IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 28, 2021, accepted March 5, 2021, date of publication March 22, 2021, date of current version March 31, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3067815

A Data-Aware Scheduling Strategy for Executing Large-Scale Distributed Workflows

SALVATORE GIAMPÀ[1], LORIS BELCASTRO[1], FABRIZIO MAROZZO[1,2], DOMENICO TALIA[1,2], AND PAOLO TRUNFIO[1,2]

[1]DIMES, University of Calabria, 87036 Rende, Italy
[2]DtoK Lab Srl, University of Calabria, 87036 Rende, Italy

Corresponding author: Fabrizio Marozzo (fmarozzo@dimes.unical.it)

This work was supported by the ASPIDE Project through the European Union's Horizon 2020 Research and Innovation Programme under Agreement 801091.

ABSTRACT Task scheduling is a crucial key component for the efficient execution of data-intensive applications on distributed environments, by which many machines must be coordinated to reduce execution times and bandwidth consumption. This paper presents ADAGE, a data-aware scheduler designed to efficiently execute data-intensive workflows in large-scale computers. The proposed scheduler is based on three key features: *i*) *critical path analysis*, for discovering the critical tasks of a workflow and reducing data transferring between nodes; *ii*) *work giving*, a new dynamic planning strategy for migrating tasks from overloaded to unloaded nodes; and *iii*) *task replication*, which executes task replicas on different nodes for improving both execution time and fault tolerance. Experiments performed on a distributed computing environment composed of up to 1,024 processing nodes show that ADAGE achieves better performances than existing scheduling systems, obtaining an average reduction of up to 66% in execution time.

INDEX TERMS Data-aware scheduler, workflow scheduling, distributed workflows, parallel programming, distributed computing, exascale computing.

I. INTRODUCTION

The term Exascale refers to the capabilities of future computing systems, still to be implemented, which should be capable of calculating at least one exaFLOPS (i.e., 10^{18} FLOPS), far exceeding the most advanced existing computing systems (about 10^{15} FLOPS). To reach the Exascale size, other than new hardware solutions, it is required to define new programming models, languages and algorithms that combine abstraction with both scalability and performance [1]. Hybrid models (based on shared/distributed memory) and communication mechanisms based on data locality and grouping are currently investigated as promising approaches. Parallel applications running on Exascale systems will require to control a high number of tasks running on a very large set of computing resources [2]. Such applications will need to avoid or limit synchronization, use less communication and remote memory, and handle software and hardware faults that can occur. In order to achieve such computational speeds, more and more novel solutions are being proposed with the aim of harnessing the computational power of a large set of machines operating in parallel [3].

The problem of coordinating many machines in a complex distributed system is widely represented as a task scheduling problem. Task scheduling has long been recognized as a NP-Hard problem, which represents a major challenge for researchers, especially if the scheduling is performed dynamically and in real time (as required by modern systems). The scheduling aims at identifying the most suitable resources for executing the workloads on time and optimizing resource utilization. In particular, it must also allow for many tasks to run simultaneously and for the exchange of large amounts of data. This paper presents ADAGE (A Data-aware scheduler based on criticAl path, work assiGnment, and rEplication), a data-aware scheduler designed to efficiently execute data-intensive workflows in a large-scale computer network. The proposed scheduler is based on three key features: *i*) *critical path analysis*, for discovering the critical tasks of a workflow and reducing data transferring between nodes; *ii*) *work giving*, a new dynamic planning strategy for migrating tasks from

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

overloaded to unloaded nodes; and *iii*) *task replication*, which executes task replicas on different nodes for improving both execution times and fault tolerance.

ADAGE is composed of the following components: *i*) *Decision Maker* (*DM*), which runs on each node and assigns the tasks to the current node or remote nodes; *ii*) *Local Ready Queue* (*LRQ*), which contains the tasks that are ready to be executed, sorted by execution priority; *iii*) *Load Balancer* (*LB*), which moves tasks from the LRQ to the least loaded neighboring nodes whenever the current node is overloaded; and *iv*) *Validator*, which checks and updates the status of the tasks in LRQ. These components work together to effectively execute the submitted workflow, which is composed of many tasks with dependencies, on a pool of computing nodes in a distributed platform.

To evaluate our strategy, we carried out several experiments on different workflows by varying both the number of tasks and computing nodes. In our evaluations, we compared the designed scheduling strategy with two state-of-the-art systems, i.e., Matrix [4] and Albatross [5]. In particular, five existing workflows (i.e., *CyberShake*, *Epigenomics*, *Inspiral*, *Montage*, *Sipht*), which can generate up to 10,000 tasks, were evaluated. Experiments performed on a HPC distributed system composed of 1,024 computing nodes show that ADAGE achieves better performance than existing scheduling systems, obtaining an average reduction of up to 66% in execution time. For the purpose of using the code of our scheduler and allowing the reproducibility of the experiments, an open-source version of ADAGE is available at https://github.com/SCAlabUnical/ADAGE.

Compared to existing techniques, our scheduler includes the following innovative aspects: *i*) it combines both static and dynamic planning strategies for reducing execution time of data-intensive workflows; *ii*) it exploits a novel algorithm for moving tasks from overloaded to unloaded nodes at runtime; *iii*) it takes advantage of task replication on different nodes to improve both execution times and fault tolerance.

The remainder of the paper is organized as follows. Section II discusses related work. Section III describes the proposed scheduling strategy. Section IV shows the experimental results and Section V concludes the paper.

II. RELATED WORK

With the increasing popularity of data-intensive workflows, several research projects have been carried out to define data-aware scheduling algorithms [6]– [8] aiming at improving scalability, energy efficiency and execution performance. In particular, due the imminent implementation of Exascale systems, task scheduling for massively parallel applications has become an important and strategic research area [3]. In particular, several algorithms and systems have been proposed to cope with the needs of large scale data-intensive applications, exploiting both static and dynamic scheduling [9]– [11].

Kosar *et al.* [12] proposed a data scheduler, namely *Stork*, that allows for planning data allocation and data transfer among computing nodes in a network. In particular, Stork uses the *ClassAd* [17] language to represent data management tasks, while computational workflows are executed using both Pegasus [18] for data-aware planning and *HTCondor DAGMan* [19] for managing task dependencies. Stork exposes four main data scheduling strategies: first fit, largest fit, smallest fit and best fit. The first three strategies are heuristics, while the fourth one is an exact algorithm that discovers the best data allocation using a greedy approach.

