Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Janus: Diagnostics and reconfiguration of data parallel programs

Vinicius Dias *, Wagner Meira Jr., Dorgival Guedes

Department of Computer Science, Universidade Federal de Minas Gerais (UFMG), Brazil

HIGHLIGHTS

- Performance diagnosis dimensions are proposed as an evaluation methodology.
- Applications representing common communication patterns are characterized.
- An extensible tool is proposed for reconfiguration of Spark applications.

ARTICLE INFO

Article history:
Received 25 April 2017
Received in revised form 14 November 2017
Accepted 26 February 2018
Available online xxxx

MSC:
68M14
68M20

Keywords:
Performance diagnosis
Dynamic reconfiguration
Spark

ABSTRACT

The increasing amount of data being stored and the variety of algorithms proposed to meet processing demands of the data scientists have led to a new generation of computational environments and paradigms. These environments simplify the task of programmers, but achieving the ideal performance continues to be a challenge. In this work we investigate important factors concerning the performance of common big-data applications and consider the Spark framework as the target for our contributions. Based on that, we present the design and implementation of Janus, a tool that automates the reconfiguration of Spark applications. It leverages logs from previous executions as input, enforces configurable adjustment policies over the collected statistics and makes its decisions taking into account communication behaviors specific of the application evaluated. In order to accomplish that, Janus identifies global parameters that should be updated, or points in the user program where the data partitioning can be adjusted based on those policies. Our results show gains of up to 1.9× in the scenarios considered.

1. Introduction

The evolution of areas like data mining, machine learning, and data analytics and the 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 be parallelized to a large number or machines, but without requiring them to express nor to handle low-level parallelism tasks. Given its wide acceptance in current big-data scenarios, in this work we consider Spark in our analysis.

Based on the algorithm description provided by the user in such high-level abstractions, it is the task of the programming environment to find the best configuration to maximize execution performance with good resource usage. That configuration is often derived from information about the data to be processed and the operations to be performed. However, in some cases the solutions proposed may not be the best ones.

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 Janus,[1] a tool that can automate the reconfiguration of Spark applications to allow the framework to achieve better performance for each application. To do that, first we present the characterization of traditional data mining algorithms 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

* Corresponding author.
E-mail address: viniciusvdias@dcc.ufmg.br (V. Dias).

[1] Janus is an ancient Roman god, which frequently symbolized change and transitions such as the progress of past to future.

https://doi.org/10.1016/j.jpdc.2018.02.030
0743-7315/© 2018 Elsevier Inc. All rights reserved.

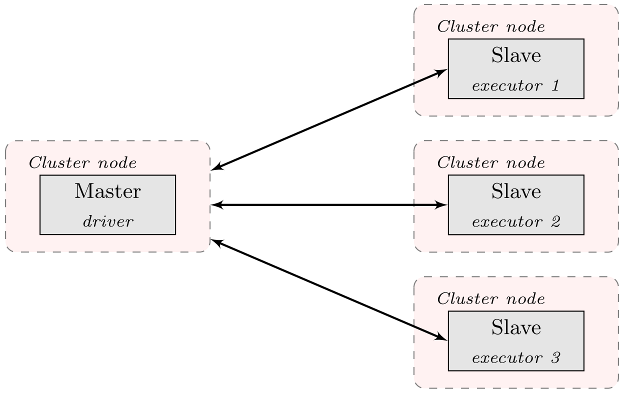


Fig. 1. Execution environment for a data-parallel application.

of recurrent applications and evaluate our solution against conservative and manual partitioning approaches. The results show consistent gains in performance and Janus's flexibility for extending policies enhances its applicability in other scenarios.

2. 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. In this context, we refer to the *Execution Environment* as the responsible for describing how the computational resource is organized in order to give support to the application computation. Then, given the execution environment, we describe the *Application Model*, with bridges the gap between the distributed computation and the programmer.

In a typical organization for data-parallel processing, the computing resources are organized as a cluster of commodity hardware, known as compute nodes, often organized in racks. This model considers that Spark parallel applications can be assigned to a subset of that cluster, using a master/slave architecture. Fig. 1, shows a Spark application assigned to four compute nodes, where we have a central coordinator (*driver*), which acts as master, and three *executors*, which play the role of slaves. The execution of an application starts with the allocation of resources for the executors and the driver.

The semantic of an application is expressed in terms of computation applied to data elements. In fact, one can think of a parallel program as a data flow, composed by operators, inputs and outputs. The operators are combined to build an algorithm, with the output of one operator serving as input to some other operator. The operations to be applied to data in frameworks like MapReduce [5], Dryad [14] and Spark [24] are described using DAGs (**D**irected **A**cyclic **G**raphs). The interpretation given to vertices and edges in those graphs depends on the system, but they always represent the way data flows through and is transformed by operators. In Spark, vertices represent immutable datasets and edges represent the transformations that are used to produce a new dataset from previously existing ones. In that way, DAGs make clear the dependencies along a program execution and are essential to identify when tasks may be parallelized and when synchronization is necessary.

In Spark, data are stored and processed as collections called Resilient Distributed Datasets (RDDs), shown in Fig. 2. The items of an RDD are organized in *partitions*, which represent an atomic concept for computation and caching. The strategy used to associate items to partitions is determined by the RDD's *partitioner*. Hashing and ranging are common strategies used for partitioning data, besides a custom partitioner defined by the programmer. Also, there is an one-to-one relation between Spark tasks and partitions.

RDDs compose the nodes of a DAG, while the operators are the edges where transformations occur. Transformations define how the output data is derived from the input data [24]. There may be *narrow* or *wide* dependencies among RDDs, based on how elements of a new RDD are derived from elements of a previous one. Mapping and filtering are examples of transformations with *narrow dependencies*, since each new element can be derived from isolated elements in the previous collection. However, when new data is produced by reductions or joins applied over existing RDDs, the original data must be reorganized, and global communication is necessary. This is called *wide dependency*, since it requires communication of data between compute nodes, what is called a *shuffle*.

2.1. Execution stages

In Spark, chains of operators with narrow dependencies, which do not require remote communication, can be grouped into *stages of execution* by pipelining those operators. During each stage, each partition of an RDD can be potentially processed in parallel, given there are no wide dependencies. On the other hand, operators with wide dependencies mark the frontiers between stages, where data shuffling occur, and constitute opportunities to change the application's degree of data parallelism for the following stage, since data partitioning may be defined anew for the resulting RDDs. As we will see throughout this work, such opportunities represent one of the most effective aspects in the process of tuning applications. We refer to those points as *adaptive points*.

To illustrate how execution stages are obtained by the Spark framework, consider the implementation of *Wordcount* in Fig. 3. Its goal is to count the occurrences of each word in a document partitioned among several compute nodes. First we consider an initial distributed collection containing lines of this document *(line 5)*, which is transformed into a new collection of individual words *(line 6)*. Each word is further transformed into a pair `(word, 1)` *(line 7)*. The collection of pairs is then aggregated by key using a summing operation *(line 8)*. Finally, we have a new collection containing words and their respective number of occurrences in the document.

The *Wordcount* code translates into a logical representation of chained *execution stages*. In this representation, distributed collections are RDDs organized according to dependencies produced by the operations applied: `flatMap`, `map` and `reduceByKey`. In Fig. 4, the transformations of the RDD containing `lines` into an RDD of `words`, and then an RDD of `pairs` are grouped together in a single execution stage, because they were tied by the *narrow dependencies a* (`flatMap`) and *b* (`map`). On the other hand, the RDD `counts` was left by itself in a second execution stage due to the fact that the dependency *c* (`reduceByKey`) is *wide* and, therefore, it marks the existence of an *adaptive point*.

At job submission time, the analysis of dependencies is done and the DAG is divided into stages. Those are handled to the execution engine, which is responsible for scheduling of stages and coordinating with the application.

3. Performance diagnosis dimensions

Before we can build our tuning tool, we must first understand the factors affecting parallel applications. To that effect, we identified a few performance diagnosis dimensions, which will guide our characterization. Our goal is to reduce the complexity of determining the source of performance bottlenecks by analyzing executions from different perspectives, which relate directly to our choice for dimensions. Those dimensions are described next.

