Scalable Script-based Data Analysis Workflows on Clouds

Fabrizio Marozzo
DIMES
University of Calabria
Italy
fmarozzo@dimes.unical.it

Domenico Talia
DIMES
University of Calabria
ICAR-CNR
Italy
talia@dimes.unical.it

Paolo Trunfio
DIMES
University of Calabria
Italy
trunfio@dimes.unical.it

ABSTRACT

Data analysis workflows are often composed by many concurrent and compute-intensive tasks that can be efficiently executed only on scalable computing infrastructures, such as HPC systems, Grids and Cloud platforms. The use of Cloud services for the scalable execution of data analysis workflows is the key feature of the Data Mining Cloud Framework (DMCF), which provides a Web interface to develop data analysis applications using a visual workflow formalism. In this paper we describe how we extended DMCF to support also the design and execution of script-based data analysis workflows on Clouds. We introduce a workflow language, named JS4Cloud, that extends JavaScript to support the implementation of Cloud-based data analysis tasks and the handling of data on the Cloud. We also describe how data analysis workflows programmed through JS4Cloud are processed by DMCF to make parallelism explicit and to enable their scalable execution on Clouds. Finally, we present a data analysis application developed with JS4Cloud, and the performance results obtained executing the application with DMCF on the Windows Azure platform.

Categories and Subject Descriptors

Computer systems organization [**Architectures**]: Distributed architectures—*Cloud computing*

Keywords

Workflows, Data analysis, Data mining, Cloud computing, Scalability, JS4Cloud

1. INTRODUCTION

Workflows are an effective paradigm to address the complexity of scientific and business applications. They provide a declarative way of specifying the high-level logic of an application while hiding the low-level details that are not fundamental for application design [14][13]. The use of workflows has proven to be very effective to describe complex data analysis processes, e.g. Knowledge Discovery in Databases (KDD) applications, which can be conveniently modelled as graphs linking together data sources, filtering tools, data mining algorithms, and knowledge models.

Data analysis workflows are often composed by many concurrent and compute-intensive tasks that can be efficiently handled only on scalable computing infrastructures, such as HPC systems, Grids and Cloud platforms. The use of Cloud services for the scalable execution of data analysis workflows is the key feature of the *Data Mining Cloud Framework* (DMCF), presented in [9]. In DMCF, data analysis workflows are designed through visual programming, which is a very effective design approach for high-level users, e.g. domain-expert analysts having a limited understanding of programming. In addition, a graphical representation of workflows intrinsically captures parallelism at the task level, without the need to make parallelism explicit through control structures [7]. On the other hand, script-based workflows can be used as an effective alternative to graphical workflows, since the formers can allow expert users to program complex applications more rapidly, in a more concise way, and with higher flexibility. Therefore, we extended the DMCF system to support also script-based data analysis workflows, as an additional and more flexible programming interface for skilled users.

In this paper we describe our solution for programming and executing parallel script-based data analysis workflows in DMCF. We introduce a workflow language, named JS4Cloud, that extends JavaScript to support the development of Cloud-based data analysis tasks and the access to data on the Cloud. The main benefits of JS4Cloud are: *i*) it is based on a well known scripting language, so that users do not have to learn a new programming language from scratch; *ii*) it implements a data-driven task parallelism that automatically spawns ready-to-run tasks to the available Cloud resources; *iii*) it exploits implicit parallelism so application workflows can be programmed in a totally sequential way, which frees users from duties like work partitioning, synchronization and communication.

The remainder of the paper is organized as follows. Section 2 shortly presents the Data Mining Cloud Framework and its visual workflow formalism. Section 3 presents JS4Cloud and discusses how workflows programmed through this language are executed by DMCF. Section 4 describes a data analysis application developed with JS4Cloud, and presents performance results obtained executing the application with DMCF on the Windows Azure platform. Sec-

tion 5 discusses related work. Finally, Section 6 concludes the paper.

2. DATA MINING CLOUD FRAMEWORK

The *Data Mining Cloud Framework* (DMCF) is a software framework for designing and executing data analysis workflows on the Cloud. DMCF supports a large variety of data mining processes, including single-task applications, parameter sweeping application, and workflow-based applications. Following the approach proposed in [2], DMCF represents knowledge discovery workflows as graphs whose nodes denote resources (datasets, data mining tools, data mining models) and whose edges denote dependencies among resources. A Web-based user interface allows users to compose workflows and to submit them for execution to Cloud platforms, following a Software-as-a-Service approach.

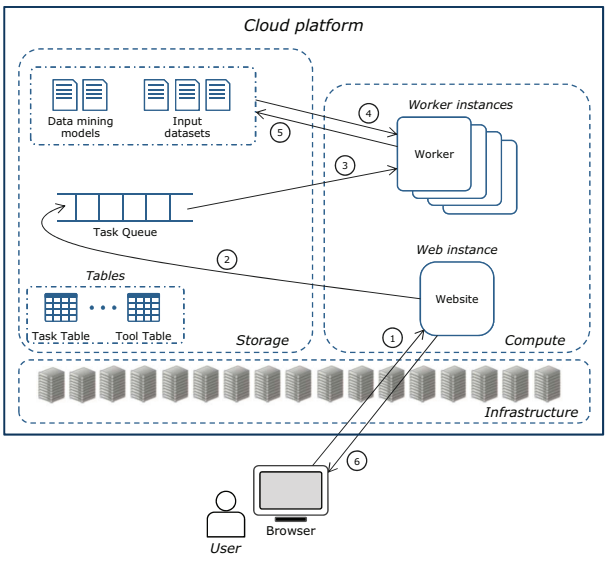


Figure 1: Architecture of the Data Mining Cloud Framework.

