2016 IEEE 28th International Symposium on Computer Architecture and High Performance Computing

Dynamic Reconfiguration of Data Parallel Programs

Vinícius Dias Wagner Meira Jr. Dorgival Guedes
Department of Computer Science
Universidade Federal de Minas Gerais
{viniciusvdias, meira, dorgival}@dcc.ufmg.br

***Abstract*—Given the large amount of data from different sources that have become available to researchers in multiple fields, Data Science has emerged as a new paradigm for exploring and getting value from that data. In that context, new parallel processing environments with abstract programming interfaces, like Spark, were proposed to try to simplify the development of distributed programs. Although such solutions have become widely used, achieving the best performance with them is still not always straight-forward, despite the multiple run-time strategies they use. In this work we analyze some of the causes of performance degradation in such systems and, based on that analysis, we propose a tool to improve performance by dynamically adjusting data partitioning and parallelism degree in recurrent applications based on previous executions. Our results applying that methodology show consistent reductions in execution time for the applications considered, with gains of up to 50%.**

I. Introduction

The evolution of areas like data mining, machine learning and data analytics, with increased availability of data sources has led to the rise of Data Science as a new way of processing and extracting value from large amounts of data. One strategy that has become popular to process such data is the use of data-parallel frameworks, like Hadoop and Spark. They provide data scientists, or domain experts, with tools that offer high-level abstractions to express complex data processing algorithms in a way that can benefit from a large number or machines, but without requiring them to express nor to handle low-level parallelism tasks. Based on the algorithm description provided by the user in that high-level abstraction, it is the task of the programming environment to find the best configuration to maximize execution performance with good resource usage. In looking for the best performance, those frameworks face three challenges, which reflect dimensions of performance diagnosis:

1) Data partitioning: parallel applications read from distributed data sources (e.g., HDFS) and often use the number of partitions as a first reference for the degree of parallelism to be used. Finding optimal partitioning can be painful because different programs can have different computational costs for the same input. The fact users may inject their own code to be executed by framework operators make costs even more unpredictable. Thus data partitioning is the task of achieving *adequate parallelism* while taking into account data distribution and properties, like its in-memory *layout* [1].

2) Load balancing: from the abstract programs provided by users, frameworks like Hadoop and Spark break computational tasks into stages, delimited by points where synchronization must occur to satisfy data dependencies. Those frameworks try to employ parallelism within every stage, but not among dependent stages. Unfortunately, such design can lead to load imbalance and skewness: even though resources may become idle in some computing nodes, the framework must wait for all the tasks in that stage to complete before it can proceed with computation. This is a challenge because little knowledge is available before execution about data distribution and processing costs.

3) Execution model flexibility: In many cases, a program may require changes of plans and reconfiguration to achieve its best performance over time. An execution model must be able to cope with such dynamic needs. While during a first part an algorithm may handle few data points and can be initially scheduled to execute in a few machines, later that dataset may expand to represent a larger application scenario. The system should be able to detect that and reconfigure itself to use more compute nodes at that point. In frameworks like Spark, that may be hard to accomplish, given that such changes may require changes in the way data is partitioned and may require the re-evaluation of all the execution design [2].

Therefore, considering that data science models and algorithms are irregular and intensive in terms of both computation and communication, performance diagnosis of parallel applications in environments such as Spark is quite a challenge. Finding the right partition for the data at each stage of execution, balancing load during execution and adjusting the environment as application behavior changes are all difficult tasks to be performed by the execution framework.

In this work we present a characterization of the kind of workload we are dealing with through three different massive data parallel applications that we believe represent most of the algorithm patterns found in the area. Next we describe our tool for adaptive reconfiguration of recurrent applications and evaluate our solution against conservative and manual partitioning approaches. The results show consistent gains in performance and the tool flexibility for extending policies enhances its applicability in other scenarios.

II. Data parallel processing execution model

It is usually the case that data parallel frameworks express their computation in terms of the computation applied to data elements. To achieve parallelism, data parallel frameworks partition data and distribute partitions among compute nodes. Programs are then expressed in terms of computation applied

978-1-5090-6108-2/16 $31.00 © 2016 IEEE
DOI 10.1109/SBAC-PAD.2016.32

to data elements. In fact, one can think of a parallel program as a data flow, composed by operators, inputs and outputs.

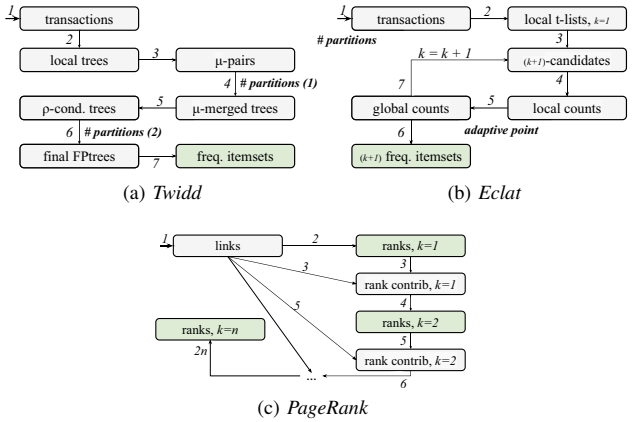
The way operators are combined to build an algorithm, with the output of one operator serving as input to some other operator, can be represented as a directed acyclic graph (DAG). The exact meaning of each node and edge may vary from system to system, but they always express the flow of data and the transformations applied to them. Whichever the representation, DAGs make clear the dependencies along a program execution and are essential to identify when tasks may be parallelized and when synchronization is necessary.