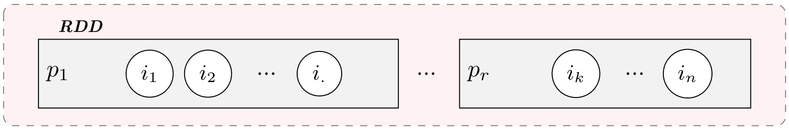


Fig. 2. RDD of n items distributed over r partitions.

```
object Wordcount {
  def main(args:  Array[String]) {
     val conf = new SparkConf().setAppName("Word Count")
     val sc = new SparkContext(conf)
     val counts = sc.textFile ("/tmp/sample").
         flatMap (_ split " ").
         map (w => (w,1)).
         reduceByKey (_ + _)
     println (counts.count + " words")
     sc.stop()
  }
}
```

Fig. 3. Wordcount algorithm in Spark.

3.1. Data layout

When programs read their raw input, they store it in data structures that carry additional semantic information. Primitive and composite types, contiguous arrays and pointers are common ways for making sense of data in memory. Object-oriented languages add a layer of abstraction that improve productivity and modularity and for that they are often used in the majority of data-parallel frameworks, like Hadoop and Spark. However, the added abstraction comes with increased overhead in memory usage. Furthermore, those frameworks run on top of the Java Virtual Machine (JVM), and the JVM's garbage collector is known to be sensitive to data memory layout and access patterns [19]. For those reasons, it is worthy to evaluate applications running on top of these systems from the perspective of their data layout. After all, applications have some flexibility to choose their trade-off between high-level abstractions (*e.g.*, objects and complex structures) for clarity and low level structures (*e.g.*, arrays and primitive types) for fine-grained performance.

3.2. Task placement

Data-parallel systems divide work by launching many tasks that run the same code (stage) on different chunks of data. With such design, scaling to the amount of data is straightforward, since data partitioning dictates the parallelism. However, given that applications process gigabytes or even terabytes of data, at any point of time, hundreds or even thousands of tasks may be ready to be dispatched for execution in the cluster. It is scheduler's role to receive all work requests, pack them into an execution plan and decide which tasks go to which resources. Furthermore, the scheduler must address the trade-off between throughput and allocation quality. In many applications, every task can have a different computational cost, or process a different amount of data. Thus, the co-allocation of heterogeneous tasks has the potential for creating unexpected performance issues. For example, assigning many heavy tasks to a same subset of resources would increase the odds of saturation in a single point and could also cause the under-utilization of others.

3.3. Adequate parallelism

Achieving the right degree of parallelism does not mean just to tune applications for performance, but also to do so with just *enough* resources at each point during execution. In data-parallel systems, the entity responsible for handling parallelism is the partitioner, which organizes the data based on a *number of partitions*, and it must be able to find that adequate parallelism for each execution. This is even more critical in new generation frameworks, where a single submitted program may have many execution stages separated by wide dependencies. Each stage may have a different optimal degree of parallelism, and the data shuffling points between stages become opportunities to adjust the partitioning accordingly. Therefore, elastic behaviors encountered along an application's life cycle must be taken into account when tuning that application. The goal, then, is to find the level of parallelism that achieves a good performance, without going beyond that level.

3.4. Load balancing

Besides the number of partitions, one must worry about the amount of work assigned to each one of then. For instance, Spark's execution model creates invisible barriers at the beginning of each stage, needed to satisfy data dependencies before proceeding with execution. Assuming applications execute stages sequentially, any imbalance in a stage's tasks leads to resource idleness, which in turn could create performance bottlenecks. Such load imbalances might be inherent to the algorithm or due to bad partitioning. In the first case, it may not be possible to change the application behavior; however, in the second case the framework may be able to improve performance by changing the partitioning scheme. In any case, only a detailed analysis of the execution may determine how to handle the problem.

4. Selected applications

Big-data applications are often based on data mining/machine learning algorithms. Because such algorithms may have different properties, we selected three of them that cover some of the major execution behaviors in this context. Our goal was to cover common application patterns, like iterativity and/or regularity, and also straightforward data analytics routines, i.e., when these characteristics are absent. Thus, we selected the following algorithms:

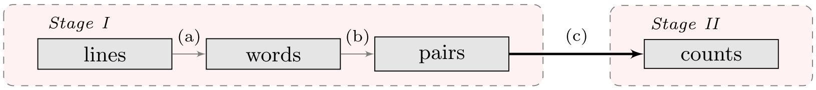


Fig. 4. *Wordcount* represented as a DAG of stages. Transformations *a* and *b* represent mapping operations, so they generate *narrow* dependencies. Transformation *c* represents the `reduceByKey` operation that executes a global reduction, so it generates a *wide* dependency.

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

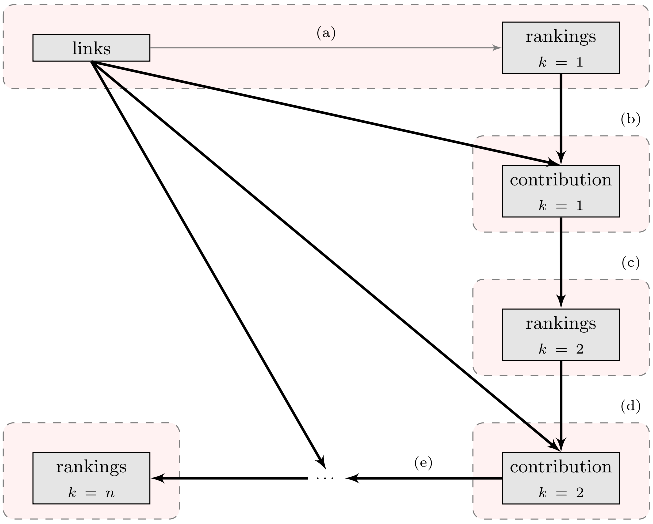


Fig. 5. Overview of *Pagerank*.

(i) PageRank, an *iterative, regular* algorithm, which executes basically the same computation on every iteration; (ii) Eclat [25], an *iterative, irregular* frequent pattern mining algorithm, where the amount of computation varies per iteration and (iii) Twidd, a parallel implementation of FPGrowth [11] that also solves the pattern mining problem, but a *non-iterative* algorithm with complex data structures. Other *iterative, regular* algorithms like *PageRank* include *KMeans* and the training step of *Collaborative Filtering with* **A***lternating* **L***east* **S***quares*. Other *iterative, irregular* algorithms like *Eclat* include Pregel [18] implementations of *Triangle Counting* and *Connected Components*. Finally, additional applications that follow a *non-iterative* pattern include *ETL workflows*, *Sorting* and *Nutch Indexing*.

4.1. 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 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 sums up those values to build a new rank. The process continues for a number of iterations or until values converge. Fig. 5 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.

4.2. Eclat

Eclat 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. It works with a vertical database layout for fast candidate counting, intersecting inverted lists of transactions. We adopted a local counting strategy [20] to implement Eclat over the RDD abstraction. Fig. 6 illustrates the algorithm.

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 filters the original set of itemsets, keeping only the frequent ones. It 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.

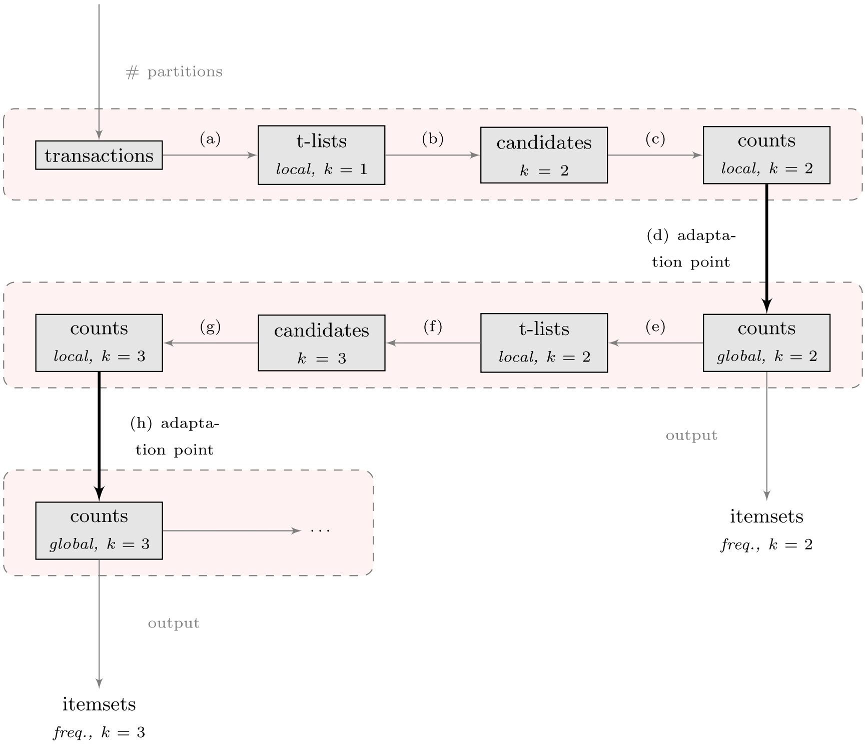


Fig. 6. Overview of *Eclat*.

