A Compiler for Agnostic Programming and Deployment of Big Data Analytics on Multiple Platforms

Beniamino Di Martino, Antonio Esposito, Salvatore D'Angelo, Salvatore Augusto Maisto, and Stefania Nacchia

Abstract—To run proper Big Data Analytics, small and medium enterprises (SMEs) need to acquire expertise, hardware and software, which often translates to relevant initial investments for activities not directly connected to the company's business. To reduce such investments, the TOREADOR project proposes a Big Data Analytics framework which supports users in devising their own Big Data solutions by keeping the inherent costs at a minimum, and leveraging pre-existent knowledge and expertise. Among the objectives of the TOREADOR framework is supporting developers in parallelizing and deploying their Big Data algorithms, in order to develop their own analytics solutions. This paper describes the Code-Based approach, adopted within the TOREADOR framework to parallelize users' algorithms and deploy them on distributed platforms, via the annotation of parallelizable code portions with parallelization primitives. The approach, which relies on the guidance of Parallel Patterns to implement the parallelization, and on Skeletons to automatically build execution and deployment templates, is realized through a source-to-source Compiler, also described in the present paper.

Index Terms—TOREADOR, parallel compiler, skeletons, parallel patterns, parallel primitives, big data analytics

1 Introduction

Nowadays we hear more and more about the benefits that a company can derive from the multiplicity, variety and quantity of data that it can or could access. The spread of Internet-based technologies and the growing interest in Smart Sensors, Smart Objects and Internet of Things (IoT) [15] have created new data sources, whose management cannot be dealt with standard IT technologies. Research in new technologies, such as the Cloud [14], has been carried out to manage IoT frameworks and deal with the resulting data deluge. There are, however, issues that make the full exploitation of Big Data difficult, especially when we intend to carry on data analysis to lead business decisions. The encountered complications are essentially: the need to hire experienced, trusted figures to carry on the analysis and the size of the investment necessary to hire such experts; the necessity to acquire hardware units suitable to the analysis purposes, in terms of processing power and memory capability; the selection and acquisition of the best, suited software to perform such analysis. While a big company can seamlessly sustain the initial investments, medium or small enterprises (SMEs) can find it a difficult hurdle to overcome. The European project *Toreador* [1] has, among its main objectives, that of overcoming the hurdles that SMEs can find in approaching Big Data, thus helping them to take full advantage of the potential of big data analysis. In particular, the project aims at providing suitable models and tools for automation and commoditization of Big Data Analytics development. In order to support SMEs in developing their own Big Data applications seamlessly, without the need to focus on the deployment and execution details, the project has defined and implemented two interleaved but distinct approaches:

A *Service-Based Approach* in which the Toreador framework extracts the requirements expressed by the user from a Declarative model (which has been defined in [4]) and then, guided by the user herself, selects, composes and orchestrates a set of available Big Data oriented services.

A *Code-Based Approach, and Compiler* that takes as input a sequential code, annotated with parallelization primitives (which have been defined within the Toreador project), and provides a parallel and distributed version of it. The execution of such distributed version is done according to specific Parallel Computational Pattern.

This paper focuses on the presentation of the Code-Based approach and Compiler, by utilizing a reference scenario taken from real use cases of the project. The approach's main objective is to support the Toreador user in developing and coding her own algorithms and Big Data analytics, and supporting her in deploying a parallelized and optimized version of her program onto a selection of platforms.

- *B. Di Martino, A. Esposito, and S. D'Angelo are with the CINI - Consorzio Interuniversitario Nazionale per l'Informatica Roma, Roma, Lazio 00198, Italy, and also with the Department of Engineering, Università degli Studi della Campania "Luigi Vanvitelli", Caserta, CE 81100, Italy. E-mail: beniamino.dimartino@unina.it, {antonio.esposito, salvatore.dangelo}@unicampania.it.*
- *S.A. Maisto and S. Nacchia are with the Department of Engineering, Università degli Studi della Campania "Luigi Vanvitelli", Caserta, CE 81100, Italy. E-mail: {salvatoreaugusto.maisto, stefania.nacchia}@unicampania.it.*

Manuscript received 13 Sept. 2018; revised 31 Jan. 2019; accepted 8 Feb. 2019. Date of publication 12 Mar. 2019; date of current version 7 Aug. 2019. (Corresponding author: Beniamino Di Martino.) Recommended for acceptance by C. Krintz. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2019.2901488

In particular, the paper describes the Parallelization Primitives and Patterns utilized to annotate and parallelize the source code of a Big Data analytics algorithm (Section 4), which is analyzed and parsed through the Compiler (Section 5), in charge of applying a set of transformation rules to fill execution and deployment Skeletons. The structure of the Compiler and the transformation steps are described, and a use case, which covers the whole parallelization process from source code annotation to deployment, is presented in Section 7, while experimental results are reported in Section 8. Finally, Section 9 addresses future developments.

2 Background

Research on patterns for parallel and concurrent programming has produced very interesting and immediately applicable results. Remarkable are the patterns presented in [18], which proposes a set of solutions for parallel programming, classified and organized according to the specific feature on which the parallelism is focused. The *Organize by Tasks* Patterns category includes patterns which organize computation on the base of the tasks to be executed; Patterns included in the *Organized by Data* category are structured on the basis of characteristic of the data structure to be analysed; the *Organize by Flow* category contains patterns which organize data according to workflows and events

If we want to consider a more practical level, it is possible to take in consideration another set of patterns, some of which are described in [20] with their implementations in C and Visual Basic. Among these patterns: The *Bag of Tasks Pattern* is used when the actual load balance is not predictable and the task executors have different capabilities; the *Loop Parallelism Pattern* focuses on source code where many loops are present.; the *Fork and Join Pattern* is used when the number of tasks to be executed cannot be predicted. The *Map Reduce* framework, originally introduced by Google [10], represents a good solution to the Data Deluge (or Data Flood) problem which affects many modern companies, since it promotes data locality and reduces communication overheads. Map Reduce organizes computation tasks according to data distribution among computing nodes so that storage, not computational resources, lead the analysis. Data are processed in parallel, executing *Map* procedures, performing filtering, sorting and distribution tasks, while results are summarized by *Reduce* procedures. Since data locality limits the parallelization, in order to reduce communications among nodes, the framework gives best results when working with huge data sets.

Algorithmic Skeletons, also referred to as Parallel Patterns, are a high-level parallel programming model which can be used to support applications design and implementation for parallel and distributed computing. There exist several definitions of Skeleton, focusing on a specific aspect of such an instrument. A very good summarization of such definitions can be found in [9]: *"An algorithmic Skeleton is parametric, reusable and portable programming abstraction, modeling a known, common and efficient parallelism exploitation Pattern"*.

Such a definition covers all the aspects for which Skeletons represent a fundamental component of the Code-based approach. Skeletons allow the declaration of high order functions as a procedural 'template', which specifies the overall structure of a computation, with gaps left for the definition of problem specific procedures and declarations [8]. A major advantage deriving from the use of skeletons is that orchestration and synchronization of the parallel activities are implicitly defined and hidden to the programmer. This implies that communication models are known in advance and cannot be modified by programmers who, in turn, are less prone to introduce errors and bugs since they are 'guided' in writing their code. Different frameworks and libraries have been defined to assist programmers using skeletons: a survey of these tools is provided in [16]. Skeletons-based approaches have been described and successfully exploited in previous works. In [7] several skeletons have been defined, with pseudo-code descriptions and implementation suggestions. In [2] the author describes a skeleton-based framework called *Muskel*, which exposes a limited number of primitives directly implementing Skeletons.