Given is wide acceptance in current big-data scenarios, in this work we consider Spark in our analysis. In Spark, data are stored and processed as collections called Resilient Distributed Datasets (RDDs), which are the edges of the DAG, and the operators are the nodes where transformations occur. Given the type of transformation, it defines how the output data depends on the input data [3]. Transformations may have narrow or wide dependencies, depending on how elements of an new RDD are derived from elements of a previous one. Mapping and filtering are examples of transformations with *narrow dependencies*, since new elements are derived from isolated elements in the previous collection. In that case, operators do not require data shuffling, i.e. network communication. When new data is produced by reductions of joins applied over existing RDDs, data must be reorganized, and global communication is necessary. This is called a *wide dependency*. In Spark, chains of operators with narrow dependencies can be grouped into *stages of execution* by pipelining operators that do not require remote communication. Furthermore, operators with wide dependencies mark the frontier between stages and constitute an opportunity for changing the application parallelism, since data partitioning may be defined anew for the resulting RDDs. We refer to these points as *adaptive points*. The identified stages are handled to the execution engine at the time of job submission, and the engine is responsible for resource allocation, scheduling of stages and coordination within the application.

In that case, resource allocation refers to the process of requesting and allocating *executor* nodes on behalf of the application. The executors have their share of cluster cores and memory for task execution. In essence, stages are scheduled according to wide dependencies and are composed by several tasks that execute the same set of commands.

III. Selected applications

Our workload for evaluation is composed by three algorithms: (i) Twidd, a parallel implementation of FPGrowth [4], a *non-iterative algorithm with complex data structures*; (ii) Eclat [5], another frequent pattern mining algorithm, like Twidd, but an *iterative, irregular* one, where the amount of computation varies per iteration; (iii) PageRank, an *iterative, regular* algorithm, which executes basically the same computation on every iteration. Our goal was to cover common application patterns present in machine learning algorithms,



(a) *Twidd* (b) *Eclat*

(c) *PageRank*

Fig. 1. Selected applications

like iterativity and/or regularity, and also straightforward data analytics routines, i.e., when these characteristics are absent.

A. Twidd

Twidd is an implementation of FPGrowth over RDDs. It solves the problem of frequent pattern mining, which is: given transactions, each one composed by sets of items, and a support threshold *minsupp*, find all subsets of items (itemsets) that occur in more than *minsupp* transactions.

An overview of Twidd is shown in Figure 1a. Briefly, the algorithm *(1)* reads transactions, *(2)* builds FPTrees over these transactions, *(3,4)* rebalances them in the cluster, *(5,6)* performs a pre-projection of itemsets and finally, *(7)* searches for the remaining frequent itemsets. For the sake of our discussion it is sufficient to state that the algorithm has two main *adaptive points* in its DAG, highlighted in the figure as *# partitions(1)* and *# partitions (2)*.

B. Eclat

Eclat is another approach for frequent pattern mining. It works with a vertical database layout for fast candidate counting by intersecting inverted lists of transactions. We adopted a local counting strategy [6] to implement Eclat over the RDD abstraction.

Figure 1b illustrates the overview of our implementation. The algorithm *(1)* reads transactions, *(2)* verticalizes the base, *(3)* generates candidates by intersecting itemsets, *(4)* outputs local counts, *(5)* aggregates local counts into global counts and filter the original set of itemsets keeping only the frequent ones. The algorithm continues with other rounds of these same steps until no frequent itemset is found. For the sake of our discussion it is sufficient to state that the algorithm has one main *adaptive point*, highlighted in the figure.

C. PageRank

PageRank is a link analysis algorithm that computes the relative importance of nodes in a network. It does so by assigning initial ranks to every node and iteratively updating each value based on the node's neighborhood. Initially, the rank of every node is set to 1.0 and on each iteration, every

Application	*Dataset*	*Size*	*Blocks*	*Details*
Twidd/Eclat	twitter	7.9 GB	63	# transactions: 233mi # distinct items: 64mi
PageRank	gplus	9.3 GB	75	# nodes: 35mi # edges: 575mi

TABLE I
ALGORITHMS AND DATASETS

node divides its own rank among its neighbors. Thus, a node with rank r and n neighbors would share $\frac{r}{n}$ of its own rank with each neighbor. Naturally, every node receives rank shares from all its neighbors and sum up those values to build a new rank. The process continues for a number of iterations or until values converge. Figure 1c illustrates the algorithm.

Despite the number of input partitions, PageRank has *adaptive points* on every iteration, as we have to *(steps 3, 5, etc.)* join ranks and links to produce contributions and *(steps 4, 6, etc.)* reduce these contributions per node.

