executor.core`. Spark allows us to store data in memory for iterative workloads like kMeans. Since `spark.executor.memory` is sized as 512MB, the system does not have enough space to cache data blocks into the executors. This means that in later stages we will have to read that block from disk, incurring in additional latency, specially if that block is in another rack.

The other example is Page Rank, a graph application. Its DAG involves many stages and the configuration will have a great impact on the application performance, as we can see in Figure 4. There is a difference of almost 14 minutes between the best execution and the worst one. In compar-

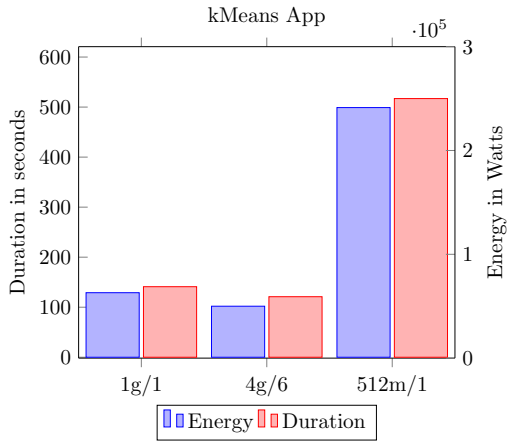

Figure 3: Run time in seconds and power consumption in watts for a kMeans App. The default, best and worst configurations are depicted

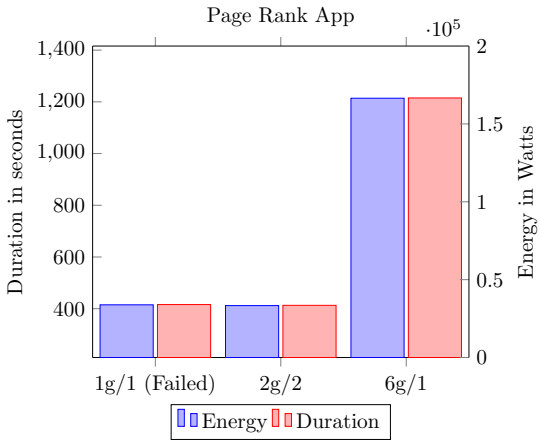

Figure 4: Run time in seconds and power consumption in watts for a Page Rank App. The default, best and worst configurations are depicted

ison to kMeans, PageRank caches data into memory that grows with every iteration, filling the heap space often. It also shuffles gigabytes of data that have to be sent through the network and buffered in memory space. The best configuration is the 2g and 2 `spark.executor.cores` configuration since it provides the best balance between memory space and number of concurrent tasks. For the default one, the executors are not able to meet the memory requirements and the application crashed with an out of memory error. Also, we observed that for the 6g and 1 `spark.executor.cores`, the parallelism is really poor with only a few tasks running concurrently in each node.

We can draw the conclusion that setting the right level of parallelism does have an impact on big data applications in terms of time and energy. The degree of this impact varies depending on the application. For PageRank it can prevent the application from crashing while for kMeans it is only a small gain. Even so, this moderate benefit may have a large impact in organizations that run the same jobs repeatedly. Avoiding bad configurations is equally important. Providing a recommendation to the user can help him to avoid out of memory errors, suboptimal running times and wasting cluster resources through constant trial and error runs.

4. Model to find the optimal level of parallelism

As we have seen in the previous section, setting the right level of parallelism can be done for each application through its `spark.executor.memory` and `spark.executor.cores` parameters. These values are supposed to be entered by the user and are then used by the resource manager to allocate JVM's. However, clusters are often seen as black-boxes where it is difficult for the user to perform this process. Our objective is to propose a model to facilitate this task. Several challenges need to be tackled to achieve this objective:

- **The act of setting these parallelization parameters cannot be detached from the status of the cluster:** Normally, the user set some parameters without considering the memory and cores available at that moment in the cluster. However, the problem of setting the right executor size is highly dependent on the memory available in each node. We need the current resource status on YARN as a variable when making these decisions.
- **Expert knowledge about the application behaviour is needed:** This needs monitoring the resources consumed and metrics of both the application plus the system side.
- **We have to build a model that adapts to the environment we are using:** Different machines and technologies give room to different configurations and parallelism setting.
- **We have to be able to apply the experience of previous observations to new executions:** This will simplify the tuning part to the user every time he launches a new application.

To solve these problems we leverage the machine learning techniques. Our approach is as follows. First, we will use our knowledge about YARN internals to calculate the number of tasks per node we can concurrently run for different configurations, depending on the available cluster memory. These calculations, together with metrics gathered from previous executions for that application, will form the input of the machine learning module. Then, the predictions will be used to find the best configuration. In the following subsections we will explain how we gather the metrics, how we build a dataset that can be used for this problem and the methodology we use to make the recommendations.

4.1. Gathering the metrics and building new features

Normally predictive machine learning algorithms need a dataset with a vector of features $x_1, x_2, ...x_n$ as an input. In this section we will explain which features we use to characterize Spark applications. They can be classified in three groups:

- Parallelism features: They are metrics that describe the concurrent tasks running on the machine and the cluster. Please note that these are different from the parallelization parameters. The parallelization parameters are the `spark.executor.cores` and `spark.executor.memory` parameters, which are set up by the users, while the parallelism features describe the effect of these parameters on the applications. We elaborate upon these features in Subsection 4.1.1.
- Application features: These are the metrics that describe the status of execution in Spark. Examples of these features are number of tasks launched, bytes shuffled or the proportion of data that was cached.
- System features: These are the metrics that represent the load of the machines involved. Examples include CPU load, number of I/O operations or context switches.

In the following subsections, we will explain how we monitor these metrics and how we build new additional metrics, like the number of tasks waves spanned by a given configuration or the RDD persistence capabilities of Spark. An exhaustive list of metrics is included as an appendix (see Table A.3), together with a notation table, to ease the understanding of this section (Table B.4).

4.1.1. Parallelism features

A Spark stage spans the number of tasks that are equal to the number of partitions of our RDD. By default, the number of partitions when reading an HDFS file is given by the number of HDFS blocks of that file. It can also be set statically for that implementation through instructions in the Spark API. Since we want to configure the workload parallelism at launch time, depending on variable factors, like the resources available in YARN, we will not consider the API approach. The goal is to calculate, before launching an application, the number of executors, tasks per executors and tasks per machine that will run with a given configuration and YARN memory available. These metrics will be later used by the model to predict the optimal configuration. To formulate this, we are going to use the calculations followed by YARN to size the containers. We need the following variables:

- Filesize (f_{size}): The number of tasks spanned by Spark is given by the file size of the input file.
- dfs.block.size (b_{hdfs}): An HDFS parameter that specifies the size of the block in the filesystem. We keep the default value of 128MB.
- yarn.scheduler.minimum.allocation-mb (min_{yarn}): The minimum size for any container request. If we make requests under this threshold it will be set to this value. We use its default value of 1024MB.
- spark.executor.memory (mem_{spark}): The size of the memory to be used by the executors. This is not the final size of the YARN container, as we will explain later, since some off-heap memory is needed.
- spark.executor.cores (cor_{spark}): The number of tasks that will run inside each executor.
- yarn.nodemanager.resource.memory-mb parameter (mem_{node}): This sets the amount of memory that YARN will have availabe in each node.
- spark.yarn.executor.memoryOverhead ($over_{yarn}$): The amount of available off-heap memory. By default, its value its 384.

- Total available nodes (N_{nodes}): This represents the number of nodes where we can run our executors.

The size of an executor is given by:

$$Size_{exec} = mem_{spark} + \max(over_{yarn}, mem_{spark} * 0.10) \quad (1)$$

The 0.10 is a fixed factor in Spark and it is used to reserve a fraction of memory for the off-heap memory. We then round up the size of the executor to the nearest multiple of yarn min_{yarn} to get the final size. For example, if we get a $Size_{exec} = 1408$, we will round up to two units of min_{yarn} resulting in $Size_{exec} = 2048$. Now we can calculate the number of executors in each node as $N_{exec} = \lfloor mem_{node}/Size_{exec} \rfloor$ and the number of task slots in each node as $slots_{node} = \lfloor N_{exec} * cor_{spark} \rfloor$. Consequently, the total number of slots in the cluster are $slots_{cluster} = \lfloor slots_{node} * N_{nodes} \rfloor$.

Finally, we also want to know the number of waves that will be needed to run all the tasks of that stage. By waves we mean the number of times a tasks slot of an executor will be used, as depicted in Figure 5. We first calculate the number of tasks that will be needed to process the input data. This is given by $N_{tasks} = \lfloor f_{size}/b_{hdfs} \rfloor$. Then the number of waves is found by dividing this task number between the number of slots in the cluster $N_{waves} = \lfloor N_{tasks}/slots_{cluster} \rfloor$. Summarizing all the previous metrics that are useful to us, we get the following set of metrics, which we will call parallelism metrics:

$$\{N_{tasks}, slots_{node}, slots_{cluster}, N_{waves}, size_{exec}, cor_{spark}\} \quad (2)$$

These are the metrics that will vary depending on the resources available in YARN and the configuration we choose for the Spark executors and that will help us to model performance in terms of parallelism.