Wei *et al.* [13] proposed a data-aware scheduling strategy obtained as the combination of two existing systems: LSF (Load Sharing Facility) [20] and GFarm [21]. LSF is a job scheduler expressively designed for HPC systems that exposes a set of scheduling tools for managing global workloads and resources. GFarm is a distributed file system that is designed for data sharing in large clusters. The proposed strategy exploits data location information retrieved from GFarm for evaluating data affinity of tasks and automatically transfer data among nodes.

Acevedo *et al.* [14] proposed a data-aware scheduling algorithm based on a variant of the critical path algorithm [22], named Critical Path File Location (CPFL). The algorithm is designed to schedule workflow tasks by declaring inter-task and data dependencies. It also allows to execute an application composed of multiple workflows by merging them in a single meta-workflow. The scheduler exploits a prescheduling stage to establish where data should be allocated. Then, the critical path algorithm is used to assign a priority value to each task of the meta-workflow. Subsequently, the scheduler manages each task using priority and assigns it to a computing node based on its data dependencies.

Marozzo *et al.* [15] proposed a composition of two systems, the Data Mining Cloud Framework (DMCF) [23] and Hercules [24], to obtain a data-aware workflow scheduling for Cloud environments. DMCF allows to process and schedule workflow tasks, while Hercules manages temporary files generated during computation. The scheduling strategy is inspired by the one proposed in [25], but it uses a new local queue on the executor node, called *locallyActivatedTask*, to obtain data-awareness. For each node, the scheduler selects the best task to run, choosing it from the global or local task queue. In particular, the scheduler tries to execute the task whose dependencies have been resolved and for which the current node is the best concerning data allocation.

MATRIX (*MAny-Task computing execution fabRIc at eXascale*) [16] is a system that implements a data-aware scheduling strategy based on work stealing [26]. It extends the classical work stealing strategy to support data-awareness by maintaining information about data dependencies of scheduled tasks. The system consists of three entities: client, scheduler and executor. The network nodes are fully connected, which means that they can communicate each other. On each node, three basic components run: executor, scheduler, and ZHT (*Zero-hop distributed HashTable*) server [27]. In particular, the last component allows to implement a shared DKVS (*Distributed Key-Value Store*) that stores information

TABLE 1. Comparison with main related systems.

Reference	Metadata Management System	Storage system	Language	Scheduler features	Evaluated metrics
Kosar et al. [12]	HTCondor DAGMan	Pegasus	Go	Stork HTCondor-G	Transfer speed
Wei et al. [13]	LSF	GFarm	C\C++	LSF plugin	-
Acevedo et al. [14]	DHT	Distributed/Local filesystem	Java	CPFL	Execution time
DMCF [15]	Microsoft Azure	Hercules Microsoft Azure	Javascript VL4Cloud	In-memory temporary file	I/O operations, Execution time
MATRIX [16]	ZHT	ZHT	C\C++	Work Stealing	Throughput
Albatross [5]	ZHT	HDFS	C\C++	Late Binding	Latency, Throughput
ADAGE	*DHT*	*Distributed/Local file system*	*Java*	*Critical Path Work Giving Backup tasks*	*Execution time Throughput*

about tasks, including data dependencies and data locality. MATRIX exploits three local queues to manage tasks, which contain respectively: tasks that are not ready to run; strictly data-dependent tasks that read much data from well-defined nodes; and not strictly data-dependent tasks that read some temporary data.

Albatross [5] is a system that improves some features of MATRIX. For example, it enhances fault tolerance by replacing the local queue containing not ready tasks with Fabriq [28]. Fabriq is a distributed message queue (DMQ) that runs on top of a distributed hash table, which prevents losing tasks when a node fails. Albatross assign tasks to nodes by using a *late-binding* technique. Specifically: *i*) when a task becomes ready (i.e., all its dependencies are solved), the load balancer pulls it from the DMQ and tries to send it to the best node according to data locality; *ii*) if the remote node is overloaded or data are local, the task is assigned to the local queue of the current node; *iii*) when the task is pulled from a local queue, the load balancer tries again to send it to the best node and, if the assignment is not possible, the task is executed on the current node.

Table 1 shows a comparison among the referred related works. For each work, the table reports the metadata management and storage systems, implementation language, features of the scheduler, and performance metrics that have been evaluated. Differently from existing techniques, ADAGE combines both static and dynamic planning strategies for improving the execution performances of data-intensive workflows. In particular, a static planning strategy, based on the critical path algorithm, is used to optimally assign tasks to the nodes during the workflow submission. Then, a novel dynamic strategy, named *work-giving*, is used by overloaded nodes for assigning tasks to other nodes. Furthermore, ADAGE exploits task replication on different nodes to improve both execution times and fault tolerance.

III. PROPOSED SCHEDULER

ADAGE is a new data-aware scheduler that exploits data locality to reduce data movement among nodes and improve the execution time of data-intensive workflows. To reach this goal, ADAGE combines both static and dynamic planning strategies.

The static planning strategy is based on the *critical path* algorithm [22], which permits to find the *critical tasks* of a workflow, i.e. tasks that cannot be delayed without delaying the execution of the entire workflow. Starting from the knowledge of the critical path, our strategy minimizes data movement and memory latency by executing a task on the node that holds the largest amount of input data.

A dynamic planning strategy is used for assigning tasks to computing nodes at runtime. We designed a new dynamic planning strategy, called *work-giving*, which is used for migrating tasks from overloaded to unloaded nodes. Specifically, if a node is overloaded, it tries to send some of its tasks to unloaded nodes in its neighborhood. Such behavior differs from the work-stealing approach, in which an entity runs on the unloaded nodes and, during the computation, searches and steals tasks from the overloaded ones. It should be noted that the stealing process is activated many times and in many nodes. This behavior can limit the scalability in large-scale computing systems (such as Exascale computers), where the unloaded nodes are usually much more than the overloaded ones. Additionally, each unloaded node competes with the others to steal tasks, which can lead to a highly random distribution of tasks in the system. Differently, the *work-giving* strategy is executed on a much smaller number of nodes, which improves the system scalability. In addition, this approach limits the random distribution of the tasks by allowing an overloaded node to assign tasks to a limited number of nodes in its neighborhood.