IV. CHARACTERIZATION

We ran our experiments in a cluster with 9 machines, each a quad-core Intel Xeon X3440 with hyperthreading and 8 MB cache, 16 GB RAM, a 1 TB 7200 RPM SATA disk, running 64-bit Linux 3.2.0. The nodes are connected with Gigabit Ethernet. The cluster is configured with Spark v1.5.1 and Hadoop/HDFS v2.5.0. We used two real world data sets in our evaluation: a set of Twitter posts, containing tweets crawled using the Twitter API; and a Google+ graph representing users (nodes) and friendships (edges) [7]. All datasets were loaded into HDFS with a replication factor of 2. Table I summarizes the setup for algorithms and datasets.

A. Factorial design

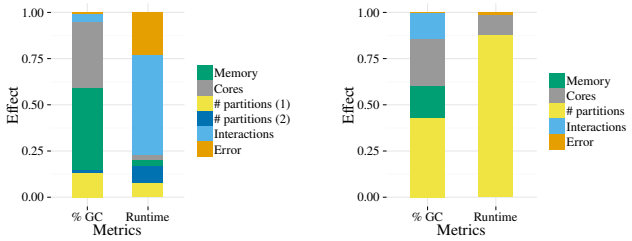
Full factorial designs are used to estimate the effects of factors in a system w.r.t. a given response variable (metric). The output is a model for each metric that associates metric variations to factors, their interactions and any error, which indicates variations not accounted to by the factors.

We adopt a 2^k factorial design in order to estimate the effects of key parameters in Spark applications under different circumstances. We evaluate Twidd and Eclat in a cluster with 9 machines. The factors were chosen based on common application parameters for each (executors memory and cores) and number of partitions in different steps on each algorithm, as highlighted in Figures 1a and 1b. The factors/levels used to configure each algorithm are summarized in Table II. Our goal was twofold: to see how each factor affected the expected execution time of the algorithms (metric run-time), and to understand how they affected the amount of time spent on the garbage collector (metric GC).

Twidd	*Exec. Mem.*	*Exec. Cores*	*# Partitions (1)*	*# Partitions (2)*
	7 GB	4	257	257
	14 GB	8	1021	1021
Eclat	*Exec. Mem.*	*Exec. Cores*	*# Partitions*	
	7 GB	4	257	
	14 GB	8	1021	

TABLE II
2^k FACTORIAL DESIGNS

Figure 2a illustrates the results for Twidd. There is little error (1%) in GC, so the time spent in the garbage collector is well defined by the factors considered. Also 80% of GC's variation is explained by memory and cores in isolation. This is expected: more memory and less cores imply in larger heap space to collect with less computational power. Different from GC, run-times are subject to error, indicating variations that could no be explained by the factors considered. Twidd make use of complex data structures, which increase garbage collection time [8], and GC itself has been recognized as one major factor that may lead to large variations in parallel applications performance [9].



(a) *Twidd* (b) *Eclat*

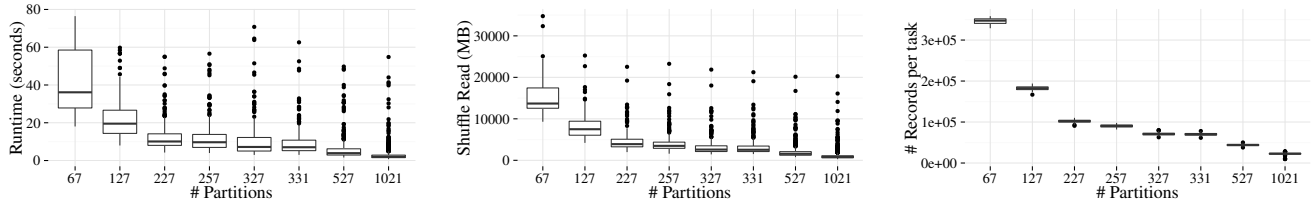
Fig. 2. Performance effects (*significance level* = 0.05). 9 machines, where one is always the execution master.

The results for Eclat are shown in Figure 2b. Error rates are low, even for run-time, so factors are determinant in all cases. Indeed, garbage collection overhead in Eclat is much lower than in Twidd, due its use of simpler data structures and arrays. Memory and cores in isolation continue to play important roles in % GC. However, the number of partitions is the factor that most impact the metrics (88% of variations). Thus, partitioning has a great impact in Eclat's performance and this fact highlights the potential importance of adaptive execution in such cases.

B. Load balancing

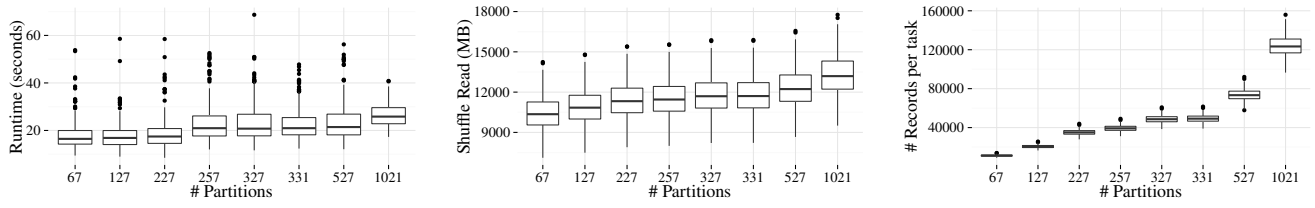
We discuss load balancing by looking closely at Twidd's 4th and 5th stages (Figure 1a), as we alter data partitioning by varying the number or partitions in the input of stage 4 (*#partitions(1)*), which also affect stage 5.