Table 1
Algorithms and datasets used for characterization.

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

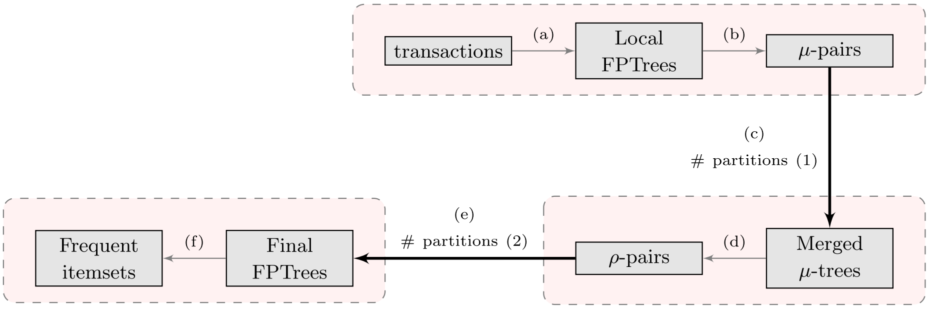


Fig. 7. Overview of *Twidd*.

4.3. Twidd

Twidd is another approach for frequent pattern mining. An overview of Twidd is shown in Fig. 7. Briefly, the algorithm *(1)* reads transactions, *(2)* builds FPTrees over those 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)*.

5. Characterization

The complexity of piled abstraction levels in data-parallel systems turns out to be an obstacle for a strictly analytical analysis. For this reason, we present an experimental evaluation to characterize sources of performance inefficiency based on the dimensions introduced in Section 3. Our observations here both confirm there is room for performance improvement and guide us with possible actions towards that desired outcome.

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 were connected with Gigabit Ethernet. The cluster was configured with Spark v1.5.1 and Hadoop/HDFS v2.5.0. We used two real world datasets 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) [17]. All datasets were loaded into HDFS with a replication factor of 2, which was enough to provide good locality to the tasks considered. Table 1 summarizes the setup for algorithms and datasets.

5.1. Task placement

Initially we observed the behavior of Twidd and Eclat under several degrees of parallelism (Fig. 8). In Twidd we varied the degree of parallelism in the adaptive point *# partitions(1)*; in Eclat we varied the degree of parallelism in the input *(# partitions)*.

Partitioning in Eclat has the expected effect: there is a sweet spot that represents the best trade-off for parallelism. On the other hand, Twidd's results are uneven. Its run-time results start behaving similarly to Eclat's; however, at the range [227, 331] we note something unexpected: execution time increases with 257 and 327 partitions and then drops again. Also, we observe a high variation in the results with 257 partitions.

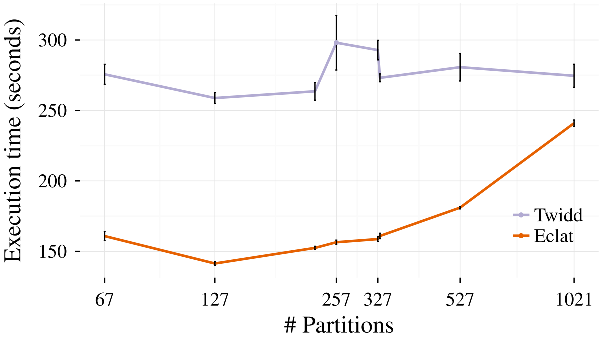


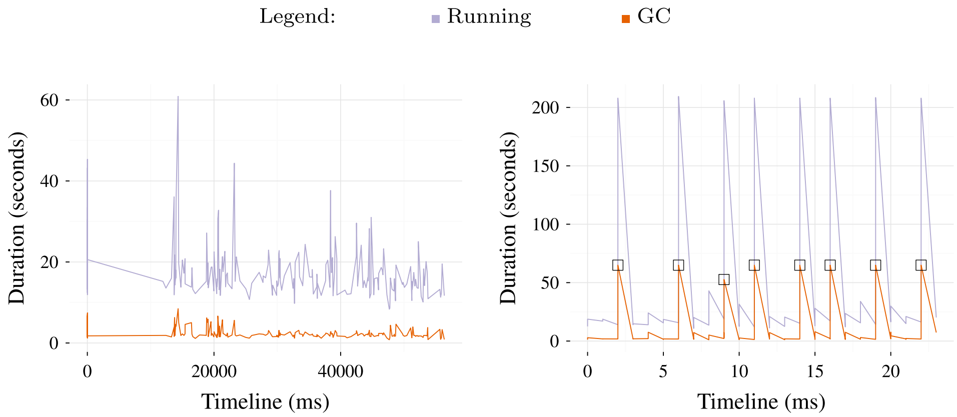
Fig. 8. In some cases finding the right degree of parallelism is not a matter of increasing/decreasing the number of partitions.

To understand that, we isolated the stage causing the high variation (the 5th stage, which runs FPGrowth on final repartitioned trees) to observe the duration of its tasks as the application evolved, and identified the best/worst case scenarios (Figs. 9(a) and 9(b)). In the best case, execution times stay between 10 and 30 s, mostly, while some times get beyond 200 s in the worst case. For the worst case we zoomed in on a shorter interval, to show tasks that had a high increase in run-times (they were 72 of 257 overall). We can see those moments coincide with an increase in the garbage collection times. Comparing the two scenarios we note that not even the most costly outliers of the best case came close to the upper bound caused by those 8 tasks in the worst case (≈200 s). To make things worse, those stragglers were scheduled nearly at the same time (delta 30 ms), in the same executor.

That shows how irregular applications are specially sensitive to parallelism and task placement. In Twidd, we identified cases where co-allocation of heavy tasks on executors running slow due to GC pauses could bring significant performance degradation to the overall execution, as just shown. That behavior is tightly coupled with the complexity of data structures employed by Twidd. In fact, that is the reason why Eclat presents such predictable behavior w.r.t. varying the degree of parallelism. In conclusion, data parallel schedulers could benefit of GC-awareness from its worker nodes.

5.2. Load balancing

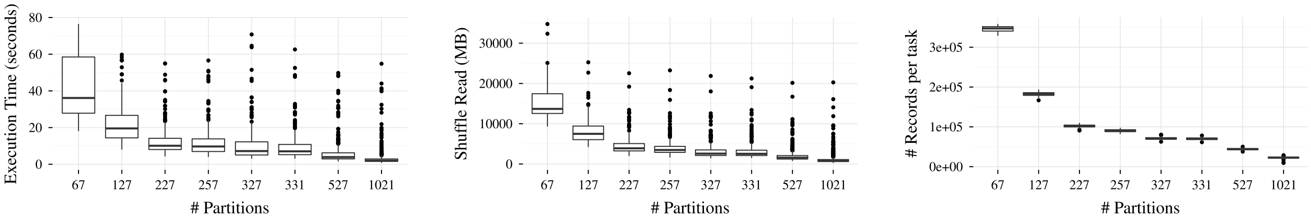
We discuss load balancing by looking closely at Twidd's 4th and 5th stages (Fig. 7), as we alter data partitioning by varying the



(a) Best case scenario.

(b) Worst case scenario.

Fig. 9. Detailed analysis of Twidd's 5th stage.