4.1.2. Application features

Spark emits an event every time the application performs an action at the task, stage, job and application levels explained in Section 2.2. These events include metrics that we capture through Slim[4], a node.js server that stores these metrics in a

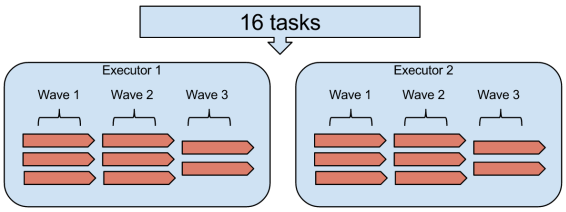

Figure 5: A set of waves spanned by two executors with three spark.executor.cores. If we have 16 tasks to be executed then it will take three waves to run them in the cluster

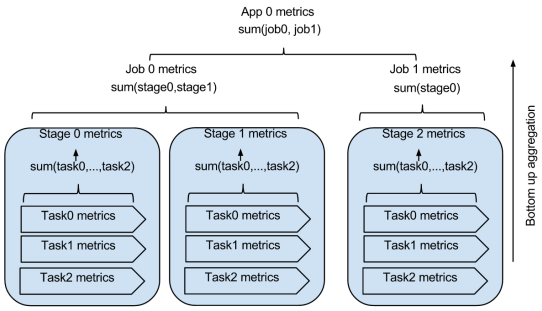

Figure 6: Slim bottom-up aggregation. If we have the metrics for the tasks, Slim can aggregate the ones belonging to a stage to calculate that stage metrics. It does the same at the job and application execution level

MongoDB[5] database and aggregates them at task, stage, job and application level in a bottom to up manner. This means that if we aggregate the counters for all the tasks belonging to a stage, we will have the overall metrics for that stage. The same is done at job level, by aggregating the stage metrics and at application level, by aggregating the jobs, as shown in Figure 6.

This however creates an issue. We want to characterize applications, but if we aggregate the metrics for each tasks then we will have very different results depending on the number of tasks launched. As we explained in Section 4.1.1, the number of tasks spanned depends on the size of the input file. This is not descriptive as we want to be able to detect similar applications, based on their behaviour and the operations they perform, even if they operate with different file sizes. Let's assume we have some metrics we gathered by monitoring the only stage of the Grep application with 64 tasks. We need these metrics to describe the application, inde-

[4]https://github.com/hammerlab/slim (last accessed Jan 2017)

[5]https://www.mongodb.com/ (last accessed Jan 2017)

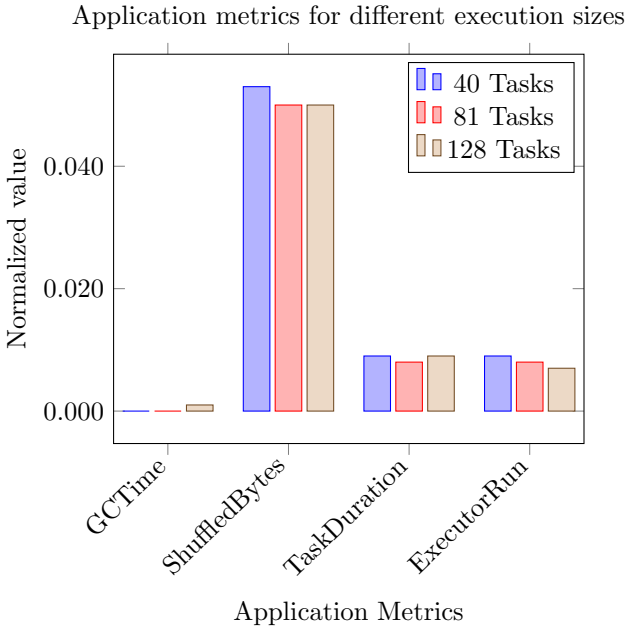

Figure 7: Mean application metrics values for a stage of a Sort application with different filesizes. We can see how the mean of the metrics is almost the same for 40, 81 and 128 tasks

pendently of the file size with which it was executed. In this example, if this same stage is launched again in the future with 128 tasks, its metrics have to be similar to the ones of the 64 tasks execution in order to apply what the model learned so far for Grep or similar behaving applications.

To even out these situations, we calculate the mean of the metrics for the tasks of that stage. Figure 7 shows that the mean gives us similar metrics independently of the filesize for a stage of Sort. The metrics are normalized from 0 to 1 to be able to plot them together, since they have different scales.

We also define an additional metric at the application level to describe how Spark persists data. We derive a persistence score vector for each stage of an application that describes how much of the total input was read from memory, local disk, HDFS or through the network. For a stage S, we have a total number of task (N_{tasks}). We count the number of these tasks that read their input from different sources as memory (N_{memory}), disk (N_{disk}), HDFS (N_{hdfs}) and network $N_{network}$. The vector for that stage V_S is:

$$V_S = \left\{ \frac{N_{memory}}{N_{tasks}}, \frac{N_{disk}}{N_{tasks}}, \frac{N_{hdfs}}{N_{tasks}}, \frac{N_{network}}{N_{tasks}} \right\}$$

This vector will describe if the stage processes data that is persisted in memory, disk or if it was not on that node at all and had to be read through the network. We also use the HDFS variable to know if the stage reads the data from the distributed file system. Usually this belongs to the first stage of an application, since it is the moment when we first read data and create the RDD.

An important issue to notice here is that V_S can change depending on the execution. Sometimes, Spark will be able to allocate tasks in the same node where the data is persisted. In some other executions, it will not be possible because all slots are occupied or because the system has run out of memory. However, the objective is not to have a detailed description of the number of tasks that read persisted data. Instead, we want to know if that stage persists data at all and how much it was able to. Later on, the model will use this to statistically infer how different configurations can affect a stage that persists data.

4.1.3. System features

There are aspects of the workload that cannot be monitored at application level, but at system level, like the time the CPU is waiting for I/O operations or the number of bytes of memory paged in and out. To have a complete and detailed signature of the behaviour of an application, we also include these kind of metrics. We include here features like cpu_usr, bytes paged in/out or bytes sent through the network interface. To capture them, we use GMone [13]. GMone is a cloud monitoring tool that can capture metrics of our interest on the different machines of the cluster. GMone is highly customizable and it allows developing plugins that will be used by the monitors in each node. We developed a plugin that uses Dstat[6]. With this, we obtain a measure per second of the state of the CPU, disk, network and memory of the machines in our cluster.

Since one task will run only in one node, we calculate the mean of the system metrics on the node during the lifespan of that task. This will give us descriptive statistics about the impact of that task at the system level. Then, for each stage we will calculate the mean of the metrics for all of its tasks in a similar bottom up way, as we explained Section

[6]http://dag.wiee.rs/home-made/dstat/ (last accessed Jan 2017)

in 4.1.2. This will represent what happened at the system level while this application was running.

4.2. Building the dataset and training the model

For each stage executed in our cluster, we get a vector that we use as a data point. We will represent this database of stages as:

$$stagesDB = \{X_{app}, X_{system}, X_{parallelism}, Y_{duration}\}$$

where the X's are the features at application (Section 4.1.2), system (Section 4.1.3) and parallelism level (Section 4.1.1) and $Y_{duration}$ is the duration of that stage or target variable to predict. Now we have to build the dataset that we will use to train the model. We want to answer the following question: *If we have a stage with metrics X_{app} and X_{system}, collected under some $X_{parallelism}$ conditions and its duration was $Y_{duration}$, what will be the new duration under some different $X_{parallelism}$ conditions?*. We can regress this effect from the data points we have from previous executions. However, we have to take into account that metrics change depending on the original conditions in which that stage was executed. For instance it is not the same executing a stage with executors of 1GB and 1 task compared to 3GB and 4 tasks. Metrics like garbage collection time will increase if tasks have less memory for their objects. In addition, CPU wait will increase if we have many tasks in the same machine competing for I/O. We can see this effect in Figure 8, where the metrics of a stage of a support vector machine implementation in Spark change depending on the number of tasks per node we choose.

To solve this variance in the metrics, we have to include two set of parallelism values in the learning process:

- The parallelism conditions under which the metrics were collected. We will call them $X_{parallelism_{ref}}$.
- The new parallelism conditions under which we will run the application. We will call them $X_{parallelism_{run}}$.

Now building the dataset is just a matter of performing a cartesian product between the metrics $\{X_{app}, X_{system}, X_{parallelism}\}$ of a stage and the $\{X_{parallelism}, Y_{duration}\}$ of all the executions we have for that same stage. For example, let's assume we have two executions of a stage with different configurations like 10 and 4 tasks per node. We also have all their features, including system plus application metrics, and the duration of the stage. The logic behind this cartesian product is: *If we have the metrics (X_{10}) and the duration (Y_{10}) of an execution with 10 tasks per node and we have the metrics (X_4) and the duration (Y_4) of an execution with 4 tasks per node (tpn), then the metrics of the 10 tasks execution together with a configuration of 4 tasks per node can be correlated with the duration of the 4 tpn execution, creating a new point like:*

$$\{X_{10app}, X_{10system}, X_{10parallelism_{ref}}, X_{4parallelism_{run}}, Y_{4duration}\} \quad (3)$$

Note here that we consider that two stages are the same when they belong to the same application and the same position in the DAG. The algorithm is shown in Algorithm 1.