Figure 3 shows additional metrics regarding Twidd's 4th stage. The x-axis refers to the number partitions in the 4th stage's input. Note that skewness in running times tends to increase as more partitions are used. If we observe the amount of data in the shuffle reads we find a similar behavior, *i.e.*, the amount of data each task reads from the shuffle becomes more skewed as the degree of parallelism increases. These results suggest that run-time grows together with the volume of data in the shuffle read. Indeed, Pearson and Spearman correlations between those two dimensions are 0.94 and 0.95 respectively. However, by looking at the number of records processed per task, we see almost no outliers, which means that the partitioner is dividing work equally based on the number of records from the shuffle write of the previous stage that are assigned to each node. Thus, load imbalance is coming



(a) Runtime (b) Shuffle Read (c) Records

Fig. 3. Behavior of Twidd's 4th execution stage as we vary *#partitions(1)*. Most of the imbalance is explained by the uneven data reads over a shuffle step. Run-time and shuffle read correlations: Pearson = 0.94; Spearman = 0.95



(a) Runtime (b) Shuffle Read (c) Records

Fig. 4. Behavior of Twidd's 5th execution stage as we vary *#partitions(1)*. Most of the imbalance is inherent to the algorithm. Run-time and shuffle read correlations: Pearson = 0.19; Spearman = 0.23

from the fact that records have different sizes and, therefore, variable processing costs.

The takeaway is that imbalance cannot always be mitigated by increasing the number of partitions of a given stage. In such cases, hash partitioning (which is the default in Spark) cannot capture and distribute the workload equally; arbitrarily increasing partitioning could result in anomalous cases [1].

A different result was found in the analysis of Twidd's 5th stage, shown in Figure 4. The running time median cannot be estimated from other metrics at any degree of parallelism, which means that tasks have variable processing costs (Fig. 4a). At this point we could not claim this is due to bad partitioning or inherent to the algorithm. However, by looking at the distribution of tasks' shuffle read volumes (Fig. 4b) and number of records (Fig. 4c) we conclude, however, that the median better represents those metrics (boxes are symmetric, except for outliers in the number of records). Thus, the cause for run-time skewness in the 5th stage is not naive partitioning, but inherent to the algorithm.

V. Adaptive execution

A stage of execution in Spark can be described in terms of its function, data inputs, data outputs, shuffle reads and shuffle writes. Many machine learning and graph processing algorithms are based on iterative and/or converging approaches. When translated into DAGs, these techniques expose opportunities for re-execution optimization. Indeed, the partitioning scheme (number of partitions and data placement) can be changed in the beginning of every stage, because data are being reorganized in every shuffle. In case of iterative algorithms, the logic of one iteration certainly will be applied to another context later on. By tying up program's logic with the data it consumes/produces, one can classify groups of stages into:

1) *Equal* (P_{eq}), i.e., they execute the same function with the same ratios of inputs/outputs.
2) *Similar* (P_{sim}), i.e., they execute the same function with different ratios of inputs/outputs.
3) *Unrelated* (P_{un}), i.e., share few or no properties.

The approach adopted by PageRank produces the same communication pattern every iteration: graph nodes share rank contributions with their neighbors. Therefore, iterations stages are P_{eq}. One simple way to optimize P_{eq} stages, given that we find the right amount of parallelism for an earlier iteration, is to simply use that knowledge for later iterations.

On the other hand, we also note algorithms that re-execute stage functions several times but in different contexts. Therefore, the problem has the potential for growing/shrinking over time. Despite that, historical information about previous executions could help the scheduler to optimize upcoming steps by estimating the desired parallelism. Eclat, for example, generates different contexts on each iteration, due to its combinatorial nature. Thus, its iterations are P_{sim}. In that case, it is erroneous to assume that the same parallelism employed initially would guarantee the same gains over time. Thus, we lack criteria to apply to Eclat, for example, the same approach we used in PageRank. The application should be able to estimate new configurations based on the common knowledge of re-executions. In this context, the common knowledge is that we ran the same stage under certain circumstances (inputs and outputs) and got some cost associated to it (e.g., run-time). Then, it becomes a matter of, given another set of inputs and prior knowledge, find the proper tuning for this new setting.

The last category (P_{un}) refers to stages that are completely unrelated during a single execution, Twidd fits this category. In this case the only information that can be leveraged is between

executions. However, it can be advantageous to learn from past events if the jobs are recurrent.

Next we present how to capture these behaviors and tune applications based on information about previous executions.