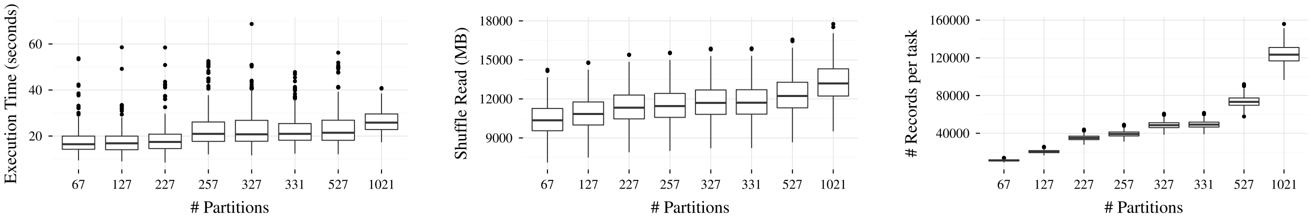


(a) Runtime.

(b) Shuffle read.

(c) Records.

Fig. 10. 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. 11. 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.

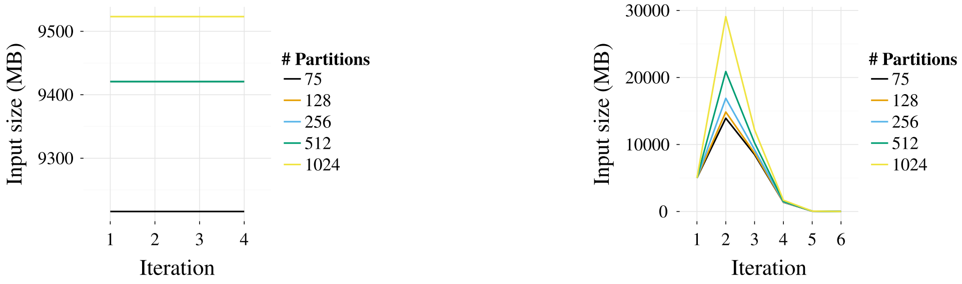
number or partitions in the input of stage 4 (*#partitions(1)*), which also affects stage 5.

Fig. 10 shows additional metrics regarding Twidd's 4th stage. The *x*-axis refers to the number partitions in the 4th stage's input. 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. We used Person and Spearman correlations to get a better understanding on this behavior [10]. Pearson correlation will allow us to determine whether the two variables increase or decrease together. Additionally, Spearman correlation helps us to determine whether this relationship is likely to be monotonic. Indeed, Pearson and Spearman correlations between those two dimensions (run-time and shuffle read) 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 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 [6].

A different result was found in the analysis of Twidd's 5th stage, shown in Fig. 11. The running time median cannot be estimated from other metrics at any degree of parallelism, which means that tasks have variable processing costs (Fig. 11(a)). At that 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. 11(b)) and number of records (Fig. 11(c)) we conclude 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.

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.



(a) PageRank: input size.

(b) Eclat: input size.

Fig. 12. Input size progression over iterations.

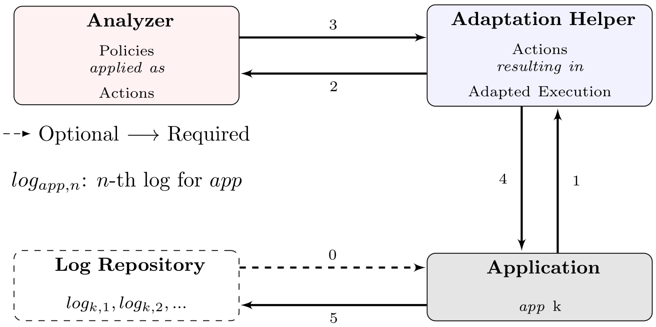


Fig. 13. Janus architecture.

5.3. Adaptive execution

Many machine learning and graph processing algorithms are based on iterative and/or converging approaches. When translated into Spark DAGs, those techniques expose opportunities for re-execution optimization. Indeed, the partitioning scheme (number of partitions and data placement method) can be changed in the beginning of every stage, since 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 (Fig. 12(a)): graph nodes share rank contributions with their neighbors. Therefore, iteration 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, there are algorithms that re-execute stage functions several times, but in different contexts. In those cases, the problem has the potential for growing/shrinking over time. Even then, 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. Then, it becomes a matter of, given another set of inputs and prior knowledge, finding 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 that case, the only information that can be leveraged is between executions, but it can still be advantageous to learn from past events if jobs are recurrent.

6. The Janus reconfiguration tool

As shown during the characterization, the adjustment of an application's parameters, specially those related to partitioning, deeply affects its performance. With that in mind, and considering that performance optimization should be transparent to the user, we present in this section our reconfiguration tool, Janus. Its main goal is to facilitate the process of tuning applications. We followed three guidelines in its design: *(1)* it should be *independent* of versioning and implementation of the targeted execution engine (Spark); *(2)* the model created by it should be *interpretable*, *i.e.*, capable of easily exposing points in the program where the adjustment is actually taking place; and *(3)* it should be *extensible*, allowing *ad-hoc* reconfiguration strategies to be implemented and added by system administrators.

Fig. 13 shows the architecture overview of Janus. 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. Next, the system internally instantiates an *Adaptation Helper* (Section 6.2) *(1)*, which then initializes and asks the *Analyzer* (Section 6.1) to parse the log and make some decisions based on the observed statistics *(2)*. Those decisions are returned to the helper as *actions* to be applied during the current execution *(3)*. The helper then translates those actions into properly partitioned RDDs back to the application *(4)*. Finally, the new log is added to the repository.

6.1. Analyzer

The purpose of the *Analyzer* is to receive execution logs from the *Adaptation Helper*, identify points for adjustment 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 configuration and partitioning over time. The analyzer also captures the communication patterns of the application, so it can identify if the

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

application is regular and/or iterative, and use that information in the decision process. The current version of Janus supports five types of action:

- `UConfAction`: changes a global configuration;
- `UNPAction`: updates the number of partitions at an *adaptive point*;
- `UPAction`: updates the partitioner of at an *adaptive point*;
- `WarnAction`: adds a warning to the application log as a mechanism to document cases where some inefficiency was detected but no automatic action to mitigate the problem is known;
- `NoAction`: indicates that no action should be taken.

For instance, `UPAction` can request the change of the partitioner from *hashPartitioner* to *rangePartitioner*. On the other hand, `WarnAction` can indicate to the user some problem that requires human intervention, like identifying the source of imbalance (Section 5.2). Also, with an `UConfAction`, for example, we are able to change the default serialization of RDDs by setting a global configuration (`spark.serializer`), which affects the whole application.

The plan of action depends on the set of policies employed by the *Analyzer*. We model policies with a pre-defined signature: they receive information about the environment and/or *adaptive point*, and return necessarily an action (or `NoAction`). More specifically, we work with two types of policies: *(i)* application policy, that considers the whole execution and updates global configurations through `UConfAction`, and *(ii)* partitioning policy, that works over an *adaptive point* through `UNPAction`/`UPAction` and receives, besides environment information, parameters and metrics regarding that specific execution point.

6.2. Adaptation Helper

The *Adaptation 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. Section 6.3 describes the Janus API. Basically, the user can feed it with previous execution information, pre-adapt and/or include new policies and call special versions of Spark operators.

The user interacts with the *Adaptation Helper* through overwritten Spark operators that are aware of adaptive points. Fortunately, 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 those special operators. Thus, the only requirement for using them is to explicitly import their definitions (*AdaptableFunctions*).

The *Adaptation Helper* can be instantiated (atomically) by one of two events: the first call to any overwritten operator is being called for the first time, or a call to `preAdapt` to include custom policies, for example. The *Adaptation Helper* also triggers the *Analyzer* initialization and execution. At that moment, the helper potentially has all actions delivered by the analyzer associated with their respective adaptive points. Its execution cost is linear with the number of tasks, basically the log parsing.

Given that actions are available, every time the user code executes some of those overwritten operators, the system verifies if there is an action to be applied at the current adaptive point. If that 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, so our solution does not break existing implementations.

Finally, we had to make 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 parameter `spark.adaptive.logpath` in *SparkConf*. For the second decision we consider with two alternatives. For most of the operators, we support an extra parameter that identifies the adaptive point. If no identifier is provided, the system considers a default name comprising the operator and the line at the user code where it was called. We plan to include automatic tracking of applications and logs in the future.