Now we can train the machine learning model with this dataset and check its prediction accuracy. First we have to choose the most accurate implementation. Since this is a regression problem, we try different state-of-the-art regression algorithms: Bayesian Ridge, Linear Regression, SGD Regressor, Lasso, Gradient Boosting Regressor, Support Vector Regression and MLPRegressor, all from the sklearn Python library [14]. First of all, we choose these regression algorithms versus other solutions, like Deep Learning, because they are interpretable. Apart from this, the chosen algorithms are easily evaluated with cross validation techniques, as K-fold, Leave P-out or Leave One Group Out, among others. From a practical point of view, these algorithms are available in the scikit-learn package, which provides implementations ready to be used right out of the box. They also allow us to create pipelines where we perform a series of preprocessing steps before training the model. We build a pipeline for each of the previous algorithms, where we first standardize the data and then fit the model. Standardization of the data is a common previous step of many machine learning estimators. It is performed for each feature, by removing its mean value, and then scaling it by dividing by its standard deviation. This ensures an equal impact of different variables on the final outcome and not being biased towards variables that have higher or lower numerical values. Then for each pipeline, we perform a ten times cross validation with 3 k-folds [15]. The results are shown in Figure 9. The metric used to evaluate the score is Mean Absolute Error (MAE) [16]. It is calculated by averaging the absolute er-

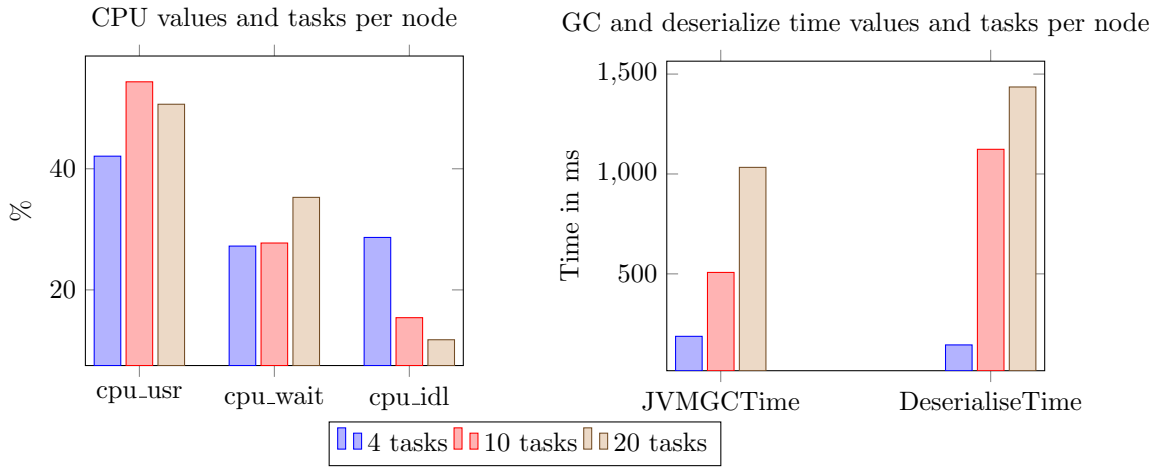

Figure 8: Changes in the metrics for a stage of a support vector machine implementation in Spark. Note how the CPU metrics change depending on the number of tasks per node. Also garbage collection and the deserialize time of an stage increases with this number

Algorithm 1 Building the training dataset for the machine learning algorithm

procedure BUILDATASET
  $stages = \{(X_{app}, X_{system}, X_{parallelism}, Y_{duration})_n\}$ ▷ The stage executions that we have
  $dataset = \emptyset$
  for all $s \in stages$ **do**
    $B = \{\, x \in stages \mid x = s \,\}$ ▷ All the stages that are the same as s
    $new = s[X_{app}, X_{system}, X_{parallelism_{ref}}] \times B[X_{parallelism_{run}}, Y_{duration}]$
    add *new* **to** *dataset*
  end for
  return *dataset*
end procedure

rors between predicted and true values for all the data points. MAE will allow us to evaluate how close the different predictors are to the real values. We do not include SGD Regressor, since its MAE is just too high to compare with the other methods. As we can see, boosted regression trees have the best accuracy amongst the seven methods and also the lowest variance. In addition, decision trees are interpretable. This means that we will be able to quantify the effect of any of the features of the model on the overall performance of the application.

4.3. Using the model: providing recommendations for applications

So far, we have only characterized and built our model for stages. An application is a DAG made out of stages. So how we provide a parallelization recommendation based on the previous explained concepts?. Remember that a configuration in Spark is given at application level. Thus, if we set some values for `spark.executor.memory` and `spark.executor.cores`, they will be applied to all the stages. If we have a list $listofconfs$ with combinations of different values for these two parameters, and we launch an application with a different combination each time, each execution will have different tasks per node, waves of tasks and memory per task settings. In other words, and following the previously introduced notation, each execution will have different $X_{parallelism}$ features. The dataset built in Algorithm 1 had data points like:

$$\{X_{app}, X_{system}, X_{parallelism_{ref}}, X_{parallelism_{run}}, Y_{duration}\} \quad (4)$$

therefore, we can plug in new values for $X_{parallelism_{run}}$ by iterating through the different configurations in $listofconfs$, and predict the new $Y_{duration}$ for the parameters. In addition, the objective is to know the effect of a given configuration on the whole application. So we can perform this process for all the stages that are part of the application and sum the predictions for each configuration. Then we only have to choose as the optimal configuration the one from the list that yields the minimum predicted duration. Algorithm 2 describes this process for a given application and its input data size. The time complexity of the boosted decision trees model, used to make the predictions, is $O(ntree{*}n{*}d)$, where $ntree$ is the number of trees of the boosted model, d is the depth of those trees and n is the number of data points to be predicted.

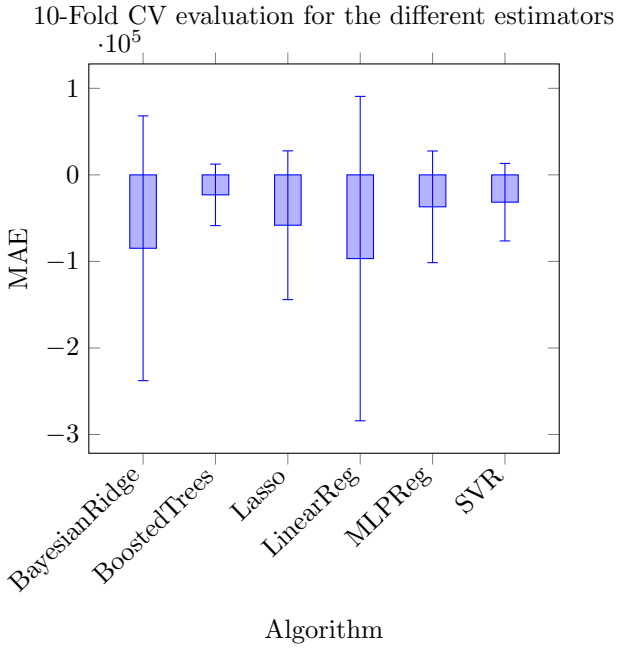

Figure 9: 10 CV with 3 K-Fold MAE score for different regression algorithms. GradientBoostingRegressor is the clear winner with a median absolute error of 23 seconds, a maximum of 94 seconds in the last 10th iteration and a minimum of 2.7 seconds for the 3rd iteration

We have to do this prediction for the $nconf$ configurations of the configuration list, which transforms it into $O(ntree{*}n{*}d{*}nconf)$. Note that the only term that is variable here is the number of data points n, that will grow as we have more executions on our system. The rest will remain constant as they are fixed the moment we train the model and are small numbers. We have to point out that we do not consider stages that launch only one task, since it is trivial that parallelism will not bring any benefit to them. The DAG engine of Spark can also detect that some stages do not have to be recomputed and so they can be skipped. We do not consider these kind of stages in the DAG either since their runtime is always 0. Figure 10 shows a graphical depiction of the workflow for a Sort app example.

4.4. Dealing with applications that fail

When we explained the process by which we predict the best parallelism configuration, we assumed that we have seen at least once all the stages of an application. However, this is not always the case, since some configurations may crash. For example, for a Spark ConnectedComponent application with a filesize of 8GB, the only configurations that

Algorithm 2 Recommending a parallelism configuration for an application and its input datasize

procedure PREDICTCONFBOOSTED($app, filesize, listofconfs$)
  $S = \{\, s \in stages \mid s \in DAG(app) \,\}$ ▷ All the stages that belong to the app
  $listofconfsandduration = \emptyset$
  for all $conf \in listofconfs$ **do**
    for all $s \in S$ **do**
      if $s.ntasks \neq 1$ **then** ▷ Some stages only have on task. We do not consider those
        $s[X_{parallelism_{run}}] = calculateParallelism(filesize, conf)$ ▷ As explained in 4.1.1
        $duration = duration + predictBoostedTrees(s)$ ▷ we accumulate the prediction of each stage
      end if
    end for
    add $(duration, conf)$ **to** $listofconfsandduration$
  end for
  return $min(listofconfsandduration)$
end procedure