A. Reconfiguration Tool

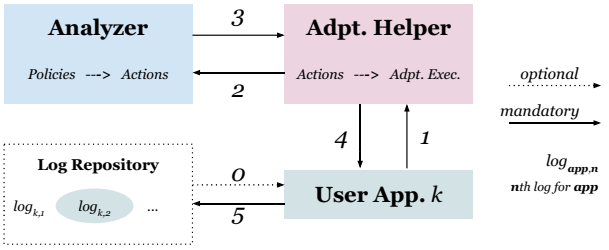


Fig. 5. Tool architecture and overview

Figure 5 shows the design and overview of the tool developed in this work. The user application starts with or without log information from a previous execution *(0)*. We leverage the standard log information provided by the Spark framework, which is based on run-time events and associated metrics. If no log information is provided the application executes as usual and registers the first log, which can serve as a feedback for future executions. The system internally instantiates an *Adaptation Helper* (section V-C) *(1)*, which then initializes and asks the *Analyzer* (Section V-B) to parse and make some decisions based on the observed statistics *(2)*. Those decisions are returned to the helper as *actions* to be applied in the current execution *(3)*. The helper then translates these action into properly partitioned RDDs back to the application *(4)*. Finally the current log is placed in the repository.

B. Analyzer

The purpose of the *Analyzer* is to receive execution logs from the *Adpt. Helper*, identify points for adaptation in the application, apply the configured policies based on the data collected and finally deliver a set of actions back to the *Adaptation Helper*. Actions in that context indicate reconfigurations in application's partitioning over time. The analyzer also captures the communication pattern of the application, i.e., it makes decisions taking into account whether the application is regular and/or iterative. Thus we employ the same premisses introduced in section V.

The current version of our tool supports three types of action: (`UNPAction`) indicates that the number of partitions of an adaptive point must be updated; (`UPAction`) indicates that the partitioning strategy (a.k.a. partitioner) must be changed; and (`WarnAction`) handles the cases where some issue was detected but any automatic action to mitigate the problem is known. For instance, `UPAction` can request the change of partitioner from *hashPartitioner* to *rangePartitioner*. On the other hand `WarnAction` can indicate to the user some problem that requires his intervention, like identifying the source of imbalance (Section IV-B).

Which action to choose depends on the current set of deployed policies in the analyzer. As a testbed, we created some policies regarding the discussed bottlenecks. It is important to point that the following policies are not extensive but enough to demonstrate the applicability of our framework:

a) ***Empty Tasks*** *(***ET***):* if there are tasks that process no data, we remove them from the number of partitions. That may lead to an update to the number of partitions at the adaptive point (`UNPAction`).

b) ***Memory/Disk Spill*** *(***SP***):* if the logs indicate a task has to perform data spill (save data for later processing due to space limitations), we try to reduce the load of each task by a factor of how much spill needed to be done in order to shuffle write the data to next stage. The potential result is an action that increases the number of partitions (`UNPAction`).

c) ***Garbage Collection*** *(***GC***):* in this case we first use the concept of skewness to determine whether garbage collection is an issue or not. If this is the case then we split each task proportionally to the minimum GC overhead observed. We refer to GC overhead as the fraction of task's run-time in which its executor was collecting garbage. The result is also an action that updates the number of partitions (`UNPAction`).

d) ***Task Imbalance*** *(***TI***):* this action automates our methodology of the load balancing analysis from Section IV-B. We consider sources of imbalance being: *(a)* inherent; *(b)* due key distribution; or *(c)* due variable costs for the same key. Due limitations of the environment, we are unable to adapt and handle some kinds of imbalance automatically in most situations. Most cases of imbalance output actions of warning (`WarnAction`) except for imbalance from key distribution which indicates that *rangePartitioner* may be a solution (`UPAction`).

The system also supports incorporating new policies to the analyzer. To accomplish that the user needs to make an extra call in its program: `preAdapt(`*sc*`,` p_1`,` p_2`, ...)`, where p_i is a policy function with the following definition:

```
def myPolicy(rdd: RDD,
             stages: List[Stage]): Action
```

In this definition `rdd` contains all the information about the RDD representing an adaptive point. Furthermore `stages` contains the list of all stages that begins in that adaptation point. For simplicity, in our presented policies, we chose the oldest created stage to extract and analyze its metrics on behalf of the adaptive point.

C. Adaptation Helper

The adaptation helper (*Adpt. Helper*) couples user applications with their respective decision makers (the *Analyzer*). Its main function is to translate actions delivered by the *Analyzer* into real modifications in the current execution, transparently to the user. Figure 6 summarizes the tool API. Basically the user can feed it with previous execution information (*log feedback*), pre-adapt and/or include new policies (*app init*) and call operators aware of adaptive points (*mod. operators*).

The user interacts with the *Adpt. Helper* through overwritten Spark operators that are aware of adaptive points. Fortunately

Log Feedback:	`conf.set("spark.adaptive.logpath",` $log_{app,n}$`)`
App. Init:	`preAdapt(`sc`,` p_1`,` p_2`, ...)`
Mod. Operators:	`textFile, reduceByKey, aggregateByKey, join, partitionBy` with an extra parameter ap, i.e., adaptive point.