The architecture of DMCF includes the following components (see Figure 1):

- A set of binary and text data containers used to store data to be mined (*Input datasets*) and the results of data mining tasks (*Data mining models*).
- A *Task Queue* that contains the workflow tasks to be executed.
- A *Task Table* and a *Tool Table* that keep information about current tasks and available tools.
- A pool of k *Workers*, where k is the number of virtual servers available, in charge of executing the workflow tasks.
- A *Website* that allows users to submit, monitor the execution, and access the results of knowledge discovery workflows.

The DMCF architecture has been designed in a sufficiently abstract and generic way to be implemented on top of the IaaS level of different Cloud systems. In fact, we are working to implement it on commercial and open source Cloud platforms. However, the current implementation we discuss here is based on Windows Azure[1].

2.1 Execution mechanisms

A user interacts with the system to perform the following steps for designing and executing a knowledge discovery application:

1. The user accesses the Website and designs the application through an HTML-5 interface. A service catalog provides her/him with the available data and tools that can be used in the application that must be developed.
2. After application submission, the runtime identifies the workflow tasks and inserts them into the Task Queue on the basis of the dependencies among them.
3. Each idle Worker picks a task from the Task Queue, and concurrently executes it on a virtual compute server.
4. Each Worker gets the input dataset from its original location.
5. After task completion, each Worker puts the result on a data storage element.
6. The Website notifies the user as soon as her/his task(s) have completed, and allows her/him to access the results.

All the potential parallelism of the workflow is exploited by using all the virtual compute servers available to the user. In addition, multi-threaded tasks exploit all the cores available on the virtual compute servers they are assigned to.

2.2 Visual workflow formalism

Visual workflows in DMCF are directed acyclic graphs whose nodes represent resources and whose edges represent the dependencies among the resources. Workflows includes two types of nodes:

- *Data* node, which represents an input or output data element. Two subtypes exist: *Dataset*, which represents a data collection, and *Model*, which represents a model generated by a data analysis tool (e.g., a decision tree).
- *Tool* node, which represents a tool performing any kind of operation that can be applied to a data node (filtering, splitting, data mining, etc.).

The nodes can be connected with each other through direct edges, establishing specific dependency relationships among them. When an edge is being created between two nodes, a label is automatically attached to it representing the kind of relationship between the two nodes.

Data and Tool nodes can be added to the workflow singularly or in array form. A *data array* is an ordered collection of input/output data elements, while a *tool array* represents multiple instances of the same tool.

Figure 2 shows an example of data analysis workflow developed using the visual workflow formalism of DMCF. In

[1]http://www.microsoft.com/windowsazure

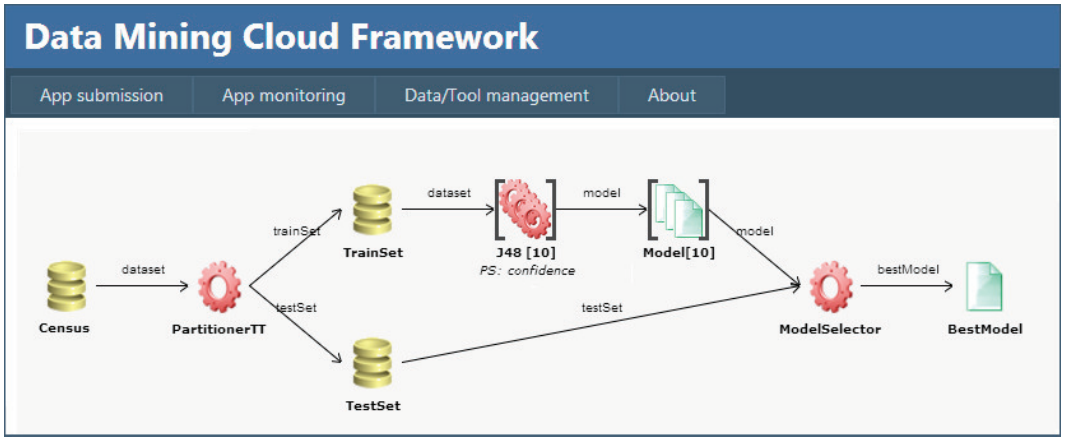


Figure 2: Example of data analysis application designed using the DMCF visual formalism.

this example, the Census dataset is split into a training set and a test set using a partitioning tool. Then the training set is analyzed in parallel by ten instances of the J48 classification tool, which are represented as a single tool array node in the workflow. The J48 instances differ each other only for the value of one input parameter (the confidence factor). The ten models generated by the J48 instances, represented as a data array, are then evaluated against the test set by a ModelSelector to identify the best model, which is the final output of the workflow.

3. THE JS4CLOUD LANGUAGE

JS4Cloud (JavaScript for Cloud) is a JavaScript-based language for programming data analysis workflows. It has been introduced as the script-based language for the Data Mining Cloud Framework (DMCF). The Web interface of DMCF allows to design and execute workflows programmed by the JS4Cloud language, by providing an environment similar to that used to develop visual workflows in the same framework. The basic difference is in the programming editor supporting JS4Cloud program design instead of the graphical composition system.