2.1 Parallelization Approaches

Big data analytics requires high programmer productivity and high performance simultaneously on large-scale clusters. Many approaches have been proposed to optimize the application execution and to ensure a higher productivity. In [4] the authors propose a new development life cycle for Big Data, combining exploration and refinement, fast deployment and controlled execution. While in [13] the authors present the benefits of applying the "code once deploy everywhere" approach to clustering of categorical data over large data sets. In [19] the authors propose a novel auto-parallelizing compiler approach where they develop a data flow algorithm that exploits domain knowledge as well as high-level semantics of mathematical operations to find the best distributions, but without using approximations such as cost models. This approach has been used to build the High Performance Analytics Toolkit (HPAT). HPAT is a compiler-based framework for big data analytics on large-scale clusters that automatically parallelizes high-level analytics programs and generates scalable and efficient MPI/C++ code. Another approach and related framework presented in [3], which addresses more specifically the streaming analysis, is FastFlow, a framework delivering both programmability and efficiency in the area of stream parallelism. FastFlow may be viewed as a stack of layers: the lower layers provide efficiency via lock-free/fence-free producer-consumer implementations; the upper layers deliver programmability by providing the application programmer with high-level programming constructs in the shape of skeletons/parallel patterns. In [17] the authors propose a parallel language and application development and execution system for creating parallel applications that run on high-performance computers and clusters; CxC lets you use as many parallel processors as you want to solve problems by defining your own network topology on these processors and a complete multiple-program/multiple-data (MPMD) model. The communication operations are one-sided, which makes parallel programming easy and efficient. Another important data parallel approach is NESL [5], which is intended to be used as a portable interface for programming a variety of parallel and vector computers, and as a basis for teaching parallel algorithms. Parallelism is supplied through a simple set of data-parallel constructs based on sequences, including a mechanism for applying any function over the elements of a sequence in parallel and a rich set of parallel functions that manipulate sequences. UPC [6] is, on the other hand, a parallel extension of the C programming language intended for multiprocessors with a common global address space. UPC has two primary objectives: 1) to provide efficient access to the underlying machine, and 2) to

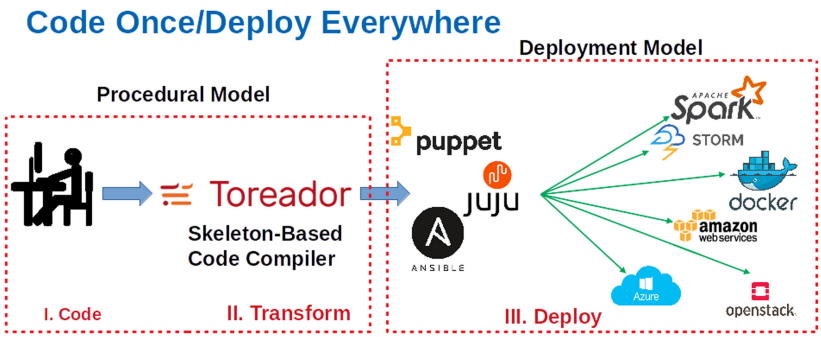

Fig. 1. The code-based approach workflow.

establish a common syntax and semantic for explicitly parallel programming in C. The quest for high performance means in particular that UPC tries to minimize the overhead involved in communication among cooperating threads.

3 The Code-Based Approach

Through the Code-Based Approach, users can express the parallel computation of a coded algorithm in terms of parallel primitives, and distribute it among computational nodes hosted by different platforms/technologies in multi-platform Big Data and Cloud environments, using State of the Art orchestrators. The approach requires that the user annotates her source code with a set of Parallel Primitives, which are then examined by the compiler. Such primitives are modeled following a set of Parallel Patterns, and guide a Skeleton Compiler in the selection of the best suiting Skeleton to fill to obtain the desired parallelization.

Fig. 1 provides an overview of the complete workflow with its three main phases:

Code. In this phase the user, which is supposed to be an expert programmer and to own a good knowledge of the sequential algorithm to be parallelized, annotates the original code with the provided Parallel Primitives.

Transform. During this phase a Skeleton-Based Code Compiler (Source to source Transformer) transforms the sequential code, annotated with the primitives, into a series of parallel versions specific to different platforms/technologies, according to a 3-steps sub-workflow. - *Parallel Pattern Selection*: selection of the Parallel Paradigm (based on the used primitives); - *Incarnation of agnostic Skeletons*: transformation of the algorithm coded using the aforementioned primitives, in a parallel agnostic version incarnating predefined code Skeletons. The whole transformation happens thanks to a set of transformation rules which operate on the source code AST, as further explained in Section 6; - *Production of technology dependent Skeletons*: specialization of the agnostic Skeletons in multiple versions of the code to be executed, but specific to the different target platforms/technologies.

Deployment. Production of Deployment Scripts.

The user is supposed to be an expert programmer: she is well aware of the potentialities (maximum flexibility and customizability, full controllability) and limitations (the application is developed from scratch) of a fully coded-based approach and she owns a good knowledge of the programming paradigms and languages involved. The Code Based Approach guides the user by proposing Parallel Computational Patterns and Skeleton-based solutions fitted to her requirements. Through a set of parallelization primitives, the user can distribute the computation among computational nodes hosted by different platforms or in multi-platform environments.

4 The Primitives

Within the Toreador approach, Parallel Primitives are constructs with a specific meaning, which are used to augment a standard and widespread language, i.e., Python, to guide the compiler in the parallelization. Ten different primitives have been defined.

Skeletons' filling is obtained by manipulating the Abstract Syntax Tree of the source code, following a set of transformation rules which are determined by the adopted Parallel Primitives and corresponding Patterns. Deployment Skeletons are also produced, according to the target platform selected by the user.

The primitives used to describe the algorithms will reflect the parallelization strategy which will be used during the Skeleton filling phase. A set of Parallel Primitives has been defined to describe the different parallel decomposition and execution strategies which can be used to implement an input algorithm.

In this section the directives' signatures are expressed in Python, which has been chosen as the target programming language to express the algorithms. A total of ten directives have been defined so far: the first five are used to describe a *Data Parallel* approach to problem decomposition and execution; the other five address a *Task Parallel* approach.

Among the Data Parallel directives, three are used for generic, i.e they do not impose a specific execution pattern explicitly.

4.1 Generic Data Parallel Directives

In this paragraph the Generic Data Parallel Directives' signature are listed and described in details.

*data_parallel_region(data, func, *params)*

The *params* input represents optional parameters which can be used both as an additional input to the *func* function, beside the *data* element, or as a flag to modify the function behavior.

Considering the definition of this directives, the computation expressed by the *func* function must be *data independent* that is, it can be performed on a single data element without the need to know the other elements or the result of computations executed over them.

So, the general workflow can be described as follows: the elements of the data input are distributed among the execution nodes, together with the optional parameters; each execution nodes executes the function func over the data chunk

which it has received and passes to result to the caller; the single results are collected and proposed as output as a list.

The data distribution logic is not embedded into this Parallel Primitive, but it depends on the Parallel Computational Pattern that the user can choose later or that is inferred from the Declarative Model.

*data_parallel_region_with_reduction(data, func, reduce_func, *params)*

This directive describes a situation which is very similar to the very generic data_parallel_region, as it requires the data elements composing the *data* input to be independent, and it considers such input to have been previously prepared. The difference is represented by the *reduce_func*, which is an operation that reduces the different data produced by the *func* function to a single output.

*data_parallel_region_dependent(data, stencil, func, *params)*

This directive describes a situation in which the input data, expressed as before by the *data*, has been previously organized in a list of elements (just as in the previous data_parallel_region directives). However, the data are not simply distributed element by element, as they can be grouped using windows with fixed dimension, which can also overlap. This is possible via the *stencil* parameter, which is composed by a three-element list: - *Window_Size* sets the dimensions of the window.