6.3. Janus API

The Janus interface incorporates functionalities from both *Analyzer* and *Adaptation Helper*. Fig. 14 describes that API. In summary, the user can feed the framework with a log of a previous execution (*Log feedback*), pre-adapt the application and/or include new custom policies (*Initialization*), call RDD operators aware of adaptive points (*Overwritten operators*) and implement new policies for a performance bottleneck or in accordance to domain specific requirements (*Policy creation*). In the next sections we describe the basic structure of a program (Section 6.3.1), the API model (Section 6.3.2) and the procedure to create new policies and add them to the framework (Section 6.3.3). Finally, Section 6.3.4 presents the sample policies that will be used in our experiments.

6.3.1. Basic structure of a program

To illustrate the use of Janus, Fig. 15 shows a WordCount application in Spark that uses it. The bold parts highlight differences in comparison to a standard implementation. 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*).

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 that would be done at the time of the *reduceByKey* call. Furthermore, as discussed, the 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 optional.

6.3.2. Janus model

The interaction between the user and Janus happens in two levels. At first, the user may be interested in using only the basic functionalities; for that, he just has to adjust the application to make use of the reconfiguration framework (Section 6.3.1). In a second level, the user may want to extend the tool and implement a custom policy for his applications; in that case, he must master the knowledge about how policies relate to the statistics present in logs of previous executions. With that in mind, we created a representation for execution logs (*Logging Model*) and other to represent types of policies and their interfaces for extension (*Policy Model*). Fig. 16 illustrates the details of the classes used in each model and relations between them.

In general, the goal of the *Logging Model* is to deliver informations and statistics of the whole application to the *Policy Model*, referred as its environment (*Environment*), or concerning a specific execution point, referred as an aggregate of stages and the RDD at the respective *adaptive point* (*AdaptivePointStats*).

The environment aggregates the specific configurations regarding the virtual machine (*Java Virtual Machine*) being used (`systemProperties`), Spark configurations (`spark Properties`), available resources, like cores and memory of executors (`executors`), and all stages observed during the execution. *Stage* represents an aggregate of tasks (*Task*), which in turn contain the metrics extracted from the execution logs. Each *AdaptivePointStats* contains the statistics of every stage that participates in an adaptive point. The *RDD* labels an adaptive point

Log feedback:	`conf.set("spark.adaptive.logpath",` $log_{app,n}$`)`
Initialization:	`preAdapt(`sc`,` p_1`,` p_2`, ...)`
Overwritten operators:	`textFile, reduceByKey, aggregateByKey, join, partitionBy` with an aditional parameter *ap* representing an adaptive point
Policy creation:	`MyPolicy extends` *ApplicationPolicy* `MyPolicy extends` *PartitioningPolicy*

Fig. 14. Janus API.

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

Fig. 15. Adaptable Wordcount.

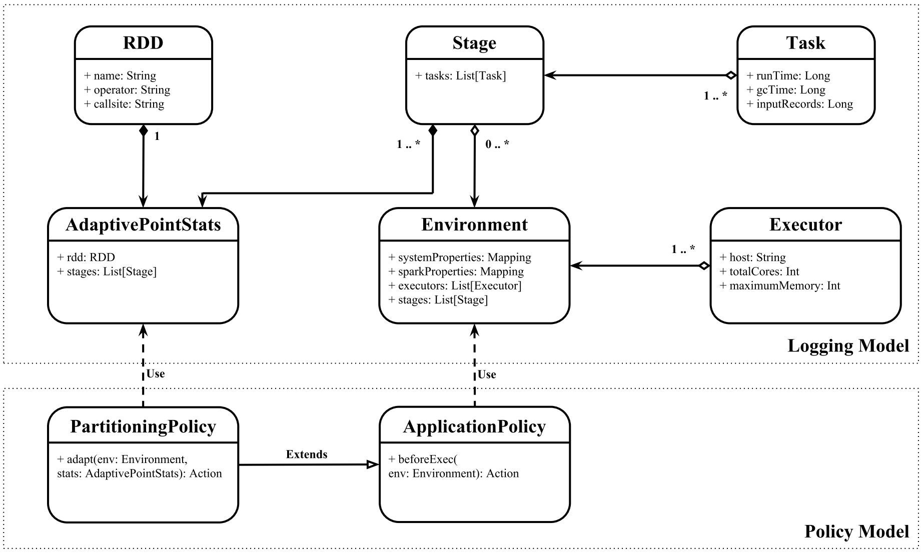


Fig. 16. Janus Reconfiguration Model.

with its name, the operator that generated it and the code line of its instantiation. The user can, alternatively, choose a name for the adaptive point, avoiding any dependency on code line numbers. *AdaptivePointStats* can receive several stages as statistics because the execution model allows it: due to the *pipelining* of *narrow* dependencies and the propagation of the same partitioning strategy, actions in the user program may force the computation of each RDD which is part of that pipeline; that can potentially generate more than one stage per adaptive point.

We identify two categories in the *Policy Model*: *(1)* ApplicationPolicy, which receives global information about the environment and applies reconfiguration actions before the start of the next execution; and *(2)* PartitioningPolicy, which can read global configurations about the environment, but also adjust the partitioning of specific adaptive points. In the next section we provide details about the implications of this model for the extension of policies.

6.3.3. *Reconfiguration policies*

To create a new reconfiguration policy, the user must extend one of the two types of policies described in Section 6.3.2 (*ApplicationPolicy* or *Partitioning Policy*) and then, include the new policy in the `preAdapt` call, before *SparkContext*'s instantiation. Several policies can be included at once: `preAdapt`($conf, p_1, p_2, \ldots$), where p_i is a Scala object that extends one type of policy.

The interface for extending policies is presented in Fig. 17. In order to create an *application policy*, the user must define an Scala object that extends `ApplicationPolicy` and implements the function `beforeExec` (*lines 15–17*). Alternatively, to create a *partitioning policy* the user must define an object that extends `PartitioningPolicy` and implements the functions `beforeExec` (if some preprocessing is necessary) and `adapt` (*lines 19–22*).

Fig. 18 shows the *WordCount* program modified to include an additional policy, named `LocalityPolicy`; the highlighted lines

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

```
1  import br.ufmg.cs.systems.sparktuner._
2  import br.ufmg.cs.systems.sparktuner.model._
3
4  /* Application Policy */
5  object MyApplicationPolicy extends ApplicationPolicy {
6
7    def beforeExec(env: Environment): Action = {
8      // return an action based on the application's environment
9    }
10 }
11
12 /* Partitioning Policy */
13 object MyPartitioningPolicy extends PartitioningPolicy {
14
15   def beforeExec(env: Environment): Action = {
16     // return an action based on the application's environment
17   }
18
19   def adapt(env: Environment, stats: AdaptivePointStats): Action = {
20     // return an action based on the application's environment (env) and
21     // specific statistics of the adaptive point (stats)
22   }
23 }
```

Fig. 17. API for creating custom *application policies* or *partitioning policies*.

```
import br.ufmg.cs.systems.sparktuner.rdd.AdaptableFunctions._
object Wordcount {
  def main(args: Array[String]) {
     val conf = new SparkConf().setAppName("WordCount").
         set("spark.adaptive.logpath", "/tmp/log1")
     preAdapt(conf, LocalityPolicy)
     val sc = new SparkContext(conf)
     val counts = sc.textFile ("/tmp/sample").
         flatMap (_ split " ").map (w => (w,1)).
         reduceByKey (_ + _, "ap-counting")
     println (counts.count + " words")
     sc.stop()
  }
}
```

Fig. 18. Adding an *application policy* to *Wordcount*.

indicate the differences to the previous version of the program (Fig. 15), when only the default policies were used. Next we describe the implementation of that custom policy.

The `LocalityPolicy` is an implementation of the simplified theoretical analysis of the *Delay Scheduling* [23], which aims at increasing the data locality of parallel tasks. Because the gain in executing tasks that have their input local is likely to be more efficient than tasks that fetch input remotely, every time one task is ready to be scheduled but there are no available cores on any machine that has that task's input locally, the scheduler delays the decision of that task assignment by a certain time. The intuition behind the strategy is that the probability of a core containing this task's data locally will increase over time, which in turn increases the odds of scheduling that task with local input. The challenge here, and also the motivation for a reconfiguration policy, is how long to delay the task assignment. Specifically, if we wait too much, the delay may become larger than the task's execution time itself; if we wait too little, the delay may be insufficient for a core with local data to appear.

