TARDIS: Optimal Execution of Scientific Workflows in Apache Spark

Daniel Gaspar, Fabio Porto, Reza Akbarinia, Esther Pacitti

TARDIS: Optimal Execution of Scientific Workflows in Apache Spark

Daniel Gaspar[1], Fabio Porto[1], Reza Akbarinia[2], and Esther Pacitti[3]

[1] LNCC - National Laboratory for Scientific Computing,
Av. Getúlio Vargas, 333, 25651-075 Petrópolis, RJ, Brazil
[2] INRIA - National Institute for Research in Computer Science and Control,
161 Rue Ada, 34095 Montpellier, France
[3] LIRMM - Montpellier Laboratory of Informatics, Robotics and Microelectronics,
860 Rue de St Priest, 34095 Montpellier, France

Abstract. The success of using workflows for modeling large-scale scientific applications has fostered the research on parallel execution of scientific workflows in shared-nothing clusters, in which large volumes of scientific data may be stored and processed in parallel using ordinary machines. However, most of the current scientific workflow management systems do not handle the memory and data locality appropriately. *Apache Spark* deals with these issues by chaining activities that should be executed in a specific node, among other optimizations such as the in-memory storage of intermediate data in *RDDs (Resilient Distributed Datasets)*. However, to take advantage of the RDDs, *Spark* requires existing workflows to be described using its own API, which forces the activities to be reimplemented in Python, Java, Scala or R, and this demands a big effort from the workflow programmers.
In this paper, we propose a parallel scientific workflow engine called *TARDIS*, whose objective is to run existing workflows inside a *Spark* cluster, using RDDs and smart caching, in a completely transparent way for the user, i.e., without needing to reimplement the workflows in the *Spark* API. We evaluated our system through experiments and compared its performance with *Swift/K*. The results show that *TARDIS* performs better (up to 138% improvement) than *Swift/K* for parallel scientific workflow execution.

1 Introduction

Over the last years, the volume of data produced by scientific simulations and experiments has been increasing in a astronomical rate. This increase is mainly a consequence of advances in sensors and the thriving of the Internet of Things, which amplify the quantities that are analysed and stored during an experiment. This leads to a field of study, called *big data*, that is interested in studying how to collect, store and process these enormous volumes of data.

Fortunately, there has also been plenty of development in the high-performance computing area. Usually, big data is stored and processed in parallel databases

running in dedicated and expensive servers. However, this approach is inappropriate for many large scale scientific applications due to its cost [14]. Besides, some scientific data is completely unstructured, or non-relational, therefore difficult to be dealt in databases. More recently, some approaches to process unstructured data over clusters of commodity hardware have appeared. These include *MapReduce* [3], *Spark* [13] and *Pegasus* [4].

In the context of scientific applications, the standard has been to express the scientific processes as workflows. Scientific workflows define a computational processing composed of *activities*. Each activity consumes input data and generates output. Activities are linked together forming a directed acyclic graph (DAG), taking in account the data dependency between them [9] [8] [6].

Typically, scientific workflows have been designed to run in clusters using a shared-disk model, inherited from HPC systems. For processing big datasets, however, moving data through the network jeopardize the computation.

By analysing current solutions for the parallel execution of scientific workflows, we observe that the great majority of them is concerned with the lack of data locality. In addition, currently there is not a complete coupling between the data management framework and the scientific workflow engine in most of the existing systems. Solutions based on MapReduce [3] or Hadoop [2] (like Pig Latin [10] or Hive [11]) do not analyse the entire workflow. They do not consider the chaining of activities when scheduling tasks to nodes. *Swift/K* is a C-like programming language for defining workflows that can be executed in clusters, among other architectures. The language coordinates the execution of activities defined as programs that consume and produce files [12]. However, the *Swift/K* engine does not ensure the data locality during the workflow execution.

Spark is an Apache open-source framework for processing big data in clusters. It allows storing data in memory and querying it repeatedly, therefore providing performance increase over *Hadoop* or other *MapReduce* implementations [13].