- *Backward* makes it possible to extend the window of data sent for elaboration to the *n* values before the current examined element of the data list.
- *Circular* is a boolean value, which states if the list has to be considered circular. The other assumptions made previously for data_parallel_region regarding the independence among the different executions of the function *func* still hold.

The distribution logic is once again not defined within the Primitive, but it is is asserted later by a specific Parallel Pattern

4.2 Specific Data Parallel Directives

In this paragraph we describe Parallel Directives which requires that functions for the correct distribution of data and their elaboration are explicitly provided. Also, such directives are automatically mapped to a specific Pattern and do not require further information from the declarative Model (save from information needed to build the deployment templates).

*map_reduce(data, map_func, reduce_func, *params)*

The distribution logic is implicit in the Primitive, as it strictly implies the application of a Map Reduce Parallel Pattern: in particular, a Distributed File System is expected to take care of data distribution and replication

*producer_consumer(data, prod_func, cons_func, *params)*

This directive implies a Producer Consumer parallelization approach.

4.3 Generic Task Parallel Directives

Only one generic task parallel directive has been defined so far.

*task_indipendent_region(data, func_list, *params)*

In this directive, the *data* input is represented by either a single value or a list of values. In the first case, the functions defined in the *func_list* are executed in parallel on the same data chunk. In the second, each function is executed separately on the corresponding data chunk, chosen according to the function index. A data preparation phase is expected to be performed before the use of the directive.

A mandatory condition is that the tasks to be executed, represented by the functions in the list, are independent.

4.4 Specific Task Parallel Directives

A series of Pattern specific directives have been defined to represent the peculiar parallelization characteristics of the Patterns which will be used to fill the Skeletons. Such directives are considered as Task Parallel as they distribute the functions to be executed among the computing nodes. The task distribution logic is not embedded in the Primitive, as it strictly depends on the Parallel Pattern which is selected by the programmer or inferred through the Declarative Model.

*divide_and_conquer(data, split_func, func, combine_func, *params)*

This directive implements a Divide and Conquer parallelization approach. The input *data* are not necessarily preelaborated. All the details regarding the communication among the nodes and the distribution of data are implicit in the functions' definitions and are not limited by the implementing Pattern.

*bag_of_task(data, func_list, *params)*

The Bag of Task directive defines a Task parallel approach in which the input functions contained in the *func_list* are taken one by one and put into a shared suitable data structure. One mandatory condition is that the functions to be executed are completely independent from one another and do not need to interact to obtain the correct result: indeed, the order of their execution cannot be ensured, since it depends on the specific data structure.

*pipeline(data, [order_func_list] *params)*

This directive describes a task parallel approach in which the functions to be executed, contained in the *order_func_list* input, need to be run in the precise order they appear in the list.

*tree_computation(data, split_func_list, func, combine_func_list, *params)*

Tree Computation is a particular Divide and Conquer recursive approach, in which the distribution of data and the elaboration to be carried on are actually separated. In this way, if a complex logic must be enforced to correctly distribute the input data, it can be distinguished from the elaboration phase. The distribution of tasks is implicit in the spit and combination functions and do not depend on the implementing Pattern.

5 The Compiler Architecture and Components

The filling process of the Skeleton will take place automatically using a Compiler, whose overall structure is shown in Fig. 2. The current Compiler is an advanced version of the prototype already presented in [11].

The compiler receives, as a first input, information coming from TOREADOR declarative models, [4], which are used to identify the best patterns to apply (when one has not been explicitly selected) and related skeletons to be filled. A second input is represented by the actual algorithm, annotated with suitable directives to identify data parallel regions which can be distributed among multiple executors, by using the selected Skeletons. The output is

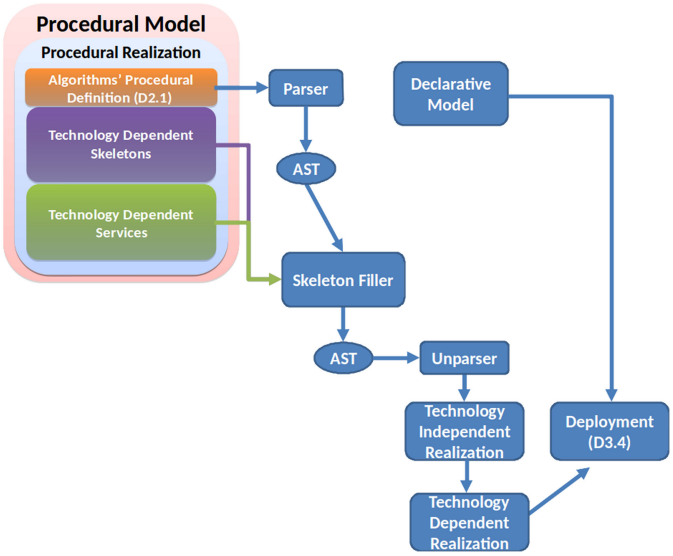

Fig. 2. Architecture of the compiler.

composed of both executable skeletons, which represent the distributed version of the input algorithm, according to a specific pattern, and deployment templates, which instead define the Services and related components to be deployed and their characteristics.

The architecture of the compiler is based on several components, namely:

Parser. That analyzes the input represented by the Algorithm's Procedural Model, and produces its AST representation.

Skeleton Filler. That is in charge of modifying the original AST in order to create a language independent representation of the final Skeleton to be produced. The transformations is not hard-coded into the compiler, but they are the result of a series of rules which are included in the external configuration files. Such files can be edited and updated anytime, without modifying the compiler itself.

Un-Parser. That transforms the final AST back into the code Skeletons, by using the desired programming language.

While this procedure produces executable Skeletons, there is still the need to configure the target environment, with information regarding the hosting nodes, their topology, supported data structures, but also physical characteristics tied to the required performances. Such aspects are considered by Deployment Models. In the following sections each component is thoroughly explained, trying to highlight the most important features

5.1 The Parser and Skeleton Filler

The *Parser* and *Skeleton Filler* represent the executive core of the compiler. Fig. 3 provides an overview of the parsing mechanism. The *Parser*, as previously mentioned, takes the algorithm written in python as input and generates its Abstract Syntax Tree representation. The tree nodes represent the key aspects of the chosen language for the definition of the algorithms and are used to define the rules that are at the base of the compiler. The algorithm is annotated with micro-functions which are recognized by the *Parser* and trigger a separate set of rules, according not only to the specific function encountered but also to the target Pattern (for the filling of the Execution Skeletons) and Services/Platform (for the construction of the Deployment Templates).

The *Skeleton Filler*, on the other hand, deals with the automatic transformation of the syntactic tree, according to the triggered rules. The filler transforms the original syntax tree, by moving/rearranging or creating nodes, which are then appended to a new tree. At the end, the produced AST is ready to be un-parsed to obtain the desired Skeletons. The Skeletons produced by the filler are grouped into three categories: - *Main Scripts* represent the entry point of the execution process managed by the Skeleton. They contain the code which cannot be distributed and the calls to the Secondary Scripts; - *Secondary Scripts* contain the code which will be executed in parallel on different computational nodes. The number of produced scripts depends on the selected Pattern; - *Deployment Templates* contain information on the characteristics of the computational nodes the filled Skeletons will be executed on.