For increasing the application reliability and finishing computation faster, ADAGE exploits *task replication* to execute speculative copies of tasks on different nodes. As stated in [29], the use of task replicas (also called backup tasks) is essential to significantly reduce the completion time of large workflow applications. In fact, some computing nodes may take an unusually long time to complete some tasks (e.g., due to overhead or hardware/software issues), negatively

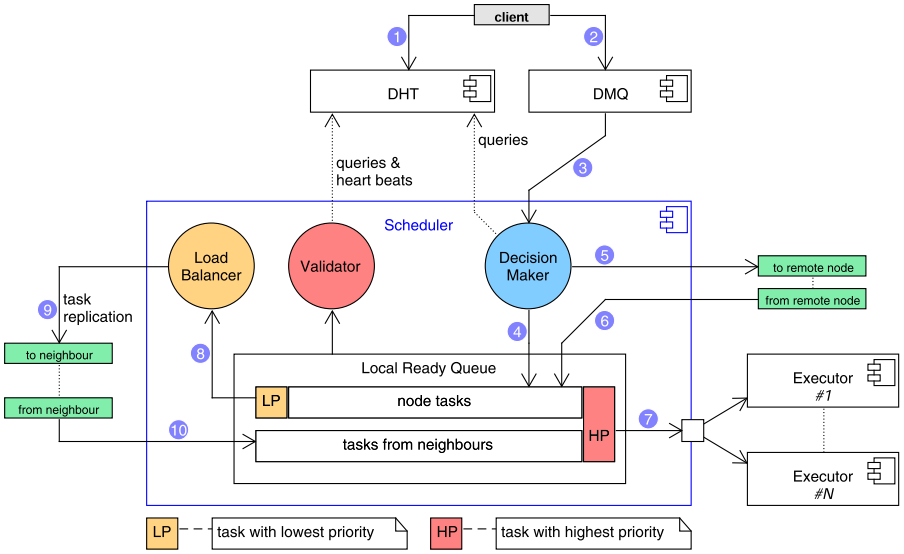


FIGURE. 1. Scheduler block diagram and execution flow.

affecting the completion time of the entire application. This mechanism marks a task as completed when the primary or a replica execution ends.

More details on the architecture, metadata and algorithms exploited by ADAGE are provided in the following sections.

A. ARCHITECTURE

The software structure of ADAGE consists of the following macro-components:

- *Client*: given a workflow composed of several tasks, it executes the critical path algorithm to pre-assign the tasks to the nodes and calculates the priority for each of them.
- *Distributed Hash Table* (*DHT*): it stores all the necessary information about tasks, such as the running state, parent tasks, and actual number of replicas.
- *Distributed Message Queue* (*DMQ*): it stores the identifiers of tasks waiting to be executed by a processing node.
- *Scheduler*: it executes a dynamic scheduling strategy, named *work-giving*, which is discussed in Section III.

An instance of the scheduler runs on each node of the system. Specifically, such a scheduler instance is composed of the following components:

- *Decision Maker* (DM): it statically assigns tasks to the current node or remote nodes.
- *Local Ready Queue* (LRQ): it contains the *ready tasks*, which are tasks whose dependencies are solved; such tasks are sorted by the execution priority calculated with the critical path algorithm.
- *Load Balancer* (LB): when the current node is overloaded, the LB selects and sends some tasks from the LRQ to less loaded neighbor nodes.
- *Validator*: it checks the completion of tasks in the LRQ; if a task is completed, the Validator removes it from the queue; otherwise, if the task execution is still pending, the heartbeat is updated and stored in the DHT.

Figure 1 shows the block diagram and execution flow of the scheduling system. Each node can dispose of one or more Executors, which pull ready tasks from the LRQ. If the LRQ is empty, the execution of the Decision Maker is triggered.

As shown in Figure 1, the client starts storing task metadata in the DHT (1) and tasks in the DMQ (2) (also specifying the preferred execution node for each of them). In particular, the client executes the critical path algorithm to find an optimal planning solution for assigning the tasks to nodes. On each node, an instance of the Decision Maker (DM) takes one or more tasks from the DMQ (3) and decides on which node they have to be executed. Specifically, if a task has been assigned by the client to the current node, it is put in the Local Ready Queue (LRQ) (4); otherwise, such a task is sent to another node, which is chosen based on data locality (5), and inserted in the LRQ (6).

The Executors get the tasks with the highest priorities from the LRQ (7) for running them, while the Load Balancer (LB) gets those with the lowest priorities (8). In the latter case, the tasks are replicated and sent to some neighbor nodes that are less charged than the current one (9). In particular, each replicated task is inserted in the LRQ of the chosen neighbor node (10). The maximum number of replicas for a task is specified by the client during the workflow submission and stored in the DHT.

Only the tasks that are assigned to the current node can be replicated by the LB. In fact, as shown in Figure 1, the LRQ is logically split in two parts so as to distinguish between the tasks that have been assigned to the current node and replicas that have been received from neighbors. More details

TABLE 2. The task metadata stored in the DHT.

Metadata	*Description*
ID	Task identifier
state	[waiting, ready, stage-in, running, failed, stage-out, complete]
heart-beat	timestamp of last heartbeat from the nodes having the task
children	set of children tasks
parents	number of parents the task is waiting for
in-list	names of input files the task reads
out-list	names of output files produced by the task
node-list	list of nodes having the task
num-backup	maximum number of backup nodes, given by the client
state-history	timestamps of task state changes (e.g., [{ts1:waiting},{ts2:ready},...])

TABLE 3. The possible states of a task.

State	*Description*
waiting	the task is waiting termination of parent tasks
ready	the task is ready to execute, because its parents are complete
stage-in	the task is ready to run, but it is waiting availability of input data
running	the task is running
failed	the task failed for some reason
stage-out	the task is complete, its output data are being written in the storage
complete	the task is complete, its output data are available from the storage

about the different components of the scheduling system are provided in the following subsections.

B. TASK METADATA AND STATES

Table 2 reports the main metadata of a task. In particular, the field *state* represents the current state of the task, which can take one of the values reported in Table 3.