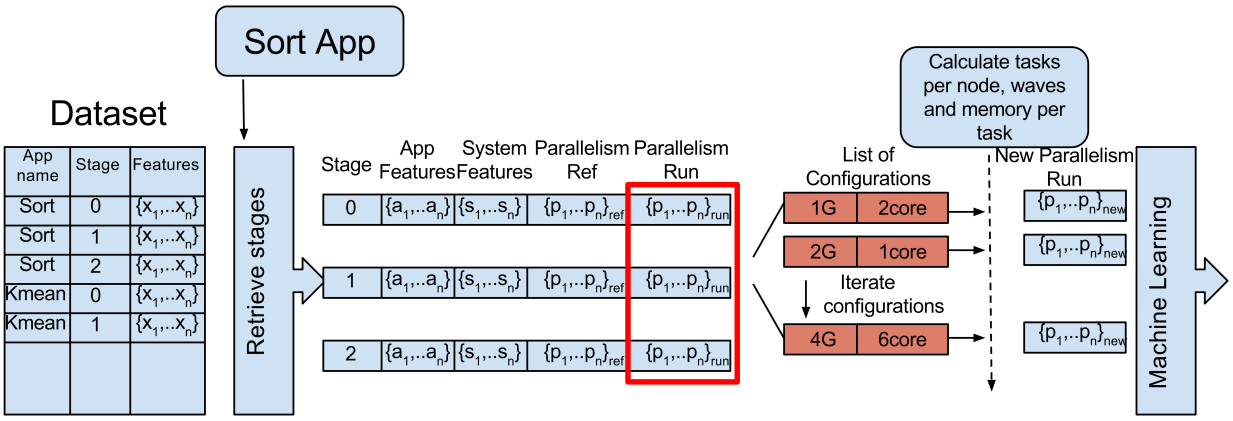

Figure 10: An example with a Sort App, where we iterate through a configuration list. Notice the different parallelism features: $\{p_1, ..., p_n\}_{ref}$ represent the parallelism conditions under which the metrics were collected, $\{p_1, ..., p_n\}_{run}$ the parallelism conditions under which the application run for that execution and $\{p_1, ..., p_n\}_{new}$ the new parallelism conditions that we want to try and the ones we will pass to the machine learning module. The rest of the metrics are kept constant. Then we just have to group by configuration and sum up each of the stages predictions to know which one is best

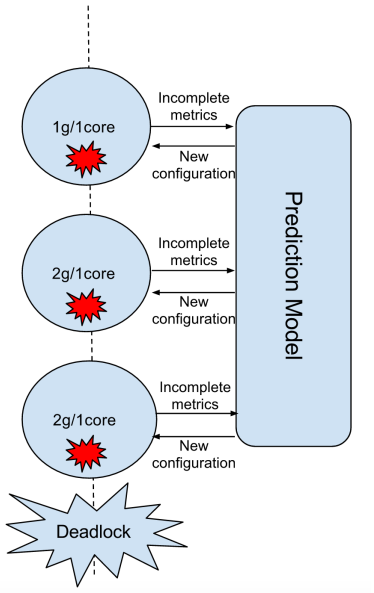

Figure 11: A deadlock situation in which the same configuration is recommended continuously. The application crashes with the default configuration. Based on whichever stages executed successfully the next recommendation also crashes. If we do not get past the crashing point the recommendation will always be the same

were able to complete successfully in our experiments were the ones with 4GB and 1 core and 6GB and 1 core. The reason behind it is that graph processing applications are resource hungry and iterative by nature. This can put a lot of pressure in the memory management, specially when we cache RDD's like in these kind of implementations. In this particular example, the 8GB file we initially read from HDFS grows to a 17.4GB cached RDD in the final stages of the application.

Thus, if we choose an insufficient memory setting, like the default 1GB configuration, the application will not finish successfully and we will not have either the complete number of stages or the metrics for that application. Incomplete information means that the next configuration recommended could be non-optimal, drawing us into a loop where the application continuously crashes, as depicted in Figure 11.

To solve this, we use the approach explained in Algorithm 3. When a user launches an application, we check if there are any successful executions for it. If we cannot find any then we retrieve the furthest stage the application got to. This can be considered as the point where the execution could not continue. For that stage we find the k-nearest neighbours from all the stages that we have seen so far. These neighbouring stages belong to a set of applications. Then we follow a conservative approach where we choose the configuration amongst these applications that resulted in the most number of stages completed. The intuition behind it is that a configuration with a high number of completed stages means better stability to the application. With this, we aim to achieve a complete execution of the application and to collect the features of all the stages with it. If the user executes the same application in the future, we can use this complete information together with the standard boosted gradient model to recommend a new configuration. By trying different values for the number of k neighbours, we found that 3 is a good number for this parameter. Going above this value can retrieve too many diverse stages, which could make the application to crash again. Going below it can take us to the same deadlock depicted in Figure 11. Remember that the objective is not to find an optimal configuration, but a conservative configuration, which enables a complete execution without failures.

The kNeighbours approach uses a K-D Tree search approach, where the complexity is $O(log(n))$ with n being the number of data points.

4.5. The recommendation workflow

Now that we have developed the models with which we make recommendations, we can connect everything together and describe the workflow we follow to find the best parallelism settings. When a user wants to execute an application, first we check if we have any information of it in our database of previously seen applications. If we do not have it, we execute it with the default configuration. This execution will give us the metrics we need to tune the application in future executions. If we do have information about that application, we check if it completed successfully or not:

- In case it did not, we apply the procedure of Algorithm 3 for crashed executions.
- In case it did, we apply the Algorithm 2 with the boosted regression trees model.

Whatever the case is, the monitoring system adds the metrics for that run to our database so they can be applied for future executions. Applications in a cluster are recurring [17]. This means that the

Algorithm 3 Finding a configuration for an application that failed

1: **procedure** PREDICTCONFKNEIGHBOURS(app)
2: $S = \{ s \in stages \mid s \in DAG(app) \wedge s[status] = failed \}$
3: $x = lastexecutedstage \in S$
4: $neighbours = find3neighbours(x, stages)$
5: $res = \{ n \in neighbours \mid n.stagesCompleted = max(n.stagesCompleted) \}$
6: **return** $res.configuration$
7: **end procedure**

precision of the model will increase with the time since we will have more data points. We assume the machine learning algorithm has been trained beforehand and we consider out of the scope of this paper and as future work, when and how to retrain it. A final algorithm for the process is depicted in Algorithm 4.

5. Evaluation

We perform an offline evaluation of the model with the traces we got from a series of experiments in Grid 5000.

Cluster setup: Six nodes of the Adonis cluster in the Grid 5000 testbed. These machines have two processors Intel Xeon E5520 with four cores each, 24GB of memory, 1 Gigabit Ethernet and 1 card InfiniBand 40G and a single SATA hard disk of 250 GB. The operating system installed in the nodes is Debian 3.2.68, and the software versions are Hadoop 2.6.0 and Spark 1.6.2. We configure the Hadoop cluster with six nodes working as datanodes, one of them as master node and the other five as nodemanagers. Each nodemanager has 21GB of memory available for containers. We set the value of vcores to the same number as physical cores. We build a benchmark that is a combination of Spark-bench [18], Bigdatabench [19] and some workloads that we implemented on our own. The latter ones were implemented to have a broader variety of stages and each one has different nature:

- RDDRelation: Here we use Spark SQL and the dataframes API. We read a file that has pairs of $(key, value)$. First we count the number of keys using the RDD API and then using the Dataframe API with a `select count(*)` type of query. After that, we do a `select from` query in a range of values and store the data in a Parquet data format[7]. This workflow consist of four stages in total.

- NGramsExample: We read a text file and we construct a dataframe where a row is a line in the text. Then we separate these lines in words and we calculate the `NGrams=2` for each line. Finally these Ngrams are saved in a text file in HDFS. This gives us a stage in total since it is all a matter of mapping the values and there is no shuffles involved.

- GroupByTest: Here we do not read any data from disk but we rather generate an RDD with pairs of $(randomkey, values)$. The keys are generated inside a fixed range so there will be duplicate keys. The idea is to group by key and then perform a count of each key. The number of pairs, the number of partitions of the RDD and consequently the number of tasks can be changed through the input parameters of the application. This gives us two stages like in a typical GroupBy operation.

We run all the applications of this unified benchmark with three different file sizes as we want to check how the model reacts when executing the same application with different amounts of data. Some implementations, like the machine learning ones, expect a series of hyperparameters. These are kept constant across executions and their effect is left out of the scope of this paper. We also try different values for `spark.executor.memory` and `spark.executor.cores` for each application and data size combinations. These executions generate a series of traces that we use to train and evaluate our model offline. Since the usefulness of our approach is that it can predict the performance of new unseen applications, we train the model by leaving some apps outside of the training set. Those apps will be later used as a test set. A description of the benchmark is depicted in Table 2. The reason for choosing this benchmark, is to have a

[7] https://parquet.apache.org/ (last accessed Jan 2017)

Algorithm 4 Providing an optimal configuration for a application and a given input file size

procedure CONFIGURATIONFORAPPLICATION($application, filesize$)
  $A = \{ apps \in appDB \mid apps.name = application) \}$
  if $A = \emptyset$ **then**
    $conf = (1g, 1)$
  else if $A \neq \emptyset$ **then**
    if $(\forall app \in A \mid app.status = failed)$ **then**
      $conf = predictconfkneighbours(app)$
    else
      $conf = predictconfboosted(app, filesize)$
    end if
  end if
  $submittospark(app, conf)$
  add *monitored* **features** *to* **stageDB** *and* **appDB**
end procedure

representative collection of batch processing applications that are affected differently by their parallelization, i.e. graph processing, text processing and machine learning. These three groups allow us to prove that applications with different resource consumption profiles have different optimal configurations. We leave as future work the effect of parallelization in data streaming applications. The files were generated synthetically. At the end we have 5583 stages to work with. As before we use the dynamic allocation feature of Spark and the DefaulResourceAllocator for YARN.

5.1. Overhead of the system

In this section, we want to evaluate the overhead of monitoring the system, training the model and calculating a prediction. The first concern is if Slim and GMone introduce some overhead when launching applications. We executed five batches of two applications running in succession: Grep and LinearRegression. We measured the time it took to execute these five batches with and without monitoring the applications. The results are depicted in Figure 12. As we can see, the overhead is negligible.