The Skeleton filling rules represent the knowledge base on which the compiler works to enact the transformations

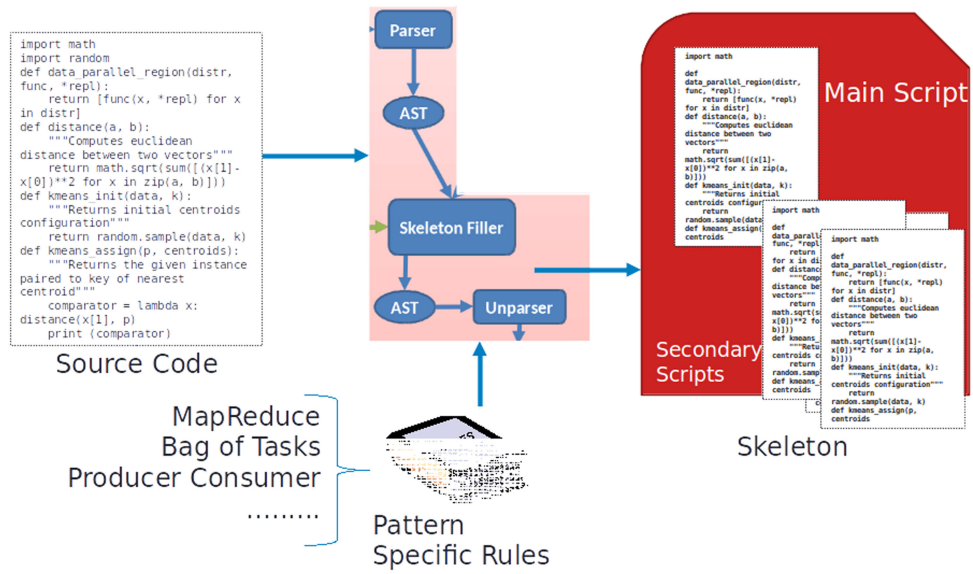

Fig. 3. The rule based parser.

of the original Algorithms' code. The rules are Pattern specific, as different patterns will require different transformations and Secondary Scripts. However, they are completely independent from the algorithm, as the Parser will treat the same micro-function in the same way whichever is the algorithm in which they are used.

5.2 The Un-Parser: From Technology Independent to Technology Dependent Realizations

The Skeletons produced by the *Parser* and *Skeleton Filler* are independent from the target platform on which the algorithm will be eventually executed. Specifically, the filled Skeletons are still parametric, with information like the number of required executing nodes, or the dimension of storage volumes, left to the user to specify. Furthermore, service calls to platform specific services or procedures are hidden behind *agnostic invocations*, that do not explicitly refer to an execution environment.

The *Un-Parser* is in charge of modifying the Technology Independent Skeletons into Technology Aware ones, by replacing parameters and agnostic calls with the ones corresponding to the users' requirements and the chosen target platform. The information needed to correctly operate the substitution are inferred from the *Declarative Model*, whose definition and use can be found in [4].

Section 7 will report examples of the transformations operated by the Compiler.

5.3 The Deployment Phase

The Deployment phase represents the last step in order to make the desired algorithm able to be executed on the target platform. Once all the necessary information have been gathered (i.e., the Target Platform, data location, possible security features) either from the Declarative Model or by asking the user directly, the deployment and execution commands are set up and launched. The Deployment is transparent to the user, so that her intervention is not required anymore, unless she wants to monitor and act on the execution of the algorithm.

Section 7 will report examples of the code produced during the deployment phase.

Target deployment environments include a range of distributed and parallel frameworks, such as Apache Storm, Spark and Map-Reduce; Cloud platforms such as Amazon and Azure; container-based systems, in particular Docker. Regarding the Docker implementation, two different strategies are currently being considered:

A centralized approach, in which the Docker containers are managed locally, within a self-contained platform which instantiates and manages them

A distributed approach, in which containers can run on remote devices, which are selected by a central offloading algorithm according to the execution schema. This second approach can be considered an instance of *Edge Computing* paradigm, where edge computing nodes are smartphones or other smart devices near to the data sources.

If the platform allows it, it is possible to exploit an automatic orchestrator for the deployment, as it has been described in [12], where the *ANSIBLE* orchestrator has been used to automatically deploy and manage the execution of a parallelized algorithm on a target platform provided by ATOS, a project's Partner.

6 The Transformation Rules

6.1 Data Parallelism: Generic Rules

In order to fill the Skeleton scripts, the compiler follows a series of simple steps described by a rule set. Here we report the core set for the filling of the main script, which is then specialized according to the chosen pattern. Such rules guide the Compiler through its visit on the source code's AST. After writing the code using the parallel primitives, the algorithm can be passed to the compiler that will take care of the translation based on the parallel paradigm and the selected target platform. The steps that are performed by the compiler are:

- Create the syntactic tree of the algorithm code: in this phase the AST python module is used for the creation of the syntactic tree starting from the python code of the algorithm
- Extract the imports: the tree is analyzed to identify the leaf nodes belonging to the import class
- Extract the classes: the tree is analyzed to identify the leaf nodes belonging to the Class Def class
- Extract the function definition: the tree is analyzed to identify the leaf nodes belonging to the Function Def class
- Analysis of the __main__ module: all the nodes are analyzed, when an assignment is found, it is checked whether the object of the assignment is an input data for the parallel primitive, in which case we proceed by adding the nodes necessary for data distribution or distributed reading
- Replacement of parallel primitives with instructions from the chosen target platform
- Recursive class analysis: the presence of parallel primitives is checked, in which case they are replaced with the nodes necessary for execution on the target platform
- Recursive analysis of the functions: the presence of parallel primitives is checked, in which case they are replaced with the nodes necessary for execution on the target platform
- The new __main__ file is produced

As for the optional parameters, if these are initialized in the original algorithm, they are encoded in a JSON file and the nodes which need to read and write them are added to the list of nodes constituting the *Body* of the main script. The steps for the completion of the main script and side scripts are repeated every time the Compiler finds a *data_parallel_region*.

6.2 The Map-Reduce Rules

The rules defined are many and different according to the pattern. To give a concrete example we examine the specific case of Map Reduce. The pattern requires to distribute the input data on a distributed file system (DFS), and all the operations on the data must include a DFS reading, thus aside from the main file, the compiler automatically produces the mapper file and the reducer file where all the instructions for the distribution and reading of data on the DFS are encapsulated.

A schematic version of the rules is shown in the Fig. 4.

Obviously in the image there are only the most important steps, which include several sub-steps within them. The overall procedure starts with the identification of the main components of the target algorithm, i.e., the Parallel region. If the compiler reaches a *data_parallel_region* during