The main benefits of JS4Cloud are: *i*) it is based on a well known scripting language, so that users do not have to learn a new programming language from scratch; *ii*) it implements a data-driven task parallelism that automatically spawns ready-to-run tasks to the available Cloud resources; *iii*) it exploits implicit parallelism so application workflows can be programmed in a totally sequential way, which frees users from duties like work partitioning, synchronization and communication.

Two strength points of JavaScript motivated its adoption as the basis for JS4Cloud: *i*) JavaScript natively provides support to arrays and calls to external functions, which are fundamental to implement parallelism and remote task execution in DMCF; *ii*) JavaScript code can be executed using the standard interpreters available in any modern Web browser, a key feature to write and execute script-based workflows using the Web interface of DMCF.

3.1 Key programming concepts

Two key programming abstractions in JS4Cloud are *Data* and *Tool*:

- *Data* elements denote input files or storage elements (e.g., a dataset to be analyzed) or output files or stored elements (e.g., a data mining model).

- *Tool* elements denote algorithms, software tools or complex applications performing any kind of operation that can be applied to a data element (data mining, filtering, partitioning, etc.).

For each Data and Tool element included in a JS4Cloud workflow, an associated descriptor, expressed in JSON format, will be included in the environment of the user who is developing the workflow.

A Tool descriptor includes a reference to its executable, the required libraries, and the list of input and output parameters. Each parameter is characterized by name, description, type, and can be mandatory or optional. An example of descriptor for a data classification tool is presented in Figure 3.

The JSON descriptor of a new tool is created automatically through a guided procedure provided by DMCF, which allows users to specify all the needed information for invoking the tool (executable, input and output parameters, etc.).

Similarly, a Data descriptor contains information to access an input or output file, including its identifier, location, and format. Differently from Tool descriptors, Data descriptors can also be created dynamically as a result of a task operation during the execution of a JS4Cloud script. For example, if a workflow W reads a dataset D_i and creates (writes) a new dataset D_j, only D_i's descriptor will be present in the environment before W's execution, whereas D_j's descriptor will be created at runtime.

Another key element in JS4Cloud is the *task* concept, which represents the unit of parallelism in our model. A task is a Tool, invoked from the script code, which is intended to run in parallel with other tasks on a set of Cloud resources.

According to this approach, JS4Cloud implements *data-driven task parallelism*. This means that, as soon as a task does not depend on any other task in the same workflow, the runtime asynchronously spawns it to the first available virtual machine. A task T_j does not depend on a task T_i belonging to the same workflow (with $i \neq j$), if T_j during its execution does not read any data element created by T_i.

```
"J48": {
 "libraryList": ["java.exe","weka.jar"],
 "executable": "java.exe -cp weka.jar
   weka.classifiers.trees.J48",
 "parameterList":[{
     "name": "dataset",
     "flag": "-t",
     "mandatory": true,
     "parType": "IN",
     "type": "file",
     "array": false,
     "description": "Input Dataset"
   },{
     "name": "confidence",
     "flag": "-C",
     "mandatory": false,
     "parType": "OP",
     "type": "real",
     "array": false,
     "description": "Confidence value",
     "value": "0.25"
   },{
     "name": "model",
     "flag": "-d",
     "mandatory": true,
     "parType": "OUT",
     "type": "file",
     "array": false,
     "description": "Output model"}]
}
```

Figure 3: Example of Tool descriptor in JSON format.

3.2 JS4Cloud Functions

JS4Cloud extends JavaScript with three additional functionalities, implemented by the set of functions listed in Table 1:

- *Data Access*, for accessing a data element stored in the Cloud;
- *Data Definition*: to define a new data element that will be created at runtime as a result of a tool execution;
- *Tool Execution*: to invoke the execution of a tool available in the Cloud.

Data Access is implemented by the `Data.get` function, which is available in two versions: the first one receives the name of a data element, and returns a reference to it; the second one returns an array of references to the data elements whose name match the provided regular expression. For example, the following statement:

```
var ref = Data.get("Census");
```

assigns to variable `ref` a reference to the dataset named `Census`, while the following statement:

```
var ref = Data.get(new RegExp("^CensusPart"));
```

assigns to `ref` an array of references (`ref[0]...ref[n-1]`) to all the datasets whose name begins with `CensusPart`.

Data Definition is done through the `Data.define` function, available in three versions: the first one defines a single data element; the second one defines a one-dimensional array of data elements; the third one defines a multi-dimensional array of data elements. For instance, the following piece of code:

```
var ref = Data.define("CensusModel");
```

defines a new data element named `CensusModel` and assigns its reference to variable `ref`, while the following statement:

```
var ref = Data.define("CensusModel", 16);
```

defines an array of data elements of size 16 (`ref[0]...ref[15]`). In both cases, the data elements will be created at runtime as result of a tool execution.

Differently from Data Access and Data Definition, there is not a named function for Tool Execution. In fact, the invocation of a tool T is made by calling a function with the same name of T. For example, the J48 tool defined in Figure 3 can be invoked as in the following statement:

```
J48({dataset:DRef, confidence:0.05, model:MRef});
```

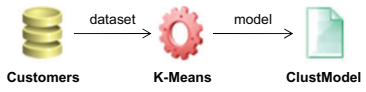
where `DRef` is a reference to the dataset to be analyzed, previously introduced using the `Data.get` function, and `MRef` is a reference to the model to be generated, previously introduced using `Data.define`.