Also we want to evaluate how much time it takes to train the model and to calculate the optimal configuration for an application. We train the model with the traces we got from Grid5000 in our local computer. The specifications for the CPU are 2,5 GHz Intel Core i7 of 4 cores and for the memory is 16GB of DDR3 RAM. The overhead of training the model depends on the number of data points, as shown in Figure 13. For the complete training set of 55139 the latency is 877 seconds. For the overhead

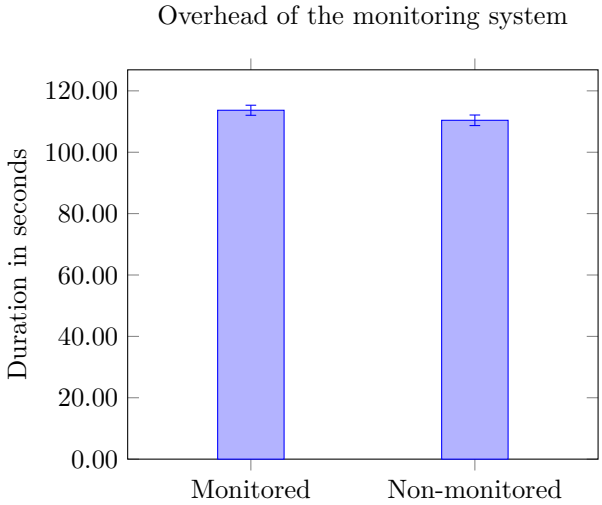

Figure 12: Overhead introduced by monitoring and storing the traces versus non-monitoring. As we can see the latency is minimum

Table 2: Different applications used. File size denotes the three different sizes used for that application. Group by doesn't read any file but creates all the data directly in memory with 100 partitions

Application	Dataset Split	File Size	Type of App
ShortestPath	Test	20GB,11GB,8GB	Graph Processing
Connected Component			
Logistic Regression		18GB,10GB,5GB	Machine Learning
Support Vector Machine			
Spark PCA Example			
Grep			Text Processing
SVDPlusPlus	Train	20GB,11GB,8GB	Graph Processing
Page Rank			
Triangle Count			
Strongly Connected Component			
Linear Regression		18GB,10GB,5GB	Machine Learning
kMeans			
Decision Tree			
Tokenizer			Text Processing
Sort			
WordCount			
RDDRelation			SparkSQL
GroupBy		100 tasks	Shuffle

of the predictions, we take all the data points in the test set for each application and predict their execution times. As we can see in Figure 14, the overhead is negligible, with a maximum latency of 0.551 seconds for the LogisticRegression App that has 10206 points in the test set. Note that this process is run on a laptop but it can be run on a node of the cluster with less load and more computation power, like the master node. We must also consider that we trained the model with sklearn, but there are distributed implementations of boosted regression trees in MLlib[8] for Spark that can speed up the process. We leave as future work when and how the model should be trained.

5.2. Accuracy of predicted times

Now we evaluate the accuracy of the predicted durations. As we mentioned earlier, we built the training and the test set by splitting the benchmark in two different groups. First, we start by evaluating the duration predictions for different stages. To do that, we take all the data points for a stage, we feed them to the model and we average the predictions for each configuration. The results can be seen in Figure 15. For some stages, the predictions are really precise, like the stage 0 of Shortest Path

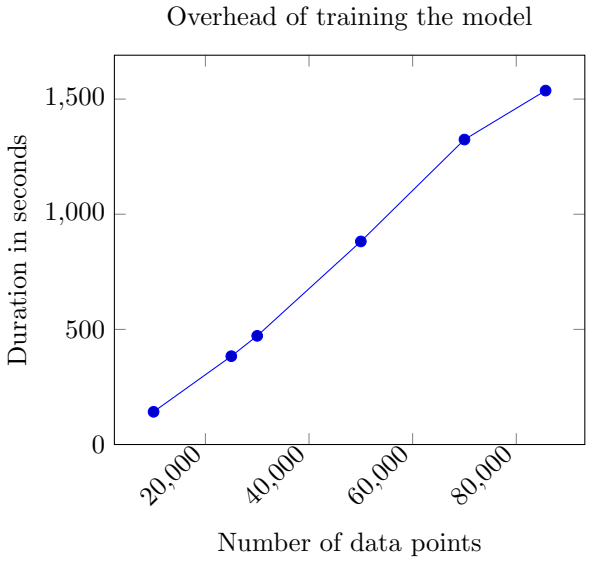

Figure 13: Duration of training the model depending on the number of data points. For the whole dataset of 85664 points the duration is 1536 seconds.

[8]http://spark.apache.org/mllib/ (last accessed Jan 2017)

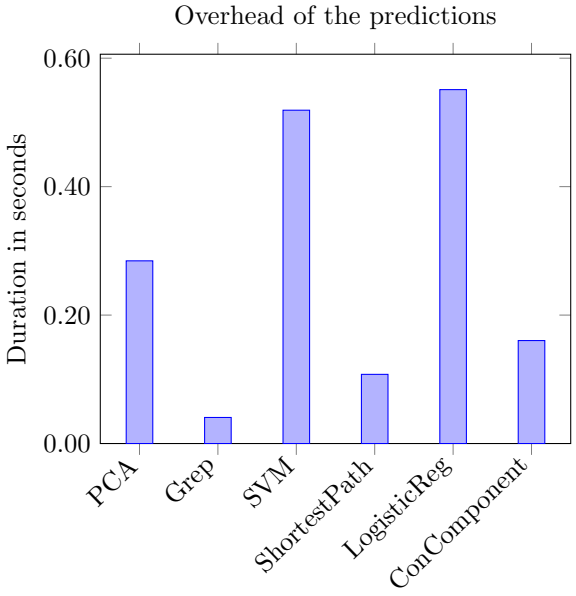

Figure 14: Latency in the predictions for different applications. The latency is always less than 1 second with a maximum of 0.519 for a Logistic Regression App

with 8GB. Some others, like the Stage 1 of Support Vector Machine for 10GB, do not follow the exact same trend as the real duration but effectively detect the minimum. We show here some of the longest running stages across applications, since optimizing their performance will have a greater impact on the overall execution. Remember that our final objective is not to predict the exact run time but rather to know which configurations will affect negatively or positively the performance of an application.

5.3. Accuracy of parallelism recommendations for an application

Now that we have seen the predictions for separate stages, we are going to evaluate the method for a complete application. The objective is to compare our recommendations with a default configuration that a user will normally use and with the best improvement the application can get. We also want to evaluate how the predictions evolve with time, as we add more data points to our database of executions. To achieve that, we are going to assume a scenario where the user launches a set of applications recurrently with different file sizes. If the application has not been launched before, the recommendation will be the default one. After that first execution, the system will have monitored metrics for that application. We can start recommending configurations and the user will always choose to launch the workload with that configuration. In the scenario, for the non-graph applications in the test set, the user executes each one with 5GB, 10GB and 18GB consecutively. For the graph applications we do the same with their 20GB, 8GB and 11GB sizes. In this case, we invert the order, since none of the 20GB executions finished correctly because of out of memory errors and we use them to get the metrics instead. The results of these executions can be seen in Figures 16 and 17. Note how we separate the figures for graphs and non-graph applications. The reason is that with one gigabyte for `spark.executor.memory`, the graph workloads always crashed, so we do not include this default execution. *Best* shows the lowest possible latency that we could get for that application. *Predicted* is the configuration proposed by our model. *Default* is the runtime for a 1g/1core configuration. *Worst* is the highest run time amongst the executions that finished successfully. We do not show here any application that did not complete successfully. For some applications the recommendations achieve the maximum efficiency like Grep with 10GB, Logistic Regression with 18GB or SVM with 18GB. For example in Grep 10GB the maximum improvement is of 11%. This means 11% of savings in energy and resource utilization, which may be significant when these applications are executed recurrently. These recommendations can also help the user to avoid really inefficient configurations, like the one in Grep 10GB. For graph executions, the benefits are more obvious. Since the initial default execution always crashed, the first recommendation by the system was drawn out of the neighbours procedure explained early. The latency for this first recommendation is not optimal, but we have to keep in mind that this is a conservative approach to get a first complete run and a series of traces for the application. After getting this information and applying the boosted model, we can get an improvement up to 50% for the case of connected component with 11GB. We also want to prove that the model can converge to a better solution with every new execution it monitors. Note that we are not talking about retraining the model, but about using additional monitored executions to improve the accuracy. As an example, we keep executing the shortest path application first with 11GB and then with 8GB until the model converges to an op-