Figure 2 shows the state diagram describing the life cycle of a task. When it is submitted by the client, a task is in the *waiting* state, which means the task is waiting for termination of some parent task. The field *parents* contains the number of parents the task is waiting for. Such a number is decreased every time a parent task terminates.

The task switches to the *ready* state when all its parents have been completed successfully. In such a state, the node can start preparing the task for execution, getting the needed data. When this happens, the task goes in the *stage-in* state. Once the stage-in process is completed, the task is executed, passing in the *running* state. If the task fails its execution due to a self-generated error (e.g., a programming error or an unhandled exception), it goes in the *failed* state and all its children fail in cascade. Alternatively, if the task successfully completes its execution, the scheduler starts to write the output data in the storage and the task is switched in the *stage-out* state. Finally, the task goes in the *complete* state when the output data have been fully stored.

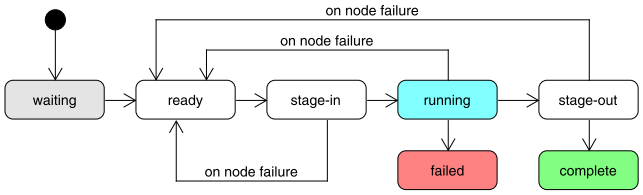


FIGURE. 2. State diagram describing the life-cycle of a task.

```
node: the current node the Executor is running on
nodeSet: the set of all nodes in the system
executorInterval: time to await between task pulling attempts

function Executor(node, nodeSet, executorInterval):
  repeat:
    if LRQ is empty:
      invoke DecisionMaker(nodeSet)
    end if

    if LRQ is not empty:
      task := get the next non-finished task from LRQ
      run the task to its completion
      set the "state" field of task to "complete"
      add a new timestamp for the "complete" state to the "state-history" field of task
      foreach child in "children" field of task:
        decrease the "parent" field of child
      end foreach
    else
      await executorInterval
    end if
  until not interrupted
end function
```

Listing 1. Pseudo-code of the Executor component. It invokes the Decision Maker when no tasks are present in the Local Ready Queue.

C. DECISION MAKER AND EXECUTOR

When an Executor terminates its current work, it selects another task from the LRQ. If the queue is empty, it activates the Decision Maker (DM), which starts to load new tasks from the DMQ to the LRQ. Then, the Executor tries again to get a task from the LRQ and, if found, executes it; otherwise it activates the DM again. However, to limit the network overload due to subsequent calls to the DMQ, the Executor awaits a short time before activating the DM again. Listing 1 shows the pseudo-code of the Executor component.

After executing a task, the Executor updates metadata of the task and its children. In particular:

- the *state* field is set to *complete*;
- a new timestamp for the *complete* state is added to the *state-history* field;
- for each *child* in the *children* field, the *parents* field is decreased.

The Decision Maker performs an initial distribution of tasks to nodes based on data locality. In particular, it performs the following steps:

1) It checks the DMQ looking for non-finished tasks that have been assigned to the current node and whose

```
nodeSet: the set of all nodes in the system

function DecisionMaker(nodeSet):
  taskFound := false
  foreach task in DMQ do:
    if task is not finished and task is not in waiting state and task is preassigned to this node and the last heartbeat of the task is too old then:
      update the heartbeat of the task
      insert task in the LRQ
      taskFound := true
    end if
  end foreach
  if taskFound is false then:
    foreach task in DMQ do:
      if task is not finished and task is not in waiting state and task is not preassigned to this node and the last the heartbeat of task is too old then:
        pre_node := the node to which the task has been preassigned
        if pre_node is not null and pre_node is online then:
          update heartbeat of the task
          insert task in the LRQ of pre_node
        else:
          best_node := GetBestNode(task, nodeSet)
          update heartbeat of the task
          insert task in the LRQ of best_node
        end if
      end if
    end foreach
  end if
end function
```

Listing 2. Pseudo-code of the Decision Maker.

heartbeats are not up-to-date. If some tasks are found, the DM inserts them in the LRQ and terminates its execution; otherwise, it proceeds to the second phase.

2) It scans the DMQ again looking for non-finished tasks that are not assigned to the current node and whose heartbeats are not up-to-date (i.e., the tasks that have been pulled from the DMQ but whose assignee node failed). If the DM gets any tasks matching such criteria, it decides where to send them. Specifically, if a task is assigned to the current node, it is inserted in the LRQ; otherwise, such a task is sent to another node in the system, which is optimally chosen based on data locality.

The DM aims at improving the fault tolerance of the system when the pre-assignment of tasks (made by the client) is no longer feasible (e.g., because some assignee node is failed or unreachable). Listing 2 shows the pseudo-code of the DM.

In particular, when a task cannot be executed by the assignee node, the DM sends it to another node for execution. The new assignee node is chosen using a heuristic that takes into account the location of the input data used by the task. Listing 3 shows the pseudo-code of the procedure used to find the best node for executing a task according to its input data locality. Similarly to what was proposed by Acevedo *et al.* [14], such a procedure aims at minimizing the total transfer time of all input data. In particular, the transfer time of a file is calculated as the ratio between file size and bandwidth of the node. Given a node, the total transfer time is calculated by considering all the input files, required by the task, that are owned by the node itself. Then, the node that grants the smallest total transfer time is chosen.

```
task: the task to search the best node for
nodeSet: the set of all nodes in the system

function GetBestNode(task, nodeSet)
  best-node := null
  best-data-transfer-time := 0
  foreach node in nodeSet do:
    if not task is already assigned to node then:
      data-transfer-time := 0;
      foreach file read by task do:
        file-nodes := get all the nodes that contains a replica of the file
        if node is in file-nodes then:
          data-transfer-time := data-transfer-time + ( size of file)/(bandwith of node)
        end if
      end foreach

      if best-node is null or data-transfer-time < best-data-transfer-time then:
        best-data-transfer-time := data-transfer-time
        best-node := node
      end if
    end if
  end foreach
  return best-node
end function
```

Listing 3. Pseudo-code of the procedure used to find the best node for executing a task.

The time complexity of the Decision Maker function is $O(t * n)$, where t is the number of submitted tasks and n is the number of nodes in the computer network.