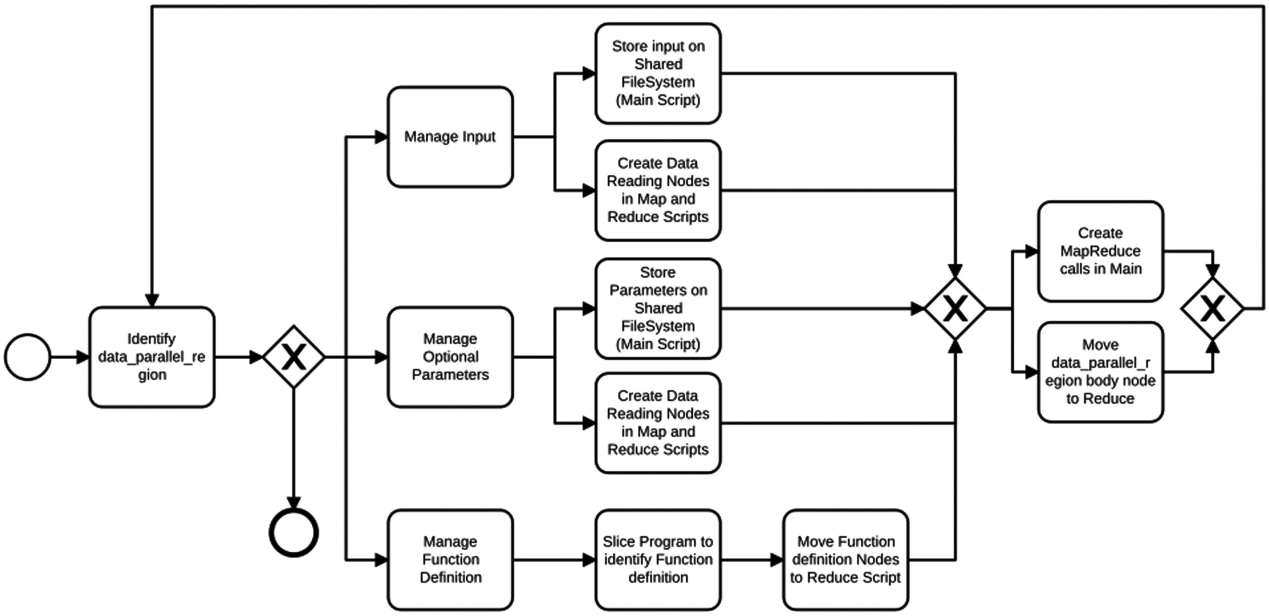

Fig. 4. The Map Reduce rules.

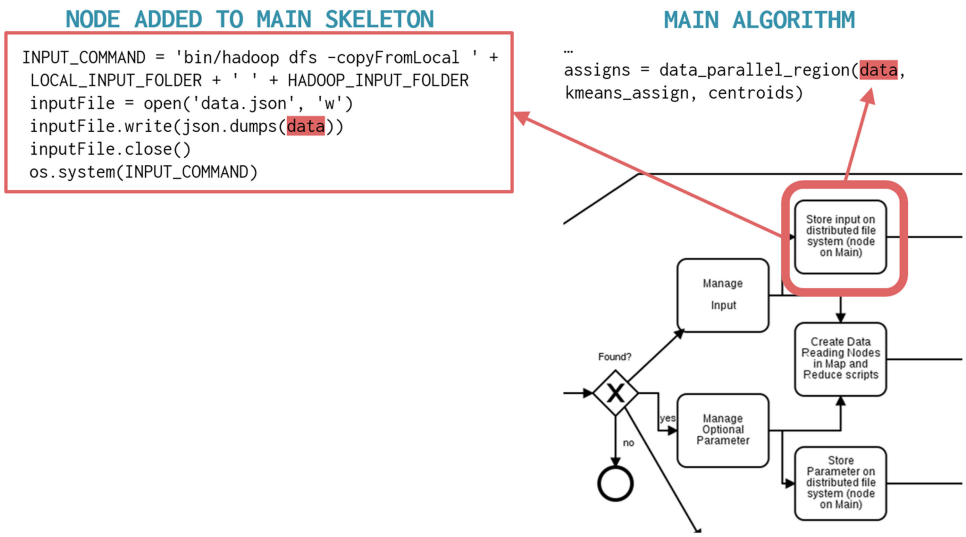

Fig. 5. The distributing nodes for the inputs.

the analysis of the syntactic tree, the following steps are performed:

- Manage Inputs
- Manage Optional Parameters
- Manage Function Definition

The *Manage Inputs* rule is entitled to handle the input data according to the pattern, in this case it needs to store them in the DFS, this specific operation is then inserted in the main script, as it can be seen more specifically in Fig. 5.

The other parallel action performed is the creation of *Data Reading Node* operations that are then inserted in the Mapper and Reducer scripts, as shown in Fig. 6.

The *Manage Optional Parameters* performs the same steps as the previous rule, but it handles optional data and parameters aside from the main data that the algorithm has to elaborate, as shown in Fig. 7.

Last but not least, as explained in the previous sections, the *data_parallel_region* has as input the function, i.e., the basic operation defined by the algorithm, that must be performed on the data in a parallel manner. This function is handled in the *Manage Function Definition*, this step is entitled to find the function definition through the program slicing and move the function in the Reducer scripts. Finally the compiler creates the Map Reduce Call in the main script, and the body of the *data_parallel_region* is moved to the Reducer script.

7 Application to the Toreador Reference Scenarios

7.1 Reference Scenarios

In this paper we will refer to a scenario that is the simplified version of a more complex case study, which has been proposed within the Toreador project by the JOT company, partner of the research project. In the general scenario the company wants to analyze the clicks made by users on online advice(ads), published on specific web-sites, in order to

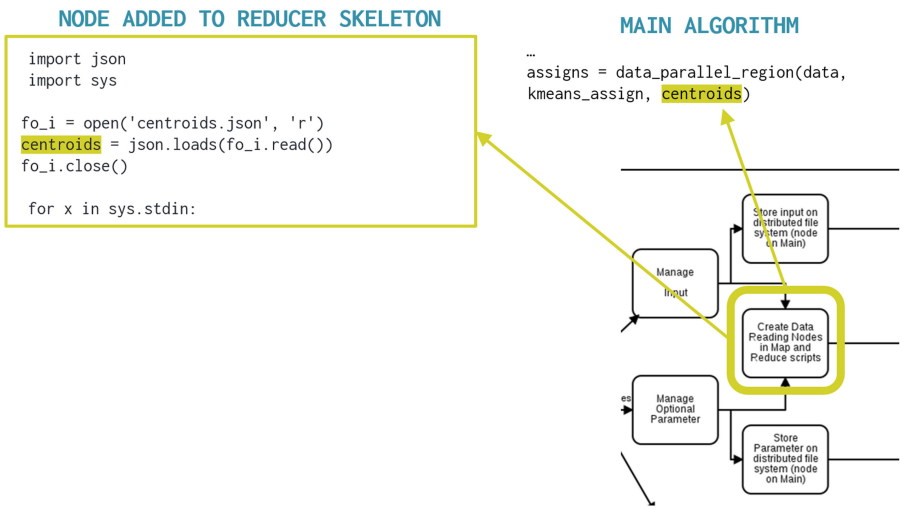

Fig. 6. The reading node in the map and reduce file.

determine the best position for such ads, and identify fake users who may exploit automatic bots to increase the clicks and, subsequently, the related revenue. In our specific case, we focus on the analysis of the IPs retrieved from the click-stream, and the categorization of such IPs according to a set of criteria: geographic position and identification of the source area, population size of the identified areas, available services, connections, and so on. This can be identified as a classification problem, which can be addressed through a series of approaches. In our case, JOT applied K-Means clustering to classify the IPs. However, due to the huge amount of input data, such a classification is extremely slow. Toreador offered two different possible solutions: to exploit one or more of the analytic Services made available by the platform (Service-based approach); or to modify the K-Means algorithm which had already been developed, through parallelization primitives, and then actually parallelize and distribute it. This second approach corresponds to the Code-based approach we are focusing on in this paper. Section 7 reports the transformation operated on the K-Means code as a result of the application of the Code-based approach, and shows the performance boost obtained through the parallelization.

This is not the only scenario on which the compiler and the approach has been tested. The SAP company (also partner of Toreador) has proposed her own clustering algorithm, namely the Rock algorithm, which has been annotated with parallel primitives to boost its performances through the compiler [13]. Lightsource (another company partner of the project Toreador) has instead proposed a scenario in which data regarding the use of batteries and solar powered cells is gathered by distributed devices located at customers' households, and different calculations are then operated (i.e., to determine the Mean Time Between Failures). In this case, the code-based approach has been used to analyze the data coming from the different devices in parallel, after a conspicuous batch had been collected. Currently we are running experiments to determine the feasibility to run part of the computation locally on tiny computing devices located at the households, thus following a Cloud-Edge Computing architectural paradigm.