To use *Apache Spark*, users should develop their applications using the provided API, e.g., in Scala, Python or R. However, usually scientific workflows activities are defined in existing software, *e.g.*, *MAFFT*, *Align2D*, *Montage* or some scripts developed by the scientists [5] [7]. It is not trivial to execute a workflow designed in *Swift/K*, or any other workflow whose activities read and write from files in *Spark*.

In this paper, we propose *TARDIS* (Task Analyser Regarding Data in Spark), a parallel scientific workflow engine that allows to run workflows using *Spark*, without needing to rewrite their activities in *Spark* API. *TARDIS* executes workflows with activities which are scientific programs, allowing them to behave in *Spark* as a code written in one of its compatible languages reading and writing data from *Spark* RDDs. In addition, it deals with the optimal partitioning of the activities inputs in the *Spark* RDDs avoiding unnecessary data transfers during workflow execution. *TARDIS* also includes new algorithms for scheduling the input data in the *Spark* cluster.

We have compared the performance of *TARDIS* with *Swift/K* for executing a *Montage* workflow to generate an image of the sky from mosaics of tiles obtained

from space catalogues. The results show that *TARDIS* performs much better than *Swift/K* for the execution of workflows.

The rest of the paper is organized as follows. In the next section, we describe the problem we are facing. In Section 3, we present some background about *Spark*, and in Section 4, we present the *TARDIS* engine. In Section 5, we report our experimental results.

2 Problem Definition

A workflow $W = (A, F, I, O)$ is composed of a set of *activities* $A = \{a_1, ..., a_n\}$, a set of *files* $F = \{f_1, ..., f_m\}$, a set of *input dependencies* $I \subset (F \times A)$ and a set of *output dependencies* $O \subseteq (A \times F)$. Activities are defined as command-line invocations of external program that reads and writes into files. Additionally, we consider $N = \{n_1, ..., n_x\}$ a set of computer nodes and a function $\mu(F) \Rightarrow N$ that allocates a file $f_i \in F$ in a node $n_j \in N$. Furthermore, $I = \{I_1, I_2, \ldots, I_v\}$ is a set of sets of input dependencies such that if $(f_i, a_j) \in I_k$ and $(f_p, a_l) \in I_k$ then $j = l$, representing the set of files that are input to a single program, enabling the parallel execution of the corresponding scientific program.

Given a workflow W, let $a, a' \in A$ be activities. We define:

- **Input of an activity**: $input(a) = \{f \in F \mid \exists (f, a) \in I\}$.
- **Output of an activity**: $output(a) = \{f \in F \mid \exists (a, f) \in O\}$.
- **Activity dependency**: We say that a depends on a', $dep(a', a)$ if $output(a') \cap input(a) \neq \emptyset$.
- **Input of a workflow**: $input(W) = \{f \in F \mid \nexists\ a \in A : f \in output(a)\}$.

The output of a workflow $output(W) \subseteq F$, can be anything defined by the user.

The problem, which we address in this paper, is how to run a workflow W in a *shared-nothing* architecture pursuing data locality and using in-memory storage for intermediate data. This can be done by solving the following four sub-problems:

- **How to distribute input files in a *shared-nothing* cluster?** We want to optimize data locality by bringing the jobs close to the data and therefore minimizing the execution time of the workflow. We should consider the data transfers among nodes during the execution of the workflow. The challenge is to allocate the input files of a workflow on nodes of the shared-nothing cluster so that data locality can be achieved.
- **How to keep *Spark* scheduling under file allocation decision?** In *Spark*, parallel function scheduling is driven by RDDs partitioning. Running *non-Spark* functions and retaining *Spark* scheduling strategy requires allocating functions to nodes where files have been allocated.
- **How to benefit from local pipelines strategies in activities that should be chained together?** *Spark* executes pipelines of activities locally using memory for storing the intermediate data. This is achieved by

analysing the dependencies among the workflows activities. If possible, different activities are executed together locally in a *Spark stage*. We want to enforce *Spark* to perform this behaviour with our program-based workflows.
- **How to ensure fault-tolerance to the workflow execution?** When dealing with distributed computing, the chances of a failure become considerably high. We need to ensure that if a node fails, or a program execution terminates with error, the workflow execution should not only detect the error but also try to solve it by running again the failed fragment.