Fig. 6. Tool API

the operators that represent adaptive points are well defined in the system (*reduceByKey*, *textFile*, *join*, etc.), i.e. any operator that allows setting the number of partitions and/or partitioner. Our system leverages scala implicits to redefine these special operators. Therefore the only requirement for using them is to explicitly import their definitions (*AdaptableFunctions*).

The *Adpt. Helper* can be instantiated (atomically) by one of two events. First if some overwritten operator is being called for the first time. Second if the user called `preAdapt` before to include their custom policies, for example. Also, the *Adpt. Helper* triggers the *Analyzer* initialization and execution. At this moment the helper potentially has all actions delivered by the analyzer associated with their respective adaptive points.

Given that actions are available, every time the user code executes some of these overwritten operators the system verifies if there is action to be applied for the current adaptive point. If this is the case, the new RDD is created considering the semantic of the action, like update the number of partitions or the partitioner. Otherwise, the execution continues with the default behavior. Therefore our solution is totally safe and does not break existing implementations.

Finally we had to settle two design decisions: *(1)* how logs are forwarded to the *Analyzer* and *(2)* how adaptive points are distinguished from each other. For the first decision we require the user to inform the log path of a previous execution by setting the configuration `spark.adaptive.logpath` in the *SparkConf*. For the second decision we work with two alternatives. We support for most of the operators an extra parameter that identifies the adaptive point. If no identifier is provided the system consider a default name composed by the operator and the line of user code it was called. We plan to include automatic tracking of applications and logs in the future.

D. Wordcount example

To illustrate the use of the tool, Figure 7 represents a Word Count application in Spark that uses it. The bold parts highlight the differences in comparison to a common implementation. Note that we *(a)* import the special functions (*line 1*); *(b)* set the feedback log of a previous execution in the *SparkConf* (*line 5*); *(c)* pre-adapt before execution without custom policies (*line 7*); and *(d)* explicitly choose an adaptive name for the reduce phase (*line 10*).

It is important to mention that from all of these changes only *(a)* and *(b)* are mandatory. In fact the user could choose to not pre-adapt its application at line 7 and then this would be done at the time of *reduceByKey* call. Furthermore, as discussed, the

```
1 import br.ufmg.cs.systems.sparktuner.rdd.AdaptableFunctions._
2 object Wordcount {
3   def main(args: Array[String]) {
4     val conf = new SparkConf().setAppName("Word Count").
5         set("spark.adaptive.logpath", "/tmp/log1")
6     val sc = new SparkContext(conf)
7     preAdapt(sc)
8     val counts = sc.textFile ("/tmp/sample").
9         flatMap (_ split " ").map (w => (w,1)).
10        reduceByKey (_ + _, "ap-counting")
11    println (counts.count + " words")
12    sc.stop()
13  }
14 }
```

Fig. 7. Adaptable Wordcount

system provides a default adaptive name for known operators, like the one used for reduction in this example. Thus setting "ap-counting" at line 10 is also optional.

VI. Evaluation

Our environment for evaluation is the same used for characterization (Sec. IV). Here we evaluate our tool considering the selected applications and the issues found. The following performance measurements include the time spent by the *Analyzer* to parse and decide actions for the applications, in case of experiments using the tool. However, in all settings, we observed no more than 3 seconds for this overhead in our naive implementation of the *Analyzer*. Section VI-A presents the results for PageRank, which accounts for *iterative* and *regular* applications. Section VI-B presents the results for Eclat, i.e., an *iterative* and *irregular* application. Finally Section VI-C presents discussions about Twidd's executions, representing *non-iterative* applications.

A. PageRank

We evaluated PageRank under several degrees of parallelism for the input partitions. Figure 8 shows the results. Initially, we ran the algorithm with different degrees of parallelism. We refer to that as *Conservative* execution because the number of input partitions, derived from the number of blocks in HDFS, is used naively as the default degree of parallelism for the reductions in every iteration. Then, after identifying that 1024 partitions would speedup the reductions, we re-ran the tests with different degrees of parallelism for the input partitions, but limiting parallelism during the iteration. We refer to that as *Adaptive.Manual* because we change the number of partitions of the reduce phase regardless of the input but we do so by manually setting this parameter in the code. The third category is the one obtained by our tool (*Adaptive.Tool*), in which we used the logs from conservative approach to optimize partitioning automatically.

Note that with *Adaptive.Manual* we improved overall performance, obtaining speedups of 2.8x, 2.2x, 1.7x and 2.3x in the first four configurations. The results with 1024 partitions are exactly the same, since it was our baseline. Finally, we

observe no gain when the number of input partitions exceed the cost of reduction (with 2048 partitions). More important, the results using *Adaptive.Tool* were able to keep up with the best case scenario of *Adaptive.Manual*, which shows that our tool and sample policies were effective, with the bonus of automatic adaptation.