D. LOAD BALANCER

The Load Balancer (LB) is a periodic thread that monitors the workload of all nodes in the neighborhood in order to efficiently distribute tasks among them.

Listing 4 shows the pseudo-code of the LB component. In particular, after setting an initial waiting time *lbTime*, the LB gets the list of neighbor nodes, which can be retrieved by using a static or a dynamic approach. According to the static approach, the neighborhood does not change over time, while using the dynamic one it can be calculated many times. For example, MATRIX [4] uses a dynamic selection strategy that randomly chooses a number of neighbor nodes equal to the square root of the total number of nodes. Then, if the local node is the most overloaded one in the neighborhood, the LB sends half of the tasks to the less overloaded neighbor node. However, such operation can fail if: *i*) the local node is not the most overloaded one in the neighborhood; *ii*) a communication error happens; *iii*) the LRQ is empty. In such cases, *lbTime* is doubled and the LB performs a new attempt after sleeping for that time. To avoid too long waits, *lbTime* is doubled until it reaches a maximum allowed value. On the other hand, if the tasks are sent successfully, the waiting time is reset to the initial value (e.g., 1 millisecond). Before sending tasks to a neighbor, the LB replicates them and maintains the original copies in the LRQ. In such a way, the different replicas of a task compete each other to be executed first. The first replica that completes its execution determines the end of the task and, consequently, causes the termination of all other running replicas.

```
node: the local machine
maxTime: the maximum waiting time

function LoadBalancer(node, maxTime):
  lbTime := 1
  repeat:
    success := false
    neighbourhood := get neighborhood of this node
    if node is the most overloaded one in neighborhood
    ↪ then:
      neighbour := get the less overloaded node in the
    ↪ neighborhood
      tasks := select half of the tasks assigned to this
    ↪ node from the LRQ
      if tasks is not empty then:
        insert the replicas of all tasks in the LRQ of
    ↪ the neighbor
        lbTime := 1;
        success := true
      end if
    end if
    if not success and lbTime < maxTime:
      double lbTime (lbTime := lbTime * 2)
    end if
    await lbTime
   until not interrupted
end function
```

Listing 4. Pseudo-code of the Load Balancer.

As it can be observed from Listing 4, the time complexity of the Load Balancer function is $O(t + n)$, where t is the number of submitted tasks and n is the number of nodes in the network.

E. VALIDATOR

As explained in Section III-D, the Load Balancer can replicate the tasks on the neighborhood of the current node. In particular a replica of a task is inserted in the LRQ of a neighbor node. An additional component, namely the Validator, monitors the LRQ looking for tasks that completed their execution. If any are found, the local replicas of such tasks are removed and not executed again. This operation can be accomplished by querying the DHT, which stores all the needed information about the tasks that are currently running in the system. The Validator also updates the heartbeats of both the tasks in the LRQ and those that are currently running. Listing 5 shows the pseudo-code of the Validator component.

On the single node, the Validator function has a time complexity that is linear in the number of tasks in the LRQ. Considering all the nodes, the total time complexity results to be $O(t)$, where t is the number of submitted tasks.

IV. CASE STUDIES AND EXPERIMENTS

We experimentally evaluated the performance of our scheduling strategy using WorkflowSim [30], a widely used toolkit for running distributed workflows, which allows to consider many aspects of the system, such as machine bandwidth, storage types (e.g., RAM, local disk or distributed file systems), hardware specifics (e.g., number of cores and clock speed) and power consumption. Each computing node in our tests has been configured with 4 CPU cores at 2,000 MIPS, 8 GB of RAM, 1 Gbps of bandwidth, and 1 TB of storage.

In our experiments, we used five existing workflows that are defined in [31]: *CyberShake*, *Epigenomics*, *Inspiral*, *Montage*, *Sipht*. To assess the effectiveness of our scheduling strategy, we compared it with two related systems: MATRIX [4] and Albatross [5]. In particular, we evaluated the execution time by varying both the number of nodes and tasks, the throughput as the number of completed tasks per second, and the distribution of the completed tasks over the execution time.

```
runningTasks: the tasks currently running on this node
validationInterval: interval of time between two
    ↪ activations of the Validator

function Validator(validationInterval):
  repeat:
    foreach task in runningTasks do:
      update heartbeat of task
    end foreach
    foreach task in LRQ do:
      if task is finished then:
        remove task from LRQ
      else:
        update heartbeat of task
      end if
    end foreach
    await validationInterval
  until not interrupted
end function
```

Listing 5. Pseudo-code of the Validator.

A. EXPERIMENT RESULTS

Figure 3 reports the elapsed execution time of the different scheduling systems when executing the five different workflows by varying the number of nodes from 1 to 1,024 (i.e., up to 4,096 cores). Specifically, each workflow has been configured to spawn 1,000 tasks. Each test has been configured with the following parameters:

- number of replicas for each task: 2;
- heartbeat expiration period: 125 s;
- period between two subsequent activations of the Validator: 62.5s (i.e., half of the heartbeat expiration period).

As shown in Figure 3(a), for the Sipht application, our strategy results to be on average 21% and 13% faster than Albatross and MATRIX respectively. For other applications, the reduction of execution time ranges from 15% to 30% for Epigenomics (Figure 3(b)), from 20% to 31% for Inspiral (Figure 3(c)), from 18% to 66% for CyberShake (Figure 3(d)), and from 1% to 23% for Montage (Figure 3(e)).