3.3 Basic patterns

In the following we describe how the basic control flow patterns can be programmed with JS4Cloud. We focus on basic patterns [1] such as *single task*, *pipeline*, *data partitioning* and *data aggregation*, and on two additional patterns provided by the visual workflow formalism of DMCF, namely *parameter sweeping* and *input sweeping*. For each pattern, we first introduce an example as a visual DMCF workflow, and then we show how the same example can be coded using JS4Cloud.

Single task

An example of single-task pattern is shown in the following figure:



This example represents a K-Means tool that produces a clustering model from a dataset. Each workflow node hides some configuration parameters that have been set by the user, e.g., the number of clusters for the K-Means tool. The following JS4Cloud script is equivalent to the visual workflow shown above:

```
var DRef = Data.get("Customers");
var nc = 5;
var MRef = Data.define("ClustModel");
K-Means({dataset:DRef, numClusters:nc, model:MRef});
```

The script accesses the dataset to be analyzed (`Customers`), sets to `5` the number of clusters, and defines the name of data element that will contain the clustering model (`ClustModel`). Then, the `K-Means` tool is invoked along with the parameters indicated in its JSON descriptor (input dataset, number of clusters, output model).

Pipeline

In the pipeline pattern, the output of a task is the input for the subsequent task, as in the following example:

Table 1: JS4Cloud functions.

Functionality	*Function*	*Description*
Data Access	Data.get(<*dataName*>);	Returns a reference to the data element with the provided name.
	Data.get(new RegExp(<*regular expression*>));	Returns an array of references to the data elements whose name match the regular expression.
Data Definition	Data.define(<*dataName*>);	Defines a new data element that will be created at runtime.
	Data.define(<*arrayName*>,<*dim*>);	Define an array of data elements.
	Data.define(<*arrayName*>,[<dim_1>,...,<dim_n>]);	Define a multi-dimensional array of data elements.
Tool Execution	<*toolName*>(<par_1>:<val_1>,...,<par_n>:<val_n>);	Invokes an existing tool with associated parameter values.

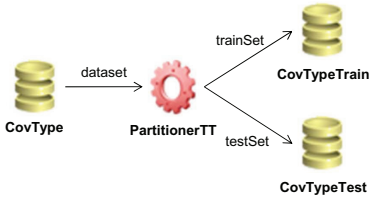
The first part of the shown example extracts a sample from an input dataset using a tool named Sampler. The second part creates a classification model from the sample using the J48 tool. This pattern example can be implemented in JS4Cloud as follows:

```
var DRef = Data.get("Census");
var SDRef = Data.define("SCensus");
Sampler({input:DRef, percent:0.25, output:SDRef});
var MRef = Data.define("CensusTree");
J48({dataset:SDRef, confidence:0.1, model:MRef});
```

In this case, since `J48` receives as input the output of `Sampler`, the former will be executed only after the end of the latter.

Data partitioning

The data partitioning pattern produces two or more output data from an input data element, as in the following example:



In this example a training set and a test set are extracted from a dataset, using a tool named PartitionerTT. With JS4Cloud, this can be written as follows:

```
var DRef = Data.get("CovType");
var TrRef = Data.define("CovTypeTrain");
var TeRef = Data.define("CovTypeTest");
PartitionerTT({dataset:DRef, percTrain:0.70,
   trainSet:TrRef, testSet:TeRef});
```

If data partitioning is used to divide a dataset into a number of splits, the DMCF's data array formalism can be conveniently used as in the following example:

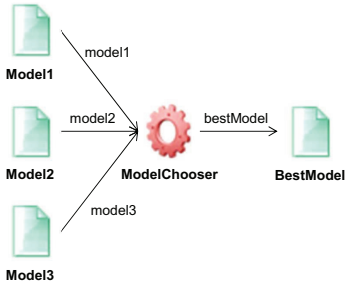
In this case, a Partitioner tool splits a dataset into 16 parts. The corresponding JS4Cloud code is:

```
var DRef = Data.get("NetLog");
var PRef = Data.define("NetLogParts", 16);
Partitioner({dataset:DRef, datasetParts:PRef});
```

Note that an array of `16` data elements is first defined and then created by the `Partitioner` tool.

Data aggregation

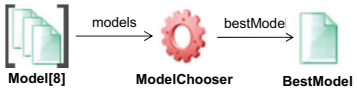
The data aggregation pattern generates one output data from multiple input data, as in the following example:



In this example, a ModelChooser tool takes as input three data mining models and chooses the best one based on some evaluation criteria. The corresponding JS4Cloud script is:

```
var M1Ref = Data.get("Model1");
var M2Ref = Data.get("Model2");
var M3Ref = Data.get("Model3");
var BMRef = Data.define("BestModel");
ModelChooser({model1:M1Ref, model2:M2Ref,
   model3:M3Ref, bestModel:BMRef});
```

DMCF's data arrays may be used for a more compact visual representation. For example, the following pattern example chooses the best one among 8 models:



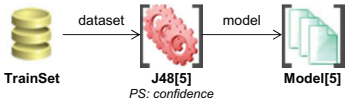
The same task can be coded as follows using JS4Cloud:

```
var BMRef = Data.define("BestModel");
ModelChooser({models:MsRef, bestModel:BMRef});
```

In this script, it is assumed that `MsRef` is a reference to an array of models created on a previous step.