The execution with 256 input partitions show an interesting fact related to the tool's operation. In its current implementation, the tool applies only one action w.r.t. only one policy per adaptive point each time it runs. In some cases, the first reconfiguration may lead to a second bottleneck becoming prevalent. We can see that the resulting time for 256 input partitions when we ran *Adaptive.Tool* once with the log information from the *Conservative* execution as input was worse than the conservative execution in a first round (notice the number or partitions chosen, 611, was close to 512, and so the performance was close to that conservative case). The action recommended by the *Analyzer* was to increase the number of partitions based on a high observed spill, i.e., an `UNPAction` that increased the number of partitions based on the policy **SP** (Section V-B). That solved the spill problem, but the new execution suffered from another bottleneck, garbage collection overhead. A second round of execution using our tool with the log of the first round was able achieve the desired optimal performance. In this case the action returned by the *Analyzer* was to increase the number of partitions (`UNPAction`) but w.r.t. the policy **GC** from Section V-B. The bar *4.Adaptive.Tool.Iteration.2* shows that last result.

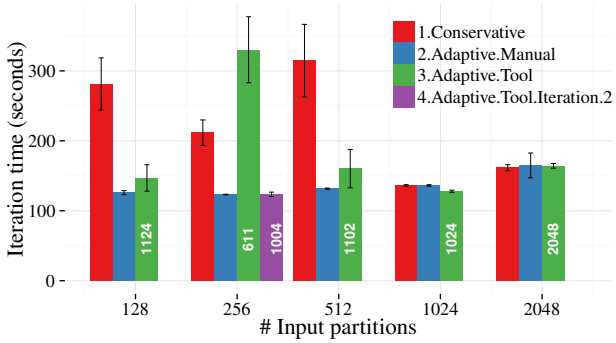


Fig. 8. Optimizing P_{eq}, i.e., stages that repeat with the same communication pattern. Adaptive re-executions (iterations) can leverage information about the previous degrees of parallelism. Numbers inside bars indicate how many partitions some iteration of the tool has decided for the reduce phase.

B. Eclat

We evaluate Eclat varying the degree of parallelism of *adaptive point* in Figure 1b. Figure 9 shows our results for three scenarios. As before, *Conservative* refers to maintaining the number of input and reduce partitions the same on every iteration, *Adaptive.Manual* considers a heuristic based on the specific behavior of the algorithm, and *Adaptive.Tool* shows the results for our tool, having *Conservative* logs as input.

To build *Adaptive.Manual*, we consider a heuristic that is used to follow the elasticity of Eclat's problem size. We observe the selectivity factor of the first operator before the shuffle took place, i.e., a factor of $selec = \frac{input}{shuffleWrite}$. This factor is used to tune the parallelism for the reduction in the first iteration. We set the number of reducer partitions to $nparts = \max(\lceil \frac{inputPartitions}{selec} \rceil, totalCores)$, setting the number of cores as a low limit. The remaining iterations are tuned proportionally to $shuffleWrite$ with $nparts$ of the first iteration.

The policy applied by the *Analyzer* in *Adaptive.Tool* is the one related to empty tasks considering the elastic behavior of the application. It makes no assumption specific to the algorithm of the application.

Both adaptive solutions are specially efficient on fixing under and overestimations in the number of partitions (75, 1024 input partitions). The gain is small in intermediary values due to the approximation to the optimal parallelism, in which candidate generation (map phase) overcome the global counting (reduce phase). Also, every result obtained by our tool (*Adaptive.Tool*) outperforms or is at least equivalent to the manual adaptation, with advantage of being an automatic approach.

Adaptive execution also has the capacity to reduce the number of launched tasks in comparison to the conservative approach, therefore reducing the total amount of resources used during each step. We indicate this gain inside the bars as the percentage of reduction in the launched tasks for the reduction stages w.r.t. the conservative approach. Note that the number of tasks launched is reduced by at least 78.4% for *Adaptive.Manual* and by at least 73.6% for *Adaptive.Tool*, even in cases where run-time gains are modest.

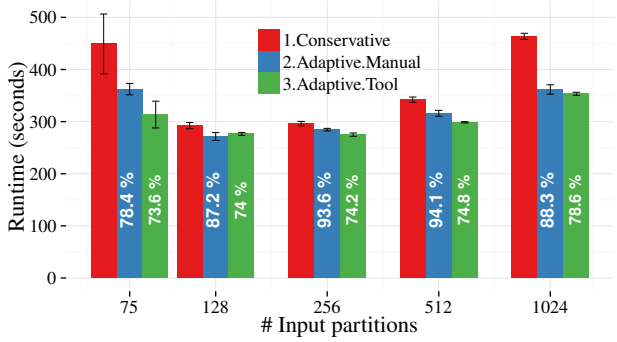


Fig. 9. Optimizing P_{sim}, i.e., stages that are repeated with different inputs. The degree of parallelism must adapt to the elasticity of the application. Percentages inside bars represent reductions in the number of launched tasks relative to the Conservative approach.