Since our scheduler was designed to support Exascale applications, which can be composed of tens of thousands of tasks, we carried out additional experiments to evaluate the execution times when the number of tasks is increased up to 10,000. For the sake of brevity, we compared the performance of the three systems using only the Montage and CyberShake workflows. Figure 4 shows the execution time obtained by increasing the number of tasks up to 10,000 on 1,024 computing nodes. In particular, as the number of tasks increases, in both Figures 4(a) and (b) the execution time of ADAGE grows slower than that of the other

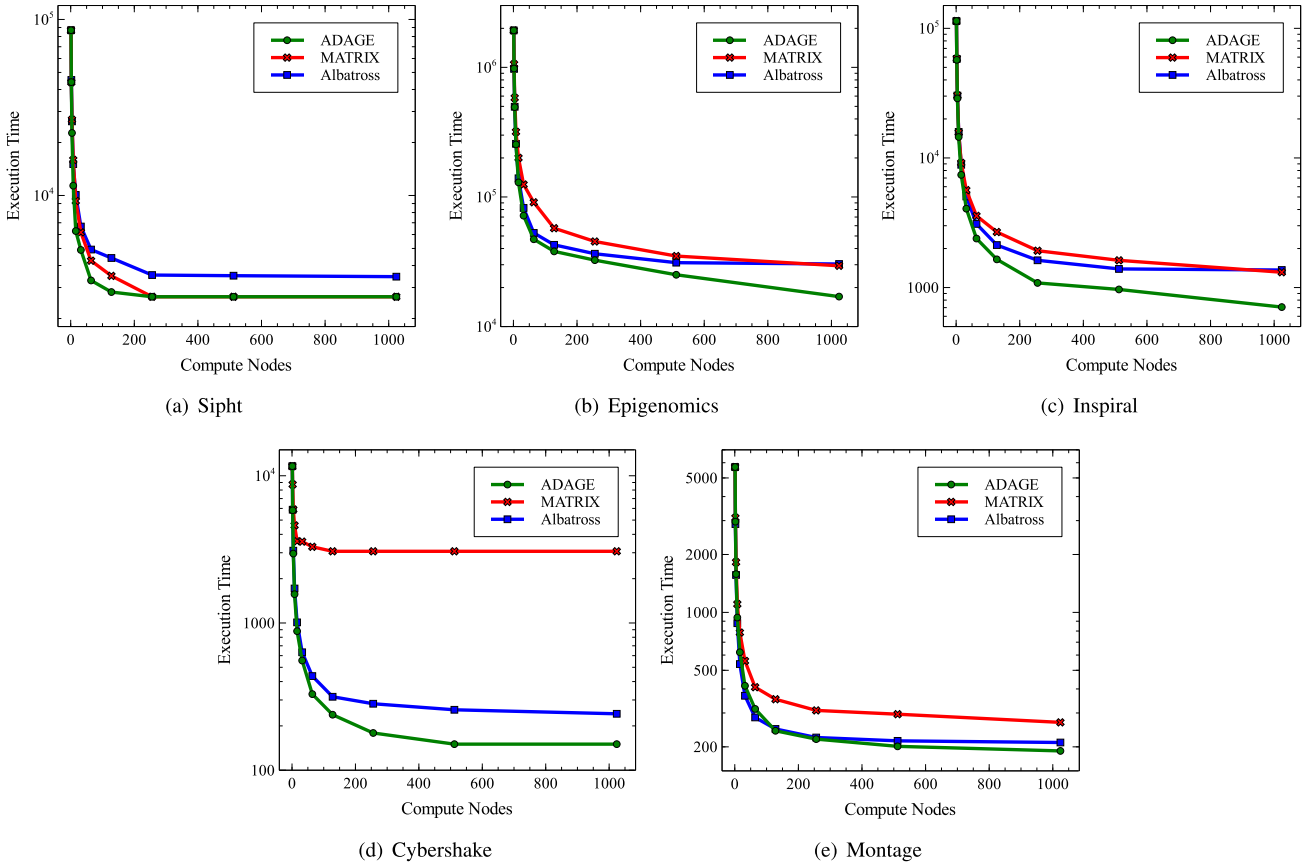


FIGURE. 3. Comparison of the execution times by varying the number of compute nodes (five workflows and 1,000 tasks).

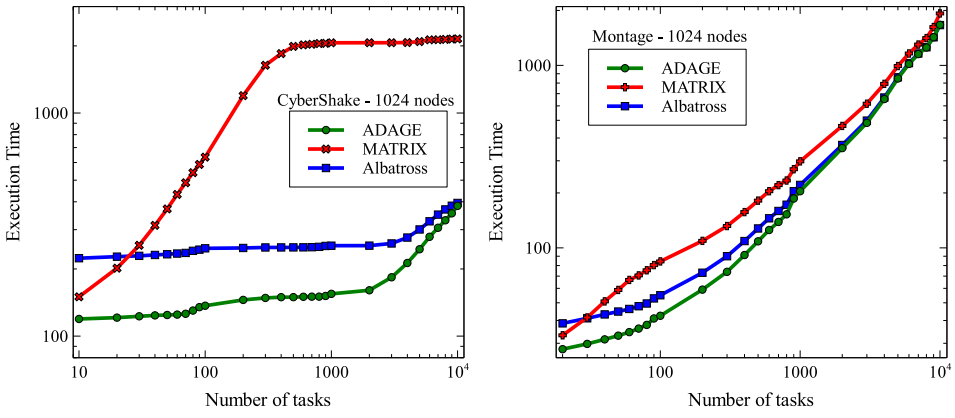
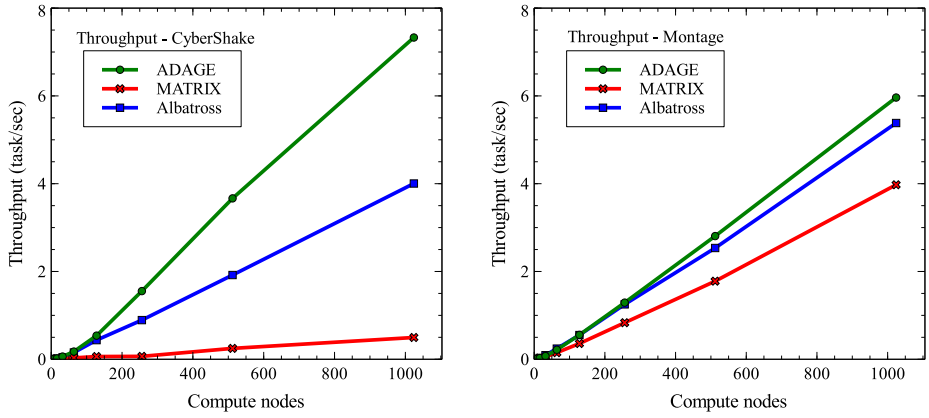


FIGURE. 4. Comparison of the execution times by increasing the number of tasks up to 10,000 (two workflows and 1,024 compute nodes).