Parameter sweeping

Parameter sweeping is a data analysis pattern in which a dataset is analyzed by multiple instances of the same tool with different parameters, as in the following example:



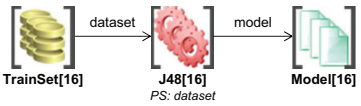
In this example, a training set is processed in parallel by 5 instances of J48 to produce the same number of data mining models. The DMCF's tool array formalism is used to represent the 5 tools in a compact form. The J48 instances differ each other by the value of a single parameter, the *confidence* factor, which has been configured (through the visual interface) to range from 0.1 to 0.5 with a step of 0.1. The equivalent JS4Cloud script is:

```
var TRef = Data.get("TrainSet");
var nMod = 5;
var MRef = Data.define("Model", nMod);
var min = 0.1;
var max = 0.5;
for(var i=0; i<nMod; i++)
   J48({dataset:TRef, model:MRef[i],
      confidence:(min+i*(max-min)/(nMod-1))});
```

In this case, the `for` construct is used to create `5` instances of `J48`, where the `i`-th instance takes as input the same training set (`TRef`), and produces a different model (`MRef[i]`), using a specific value for the confidence parameter (`0.1` for `J48[0]`, `0.2` for `J48[1]`, and so on). It is worth noticing that the tools are independent each other, and so the runtime can execute them in parallel.

Input sweeping

Input sweeping is a pattern in which a set of input data is analyzed independently to produce the same number of output data. It is similar to the parameter sweeping pattern, with the difference that in this case the sweeping is done on the input data rather than on a tool parameter. An example of input sweeping pattern is represented in the following figure:



In this example, 16 training sets are processed in parallel by 16 instances of J48, to produce the same number of data mining models. Data arrays are used to represent both input data and output models, while a tool array is used to represent the J48 tools. The following JS4Cloud script corresponds to the example shown above:

```
var nMod = 16;
var MRef = Data.define("Model", nMod);
for(var i=0; i<nMod; i++)
   J48({dataset:TsRef[i], model:MRef[i],
      confidence:0.1});
```

It is assumed that `TsRef` is a reference to an array of training sets created on a previous step. The `for` loop creates `16` instances of `J48`, where the `i`-th instance takes as input `TsRef[i]` to produce `MRef[i]`. Also in this case, since the tools are independent each other, they can be executed in parallel by the runtime.

3.4 Parallelism exploitation

As explained above, as soon as a task in a JS4Cloud workflow does not depend on any other task, the DMCF runtime asynchronously spawns it to the first available virtual machine. To better explain how parallelism is exploited with this approach, let us consider again the visual workflow shown in Figure 2, which performs a data classification with parameter sweeping. The equivalent JS4Cloud workflow is shown in the left part of Figure 4.

The workflow can be seen as composed of three steps. In the first step, PartitionerTT splits the input dataset into training set and test set (task $T1$). The second step consists in the concurrent execution of 10 instances of J48 (tasks $T2_0 ... T2_9$). During the third step, ModelSelector chooses the best model (task $T3$).

Overall, the workflow generates 12 tasks that are related each other as specified by the dependency graph shown in the right part of Figure 4. The graph shows that, as soon as $T1$ completes, tasks $T2_0 ... T2_9$ can be executed. After completion of all such tasks, $T3$ can be finally executed. The parallelism exhibited by the graph is fully exploited by executing the dependency-free tasks on the available virtual machines. In this case, tasks $T2_0 ... T2_9$ will run in parallel, thus resulting in a significant execution speedup.

4. PERFORMANCE EVALUATION

In this section we present some experimental performance results obtained executing a JS4Cloud workflow with the Data Mining Cloud Framework. The Cloud environment used for the experiment was composed by 64 virtual servers, each one equipped with a single-core 1.66 GHz CPU, 1.75 GB of memory, and 225 GB of disk space.

The workflow used for this evaluation analyzes a dataset using n instances of the J48 classification algorithm that work on n partitions of the training set and generate n knowledge models. By using the n generated models and the test set, n classifiers produce in parallel n classified datasets (n classifications). In the final step of the workflow, a voter generates the final classification (in the file `FinalClassTestSet`) by assigning a class to each data item. This is done by choosing the class predicted by the majority of the models [16].

The input dataset, containing about 46 million tuples and with a size of 5 GB, was generated from the *KDD Cup 1999*'s dataset[2], which contains a wide variety of simulated intrusion records in a military network environment.