In this paper, our objective is to enable the efficient execution of existing scientific workflows in a shared-nothing execution architecture, using *Spark*, where programs run on local data and produce intermediate results using local memory.

3 Background

3.1 Spark

Apache Spark is an open-source parallel data processing framework developed and maintained by *Apache* that generalizes the *MapReduce* model [13, 1]. *Spark* manages data by introducing the RDD (*Resilient Distributed Datasets*) abstraction. These are read-only in-memory collections of objects distributed into the machines of a cluster.

A workflow may be executed in *Spark* via the API provided in *Scala*, *Python*, *Java* or *R* languages. Activities of a *Spark* workflow may be *transformations* or *actions*. *Transformations* consume a RDD and output another. *Actions* consume a RDD and output objects like the ones that compose the input RDD. For example, the *collect* action returns all of the objects present in a RDD and the *map* transformation applies a function to each object of the input RDD and outputs a new RDD with the transformed objects.

In *Spark*, the transformations are executed when their output RDD is needed for an action. Actually, the *Spark* engine knows what transformations should be done to generate a given RDD. When an action is called, *Spark* will execute all transformations necessary to generate the input data to that action. RDDs do not persist in memory, but if required by another operation, they can be recreated on-the-run.

Scientific workflows are typically executed in shared-disk architectures, and this limits data locality. Conversely, *Spark* is designed to explore in memory data locality. Unfortunately, *Spark* workflows constrain activities to be coded in one of is compatible languages which limits its applicabilities to a scientific workflow already existent.

4 TARDIS Engine

TARDIS (*Task Analyser Regarding Data In Spark*), is a parallel scientific workflow engine developed in Python that runs existing workflows over a shared-nothing *Spark* cluster. Each *Spark* node executes some activities of the workflow, and the data is spread throughout the cluster in the RDDs.

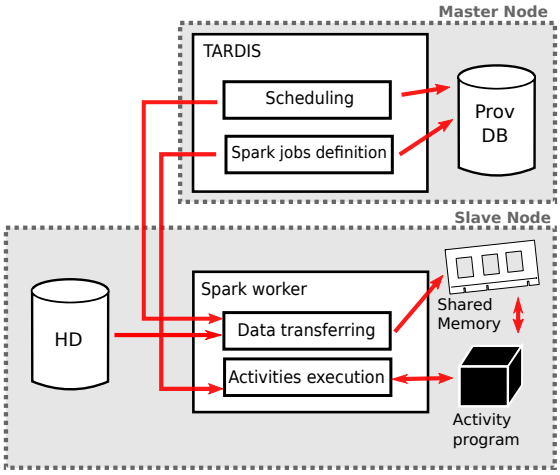

Fig. 1. TARDIS architecture.

4.1 Architecture

The *TARDIS* system is composed of two main modules: the master and slaves. The first is responsible for executing the files allocation scheduler and maintaining a *SQLite* database for provenance reasons. It connects to a *Spark* cluster and submits jobs to it. The *Spark* workers behave as *TARDIS* slave nodes. The main components of the master module are as follows:

- Job parsing: users provide a *workflow descriptor* to TARDIS using our own language. This component is responsible for parsing the job description given by the user;
- Job reformulation: User inputs his or her workflow using TARDIS defined activities types (such as *map* and *partial reduce*. During the initial workflow analysis, these activities are converted to a set of *Spark* activities. Extra ones are included in order to perform the correct execution of the workflow (i.e. to transfer data among nodes);
- Scheduling: it is responsible of deciding in which nodes the input files of activities should be consumed.