7.2 Code Implementation

As already stated in Section 7.1, in our reference scenario a K-means Clustering algorithm has been employed to categorize users of online web-sites, clicking on ads put on the site pages, and collect information regarding their provenience, general background, and services they could access

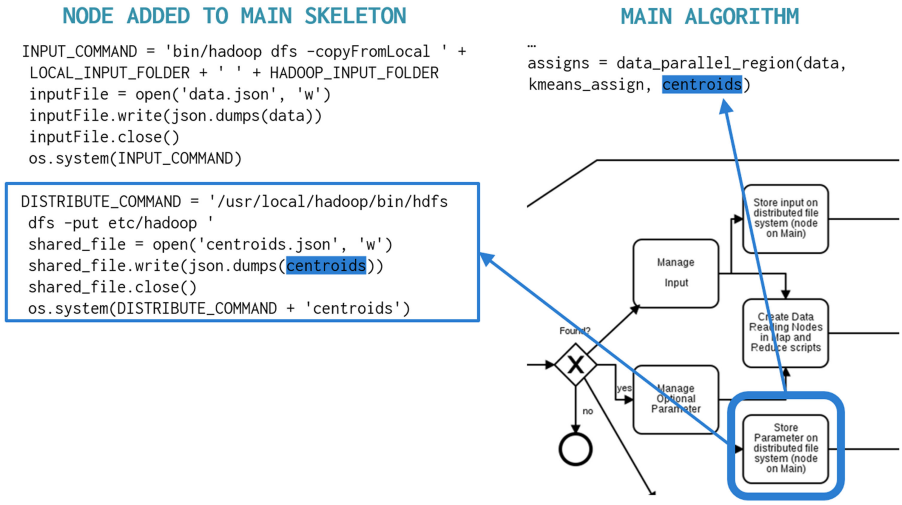

Fig. 7. The distributing node for the optional parameters.

to in their geographical area. Parallel Primitives have been applied to the code In this section we are going to present two case studies that employ the *K-means Clustering*, which is considered as a very effective, yet simple, algorithm to be used in a Big Data scenario.

The target platforms for which we are going to produce the Vendor and Parallel Pattern specific versions are:

1) a Hadoop cluster on which the Map Reduce paradigm is implemented;
2) an Apache Spark cluster to implement the Producer-Consumer parallelization paradigm.

Source Code 1. K-Means Without Parallel Primitives

```
import math,random
def distance(a,b):
    return......
def kmeans_init(data,k):
    return ...
def kmeans_assign(p,centroids):
    comparator=lambda x: distance(x[1],p)
    nearest=min(enumerate(centroids),key...
    return (p,nearest[0])
def kmeans_centroid(cluster):
    csum = map(sum,zip(*cluster))
    return [x/len(cluster) for x in csum]
if __name__=="__main__":
    data=...
    k=2
    centroids=kmeans_init(data, k)
    iteration=0
    while iteration==0 or previousCentro...
        previousCentroids=centroids
        assigns = [kmeans_assign(x, centroids) for
        x in data]
        clusters=[[] for x in centroids]
        for x in assigns:
            clusters[x[1]].append(x[0])
            centroids = [kmeans_centroid(x) for x in
            clusters]
            iteration+=1
```

In particular, the Hadoop platform used for the first Use Case has been set-up using a cluster created on a private workstation, but it could be easily migrated to any Hadoop cluster. On the other hand, the Apache Spark cluster has been created in a completely automatic fashion, by using an Ansible-enabled platform. Such a platform can be reached through a Rest API, which allows to request and deploy a cluster and execute the algorithm compiled for that specific platform.

The reported code has some specific lines highlighted with a bold /italic style, these lines represent the code that has been added and/or substituted. The code reported in box 1 reports the main parts of the original code (cuts have been made where the full code was not necessary). The reported code represents a standard K-means implementation, with a *distance* function to calculate the distance among the clusters'centroids, a set of functions to actually instantiate the clusters and initialize the centroids. The distance function reported acts on two parameters, which represent characteristics vectors.

Source Code 2. K-Means with Parallel Primitives

```
import math,random, toreador as t
def distance(a,b):
    return ...
def kmeans_init(data,k):
    return ...
def kmeans_assign(p, centroids):
    comparator=lambda x: distance(x[1],p)
    nearest=min(enumerate(centroids),key...
    return (p,nearest[0])
def kmeans_centroid(cluster):
    csum=map(sum,zip(*cluster))
    return [x/len(cluster) for x in csum]
if __name__=="__main__":
    data=...
    k = 2
    centroids=kmeans_init(data, k)
    iteration=0
    while iteration==0 or previousCentro...
        previousCentroids=centroids
        assigns = t.data_parallel_region(data,
        kmeans_assign, centroids)
        clusters=[[] for x in centroids]
        for x in assigns:
            clusters[x[1]].append(x[0])
        centroids = t.data_parallel_region(clus-
        ters, kmeans_centroid)
        iteration+=1
```

We have therefore re-written the algorithm with parallel primitives which, as we shall see, does not involve complex procedures, since it does not require particular constructs or languages. During the parallelization process of the K-means algorithm, the *data_parallel_region* primitive has been exploited in two different phases. Source Code 2 provides an excerpt of the algorithm code using such parallel primitives in two different locations: during the assignation of data points to centroids and during the re-calculation of the centroids themselves. This two operation are data parallel tasks, and they can be performed by different nodes at the same time on different data.

Once the code has been annotated with the parallel primitives, it can be fed as input to the compiler. In the first phase, the code is analyzed and parallel primitives are identified.

7.3 Production of the Skeletons

In this specific example, we have selected Map-Reduce as the reference Parallel Patterns and an Hadoop environment, deployed on a local cluster, as a target platform.

In order to implement the algorithm according to the Map Reduce Pattern, in our methodology, three different files are produced starting from empty Skeletons:

1) *Main file*: this is the main script where the sequential (not parallelizable) operations of the algorithm are executed and Map Reduce jobs are launched;
2) *Mapper file*: it deals with distributing, filtering, and sorting input data;
3) *Reduce file*: it deals with operations that "reduce" the object of evaluation, by gathering the partial results of the executions operated on the distributed nodes. A very common operation consists in simply summing the partial results.