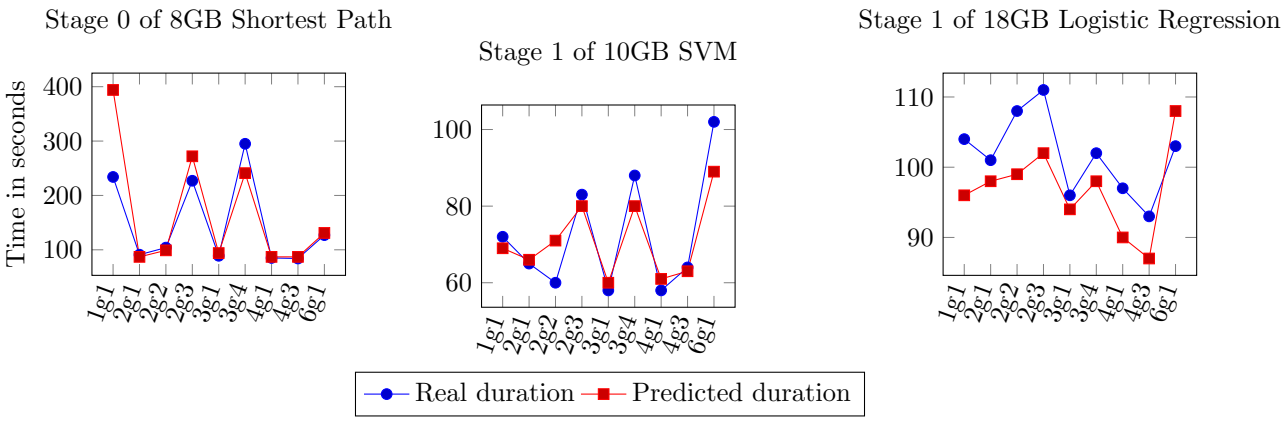

Figure 15: Predictions for stages of different apps with the highest duration and so highest impact inside their applications. For Shortest Path the predictions are pretty accurate. For SVM and Logistic Regression the trend of the predicted values follows the real ones.

timal solution. The evolution can be seen in Figure 18. For the 11GB execution, the first point is the duration for the neighbours approach. After that, the execution recommended with the boosted regression model fails, but the model goes back to an optimal prediction in the next iteration. In the 8GB case, it goes down again from the conservative first recommendation to a nearly optimal one. This is an added value of the model. The system can become more intelligent with every new execution and will help the user to make better decisions about parallelism settings.

Finally, we include in Figure 19 the time that it takes to find an optimal configuration for each application. For the boosted decision trees approach, the average overhead is 0.3 seconds and it increases slightly for those applications that have more stages, like logistic regression. We consider this negligible, since a 0.3 seconds delay will not be noticeable by the user. The kNeighbours approach is faster, but it is only applied to those applications that crash with the default execution of 1g/1core.

5.4. Interpretability of the model

Another advantage of decision trees is that they are interpretable. That means that we can evaluate the impact of certain features on the outcome variable (duration in our case). One of the possible applications is to explain how the number of tasks per node affects a given workload. This information can be valuable when the user wants to manually choose a configuration. For example, in Figure 20 we see the partial dependence of three applications. Shortest Paths is a graph processing application and, as we saw earlier, it benefits from a low number of tasks per node. KMeans sweet spot seems to be around a default 10-12 tasks per node. For PCA, the points between 0 and 5 correspond to stages with only one or few tasks, something very common in machine learning implementations. Again everything from 8 to 15 is a good configuration while more than that, it is counterproductive. Also note how the partial dependence shows us the degree to which parallelism affects that workload. For example in Shortest Paths it ranges from 2000 to -4000 while in kMeans it goes from 100 to -100. Indeed, as we saw in the experiments of the motivation section, the benefit of parallelism is more obvious on graph applications than in kMeans. This feature of a decision tree proves to be useful and can help the user to understand how different parameters affect their workloads.

6. Related Work

There are several areas of research that are related to our work:

Using historical data to predict the outcome of an application: Several lines of research try to optimize different aspects of applications. In [20], the authors present the Active Harmony tuning system. It was born from the necessity to automatically tune scientific workflows and e-commerce applications. The challenge lies on choosing an optimal parametrization from a large search space. To narrow this space, it considers only the parameters that affect performance the most. In contrast, we

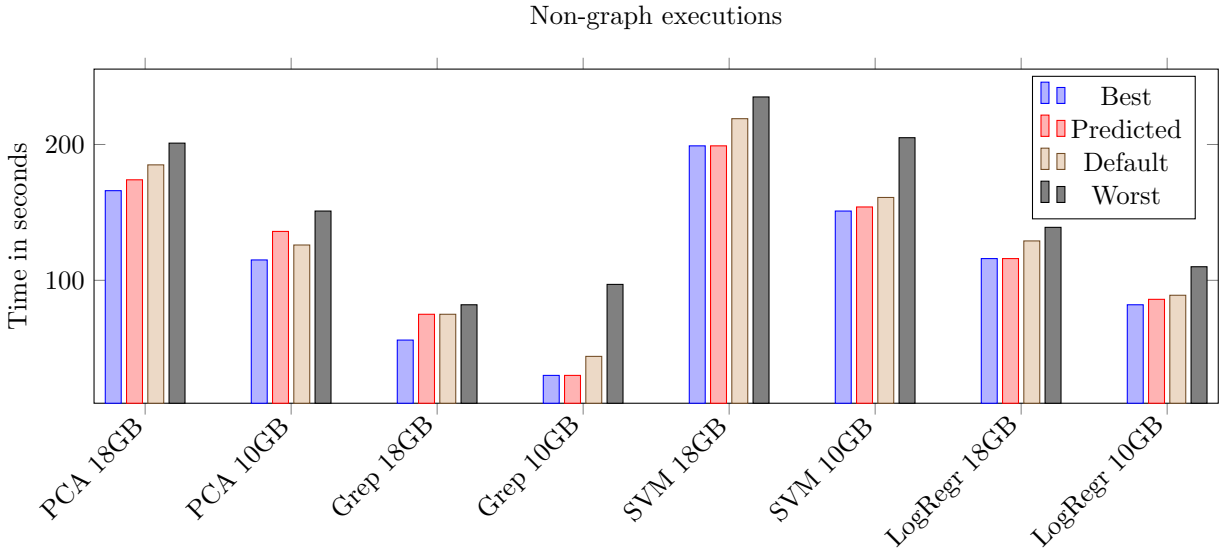

Figure 16: Executions for the non-graph processing applications of the test set. Best shows the lowest possible latency that we could get for that application. Predicted is the configuration proposed by our model. Default is the runtime for a 1g and 1 task slot per executor configuration. Worst is the highest run time amongst the executions that finished successfully

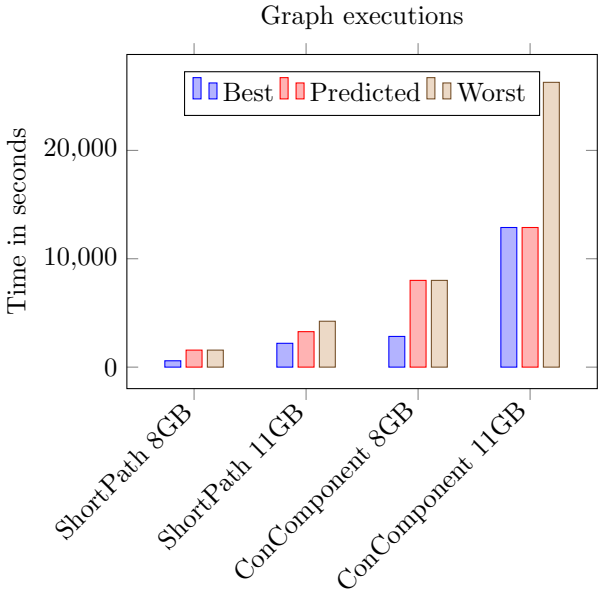

Figure 17: Executions for the graph processing applications of the test set. Best shows the lowest possible latency that we could get for that application. Predicted is the configuration proposed by our model. Worst is the worst running time. All the default configurations crashed for these kinds of applications

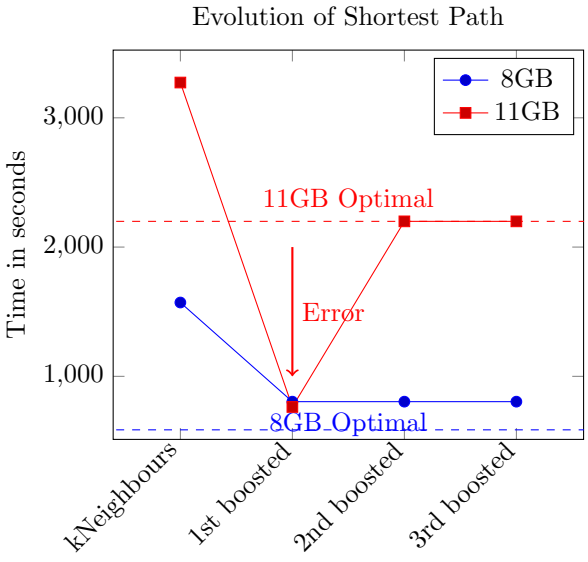

Figure 18: Evolution for the predictions of a recurrently executed shortest path application. From the conservative kNeighbours approach, we go to nearly optimal execution times. Note how the first prediction of the boosted model results in an OOM error