The slave module contains the following main components:

- Data placement: whenever the required files for an activity execution are not available in local shared memory, slave nodes will download them from other nodes or, if locally available, copy them from local disk;
- Workflow execution: finally, activities are executed by running external software over in-memory files.

The execution model of TARDIS achieves good performance by combining the scheduling of files to nodes and the corresponding Spark executors to achieve

data locality in activities execution. Thus, once the TARDIS scheduler defines a file allocation the system must ensure that the files are in their respective nodes once the execution of activities begin. In this context, an initial job activity downloads the files to executing nodes according to the schedule.

This transfer is achieved by running a lightweight HTTP server in each node, which exposes the in-memory files to the network. The input files are loaded into its own node shared memory and then other nodes may do an HTTP request to download the required files. This may happen also for intermediate files during the execution of the workflow, especially after a *Spark* shuffle.

These downloaded files are moved to a shared-memory file system mounted at each node to enable efficient in-memory pipelined activity execution of black-box programs. The latter run under the control of the *Activity Execution* (AE) module. The execution of black-box software should privilege data locality. The AE module must identify the file allocated to its node and pass it to its associated black-box program.

At the end of execution, the master collects the output files and stores them in a local folder.

The architecture of the TARDIS engine is depicted in Figure 1. *Prov DB* is a *SQLite* database and currently only records some data used for provenance. *HD* depicts the local storage of files in slave nodes.

4.2 TARDIS Language

TARDIS offers an adaptation of *Spark* API to enable the instrumentation of proposed execution as *TARDIS* activities. Users define their workflows using a *Python* script, calling our methods. Our system allows workflow activities to be defined as *maps*, *partial reduces* and *reduces*.

Map Activities. A *map activity* is an activity that consumes only one file of the input RDD and outputs one file. This activity will be performed in parallel over all files of the input.

It is defined using the *TARDIS* method `map_activity`. This method expects up to four parameters: the activity name or ID, the command that performs the activity, the input RDD and the pattern of files that will be affected by it. The name or ID is used only for provenance reasons. The command is given by the path to the executable that should be run over the files with all of its expected parameters. A special keyword `@!input` is replaced by the respective filename in each execution of the map.

The *pattern of files* describes in a *bash*-alike expression which files of the input RDD should be consumed by the activity. The default case is "`*`", which means that the command specified will be run over every file in the input RDD.

This method returns a transformed RDD with the respective output files and the files from the input that were not used.

For example, if a user wants to reverse all `txt` files in an RDD (the first line becomes the last, and do on), he or she can use the following command in Python:

```
reverseRDD = tardis.map_activity("reverse", "tac @!input >
             @!input.rev", filesRDD, "*.txt").
```

For instance, if `filesRDD` had three files: `text1.txt`,
`text2.txt` and `photo.jpg`, after running the previous command, `reverseRDD` would have also three files:
`text1.txt.rev`, `text2.txt.rev` and `photo.jpg`.

Partial Reduce Activities. A *partial reduce activity* consumes more than one file and outputs only one. This allows the user to specify more than one disjoint set of files that will be consumed together, so that the different sets can be executed in parallel. For instance, the user can reduce all *txt* files and all *jpg* files in parallel.

To run an activity like this, users can use the *TARDIS* method called `partial_reduce_activity`. The first 3 parameters are the same as the map method: the activity name or ID, the command that performs the activity and the input RDD. The fourth is an array of patterns of files that will be affected by it. The only difference is that in a partial reduce, the user can specify more than one pattern.

In the command string, `@!input` will be translated to all the affected filenames separated by a space, and there is a new placeholder, `@!output`, which will be translated to a filename-safe version of the current pattern.

For example, if the user wants to concatenate all `txt` files in one big file and all the `jpg` files in another, the following command can be used:

```
concatenatedRDD = tardis.partial_reduce_activity("cat", "cat
   @!input > all@!output", filesRDD, ("*.txt", "*.jpg")).
```

If we had four files in `filesRDD`: `text1.txt`, `text2.txt`, `photo1.jpg` and `photo2.jpg`, after the command `concatenatedRDD` we would have two files: `all.jpg` and `all.txt`.