Figure 5 shows the JS4Cloud code of the workflow. At the beginning, the input dataset is split into training set and test set by a partitioning tool (*line 3*). Then, the training set is partitioned into 64 parts using another partitioning tool (*line 5*). As third step, the training sets are analyzed in parallel by 64 instances of the J48 classification algorithm, to produce the same number of classification models (*lines 7-8*). The fourth step classifies the test set using the 64

[2]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99

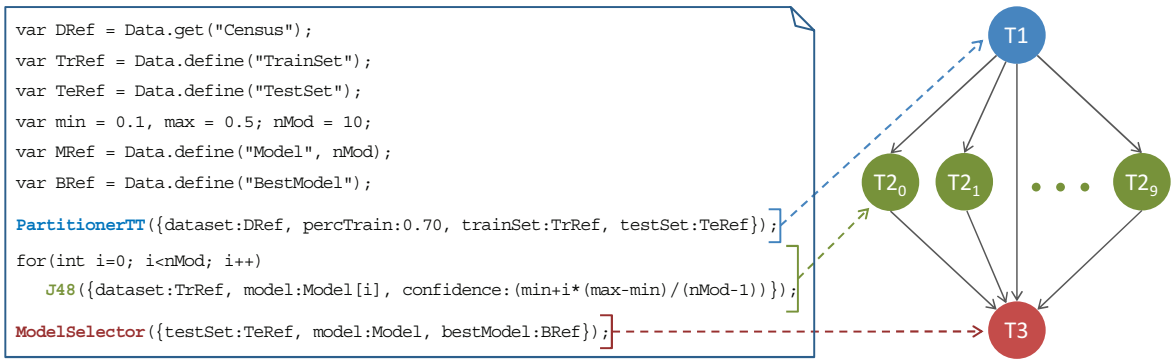


Figure 4: JS4Cloud script equivalent to the workflow in Figure 2, with associated task dependency graph.

```
 1: var n = 64;
 2: var DRef = Data.get("KDDCup99_5GB"),
        TrRef = Data.define("TrainSet"),
        TeRef = Data.define("TestSet");
 3: PartitionerTT({dataset:DRef, percTrain:0.7,
        trainSet:TrRef, testSet:TeRef});
 4: var PRef = Data.define("TrainsetPart", n);
 5: Partitioner({dataset:TrRef, datasetPart:PRef});
 6: var MRef = Data.define("Model", n);
 7: for(var i=0; i<n; i++)
 8:     J48({dataset:PRef[i], model:MRef[i],
           confidence:0.1});
 9: var CRef = Data.define("ClassTestSet", n);
10: for(var i=0; i<n; i++)
11:     Classifier({dataset:TeRef, model:MRef[i],
           classDataset:CRef[i]});
12: var FRef = Data.define("FinalClassTestSet");
13: Voter({classData:CRef, finalClassData:FRef});
```

Figure 5: JS4Cloud workflow for data classification.

models generated on the previous step (*lines 10-11*). The classification is performed by 64 classifiers that run in parallel to produce 64 classified test sets. As the last operation, the 64 classified test sets are passed to a voter that produces the final classified test set. The workflow is composed of 131 tasks, which are related each other as specified by the dependency graph shown in Figure 6.

Figure 7 shows a snapshot of the workflow taken during its execution in the DMCF's user interface. Beside each code line number, a colored circle indicates the status of execution. The green circles at lines 3 and 5 indicate that the two partitioners have completed their execution; the blue circle at line 8 indicates that J48 tasks are still running; the orange circles indicates that the corresponding tasks are waiting to be executed.

Figure 8 shows the turnaround times of the workflow, obtained varying the number of virtual servers used to run it on the Cloud from 1 (sequential execution) to 64 (maximum parallelism). As shown in the figure, the turnaround time decreases from more than 107 hours (4.5 days) by using a single server, to about 2 hours on 64 servers. This is an evident and significant reduction of time, which proves the system scalability.

The scalability achieved by the system can be further

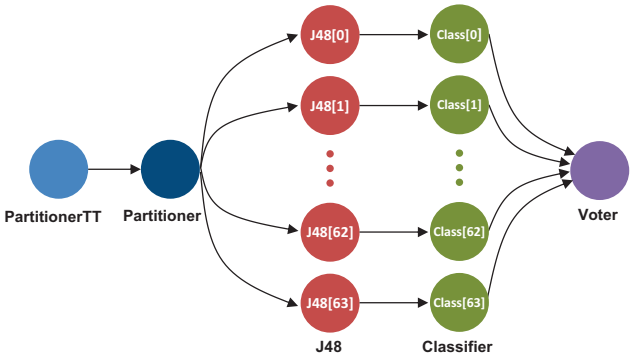


Figure 6: Task dependency graph associated with the workflow in Figure 5.

evaluated through Figure 9, which illustrates the relative speedup achieved by using up to 64 servers. As shown in the figure, the speedup increases from 7.64 using 8 servers to 50.78 using 64 servers. This is a very positive result, taking into account that some sequential parts of the implemented application (namely, partitioning and voting) do not run in parallel.

Figure 10 shows the application efficiency, calculated as the speedup divided by the number of used servers. As shown in the figure, efficiency on 32 servers is equal to 0.9 whereas on 64 servers it is equal to 0.8. Thus in this case, 80% of the computing power of each used server is exploited.

5. RELATED WORK

Several systems have been proposed to design workflows using script-based or visual formalisms [13], but only some of them currently work on the Cloud. In the following, we briefly discuss the most representative Cloud-based workflow management systems that support either script-based or visual workflow design.