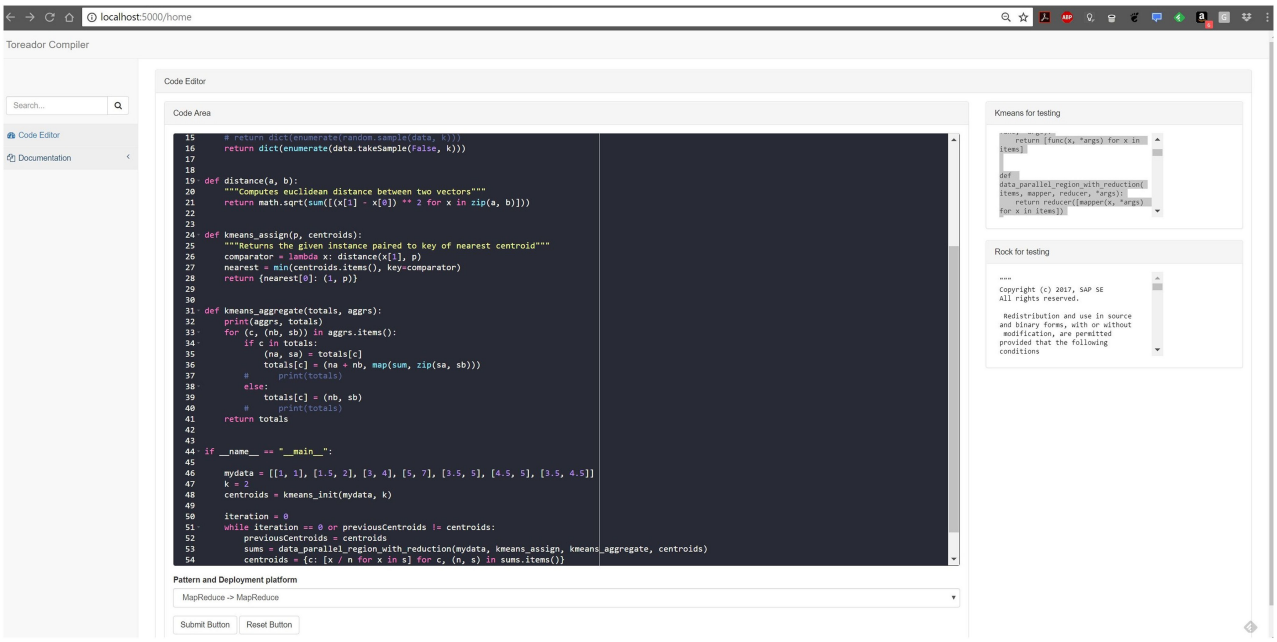

Fig. 8. The web interface for the compiler.

Source Code 3. Main.py - The Technology Dependent Realization

```
import math,random ,os
LOCAL_INPUT_FOLDER='/input'
HADOOP_INPUT_FOLDER='/tmp/hadooptest/'
DISTRIBUTE_COMMAND='/usr/local/hadoop/hd...
INPUT_COMMAND='bin/hadoop dfs -copyFromL...
OUTPUT_COMMAND='/usr/local/hadoop/bin/hd...
CAT_OUTPUT='cat output'
def file_write(name,data):
    ...
def kmeans_init(data,k):
    return ...
if __name__=='__main__':
    data=...
    k=2
    centroids=kmeans_init(data, k)
    iteration=0
    while iteration==0 or previousCentro...
        previousCentroids=centroids
        file_write('data.json',data)
        os.system(INPUT_COMMAND)
        file_write('centroids.json',cent...
        os.system(DISTRIBUTE_COMMAND+'ce...
        LAUNCH_COMMAND_0=('/usr/local/ha...
        os.system(LAUNCH_COMMAND_0)
        assigns=os.popen(CAT_OUTPUT).rea...
        clusters = [[] for x in centroids]
        for x in assigns:
            clusters[x[1]].append(x[0])
        file_write('clusters.json',clust...
        os.system(INPUT_COMMAND)
        LAUNCH_COMMAND_1=('/usr/local/ha...
        os.system(LAUNCH_COMMAND_1)
        centroids=os.popen(CAT_OUTPUT).r...
    iteration+=1
```

In this specific case, two sets of Mapper and Reducer have been produced, one for each *data_parallel_region* found in the parallelized code, while only one Main file is necessary.

Source Code 4. Mapper.py - The Technology Dependent Realization

```
import sys,json
def read_input(file):
    for line in file:
        yield json.loads(line.strip('n'))
data = read_input(sys.stdin)
for element in data:
    print(element)
```

Also in this section the reported code has some specific lines highlighted with a bold /italic style, these lines represent the code that has been retained from the sequential code and inserted in the final skeletons.

Source Code 5. Reducer.py - The Technology Dependent Realization

```
import json, math ,sys
def distance(a,b):
    return...
def kmeans_assign(p, centroids):
    comparator=lambda x: distance(x[1],p)
    nearest=min(enumerate(centroids),key...
    return (p,nearest[0])
def file_read(name):
    fp=open(name,'r')
    data=json.loads(fp.read())
    fp.close
    return data
centroids=file_read('centroids.json')
for x in sys.stdin:
    print(kmeans_assign(x, centroids))
```

The Source Code shown in 3 reports the Main file which has been created with information relative to the target platform. Commands stored in local variables, together with

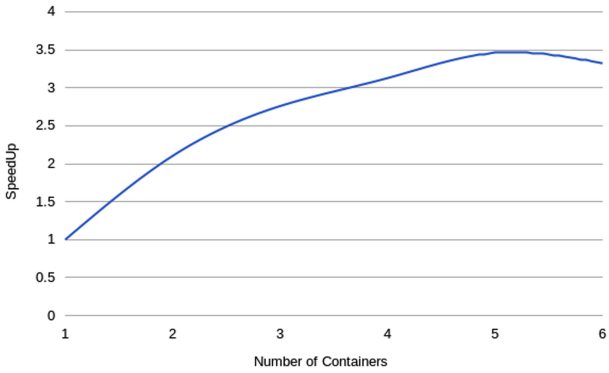

Fig. 9. Speed-Up results for Kmeans.

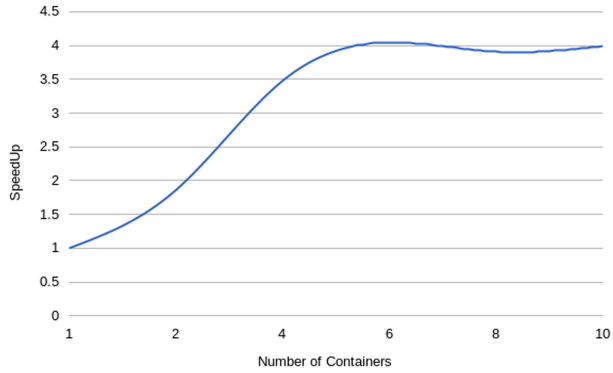

Fig. 10. Speed-Up results for MTBF.

data location information, are added to an empty Skeleton by the compiler, to adapt it to a specific target.

The Mapper and Reducer files are indeed very simple: as it can be seen in code boxes 4 and 5 respectively, the Mapper is in charge of distributing the incoming data, exploiting the Standard Input and Standard Output of the Python program running on the Hadoop Cluster, while the Reducer calculates the result of the function passed through the Primitive (a distance in our case). The other two Mapper and Reducer files have not been reported for brevity.

7.4 The Compiler Web Interface

A web graphical interface, available online, has been provided to interact with the compiler. When the modified source code is presented to the Web Compiler, it is analyzed and transformed, according to the specific Parallel Pattern which has been chosen for the parallelization. As it can be seen from Fig. 8, the code can be pasted in the central area, while the Pattern and the Deployment Platform can be selected from the menu just below the code area.

8 Results

The experimental results (speed-up figures) shown in Fig. 9 have been obtained by running the Kmeans example illustrated in previous sections, parallelized according to the Producer-Consumer pattern, targetted on a Docker executing environment, and ran on a workstation (based in our laboratory) equipped with one node / 4 cores (8 threads). The elapsed time figures have been averaged from 4 independent runs. The figure shows satisfactory speedup, up to 3 consumer containers running on distinct cores (the fourth one dedicated to the Producer and queue-manager containers. With more than 3 consumer containers, the speed-up slows down, due to the fact that some of the cores are shared by the different containers. We have used a *docker-compose.yml* version 2, which allows to allocate the containers on specific cores (thus allowing to control the execution of dockers on dedicated cores).

Fig. 10 reports the speed-Up results obtained on the real use case (LightSource's MTBF analytics) illustrated in Section 7.1, parallelized according to the Producer-Consumer pattern, targetted on a Docker executing environment, and ran on a workstation (based at LightSource premises) equipped with 2 nodes, 4 cores each. The speed-up scalability behaviour is here again satisfactory until the consumer containers run on distinct cores (that is, up to 6 cores, the remaining 2 cores utilized by the producer and queue-manager containers).