Because our goal is to find an optimal delay for the application, the implementation of *LocalityPolicy* requires the extension of `ApplicationPolicy` and the implementation of the function `beforeExec` (Fig. 17, lines 4–10). In particular, the function receives the environment information collected from a previous execution and must return an action that affects the whole application. Then, in the context of a locality policy, we are actually interested in an action `UConfAction`, since with it we will be able to update the configuration `spark.locality.wait`, which controls the scheduler's delay parameter in Spark. The function body must estimate the ideal delay in order to guarantee a minimal proportion of local tasks. That proportion serves as a parameter for the policy. For example, the user may want to find the optimal delay to achieve at least 90% of local tasks. We can further extrapolate that procedure for each stage of the application and choose the greater value as the final delay, in order to cover all stages. The implementation of that policy is shown in Fig. 19. Finally, a similar procedure can be applied to develop *partitioning policies*, with an additional implementation of the function `adapt`, which will take care of *adaptive points* in the applications.

6.3.4. *Sample policies*

As a proof of concept, we developed partitioning policies that handle some of the bottlenecks discussed in Section 5. Here we provide the pseudo-code for each strategy; an actual implementation would implement the algorithms as the `adapt` function of a *PartitioningPolicy* (Section 6.3.3). The policies presented next are not extensive, but are sufficient to demonstrate Janus's applicability.

Empty Tasks (**ET**, *Algorithm 1*). 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`).

Algorithm 1 Empty Tasks

```
Function: OPT-EMPTY-TASKS( rdd, stages )
    repr ← FILTER(stages, "most recent")
    numEmptyTasks ← FILTER(repr.tasks, task.input == 0).size
    if numEmptyTasks > 0 then
        return UNPACTION(rdd.name, repr.tasks.size − numEmptyTasks)
    else
        return NOACTION(rdd.name)
```

Memory/Disk Spill (**SP**, *Algorithm 2*). 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`).

```
object LocalityPolicy extends ApplicationPolicy {
  private val replFactor:  Int = 3
  private val localityTarget:  Double = 0.9
  private def ln(n:  Double):  Double = {
     scala.math.log(n) / scala.math.log(scala.math.E)
  }
  private def stageDelay(numExecutors:  Int, coresPerExecutor:  Int,
     stage:  Stage):  Double = {
     val taskRunTimes = stage.taskRunTimes
     val avgTaskLength = (taskRunTimes.sum / taskRunTimes.size.toDouble)
     val minSchedDelay = - (numExecutors / replFactor) *
                     ln( (1 - localityTarget) / (1 + (1 - localityTarget)) )
     (minSchedDelay / coresPerExecutor) * avgTaskLength
  }
  def beforeExec(env:  Environment):  Action = {
     val numExecutors:  Int = env.executors.size
     val coresPerExecutor:  Int = (env.executors.map (_.totalCores).sum /
                         numExecutors).toInt
    val schedDelay:  Double = env.stages.map (
        stage => stageDelay(numExecutors, coresPerExecutor, stage)).max
     UConfAction("spark.locality.wait", s"${schedDelay/1000}s")
  }
}
```

Fig. 19. Application policy for *Delay Scheduling*.

Algorithm 2 Memory/Disk Spill

```
1:  Function: OPT-SPILL( rdd, stages )
2:      repr ← FILTER(stages, "most recent")
3:      bytesSpilled ← repr.bytesSpilled
4:      shuffleWriteBytes ← repr.shuffleWriteBytes
5:      if shuffleWriteBytes > 0 then
6:          factor ← $\frac{bytesSpilled}{shuffleWriteBytes}$
7:          if factor > 0 then
8:              numPartitions ← $repr.numTasks + \lceil factor * repr.numTasks \rceil$
9:              return UNPACTION(rdd.name, numPartitions)
10:         else
11:             return NOACTION(rdd.name)
12:     else
13:         return NOACTION(rdd.name)
```

Garbage Collection (**GC**, *Algorithm 3).* 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 median 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`).

Algorithm 3 Garbage Collection

```
1:  Function: OPT-GC( rdd, stages )
2:      repr ← FILTER(stages, "most recent")
3:      gcOverheads ← repr.taskGcOverheads
4:      sk ← SKEWNESS(gcOverheads)
5:      if HIGHSKEWNESS(sk) then
6:          target ← MEDIAN(gcOverheads)
7:          normalized ← NORMALIZE(gcOverheads, target)
8:          numPartitions ← SUM(normalized)
9:          return UNPACTION(rdd.name, numPartitions)
10:     else
11:         return NOACTION(rdd.name)
```

*Task Imbalance (***TI***, Algorithm 4).* This action automates our methodology of the load balancing analysis from Section 5.2. We consider sources of imbalance being: *(a)* inherent; *(b)* due to key distribution; or *(c)* due to variable costs for the same key. Due to limitations of the environment, we are unable to adapt and handle some kinds of imbalance automatically in most situations. Most cases of imbalance output warnings (`WarnAction`) except for imbalance due to key distribution, where a *rangePartitioner* may be a solution (`UPAction`).

Algorithm 4 Task Imbalance

```
1:  Function: OPT-TASK-IMBALANCE( rdd, stages )
2:      repr ← FILTER(stages, "most recent")
3:      runTimes ← repr.taskRunTimes
4:      sk ← SKEWNESS(runTimes)
5:      if not HIGHSKEWNESS(sk) then
6:          return NOACTION(rdd.name)
7:      corr1 ← CORRELATION(runTimes, repr.taskShuffleReadBytes)
8:      corr2 ← CORRELATION(runTimes, repr.taskShuffleReadRecords)
9:      HighCorr1 ← HIGHCORRELATION(corr1)
10:     HighCorr2 ← HIGHCORRELATION(corr2)
11:     if HighCorr1 and HighCorr2 then
12:         return UPACTION(rdd.name, rangePartitioner)
13:     else if not HighCorr1 and not HighCorr2 then
14:         return WARNACTION(rdd.name, Inherent)
15:     else
16:         return WARNACTION(rdd.name, VariableCost)
```

7. Evaluation

Our environment for evaluation is the same used for characterization (Section 5). Here we evaluate Janus 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 s for that overhead in our implementation of the *Analyzer*. Section 7.1 presents the results for PageRank, which represents *iterative*, *regular* applications. Section 7.2 presents the results for Eclat, an *iterative*, *irregular* application. Section 7.3 discusses Twidd's executions, representing *non-iterative* applications. Finally Section 7.4 presents an additional use-case with Collaborative Filtering (**Al**ternating **L**east **S**quares) [26] executions, representing applications composed by several communication patterns combined.

7.1. PageRank

We evaluated PageRank under several degrees of parallelism for the input partitions (Fig. 20). Initially, we ran the algorithm with no explicit reconfiguration. We refer to that as *Conservative* execution because the number of input partitions, derived from the number of blocks in HDFS, is used naively by Spark as the default degree of parallelism for the reductions in every iteration. By doing

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

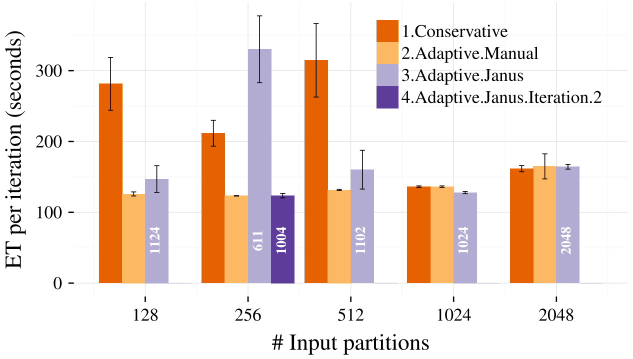


Fig. 20. Optimizing P_{eq}, *i.e.*, stages that repeat with the same communication pattern. Adaptive re-executions (iterations) can leverage information about the previous degrees of parallelism. Numbers inside bars indicate how many partitions some iteration of Janus has decided for the reduce phase.