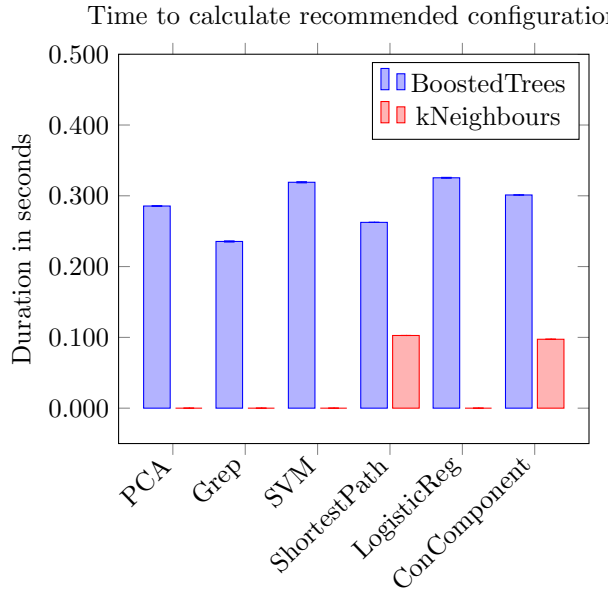

Figure 19: Time needed to calculate a recommended configuration with the boosted regression method and the kNeighbours method for each of the applications in our simulated scenario. Note that some of the applications do not use the kNeighbours method, since it is needed only when the application crashes with the default configuration of 1g and 1 core

do not need to narrow the parameter space, because we evaluate the performance of a configuration through the machine learning module, which has a negligible overhead. Another different approach in [21] is to use a benchmark of HPC workloads from different domains to create a reference database of execution profiles. These profiles are called kernels. The paper presents a framework with the reference kernel implementations, their execution profiles and a performance model already trained that can predict their execution time with other problem sizes and processor. Our solution does not need this reference database already built and it dynamically creates the model with every new execution that comes in. There is work that considers the characteristics of the input data as an input to the learning model. In [22], the authors sample from the data a series of characteristics (e.g number of numerical and categorical attributes of the dataset) that constitute the variables of a linear regression model that predicts the run time of a C4.5 algorithm. However, they do not consider how we can speed up applications or any configuration parameters. Multilayer neural networks were used in [23] to predict execution times on the parallel application SMG2000. It explains a typical machine learning process where data from different executions is gathered into a training and a test set. A problem of neural networks though is that they are not interpretable. This does not suit our objective of assisting the user when using big data platforms.

Optimizing and predicting performance of Hadoop: When Hadoop was the main tool for big data analytics, a series of publications focused on optimizing its default settings. Starfish [24] is an auto-tuning system for MapReduce that optimizes the parametrization through a cost based model. However, this model introduces a profiling overhead and is rigid, meaning that if the underlying infrastructure changes, so should the model. A machine learning approach can adapt itself by training on new data. The authors in [25] propose a method to predict the runtime of Jaql queries. It trains two models for each query type: one to predict the processing speed of the query and a second one to predict the output cardinality. A query is considered similar to another one if they have the same jaql functions. With these two models we can estimate the execution time. This is a really restrictive model, especially to new incoming queries that does not have any other similar observations. Our model does not need any features of a specific domain like jaql but relies instead on more general Spark and machine metrics. It can also be applied to any new incoming observations. ALOJA-ML [26] uses machine learning techniques to tune the performance of Hadoop and discover knowledge inside a repository of around 16.000 workload executions. This allows the authors to evaluate the impact of variables like hardware configurations, parameters and cloud providers. However, Spark works in a different way compared to Hadoop, specially because of its in-memory computing capabilities. In [27], the authors also study how to configure a MapReduce job in addition to calculating the number of virtual machines and its size needed to run that application. It does so by clustering applications that show a similar CPU, memory and disk usage rate. For each cluster, it builds a SVM regression model. In contrast, our approach does not need to cluster similar applications together but generalizes depending on the features of the workload. MROnline [28] provides an online tuning system that is based on statistics of the job and a gray-box smart hill climbing algorithm. The tuning is made at task level. This system is oriented to MapReduce and

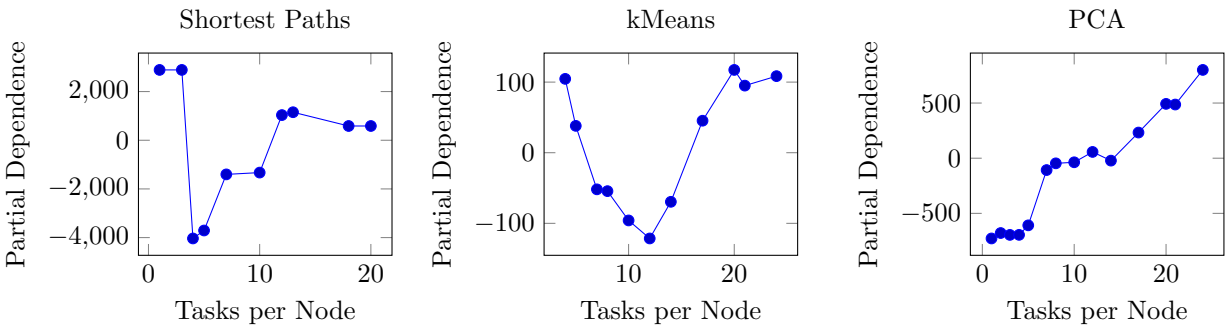

Figure 20: Partial dependence for different applications. Shortest Paths works better with a low number of tasks per node. kMeans best setting is 12 tasks per node. For PCA the points between 0 and 5 corresponds to stages with only one or few tasks, while with many tasks 7 is a good configuration

needs tasks test runs to converge into the optimal solution. Cheng et al. [29] use genetic algorithms to configure MapReduce tasks in each node, taking into account cluster heterogeneity. The parameters are changed, depending on how well a parameter value works in a machine. However, it does not work well for short jobs, since it needs time to converge into a good solution.

Optimizing and predicting performance of Spark: Spark is a relatively new technology and its popularity is on the rise. Tuning up its performance is an important concern of the community and yet there is not much related work. In [30], the authors present, to the best of our knowledge, the only Apache Spark prediction model. Again sampling the application with a smaller data size is used to get statistics about the duration of the tasks and plugged into a formula that gives an estimation of the total run time for a different file size. However, it does not considers tuning any parameters and their effect on duration. MEMTune [31] is able to determine the memory of Spark's executors by changing dynamically the size of both the JVM and the RDD cache. It also prefetchs data that is going to be used in future stages and evicts data blocks that are not going to be used. However, the method is an iterative process that takes some time to achieve the best result and it does not work well with small jobs. It does not learn from previous executions either. In [32], the authors perform several runs of a benchmark and iterate through the different parameters of Spark to determine which ones are of most importance. With the experience acquired from these executions, they build a block diagram through a trial and error approach of different parameters. The obvious drawback is that we have to execute the application several times and follow the process every time the input size changes. Tuning garbage collection has also been considered in [33]. The authors analyse in depth the logs of JVM usage and come to the conclusion that G1GC implementation improves execution times. Finally, in [34], a new model for shuffling data across nodes is proposed. The old model of creating a file in each map tasks for each reduce task was too expensive in terms of IO. Their solution is to make the map tasks running in the same core write to the same set of files for each reduce task.

Task contention: One of the objectives of our model is to detect task contention and act accordingly, by increasing or decreasing the number of tasks. This topic has also been explored in [35]. Bubble up proposes a methodology to evaluate the pressure that a given workload generates in the system and the pressure that this same workload can tolerate. With this characterization, better colocation of tasks can be achieved. Nonetheless, this requires using the stress benchmark in all the new applications that we launch. Paragon [36] takes into account both heterogeneity of machines and contention between tasks when scheduling. By using SVD, it can calculate performance scores of a task executed in a given machine configuration and co-located with another workload. Same as in previous references, it needs benchmarking each time a different workload comes in. Moreover, we are not trying to create a scheduler, but to assist the user in using big data platforms. In [37], the authors propose another scheduler that is based on the assumption that resource demands of the differ-

ent stages can be known in advance. By considering the maximum amount of resources an executor will use, Prophet can schedule JVM's on different machines to avoid over-allocation or fragmentation of resources. This results in a better utilization of the cluster and less interference between applications. This work assumes again that the user already knows the parallelism settings for the application and tries to optimize from there. Our work helps the user to choose a right configuration before sending it to the cluster manager and also gives an interpretation of this decision.

7. Conclusions and Future Work

The results of this paper show that it is possible to accurately predict the execution time of a big data-based application with different file sizes and parallelism settings using the right models. This is a first step in a longer path towards a much higher control of big data analytics processes. It is necessary to reduce complexity and allow centralized management of environments with multiple technologies. To achieve this, the next generation of big data infrastructure solutions need to be vendor-agnostic and user friendly. In addition, processes, monitoring and incident response must be increasingly automated to boost speed and eliminate human error, thereby increasing system uptime.

Regarding our research plans, we aim at applying these concepts to an heterogeneous environment with different machine specifications. This could be done by modeling at task-level, instead of stage-level. We also want to consider choosing these parallelism settings when there are already other applications running in the cluster and modeling their interference. We also plan to focus on mitigating issues related to data skew, where some executors receive more shuffled data than others, causing out-of-memory errors if the JVM is not sized correctly. As the IT arena expands and large workloads execution pave the way for it, it is important to develop concrete methods to reduce the resource wastage and optimize utilization at every stage of execution.

All in all, both industry and academy need to continue collaborating to create solutions that make it possible to manage the lifecycle of applications, data and services. In particular, new solutions will be necessary that simplify management of big data technologies and generate reliable and efficient root cause analysis mechanisms to understand the health of big data systems and to optimize its performance.

Acknowledgment

This work is part of the BigStorage project, supported by the European Commission under the Marie Skłodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] A. Nadkarni, D. Vesset, Worldwide Big Data Technology and Services Forecast, 2016–2020. International Data Corporation (IDC).

[2] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets., HotCloud 10 (2010) 10–10.

[4] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al., The stratosphere platform for big data analytics, The VLDB Journal 23 (6) (2014) 939–964.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, ACM, 2013, p. 5.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center., in: NSDI, Vol. 11, 2011, pp. 22–22.