Reduce Activities. A *reduce activity* is a specific case of the *partial reduce* one where only one pattern is given: `"*"`. This means that all files from the RDD will be consumed by only one instance of the command that performs the activity.

This is different from a reduce in a *MapReduce* or a *Spark* paradigm. In those cases, it can be executed in parallel, because the operation is transitive and binary (so it can be executed in a binary tree). As our operator is a black-box, we cannot assure that the operation is transitive and we cannot modify the input to receive only a pair of files.

Users can define this activity with the `reduce_activity` method. It expects three parameters: the activity name or ID, the command that performs the activity and the input RDD. The `@!input` placeholder can be used in the command to be translated to a list of all files available in the RDD. For instance, the user wants to concatenate all of the files in an RDD to a single file, the following command can be used:

```
finalRDD = tardis.reduce_activity("final_cat", "cat @!input >
                    allFiles", filesRDD).
```

In this example, the `*` wildcard could replace `@!input` with no loss of generalization. The `finalRDD` would have only one file, named `allFiles`.

4.3 Data Placement

As discussed in section 4.1, TARDIS distributes the files through nodes of the shared-nothing cluster, such that the scientific black-box softwares can be scheduled to access their input files locally. In this section, we present our file allocation algorithms for mapping each input file $f \in F$ to a node $n \in N$ in the cluster.

The file allocation algorithm should take into account the size of the file, their initial allocation and the cost to transfer the file from one node to another. This cost may vary if the nodes are not in the same network. Additionally, to avoid skew, the scheduler should consider the *ideal load* of each node. This ideal is given by the ratio of the computing capability of the respective node in relation to the general computing capability of the cluster. At this moment, we consider only the quantity of cores and the respective CPU frequency as a measure of this capability.

Let c_i be the computing capability of node n_i. $\sum_j c_j$ is the sum of the capabilities of the entire cluster. Supposing that to run a workflow W, the partition p_i is allocated to node n_i, the ideal size of p_i is given by

$$ideal\_size(p_i) = \frac{c_i \times \sum_{d \in input(W)} size(d)}{\sum_j c_j}. \quad (1)$$

We propose four different file allocation algorithms: *only local*, *greedy allocation*, *locals first* and *lazy allocation*. They are presented in the next sections.

File Allocation Algorithms.

Only local

This algorithm tries to minimize the data transfer during the execution of workflows by maximizing the data locality. It trivially allocates each file to be run in the same node where it is already stored. Algorithm 1 describes the *only local* allocation. The *FQDN* (fully qualified domain name) of the node is used as its identification.

Greedy allocation

The *Greedy allocation* algorithm pursues the optimal solution by reducing the skew of data according to the computing capability of each node. As discussed previously, each node is attributed an ideal fraction of the total input volume size (as computed by Equation 1). As shown in Algorithm 2, our greedy algorithm allocates the biggest files to the nodes with highest capability.

The algorithm proceeds as follows. After sorting the nodes and the files in descending order (lines 2 and 3), we allocate the biggest files to the node with

Algorithm 1 Only local algorithm

```
1: procedure LOCAL(nodes,objects)
2:     for obj in objects do
3:         for no in nodes do
4:             if obj.node.fqdn == no.fqdn then
5:                 no.alloc_obj_to(obj)
```

biggest capability that can still fit this file (line 6-10). If the file cannot fit any host, the host with the biggest capability, which has less tasks than it should, will receive the task (lines 11-15) even if this makes it have more tasks than what is required.