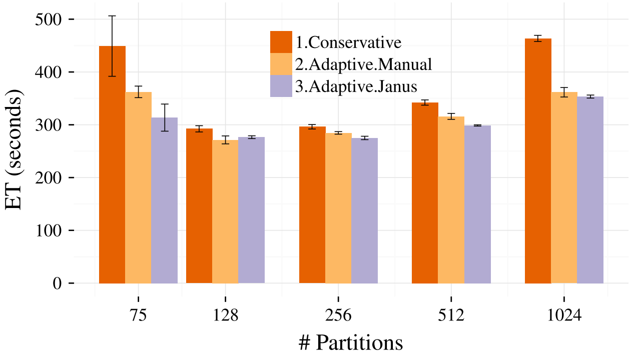


Fig. 21. Optimizing P_{sim}, *i.e.*, stages that are repeated with different inputs. The degree of parallelism must adapt to the elasticity of the application. Percentages inside bars represent reductions in the number of launched tasks relative to the Conservative approach.

a careful examination on the performance logs provided by Spark, we noticed a high garbage collection activity in the configurations with less than 1024 partitions, i.e., the memory available for each task was not sufficient. As expected, such overhead severely degraded PageRank's performance. 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 the parallelism to 1024 partitions during each 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 that parameter in the code. The third category is the one obtained by our tool (*Adaptive.Janus*), in which we used the logs from *Conservative* approach as input.

Adaptive.Manual improved overall performance, obtaining speedups of 2.2×, 1.7× and 2.3× in the first three configurations, considering the observations regarding adaptive execution made in Section 5.3. 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 exceeds the cost of reduction (with 2048 partitions). More important, the results using *Adaptive.Janus* were able to keep up with the best case scenario of *Adaptive.Manual*, which shows that the policies created were effective, with the bonus of automatic adaptation.

The execution with 256 input partitions shows an interesting fact related to the tool's operation. In its current implementation, Janus applies only one action, from one policy per adaptive point each time it runs. In some cases, the first reconfiguration may cause a second bottleneck to appear. The resulting time for 256 input partitions when we ran *Adaptive.Janus* once with the log information from the *Conservative* execution 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 6.1). That solved the spill problem, but the new execution suffered from another bottleneck due to garbage collection overhead. A second round of execution using Janus with the log of the first round was able to achieve the desired optimal performance. In that case the action returned by the *Analyzer* was to increase the number of partitions again (`UNPAction`) but w.r.t. the policy **GC** from Section 6.1. The bar *4.Adaptive.Janus.Iteration.2* shows that last result. In other cases, like with 128 and 512 input partitions, the heuristics lead directly to a decision closer to the optimal solution.

7.2. Eclat

We evaluate Eclat varying the degree of parallelism of *adaptive point* in Fig. 6. Fig. 21 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.Janus* shows the results for Janus using *Conservative* logs as input.

To build *Adaptive.Manual*, we considered a heuristic to follow the elasticity of Eclat's problem size (Fig. 12(b)). 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.Janus* is the one related to empty tasks, considering the elastic behavior of the application. It makes no assumptions about the algorithm.

Both adaptive solutions are effective on fixing under- and overestimations in the number of partitions (75 and 1024 input partitions, respectively). The gain is small in intermediary values, as they approach the optimal parallelism, when candidate generation (map phase) overcomes the global counting (reduce phase). Note that because we are comparing two similar heuristics regarding the adaptive execution, some results tend to favor the manual approach, e.g., in the experiments with 128 partitions. However, every result obtained by our tool (*Adaptive.Janus*) outperforms or is at least statistically not different from the manual adaptation, with the advantage of being an automatic approach.

Adaptive execution also has the capacity to reduce the number of tasks launched in comparison to the conservative approach, therefore reducing the total amount of resources used during each step. We indicate that gain as the percentage of reduction in the tasks launched for the reduction stages compared to the conservative approach (Fig. 22). The number of tasks launched is reduced by at least 78.4% for *Adaptive.Manual* and by at least 73.6% for *Adaptive.Janus*, even in cases where run-time gains are modest.

7.3. Twidd

Fig. 23 shows the warning messages generated by Janus when we execute Twidd with the log discussed in Section 5.2. It captured the same sources of imbalance discovered in that analysis, *i.e.*, stage 4, where *muTrees* are merged, suffers from imbalance due to variable costs for the same key, and stage 5, where *rhoTrees* are

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

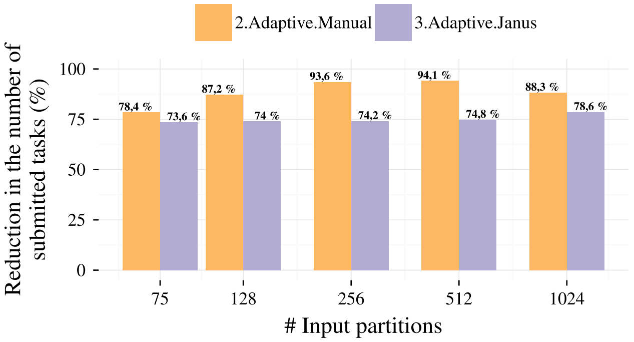


Fig. 22. Percentages indicate a reduction in the resource utilization of the adaptive approaches against the conservative one *(1.Conservative)*, shown in Fig. 21.

merged, suffers from imbalance inherent to the application. Since those issues cannot be handled easily by the current version of our tool, the actions are merely warnings, indicating user intervention is necessary.

7.4. Collaborative filtering with ALS

We also evaluated Janus in a broader scenario considering a typical machine learning workflow. More specifically we ran a *Collaborative Filtering (ALS)* [26] application that comprises training and testing. During the training phase the main behavior is iterative and regular, however during the test phase we observe some non-iterative communication patterns. Therefore, our goal is to evaluate Janus under a heterogeneous execution flow, potentially composed by several communication patterns altogether. The dataset used in this experiment is a deduplicated version of the Amazon Reviews Dataset [12], which contains 82.83 million reviews of 6.6 million users on 2.4 million products. The results of varying the number of input partitions are shown in Fig. 24.

Note that we see significant performance improvement in the first three configurations. Janus fixed underestimations in the number of partitions (72) and introduced small adjustments in an optimal partitioning (144). The third configuration (216) required two Janus iterations to exhibit improvements because we reached a high-skewed load distribution with the actions applied in the first iteration, similarly to our discussion in Section 7.1. Finally, in the fourth configuration, we did not get significant gains as the cost of spawning more tasks starts to overcome the cost of processing.

8. Related work

Several works propose job optimization for data-parallel frameworks [1,3,15]. Those systems work with structured data (SQL-like) or domain-specific languages (e.g. for machine learning), where the knowledge about data organization exposes more opportunities for optimization. Our work considers general purpose data processing, where there are no assumptions about the input.

Optimization of recurring jobs can be made by creating a statistics repository of applications executed in the cluster as done in Scope [4]. 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.

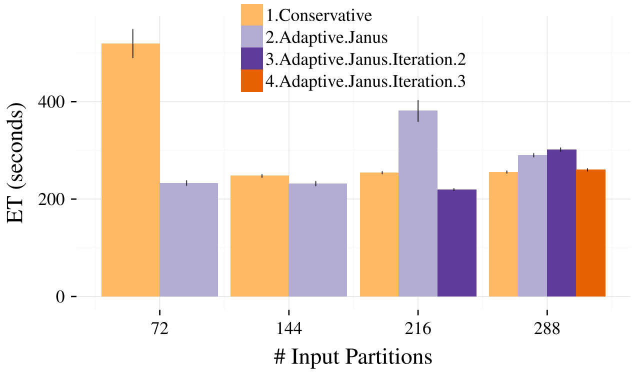


Fig. 24. Optimizing heterogeneous applications, i.e., composed by several communication patterns combined. Janus isolates points in the application with different communication patterns and applies the policies according to that specific behavior.

Automatic reconfiguration of recurring jobs is another option in performance optimization. Starfish [13] is able to construct job profiles based on previous executions, exploring the space of Hadoop internal configurations (using a *What-If* engine) to find the best setup for later jobs. However, it optimizes only static parameters that must be set prior to the application submission and cannot be changed after that. Therefore, it is also orthogonal to our approach. Furthermore, because our framework is based on configurable policies, it is possible to implement and quickly try new custom partitioning strategies [2,9] and expand the applicability of our tool.

Adaptive query execution is an ongoing issue in the Spark project that proposes improvements to the run-time engine [22]. The main idea is to postpone stage submissions to allow partitioning focused on statistics collected in earlier stages. That new feature, which aims the optimization in general cases, could benefit from extra knowledge similar to that provided by Janus.