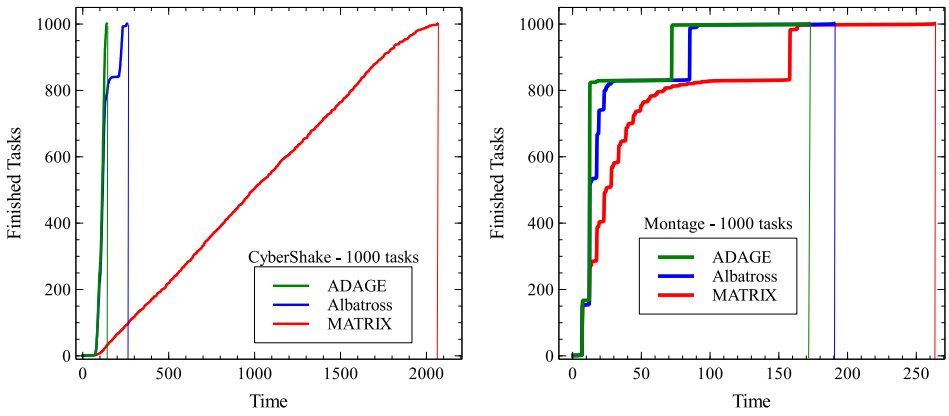
two systems. This experiment demonstrates a greater ability of our scheduler to manage large computational resources.

Figure 5 illustrates the throughput of the different scheduling systems, which has been calculated as the number of completed tasks per second. As shown, ADAGE achieves significantly better results than the other systems. In particular, as the number of available nodes increases, the throughput of our scheduler considerably increases compared to that of the two other strategies, which demonstrates how ADAGE is particularly suitable for very large distributed computation systems. By executing the CyberShake workflow with 1,024 compute nodes, ADAGE obtains a throughput that is 83% and 1377% greater than that of Albatross and MATRIX respectively (Figure 5(a)). In the Montage workflow case, using 1,024 nodes, the throughput of ADAGE is 11%



(a) CyberShake (b) Montage

FIGURE. 5. Comparison of number of tasks completed per second (throughput) vs the number of compute nodes.



(a) CyberShake (b) Montage

FIGURE. 6. Comparison of task completion times using 1,024 compute nodes.

and 50% greater (Figure 5(b)) of Albatross and MATRIX respectively.

Figure 6 shows the distribution of the completed tasks over the execution time. The plotted results show that ADAGE achieves the peak of completed tasks much faster than the other two systems. This figure reports how the execution time of tasks is greatly reduced by using ADAGE, which means that computational resources are released in a shorter time with reference to the other two approaches.

Overall, the obtained results and the wide number of experiments carried out on different workflows demonstrated how the proposed scheduling strategy offers better performance than other existing systems. This is especially true when a very large number of nodes is used. This feature makes the proposed algorithm particularly interesting for supporting massive task execution in the upcoming Exascale systems.

V. CONCLUSION

In this paper we presented ADAGE, a new data-aware scheduling strategy for large distributed computation environments, such as the upcoming Exascale systems. Differently from existing techniques, ADAGE combines both static and dynamic planning strategies for improving the execution time of data-intensive workflows. In particular, it is based on three key features: *i*) *critical path analysis*, for discovering the critical tasks of a workflow and reducing data transferring between nodes; *ii*) *work giving*, a new dynamic planning strategy for migrating tasks from overloaded to unloaded nodes; and *iii*) *task replication*, which executes task replicas on different nodes for improving both execution times and fault tolerance. Experiments performed on a distributed environment composed of up to 1,024 compute nodes showed that ADAGE achieves better performances than existing techniques, obtaining a reduction of up to 66% in execution time. Moreover, as the number of available nodes increases, ADAGE outperformed the other techniques in terms of throughput, demonstrating that is particularly suitable for very large distributed computing systems.

In future work, additional research issues will be investigated. In particular, since our scheduler supports different workflows patterns (e.g., map-reduce, divide-and-conquer, pipeline), we plan to investigate its usability in combination

with Apache Hadoop and Spark, which are widely used for developing and executing general-purpose high performance applications.

DATA AND CODE AVAILABILITY STATEMENT

For the purpose of using the code of our scheduler, an open-source version of ADAGE is available at https://github.com/SCAlabUnical/ADAGE along with some sample workflows and instructions for running experiments.

REFERENCES

[1] G. Da Costa, T. Fahringer, J.-A. Rico-Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, D. Talia, P. Trunfio, and H. Astsatryan, "Exascale machines require new programming paradigms and runtimes," *Supercomput. Frontiers Innov.*, vol. 2, no. 2, pp. 6–27, 2015.

[2] L. Belcastro, F. Marozzo, and D. Talia, "Programming models and systems for big data analysis," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 34, no. 6, pp. 632–652, Nov. 2019.

[3] D. Talia, P. Trunfio, F. Marozzo, L. Belcastro, J. Garcia-Blas, D. D. Rio, P. Couvée, G. Goret, L. Vincent, A. Fernández-Pena, D. M. de Blas, M. Nardi, T. Pizzuti, A. Spătaru, and M. Justyna, "A novel data-centric programming model for large-scale parallel systems," in *Euro-Par 2019: Parallel Processing Workshops* (Lecture Notes in Computer Science). Gottingen, Germany: Springer, Aug. 2020, pp. 452–463.

[4] A. Rajendran, K. Wang, and I. Raicu, "MATRIX: MAny-Task computing execution fabRIc at eXascale," in *Proc. 2nd Greater Chicago Area Syst. Res. Workshop (GCASR)*, Chicago, IL, USA, 2013.

[5] I. Sadooghi, G. Kumar, K. Wang, D. Zhao, T. Li, and I. Raicu, "Albatross: An efficient cloud-enabled task scheduling and execution framework using distributed message queues," in *Proc. IEEE 12th Int. Conf. e-Sci.*, Oct. 2016, pp. 11–20.

[6] T. Kosar and M. Balman, "A new paradigm: Data-aware scheduling in grid computing," *Future Gener. Comput. Syst.*, vol. 25, no. 4, pp. 406–413, Apr. 2009.

[7] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2014, pp. 301–316.

[8] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2011, pp. 295–304.

[9] C. L. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, Aug. 2014.

[10] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *J. Grid Comput.*, vol. 13, no. 4, pp. 457–493, Dec. 2015.