Algorithm 2 Greedy allocation algorithm

```
1: procedure GREEDY(nodes,objects)
2:     sort nodes by capability into descending order
3:     sort objects by size into descending order
4:     for object in objects do
5:         obj_allocated ← False
6:         for no in nodes do
7:             if obj.size < no.avail_size then
8:                 no.alloc_obj_to(obj)
9:                 obj_allocated ← True
10:                break
11:        if not obj_allocated then
12:            for no in nodes_greedy do
13:                if no.ideal_size > no.curr_size then
14:                    no.alloc_obj_to(obj)
15:                    break
```

Locals first

The *locals first* is a **hybrid** algorithm, in the sense that they tries to reduce both the skew and the data transfer among the nodes. The *locals first* algorithm tries to allocate the data to its local node, while this node has space (lines 2-6). After filling every node, the remained data will be allocated to the closest node with space available (lines 7-12). This is done by ordering the nodes with respect to the transfer cost to where the object currently is.

Lazy allocation

The *lazy allocation* algorithm, also hybrid, starts by allocating all data to the local node, like the *only local* algorithm (lines 2-5). Then, it redistributes the overflowing data to other nodes, as shown in Algorithm 4. This is done by ordering the files in a overflowing node by their size and removing the smallest ones until the node is close to its ideal size (lines 6-11). Then each removed file will be allocated to the closest node with available space (lines 12-16).

Algorithm 3 Locals first algorithm

```
1: procedure FIRST(nodes,objects)
2:     for obj in objects do
3:         for no in nodes do
4:             if obj.node.fqdn == no.fqdn and no.avail_size ≥ obj.size then
5:                 no.alloc_obj_to(obj)
6:                 objects.remove(obj)
7:     for obj in objects do
8:         sort nodes by transfer costs from obj.node into ascending order
9:         for no in nodes do
10:            if no.avail_size > 0 then
11:                node.alloc_obj_to(obj)
12:                break
```

Algorithm 4 Lazy allocation algorithm

```
1: procedure LAZY(nodes,objects)
2:     for obj in objects do
3:         for no in nodes do
4:             if obj.node.fqdn == no.fqdn then
5:                 no.alloc_obj_to(obj)
6:     for no in nodes do
7:         if no.avail_size < 0 then
8:             sort no.objs_to_run by size in ascending order
9:             while no is still overflowing do
10:                obj_taken ← no.objs_to_run[0]
11:                remove no.objs_to_run[0] from no
12:                ordered_nodes ← sort nodes by transfer costs from no into ascend-
    ing order
13:                for other_no in ordered_nodes do
14:                    if other_node.avail_size > 0 then
15:                        other_node.alloc_obj_to(obj_taken)
16:                        break
```

4.4 Scheduling

Given the *TARDIS* file allocation scheduling, we must conceive a strategy that would oblige *Spark* scheduling to follow the proposed solution. This is done by proceeding four main steps.

Step 1: Placeholders distribution. The output of the previous allocation algorithms is a table, with the input files and theirs respective nodes to be allocated to. We create an RDD containing $n \times f$ unique integers, where n is the number of nodes and f is the number of input files. The $n \times f$ copies of the scheduling table reserves f slots for each of the n nodes. The f slots are placeholders for possible files at each node. Each place-holder has an individual *id*. Later, each slot may hold a pointer to file allocated by the scheduling algorithm.

This is needed because it is not possible to increase the number of elements in the RDD later without *Spark* performing a shuffling of the data.

By invoking the `parallelize` method of *Spark* with the parameter n, we create n partitions of the allocation table and send them to the cluster. We cannot assure that each node will receive a RDD partition with f elements. So we check if the number of elements allocated to each node is higher or equal to the number of files that should be allocated to that specific node. If this condition is not assured, we restart this algorithm, this time creating an RDD with $m \times n \times f$ integers. m is initialized to 2, and is increased until the above condition is satisfied. This condition is checked in the *master node* after executing a *Spark* **map** transformation over the RDD which tags the element with the current node name. After caching it in the nodes and collecting all elements to the master, it knows not only the quantity of placeholders of each node, but also which exact ids have been allocated to each node. This information will be used for fault-tolerance reasons.

Step 2: Distribution of the allocation table. At this point, the *master node* has two tables, a file allocation one, generated by the algorithms presented in the previous section and an id allocation, collected by the previous step. By joining both tables we generate (id, file) pairs, associating different ids of one node to the files it should store. This new table will be broadcast to all nodes.