C. Twidd

Figure 10 shows the warn messages generated by the tool when a Twidd log discussed in Section IV-B is used. Note that it captured the same sources of imbalanced discovered in that analysis, i.e., stage 4, where *muTrees* are merged, suffers from imbalance due variable costs for the same key, and stage 5, where *rhoTrees* are merged, suffers from imbalance inherent to the application. As those issues cannot be handled trivially

```
WARN OptHelper: WarnAction(muTrees, VariableCost)
   (task-imbalance,257)
WARN OptHelper: WarnAction(rhoTrees, Inherent)
   (task-imbalance,257)
```

Fig. 10. Tool warnings

by the current version of Spark's execution engine, the actions are merely warnings, since user's intervention is necessary.

VII. Related Work

Several works propose job optimization for data-parallel frameworks [10], [11]. Those systems work with structured data (SQL-like), where the knowledge about data organization exposes more opportunities for optimization. Our work regards general purpose data processing, where one have little or no prior information about the inputs.

Optimization of recurring jobs can be made by creating a statistics repository of applications executed in the cluster as done in Scope [12]. The approach in that case aims at future re-executions and assumes that similar jobs process data with similar properties. Thus, it is orthogonal to our discussion.

Automatic reconfiguration of recurring jobs is another option in performance optimization. Starfish [13] is able to construct job profiles based on previous executions. Then, it explores the space of Hadoop internal configurations (using a *What-If* engine) to find the best setup for later jobs. However, Starfish optimizes just static parameters that must be set prior to the application submission and cannot be changed after that. Therefore, this is also orthogonal to our approach.

Adaptive query execution is an ongoing issue in the Spark project that proposes improvements to the run-time engine [2]. The main idea is to postpone stage submissions to allow partitioning focused on statistics collected in earlier stages. That new feature, which aims to optimized general cases, could benefit from extra knowledge of iterative algorithms and stage re-executions, as discussed here.

Most of the data parallel frameworks run on garbage-collected languages and this is the case for Spark with the Java Virtual Machine (JVM). Several algorithms for garbage collection suffer from long GC pauses and memory overheads. Broom [14] uses a region-based memory management that co-locates objects having the same collection footprints, which avoids large heap scans every time a full collection occur. Holistic approaches [9] plan to coordinate collections in a distributed environment. The latter could improve scheduler's decisions by providing worker state awareness.

Memory management in Spark is moving towards a manual, optimized memory layout [15], which would remove garbage collection for critical steps of execution w.r.t. memory, like aggregations. Despite that, optimizers could still benefit from executor load feedback, apart from this knowledge coming from virtual machines or manual instrumentation.

VIII. Conclusion

Data-parallel frameworks simplify the task of data scientists, allowing them to write parallel applications using high-level abstractions. However, they still face some challenges, like to create self tuning systems in scenarios where the user has total freedom to write any code.

We characterized sources of inefficiency in three representative applications, showing the impact of partitioning to applications performance and also that in order to handle imbalance one must identify one of its several potential causes. Then we presented the concept of adaptive execution and how the communication pattern of the applications can be used to optimize their performance. Finally, with all that background, we built a tool for automatic reconfiguration of recurrent applications. Our solution is configurable, compatible with any Spark core application and allows pluggable policies for bottleneck mitigation. The tool receives as feedback logs of previous executions and automatically decides new partitioning parameters for the adaptive points. We evaluated our tool against a conservative and an adaptive/manual approaches. We were able to observe consistent improvements w.r.t. the conservative approach and fairly equivalent results when compared to manual adaptation. We believe that our work is a step towards self tuning data parallel systems.

Acknowledgements

This work was partially funded by Fapemig, CNPq, CAPES, and by projects InWeb (MCT/CNPq 573871/2008-6), MASWeb (FAPEMIG-PRONEX APQ-01400-14), and EUBra-BIGSEA (H2020-EU.2.1.1 690116, Brazil/MCTI/RNP GA-000650/04).

References

[1] V. S. Dias, R. E. A. Moreira, W. Meira Jr., and D. Guedes, "Diagnosing performance bottlenecks in massive data parallel programs," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, May 2016.

[2] M. Zaharia, "Adaptive execution in spark," 2015. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-9850

[3] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI 12*. San Jose, CA: USENIX, 2012, pp. 15–28.

[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[5] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *KDD '97*, 1997.

[6] A. Veloso, W. Meira, Jr., R. Ferreira, D. Guedes, and S. Parthasarathy, "Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining," in *PKDD '04*, 2004.

[7] G. Magno *et al.*, "New kid on the block: Exploring the google+ social graph," in *IMC '12*, 2012.

[8] K. Nguyen *et al.*, "Facade: A compiler and runtime for (almost) object-bounded big data applications," in *ASPLOS '15*, 2015.

[9] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *HotOS XV*. USENIX Association, May 2015.

[10] Q. Ke, M. Isard, and Y. Yu, "Optimus: A dynamic rewriting framework for data-parallel execution plans," *Proceedings of the 8th ACM European Conference ...*, pp. 15–28, 2013.

[11] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI'12*, 2012.

[12] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou, "Recurring job optimization in scope," in *SIGMOD '12*, 2012.

[13] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, 2011, pp. 261–272.

[14] I. Gog *et al.*, "Broom: Sweeping out garbage collection from big data systems," in *HotOS XV*. USENIX Association, May 2015.

[15] R. Xin, "Project tungsten," 2015. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-7075