[7] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, L. Sarzyniec, Adding virtualization capabilities to the Grid'5000 testbed, in: I. Ivanov, M. Sinderen, F. Leymann, T. Shan (Eds.), Cloud Computing and Services Science, Vol. 367 of Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20. doi:10.1007/978-3-319-04519-1_1.

[8] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types., in: NSDI, Vol. 11, 2011, pp. 24–24.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, 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, USENIX Association, 2012, pp. 2–2.

[10] Dynamic allocation in spark, http://spark.apache.org/docs/latest/job-scheduling.html/.

[11] Electrical consumption on grid'5000 lyon site, https://intranet.grid5000.fr/supervision/lyon/wattmetre/.

[12] Managing cpu resources, https://hadoop.apache.org/docs/r2.7.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[13] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, G. Antoniu, Gmone: A complete approach to cloud monitoring, Future Generation Computer Systems 29 (8) (2013) 2026–2040.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[15] R. Kohavi, et al., A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Ijcai, Vol. 14, 1995, pp. 1137–1145.

[16] C. J. Willmott, K. Matsuura, Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance, Climate research 30 (1) (2005) 79–82.

[17] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, J. Zhou, Recurring job optimization in scope, Proceedings of the 2012 international conference on Management of Data - SIGMOD '12 (2012) 805doi:10.1145/2213836.2213959.

[18] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al., Bigdatabench: A big data benchmark suite from internet services, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2014, pp. 488–499.

[20] I. H. Chung, J. K. Hollingsworth, Using Information from Prior Runs to Improve Automated Tuning Systems, Proceedings of the ACM/IEEE SC 2004 Conference: Bridging Communities 00 (c). doi:10.1109/SC.2004.65.

[21] A. Jayakumar, P. Murali, S. Vadhiyar, Matching Application Signatures for Performance Predictions Using a Single Execution, Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015 (2015) 1161–1170doi:10.1109/IPDPS.2015.20.

[22] T. Miu, P. Missier, Predicting the execution time of workflow activities based on their input features, Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012 (2012) 64–72doi:10.1109/SC.Companion.2012.21.

[23] E. Ipek, B. R. De Supinski, M. Schulz, S. A. McKee, An approach to performance prediction for parallel applications, in: European Conference on Parallel Processing, Springer, 2005, pp. 196–205.

[24] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu, Starfish: A self-tuning system for big data analytics., in: CIDR, Vol. 11, 2011, pp. 261–272.

[25] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, A. Ailamaki, Same queries, different data: Can we predict runtime performance?, Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012 (2012) 275–280doi:10.1109/ICDEW.2012.66.

[26] J. L. Berral, N. Poggi, D. Carrera, A. Call, R. Reinauer, D. Green, ALOJA-ML : A Framework for Automating Characterization and Knowledge Discovery in Hadoop Deployments, Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2015) 1701–1710doi:10.1145/2783258.2788600.

[27] P. Lama, X. Zhou, Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud, in: Proceedings of the 9th international conference on Autonomic computing, ACM, 2012, pp. 63–72.

[28] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, N. Fuller, Mronline: Mapreduce online performance tuning, in: Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, ACM, 2014, pp. 165–176.

[29] D. Cheng, J. Rao, Y. Guo, X. Zhou, Improving mapreduce performance in heterogeneous environments with adaptive task tuning, in: Proceedings of the 15th International Middleware Conference, ACM, 2014, pp. 97–108.

[30] K. Wang, M. M. H. Khan, Performance prediction for apache spark platform, in: High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, IEEE, 2015, pp. 166–173.

[31] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, Z. Z. Hu, MEMTUNE : Dynamic Memory Management for In-memory Data Analytic Platforms, Proceedings - 2016 International Parallel and Distributed Processing Symposium (2016) 1161—-1170doi:10.1109/IPDPS.2016.105.

[32] P. Petridis, A. Gounaris, J. Torres, Spark parameter tuning via trial-and-error, in: INNS Conference on Big Data, Springer, 2016, pp. 226–237.

[33] J. Huang, Tuning Java Garbage Collection for Spark Applications Introduction to Spark and Garbage Collection How Java ' s Garbage Collectors Work, https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html (2015).

[34] A. Davidson, A. Or, Optimizing Shuffle Performance in Spark, University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep.

[35] J. Mars, L. Tang, R. Hundt, K. Skadron, M. L. Soffa, Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations, Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11 (2011) 248doi:10.1145/2155620.2155650.
URL http://dl.acm.org/citation.cfm?doid=2155620.2155650

[36] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware Scheduling for Heterogeneous Datacenters, Proceedings of the eighteenth international conference on Architec-

tural support for programming languages and operating systems - ASPLOS '13 (2013) 77–88doi:10.1145/2451116.2451125.

[37] G. Xu, C.-z. Xu, S. Jiang, Prophet : Scheduling Executors with Time-varying Resource Demands on Data-Parallel Computation Frameworks, in: Autonomic Computing (ICAC), 2016, 2016, pp. 45,54.

Table A.3: In this table we summarize the different features we used as input for the machine learning models. The monitor system column refers to the system that gathered the metric, namely Gmone, Slim or None, the latter meaning that this metric was derived from other ones.

Feature/Metric	**Description**	**Monitor System**	**Feature Group**
BytesReadDisk	Number of bytes read from Disk	Slim	Application
BytesWrittenDisk	Number of bytes written to disk	Slim	Application
ExecutorDeserializeTime	Time that the stage spends deserialising the tasks in the executor	Slim	Application
JVMGCTime	Time that the stage spends doing Garbage Collection	Slim	Application
MemoryBytesSpilled	Memory bytes used before spilling to disk	Slim	Application
DiskBytesSpilled	Disk bytes used to spill to disk	Slim	Application
ShuffleBytesRead	Number of inputs Bytes Read from shuffle	Slim	Application
ShuffleBytesWritten	Number of output bytes written for shuffling	Slim	Application
ShuffleReadTime	Time to read all the shuffle input	Slim	Application
ShuffleWriteTime	Time to write all the shuffle output	Slim	Application
TaskCountsFailed	Number of tasks that failed for that stage	Slim	Application
TaskCountsNum	Total number of tasks that need to run for that stage	Slim	Parallelism
TasksThatRunned	Total number of tasks that finished successfully	None(Derived)	Application
totalTaskDuration	Total duration of the tasks in the stage	Slim	Application
spark.executor.bytes	Number of memory bytes allocated to the executor	Slim	Parallelism
spark.executor.cores	Number of cores allocated to the executors. Equivalent to the number of tasks running inside the executors	Slim	Parallelism
tasksPerNode	Number of tasks slots per node	None (Derived)	Parallelism
tasksPerCluster	Number of tasks slots in the whole cluster	None (Derived)	Parallelism
memoryPerTask	Bytes of memory for each task. Is calculated by dividing spark.executor.bytes/spark.executor.cores	None (Derived)	Parallelism
nWaves	Number of waves of tasks needed to complete the stage	None (Derived)	Parallelism
cpu_wait	% of cpu on wait	GMone	System
cpu_usr	% of cpu on usage	GMone	System
cpu_idl	% of cpu that is idle	GMone	System
paging_in	number of bytes paged in	GMone	System
paging_out	number of bytes paged out	GMone	System
io_total_read	number of IO read operations	GMone	System
io_total_write	number of IO write operations	GMone	System
sys_contswitch	Number of system context switchs	GMone	System
sys_interrupts	Number of system interrupts	GMone	System
Disk	Percentage of tasks that read their data from Disk	None (Derived)	Application
Hadoop	Percentage of tasks that read their data from HDFS	None (Derived)	Application
Memory	Percentage of tasks that read their data from Memory	None (Derived)	Application
Network	Percentage of tasks that read their data through the Network	None (Derived)	Application

Appendix A. Table with the variables and metrics used for machine learning

Table B.4: Here we provide a notation table for the different terms we have used through the paper. We also include the section in which the term was first mentioned

Notation	Description	Mentioned in Section
f_{size}	File size of the input file	Section 4.1.1. Parallelism features
b_{hdfs}	Size of the block in HDFS	
min_{yarn}	Minimum size of a container request	
mem_{spark}	Size of the memory to be used by executors	
cor_{spark}	Number of tasks that will run inside each executor	
mem_{node}	This sets the amount of memory that YARN has available in each node	
$over_{yarn}$	The amount of available off-heap memory	
N_{nodes}	The total number of nodes	
$Size_{exec}$	The total size of an executor in Spark	
N_{exec}	The number of executors in each node	
$slots_{node}$	The number of task slots per node	
$slots_{cluster}$	The number of task slots per cluster	
N_{tasks}	Number of tasks needed to process the data	
N_{waves}	Number of waves of tasks needed to process the data	
V_s	Data persistence vector for one stage	
N_{memory}	Number of tasks that read from memory	Section 4.1.2. Application features
N_{disk}	Number of tasks that read from disk	
N_{hdfs}	Number of tasks that read from HDFS	
$N_{network}$	Number of tasks that read from network	
X_{app}	The vector with the features at application level for one stage	Section 4.2. Building the dataset
X_{system}	The vector with the features at system level for one stage	
$X_{parallelism}$	The vector with the features at parallelism level for one stage	
$X_{parallelism_{ref}}$	The parallelism vector under which the other metrics were collected	
$X_{parallelism_{run}}$	The new parallelism conditions under which the stage will be run	
$Y_{duration}$	The duration of the stage	
$listofconfs$	The list of configurations which we will iterate through, in order to find an optimal one	Section 4.3. Using the model

Appendix B. Notation table with the terms used in the text