Step 3: Memory allocation. Then, we run a *Spark* `map` over this RDD with a *TARDIS* method that will be executed over each element of this array. This method will copy into memory the file related to the current id. This map returns a pointer to the file allocated in memory as well as its name. This copy will be done by locally reading the disk, if the file is already available in this node, or by transferring it via the network. This is done, currently, using the `rsync` utility. To allocate and access files in the memory, we use the *shared memory* area (SHM) of the operational system (i.e. Linux).

Step 4: Unused placeholders removal. Finally, a *Spark* `filter` transformation with a *TARDIS* routine is executed over the RDD to remove the elements flagged with zero. Thus, in this step we have an RDD full of filenames with pointers to its respective positions in local memory. Besides, we also have this RDD partitioned according to our scheduling algorithm.

4.5 Collecting Output Files

After executing the workflow, the `collect_files` *TARDIS* method should be used to write the files in the output RDD to the master node hard disk.

A folder called `output<N>` will be created in the current directory with the files inside it. `<N>` is a random integer that should allow the user to execute the workflow many times and keep the outputs. *TARDIS* prints to the standard output the name of the created folder.

5 Experiments

In this section, we report the results of our experiments done for validating *TARDIS* and evaluating its performance.

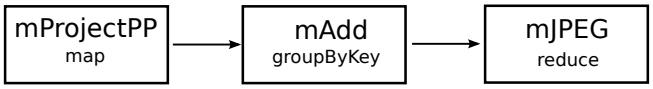

Fig. 2. The *Montage* workflow used in experiments

We tested *TARDIS* with a workflow using *Montage* to produce an image in a cluster. This workflow consumes 3.7 GB of input data and 20 GB of intermediate data. *Montage* is a toolkit to assemble astronomical images to generate mosaics, or panoramas [5]. It can be used for generating a colorful image of a certain part of the sky. To generate a color image, *Montage* uses tiles obtained from the DSS2 (Digitalized Sky Survey) with three filters: blue, red and infra-red. These are catalogs that are generated by using the pictures taken by telescopes.

Table 1. Execution time for different experiments

		6 nodes	3 nodes	1 node
TARDIS with *only local* allocation	average	8 min 04 sec	13 min 32 sec	19 min 57 sec
	maximum	8 min 05 sec	13 min 34 sec	20 min 18 sec
	minimum	8 min 03 sec	13 min 30 sec	19 min 44 sec
TARDIS with *greedy* allocation	average	8 min 26 sec	14 min 28 sec	
	maximum	8 min 37 sec	14 min 30 sec	
	minimum	8 min 21 sec	14 min 25 sec	
TARDIS with *locals first* allocation	average	8 min 03 sec	13 min 34 sec	
	maximum	8 min 04 sec	13 min 35 sec	
	minimum	8 min 03 sec	13 min 33 sec	
TARDIS with *lazy* allocation	average	8 min 10 sec	13 min 33 sec	
	maximum	8 min 15 sec	13 min 38 sec	
	minimum	8 min 06 sec	13 min 32 sec	
Swift/K	average	16 min 51 sec	16 min 20 sec	10 min 42 sec
	maximum	19 min 13 sec	23 min 03 sec	12 min 27 sec
	minimum	14 min 20 sec	12 min 15 sec	9 min 17 sec

During the required image generation, the tiles (photos obtained from telescopes) need to be normalized and unified to cover the area requested by the user. Besides, after doing this for the three filters, they need to be combined to

generate a color image. Although all the tiles of the three channels are used to generate the final image, intermediate activities perform a *reduce* operation in each channel separately.

Figure 2 shows the Montage workflow, with its activities and types. The *mProjectPP* activity projects some tiles to the coordinates of the final image. The *mAdd* concatenate the different tiles, already reprojected. This is done for three color-channels: red, blue and infrared. Finally, *mJPEG* creates a colorful picture using the three channels.