Pegasus [3], developed at the University of Southern California, includes a set of technologies to execute workflow-based applications in a number of different environments, including desktops, clusters and Grids. It has been used in several scientific areas including bioinformatics, astronomy, earthquake science, gravitational wave physics, and ocean science. The Pegasus workflow management system can manage the execution of an application formalized as a visual workflow by mapping it onto available resources and

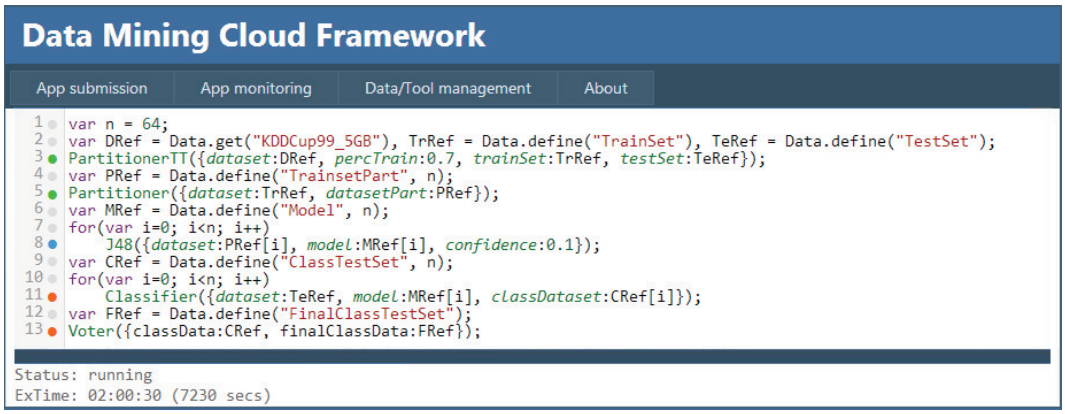


Figure 7: Snapshot of the JS4Cloud workflow running in the DMCF's user interface.

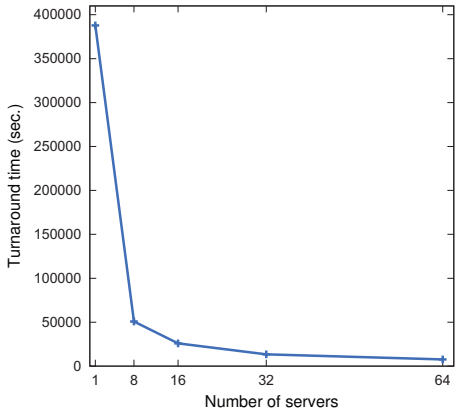


Figure 8: Turnaround time vs number of available servers.

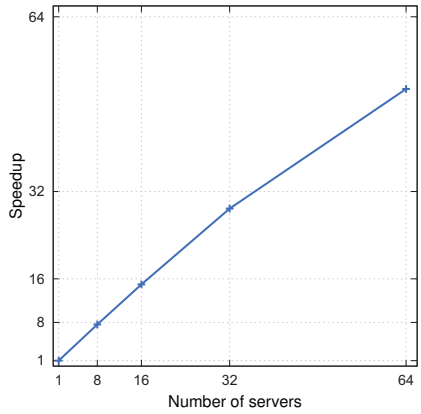


Figure 9: Speedup vs number of available servers.

executing the workflow tasks in the order of their dependencies. Recent research activities carried out on Pegasus investigated the system implementation on Cloud platforms and how to manage computational workflows in the Cloud for developing scalable scientific applications [5].

Taverna [15] is a workflow management system developed at the University of Manchester. Its primary goal is supporting the life sciences community (biology, chemistry, and medicine) to design and execute scientific workflows and support in silico experimentation, where research is performed through computer simulations with models closely reflecting the real world. Even though most Taverna applications lie in the bioinformatics domain, it can be applied to a wide range of fields since it can invoke any web service by simply providing the URL of its WSDL document. This feature is very important in allowing users of Taverna to reuse code (represented as a service) that is available on the internet. Therefore, the system is open to third-part legacy code by providing interoperability with Web services.

ClowdFlows [6] is a Cloud-based platform for the composition, execution, and sharing of interactive data mining workflows. According with the Software-as-a-Service approach, ClowdFlows provides a user interface that allows programming visual workflows in any Web browser. In addition, its service-oriented architecture allows using third party services (e.g., Web services wrapping open-source or custom data mining algorithms). The server side consists of methods for the client side workflow editor to compose and execute workflows, and a relational database of workflows and data.

E-Science Central (e-SC) [4] is a Cloud-based system that allows scientists to store, analyze and share data in the Cloud. Like ClowdFlows, e-Sc provides a user interface that allows programming visual workflows in any Web browser. Its in-browser workflow editor allows users to design a workflow by connecting services, either uploaded by themselves or shared by other users of the system. One of the most common use cases for e-Sc is to provide a data analysis back end to a standalone desktop or Web application. To this end, the e-SC API provides a set of workflow control methods and data structures. In the current implementation, all the workflow services within a single invocation of a workflow execute on the same Cloud node.

Differently from the systems above, which support visual workflow design, we provide both visual and script-based workflow programming, so as to meet the needs of both high-level users and skilled programmers. In addition, the DMCF's runtime differs from that of ClowdFlows and E-Science Central because it is able to parallelize the execution of the tasks of each workflow, an important feature to ensure scalable data analysis workflows execution on the Cloud.

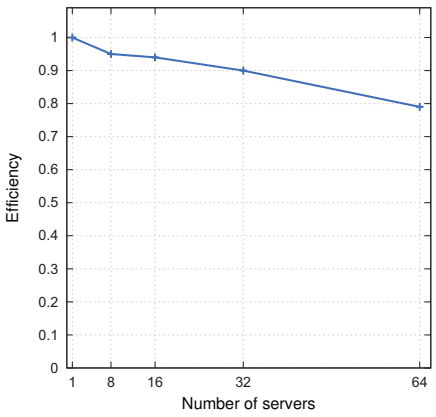


Figure 10: Efficiency vs number of available servers.