9 Conclusions and Future Work

In this paper the Code Based Approach and Compiler, developed within the TOREADOR project, has been presented. The article has focused on a Key-means algorithm applied in a simplified scenario, but several other algorithms have been considered during experimentation, taken from the top 10 most used one in Data Mining and Big Data analytics: C4.5, APriori, Support vector machines, PageRank, k-NN, Nayve Bayes.

The supported deployment platforms have been briefly listed in Section 5.3 but, also in this case, we plan to further extend the support to platforms which can offer Stream analysis capabilities. Research is being carried on the possible composition of Apache Kafka and Spark, as a target environment, to support data Streaming.

In the future, it has been planned further experiments that will be run to determine the performance improvement obtained with different configurations of the execution parameters (number and power of the execution nodes, data distribution paradigms). The acquired data, and the instruments that will be used to collect them, will become part of the parallelization suite which will be offered to the final user, in order to support her in the optimization of the code.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the TOREADOR project, grant agreement Number 688797.

References

[1] [Online]. Available: http://www.toreador-project.eu/. Accessed on: Feb. 2019.

[2] M. Aldinucci, M. Danelutto, and P. Dazzi, "MUSKEL: An expandable skeleton environment," *Scalable Comput.: Practice Exp.*, vol. 8, no. 4, pp. 325–341, 2007.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-level and efficient streaming on multi-core, " in, *Programming Multi-Core and Many-Core Computing Systems*. Hoboken, NJ, USA: Wiley, 2014.

[4] C. A. Ardagna, V. Bellandi, P. Ceravolo, E. Damiani, B. Di Martino, D. Salvatore, and A. Esposito, "A fast and incremental development life cycle for data analytics as a service," in *Proc. IEEE Int. Congr. Big Data*, 2018, pp. 174–181.

[5] G. E. Blelloch, "NESL: A nested data-parallel language (version 3.1)," School Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-95–170, 1995.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center Comput. Sci., Bowie, MD, USA, Tech. Rep. CCS-TR-99–157, 1999.

[7] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.
[8] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. London, U.K.: Pitman, 1989.
[9] M. Danelutto, "Distributed systems: Paradigms and models," *Support Material-Laurea Magistrale Comput. Sci. Netw.* Pisa, versione Sep. 25, 2013, Available online at http://backus.di.unipi.it/~marcod/SPM1415/spmSept14Epub.pdf, last checked Mar. 2019.
[10] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[11] B. Di Martino, S. D'Angelo, and A. Esposito, "A platform for MBDAaaS based on patterns and skeletons: The python based algorithms compiler," in *Proc. IEEE 14th Int. Conf. Netw. Sens. Control*, 2017, pp. 400–405.
[12] B. Di Martino, S. D'Angelo, A. Esposito, I. Martinez, J. Montero, and T. Pariente Lobo, "Parallelization and deployment of big data algorithms: The TOREADOR approach," in *Proc. 32nd IEEE Int. Conf. Adv. Inf. Netw. Appl.*, May 2018, pp. 408–412.
[13] B. Di Martino, S. D'Angelo, A. Esposito, R. Cappuzzo, and A. S. de Oliveira, "ROCK algorithm parallelization with TOREADOR primitives," in *Proc. 32nd Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2018, pp. 402–407.
[14] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Integration of agent-based and cloud computing for the smart objects-oriented IoT," in *Proc. IEEE 18th Int. Conf. Comput. Supported Cooperative Work Des.*, May 2014, pp. 493–498.
[15] G. Fortino and P. Trunfio, *Internet of Things Based on Smart Objects*, G. Fortino & P. Trunfio, Eds., Berlin, Germany: Springer, 2014.
[16] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Softw.: Practice Exp.*, vol. 40, no. 12, pp. 1135–1160, 2010.
[17] M. Oberdorfer and J. Gutowski, "CxC parallel programming," *C/C++ Users J.*, CMP Media LLC, United States, 2004, pp. 42–47.
[18] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. London, U.K.: Pearson Education, 2004.
[19] E. Totoni, T. A. Anderson, and T. Shpeisman, "HPAT: High performance analytics with scripting ease-of-use," in *Proc. Int. Conf. Supercomput.*, 2017, Art. no. 9.
[20] S. Toub, "Patterns for parallel programming: understanding and applying parallel patterns with the .NET framework 4," Available online at https://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee, Last checked Mar. 2019.

Beniamino Di Martino is a full professor of information systems with the Engineering Department, University of Campania "Luigi Vanvitelli" (Italy). He is author of 14 international books and more than 300 publications in international journals and conferences. He has been coordinator of EU funded FP7-ICT Project mOSAIC, and participates to various international research projects with various leadership roles (among them five EC FP7 and H2020 projects). He is editor / associate editor of seven international journals and EB member of several international journals. He is vice chair of the IEEE CS Technical Committee on Scalable Computing. He is member of many Technical Committees, including: IEEE WG for the IEEE P3203 Standard on Cloud Interoperability, IEEE Intercloud Testbed Initiative, IEEE Technical Committees on Scalable Computing (TCSC), on Big Data (TCBD), on Data Engineering (TCDE), on Semantic Computing (TCSEM), on Services Computing (TCSVC), on Intelligent Informatics (TCII), on Pattern Analysis and Machine Intelligence (TCPAMI), on Software Engineering (TCSE), on Distributed Processing (TCDP), on Parallel Processing (TCPP), on Cloud Computing (TCCC), of Cloud Standards Customer Council, of OMG Cloud Working Group, of Cloud Computing Experts' Group of the European Commission. He is chair of Nomination Committee for the "IEEE TCSC Award of Excellence in Scalable Computing" and member of Nomination Committee for the "IEEE TCSC Award for Medium Career Researchers". His research interests include: Knowledge discovery and management, Semantic Web and Semantic Web services, Semantic based information retrieval, cloud computing, big data intelligence, high performance computing and architectures. Mobile and intelligent agents and mobile computing, reverse engineering, automated program analysis and transformation, algorithmic patterns recognition and program comprehension, image analysis.

Antonio Esposito received the PhD degree in December 2016, with a thesis on a pattern-guided Semantic approach to the solution of portability and interoperability issues in the Cloud enabling automatic services composition. He is a postdoc with the Engineering Department, University of Campania "Luigi Vanvitelli" (Italy). He participated in research projects supported by international and national organizations, such as EU-FP7-SMARTCITIES CoSSmiC and EU-H2020-BIGDATA Toreador. His main interests include software engineering, cloud computing, design and cloud patterns, Semantic based information retrieval.

Salvatore D'Angelo received the master's degree in computer engineering, in 2016. He is working toward the PhD degree in computer and electronic engineering in the Department of Engineering, University of Campania "Luigi Vanvitelli". His interests include research activities dealing with big data, machine learning, cloud computing, and high performance computing.

Salvatore Augusto Maisto received the master's degree in computer engineering, in 2016. He is working toward the PhD degree in computer and electronic engineering in the Department of Industrial and Information Engineering, University of Campania "Luigi Vanvitelli" (Italy). His interests include research activities dealing with Semantic Web, knowledge discovery, cloud computing, business process management, microservices, and software modernization.

Stefania Nacchia received the master's degree in computer engineering, in 2016. She is working toward the PhD degree in computer and electronic engineering in the Department of Engineering, University of Campania "Luigi Vanvitelli" (Italy). She is involved in research activities dealing with Semantic Web, knowledge discovery, cloud computing, business process management, and machine learning.