Most data-parallel frameworks use garbage-collected languages and that is the case of Spark with the Java Virtual Machine (JVM). Several algorithms for garbage collection suffer from long GC pauses and memory overheads. Broom [8] 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 [16] plan to coordinate collections in a distributed environment. The latter could improve our scheduler's decisions by providing worker state awareness.

Memory management in Spark is moving towards a manual, optimized memory layout [21], 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.

This paper extends results previously published [6,7]: it provides a detailed description of the tool implementation, not presented before, as well as some new analysis of the evaluation results.

9. Conclusion

Data-parallel frameworks simplify the task of data scientists, allowing them to write parallel applications using high-level abstractions. However, those frameworks still face some challenges,

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

Fig. 23. Warnings produced by Janus for a given execution of Twidd.

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.

such as to create self tuning solutions 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, we must identify its causes, among several potential ones. 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 that background, we built Janus, a tool for automatic reconfiguration of recurrent applications. Our solution is configurable, compatible with any Spark core application, and it allows pluggable policies for bottleneck mitigation. Janus receives as input logs of previous executions and automatically decides new partitioning parameters for the adaptive points it identifies. We evaluated our tool against both a conservative and an adaptive/manual approach. 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.

Acknowledgments

This work was partially funded by projects InWeb (MCT/CNPq 573871/2008-6), MASWeb (FAPEMIG-PRONEX APQ-01400-14), INCT-Cyber (CNPq 465714/2014-5), and EUBra-BIGSEA (H2020-EU.2.1.1 690116, Brazil/MCTI/RNP GA-000650/04).

References

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, J. Zhou, Re-optimizing data-parallel computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 21–21. URL http://dl.acm.org/citation.cfm?id=2228298.2228327.

[2] M. Bertolucci, E. Carlini, P. Dazzi, A. Lulli, L. Ricci, Static and dynamic big data partitioning on apache spark, in: Parallel Computing: On the Road To Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK, 2015, pp. 489–498. http://dx.doi.org/10.3233/978-1-61499-621-7-489.

[3] M. Boehm, M.W. Dusenberry, D. Eriksson, A.V. Evfimievski, F.M. Manshadi, N. Pansare, B. Reinwald, F.R. Reiss, P. Sen, A.C. Surve, S. Tatikonda, SystemML: declarative machine learning on spark, Proc. VLDB Endow. 9 (13) (2016) 1425–1436. http://dx.doi.org/10.14778/3007263.3007279.

[4] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, J. Zhou, Recurring job optimization in scope, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, ACM, New York, NY, USA, 2012, pp. 805–806. http://dx.doi.org/10.1145/2213836.2213959.

[5] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. http://dx.doi.org/10.1145/1327452.1327492.

[6] V. Dias, R. Moreira, W. Meira, D. Guedes, Diagnosing performance bottlenecks in massive data parallel programs, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, 2016, pp. 273–276, http://dx.doi.org/10.1109/CCGrid.2016.81.

[7] V.V. dos Santos Dias, W.M. Jr., D.O. Guedes, Dynamic reconfiguration of data parallel programs, in: 28th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2016, Los Angeles, CA, USA, October 26–28, 2016, pp. 190–197. http://dx.doi.org/10.1109/SBAC-PAD.2016.32.

[8] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, D. Murray, S. Hand, M. Isard, Broom: sweeping out garbage collection from big data systems, in: Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, in: HOTOS'15, USENIX Association, Berkeley, CA, USA, 2015. pp. 2–2. URL http://dl.acm.org/citation.cfm?id=2831090.2831092.

[9] A. Gounaris, G. Kougka, R. Tous, C.T. Montes, J. Torres, Dynamic configuration of partitioning in spark applications, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 1891–1904. http://dx.doi.org/10.1109/TPDS.2017.2647939.

[10] F. Gravetter, L. Wallnau, Essentials of statistics for the behavioral science, in: Available Titles Aplia Series, Cengage Learning, 2007. URL https://books.google.com.br/books?id=hcoYNW4BujYC.

[11] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD Rec. 29 (2) (2000) 1–12. http://dx.doi.org/10.1145/335191.335372.

[12] R. He, J. McAuley, Ups and downs: modeling the visual evolution of fashion trends with one-class collaborative filtering, in: Proceedings of the 25th International Conference on World Wide Web, WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 2016, pp. 507–517. http://dx.doi.org/10.1145/2872427.2883037.

[13] 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, 2011, pp. 261–272.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, ACM, New York, NY, USA, 2007, pp. 59–72. http://dx.doi.org/10.1145/1272996.1273005.

[15] Q. Ke, M. Isard, Y. Yu, Optimus: A dynamic rewriting framework for data-parallel execution plans, in: Proceedings of the 8th ACM European Conference …, 2013, pp. 15–28. http://dx.doi.org/10.1145/2465351.2465354.

[16] M. Maas, T. Harris, K. Asanovic, J. Kubiatowicz, Trash day: Coordinating garbage collection in distributed systems, in: Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS'15, USENIX Association, Berkeley, CA, USA, 2015 pp. 1–1. http://dl.acm.org/citation.cfm?id=2831090.2831091.

[17] G. Magno, G. Comarela, D. Saez-Trumper, M. Cha, V. Almeida, New kid on the block: Exploring the google+ social graph, in: Proceedings of the 2012 Internet Measurement Conference, IMC '12, ACM, New York, NY, USA, 2012, pp. 159–170. http://dx.doi.org/10.1145/2398776.2398794.

[18] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146. http://dx.doi.org/10.1145/1807167.1807184.

[19] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, G. Xu, FACADE: A compiler and runtime for (almost) object-bounded big data applications, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, ACM, New York, NY, USA, 2015, pp. 675–690. http://dx.doi.org/10.1145/2694344.2694345.

[20] A. Veloso, W. Meira Jr., R. Ferreira, D.G. Neto, S. Parthasarathy, Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining, in: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, in: PKDD '04, Springer-Verlag New York, Inc., New York, NY, USA, 2004, pp. 422–433. URL http://dl.acm.org/citation.cfm?id=1053072.1053111.

[21] R. Xin, Project tungsten, 2015. URL https://issues.apache.org/jira/browse/SPARK-7075.

[22] M. Zaharia, Adaptive execution in spark, 2015. URL https://issues.apache.org/jira/browse/SPARK-9850.

[23] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling, in: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, ACM, New York, NY, USA, 2010, pp. 265–278. http://dx.doi.org/10.1145/1755913.1755940.

[24] 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, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 2–2. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

[25] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, in: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, KDD'97, AAAI Press, 1997, pp. 283–286. URL http://dl.acm.org/citation.cfm?id=3001392.3001454.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, R. Pan, Large-Scale parallel collaborative filtering for the netflix prize, in: Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management, AAIM '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–348. http://dx.doi.org/10.1007/978-3-540-68880-8_32.

Vinicius Dias is a Master Student of the Computer Science Department at Universidade Federal de Minas Gerais (UFMG), Brazil. He received his Bachelor degree at Universidade Federal de Uberlândia in 2013. His research interests include large-scale distributed systems, computer networks and data mining.

Wagner Meira Jr. obtained his Ph.D. from the University of Rochester in 1997 and is Full Professor at the Computer Science Department at Universidade Federal de Minas Gerais, Brazil. He has published more than 200 papers in top venues and is co-author of the book Data Mining and Analysis — Fundamental Concepts and Algorithms published by Cambridge University Press in 2014. His research focuses on scalability and efficiency of large scale parallel and distributed systems, from massively parallel to Internet-based platforms, and on data mining algorithms, their parallelization, and application to areas such as information retrieval, bioinformatics, and e-governance.

Dorgival Guedes is an Associate Professor of the Computer Science Department at Universidade Federal de Minas Gerais (UFMG). He received his Ph.D. from the University of Arizona, Tucson, in 1999. He also holds a BSEE and an M.S. in Computer Science from UFMG. His research interests include Computer Networks, Distributed Systems and Operating Systems, specially where they relate to the scalability of globally distributed applications, including areas such as Cloud Computing, Big-Data, and Software Defined Networks.

Please cite this article in press as: V. Dias, et al., Janus: Diagnostics and reconfiguration of data parallel programs, J. Parallel Distrib. Comput. (2018), https://doi.org/10.1016/j.jpdc.2018.02.030.