COMPSs [8] is a programming model and an execution runtime, whose main objective is to ease the development of workflows for distributed environments, including private and public Clouds. With COMPSs, users create a sequential application and specify which methods of the application code will be executed remotely. This selection is done by providing an annotated interface where these methods are declared with some metadata about them and their parameters. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run. This COMPSs strategy is similar to that exploited in DMCF for parallelizing JS4Cloud workflows. However, while in COMPSs users must provide explicit annotations to specify which methods will be executed remotely, JS4Cloud is a pure implicit parallel language, since no special directives or annotations are needed to enable parallel execution.

We finally mention two script-based workflow languages, Gscript [7] and JOLIE [11], which are related to JS4Cloud, even if they are not explicitly designed for Cloud-based systems.

Gscript is a script-based workflow language, designed to be semantically equivalent to GWENDIA, a visual language to express scientific workflows involving complex data flow patters [10]. A Gscript program is composed of a series of statements, blocks, scalar or array expressions. Each statement defines a processor, its inputs and outputs, and the iteration strategies in a single statement.

JOLIE allows programmers to compose statements in a workflow by making sequences, parallelism and non-deterministic choices. Using its communication primitives and its compositional operators, JOLIE can compose other services by exploiting their input operations. JOLIE provides also statements for user input/output console interaction.

Both Gscript and JOLIE are custom languages with a given syntax to write a workflow and to invoke services from it, while JS4Cloud relies on the widely-known JavaScript language. Furthermore, Gscript and JOLIE require the user to explicitly deal with parallelism (through specific control structures in the case of Gscript; by specifying operators between statements in the case of JOLIE), whereas JS4Cloud relies on sequential JavaScript programming and leaves to the runtime the task of exploiting workflow parallelism.

6. CONCLUSION

The *Data Mining Cloud Framework* (DMCF) is a software system for designing and executing data analysis and knowledge discovery workflows on the Cloud. In this paper we described our solution for programming and executing parallel script-based data analysis workflows in DMCF. We introduced a workflow language, named JS4Cloud, that extends JavaScript to support the development of Cloud-based data analysis tasks and the access to data on the Cloud.

The main benefits of JS4Cloud are: *i*) it is based on a well known scripting language, so that users do not have to learn a new programming language from scratch; *ii*) it implements a data-driven task parallelism that automatically spawns ready-to-run tasks to the available Cloud resources; *iii*) it exploits implicit parallelism so application workflows can be programmed in a totally sequential way, which frees users from duties like work partitioning, synchronization and communication.

Experimental performance results, obtained designing and executing JS4Cloud workflows in DMCF, have proven the effectiveness of the proposed language for programming data analysis workflows, as well as the scalability that can be achieved by executing such workflows on a public Cloud infrastructure. Cloud environments like DMCF and its visual and script-based programming interfaces are important elements for helping researchers and developers in the implementation of big data analysis applications [12].

7. REFERENCES

[1] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, 2008.

[2] E. Cesario, M. Lackovic, D. Talia, and P. Trunfio. Programming knowledge discovery workflows in service-oriented distributed systems. *Concurrency and Computation: Practice and Experience*, 25(10):1482–1504, July 2013.

[3] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[4] H. Hiden, S. Woodman, P. Watson, and J. Cala. Developing cloud applications using the e-Science Central platform. *Philosophical Transactions of the Royal Society A: Mathematical,Physical and Engineering Sciences*, 371(1983), Jan. 2013.

[5] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, SC '10, pages 1–9. IEEE, Nov. 2010.

[6] J. Kranjc, V. Podpečan, and N. Lavrač. ClowdFlows: A Cloud Based Scientific Workflow Platform. In P. Flach, T. Bie, and N. Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases*,

volume 7524 of *Lecture Notes in Computer Science*, pages 816–819. Springer, Heidelberg, Germany, 2012.

[7] K. Maheshwari and J. Montagnat. Scientific workflow development using both visual and script-based representation. In *Proceedings of the 2010 6th World Congress on Services*, SERVICES '10, pages 328–335, Washington, DC, USA, 2010. IEEE Computer Society.

[8] F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia. Enabling cloud interoperability with compss. In *Proc. of the 18th International European Conference on Parallel and Distributed (Europar 2012)*, volume 7484, pages 16–27, Rhodes Island, Greece, 27-31 August 2012. Lecture Notes in Computer Science.

[9] F. Marozzo, D. Talia, and P. Trunfio. Using clouds for scalable knowledge discovery applications. In *Euro-Par Workshops*, pages 220–227, Rhodes Island, Greece, August 2012.

[10] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. B. Fornarino. A data-driven workflow language for grids based on array programming principles. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[11] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. Jolie: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.

[12] D. Talia. Clouds for scalable big data analytics. *IEEE Computer*, 46(5):98–101, 2013.

[13] D. Talia. Workflow systems for science: Concepts and tools. *ISRN Software Engineering*, 2013.

[14] D. Talia and P. Trunfio. *Service-oriented distributed knowledge discovery*. Chapman and Hall/CRC, October 2012.

[15] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, July 2013.

[16] Z.-H. Zhou and M. Li. Semi-supervised learning by disagreement. *Knowl. Inf. Syst.*, 24(3):415–439, 2010.