[11] K. Wang, K. Qiao, I. Sadooghi, X. Zhou, T. Li, M. Lang, and I. Raicu, "Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales," *Concurrency Comput., Pract. Exp.*, vol. 28, no. 1, pp. 70–94, Jan. 2016.

[12] T. Kosar, M. Balman, E. Yildirim, S. Kulasekaran, and B. Ross, "Stork data scheduler: Mitigating the data bottleneck in e-science," *Philos. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 369, pp. 3254–3267, Aug. 2011.

[13] X. Wei, W. Li, O. Tatebe, G. Xu, L. Hu, and J. Ju, "Implementing data aware scheduling in Gfarm$^{(R)}$ using LSF$^{(TM)}$ scheduler plugin mechanism," in *Proc. Int. Conf. Grid Comput. Appl.*, Jan. 2005, pp. 3–10.

[14] C. Acevedo, P. Hernández, A. Espinosa, and V. Méndez, "A critical path file location (CPFL) algorithm for data-aware multiworkflow scheduling on HPC clusters," *Future Gener. Comput. Syst.*, vol. 74, pp. 51–62, Sep. 2017.

[15] F. Marozzo, F. R. Duro, J. G. Blas, J. Carretero, D. Talia, and P. Trunfio, "A data-aware scheduling strategy for workflow execution in clouds," *Concurrency Comput., Pract. Exp.*, vol. 29, no. 24, Dec. 2017, Art. no. e4229.

[16] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 119–128.

[17] R. Raman, M. Solomon, M. Livny, and A. Roy, "The classads language," in *Grid Resource Management*. Norwell, MA, USA: Kluwer, Jan. 2004, pp. 255–270.

[18] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.

[19] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," *Concurrency Comput., Pract. Exp.*, vol. 17, nos. 2–4, pp. 323–356, 2005.

[20] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," *Softw., Pract. Exp.*, vol. 23, no. 12, pp. 1305–1336, Dec. 1993.

[21] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, "Grid datafarm architecture for petascale data intensive computing," in *Proc. 2nd IEEE/ACM Int. Symp. Cluster Comput. Grid (CCGRID)*, May 2002, p. 102.

[22] J. E. Kelley, "Critical-path planning and scheduling: Mathematical basis," *Oper. Res.*, vol. 9, no. 3, pp. 296–320, Jun. 1961.

[23] F. Marozzo, D. Talia, and P. Trunfio, "A workflow management system for scalable data mining on clouds," *IEEE Trans. Serv. Comput.*, vol. 11, no. 3, pp. 480–492, May 2018.

[24] F. R. Duro, J. G. Blas, and J. Carretero, "A hierarchical parallel storage system based on distributed memory for large scale systems," in *Proc. 20th Eur. MPI Users-Group Meeting (EuroMPI)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 139–140.

[25] F. Marozzo, D. Talia, and P. Trunfio, "JS4Cloud: Script-based workflow programming for scalable data analysis on cloud platforms," *Concurrency Comput., Pract. Exp.*, vol. 27, no. 17, pp. 5214–5237, Dec. 2015.

[26] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[27] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 775–787.

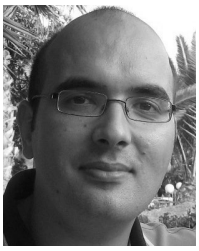
[28] I. Sadooghi, K. Wang, D. Patel, D. Zhao, T. Li, S. Srivastava, and I. Raicu, "FaBRIQ: Leveraging distributed hash tables towards distributed publish-subscribe message queues," in *Proc. IEEE/ACM 2nd Int. Symp. Big Data Comput.*, Dec. 2015, pp. 11–20.

[29] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[30] W. Chen and E. Deelman, "WorkflowSim: A toolkit for simulating scientific workflows in distributed environments," in *Proc. IEEE 8th Int. Conf. E-Sci.*, Oct. 2012, pp. 1–8.

[31] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proc. 3rd Workshop Workflows Support Large-Scale Sci.*, vol. 10, Nov. 2008, pp. 1–10.

SALVATORE GIAMPÀ received the master's degree in computer engineering in 2019. He is currently a Research Fellow of computer engineering with the University of Calabria, Italy. His research interests include distributed and parallel computing, programming framework, and software engineering.



LORIS BELCASTRO received the Ph.D. degree in information and communication engineering with the University of Calabria, Italy. He is currently a Research Fellow of computer engineering with the University of Calabria. His research interests include cloud computing, social media and big data analysis, distributed knowledge discovery, and data mining. In 2012, he received the scholarship from the Institute for High-Performance Computing and Networking of the Italian National Research Council (ICAR-CNR).

FABRIZIO MAROZZO received the Ph.D. degree in systems and computer engineering with the University of Calabria. From 2011 to 2012, he visited the Barcelona SuperComputing Center for a research internship, with the Grid Computing Research Group, Computer Sciences Department. He is currently an Assistant Professor of computer engineering with the University of Calabria. His research interests include distributed systems, data mining, cloud computing, social media, big data analysis, and peer-to-peer networks. He is also serving as an Associate Editor for IEEE Access and the *International Journal of Intelligent Systems Technologies and Applications*.

DOMENICO TALIA is currently a Professor of computer engineering with the University of Calabria. His research interests include parallel and distributed data mining algorithms, cloud computing, grid services, distributed knowledge discovery, peer-to-peer systems, and parallel programming models. He is also a member of the Editorial Board of *Future Generation Computer Systems*, IEEE Transactions on Parallel and Distributed Systems, the *International Journal of Web and Grid Services*, the *Journal of Cloud Computing: Advances, Systems and Applications*, *Scalable Computing: Practice and Experience*, and the *International Journal of Next-Generation Computing*.

PAOLO TRUNFIO was a Research Collaborator with the Institute of Systems and Computer Science of the Italian National Research Council (ISI-CNR), from 2001 to 2002. In 2007 he was a Visiting Researcher with the Swedish Institute of Computer Science (SICS), Stockholm. He is currently an Associate Professor of computer engineering with the University of Calabria. His research interests include cloud computing, social-media analysis, service-oriented architectures, distributed knowledge discovery, and peer-to-peer systems. He is also in the Editorial Board of *Future Generation Computer Systems*, IEEE Transactions on Cloud Computing, and *Journal of Big Data*.