The experiments were run in a cluster running Linux (CentOS distribution). Each node has two *Intel(R) Xeon(R) CPU E5-2630 v3* running at 2.40GHz with eight physical cores each. The RAM memory available in each node is 94 GB. The cluster nodes were used for running this workflow using six, three and one node. We compare the execution time of *TARDIS* and its different scheduling algorithms with *Swift/K*, which is the state-of-art workflow engine.

After performing 10 times each experiment, the averages, maximums and minimums of the execution time are presented in Table 1.

We see that *TARDIS* performs better than *Swift/K* for the execution of the workflow when the number of nodes is higher than one. With 6 nodes, the best algorithm is *TARDIS* with *locals first allocation*. *Swift/K* does not pursue data locality, thus its performance is not very good compared to *TARDIS* when the number of nodes increases. However, it performs better than *TARDIS* when running in only one node, i.e., not parallel. The reason is that *TARDIS* is optimized for running in distributed environments, particularly its data allocation and scheduling modules.

6 Conclusion

In this paper, we proposed *TARDIS*, a *Spark*-based parallel workflow execution system that pursues data locality and minimizes the skew among nodes. It executes existing workflows, without needing to re-implement workflow activities in Spark API. It uses the nodes memory to export the data from *Spark* to activities that consume and generate data from/to files. We also proposed four data allocation algorithms to be used with *TARDIS*. In our experiments, jobs with different allocations incurred in very close elapsed-time. This is mainly due to the fact that files were already evenly distributed among cluster nodes. We intend to further investigate the impact of allocation algorithms in future work. Moreover, we proposed techniques for modifying the scheduling of Spark in order to optimize the workflow execution in *TARDIS* and to take into account our data placement strategies.

We evaluated the performance of *TARDIS* and compared it with *Swift/K*. The results show that the data locality and in-memory execution of *TARDIS* is highly adequate to parallel workflow execution in distributed environments, leading to an improvement of up to 138% in execution time.

References

1. Apache: Apache spark programming guide. https://spark.apache.org/docs/2.0.1/programming-guide.html
2. Apache: Hadoop. http://hadoop.apache.org/
3. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008), http://doi.acm.org/10.1145/1327452.1327492
4. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., et al.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming 13(3), 219–237 (2005)
5. Jacob, J.C., Katz, D.S., Berriman, G.B., Good, J.C., Laity, A., Deelman, E., Kesselman, C., Singh, G., Su, M.H., Prince, T., et al.: Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. International Journal of Computational Science and Engineering 4(2), 73–87 (2009)
6. Liroz-Gistau, M., Akbarinia, R., Pacitti, E., Porto, F., Valduriez, P.: Dynamic workload-based partitioning for large-scale databases. Database and Expert Systems Applications (Jan 2012), http://dx.doi.org/10.1007/978-3-642-32597-7_16
7. Ocaña, K., de Oliveira, D.: Parallel computing in genomic research: advances and applications. Advances and applications in bioinformatics and chemistry: AABC 8(23) (2015)
8. Oliveira, D., Boeres, C., Porto, F., Fausti, A.: Avaliação da localidade de dados intermediários na execução paralela de workflows bigdata. In: SBBD Proceedings 2015 (2015)
9. de Oliveira, D.E.M., Boeres, C., Porto, F.: Análise de estratégias de acesso a grandes volumes de dados. In: SBBD Proceedings 2014 (2014)
10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: A not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 1099–1110. SIGMOD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1376616.1376726
11. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A warehousing solution over a map-reduce framework. Proc. VLDB Endow. 2(2), 1626–1629 (Aug 2009), http://dx.doi.org/10.14778/1687553.1687609
12. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing 37(9), 633–652 (2011)
13. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (2012)
14. Zhou, J., Bruno, N., Wu, M.C., Larson, P.A., Chaiken, R., Shakib, D.: Scope: Parallel databases meet mapreduce. The VLDB Journal 21(5), 611–636 (Oct 2012), http://dx.doi.org/10.1007/s00778-012-0280-z