Efficient scheduling of scientific workflow actions in the Cloud based on required capabilities

Michel Krämer[1,2][0000−0003−2775−5844]

[1] Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283 Darmstadt, Germany
[2] Technical University of Darmstadt, 64289 Darmstadt, Germany
michel.kraemer@igd.fraunhofer.de

Abstract. Distributed scientific workflow management systems processing large data sets in the Cloud face the following challenges: (a) workflow tasks require different capabilities from the machines on which they run, but at the same time, the infrastructure is highly heterogeneous, (b) the environment is dynamic and new resources can be added and removed at any time, (c) scientific workflows can become very large with hundreds of thousands of tasks, (d) faults can happen at any time in a distributed system. In this paper, we present a software architecture and a capability-based scheduling algorithm that cover all these challenges in one design. Our architecture consists of loosely coupled components that can run on separate virtual machines and communicate with each other over an event bus and through a database. The scheduling algorithm matches capabilities required by the tasks (e.g. software, CPU power, main memory, graphics processing unit) with those offered by the available virtual machines and assigns them accordingly for processing. Our approach utilises heuristics to distribute the tasks evenly in the Cloud. This reduces the overall run time of workflows and makes efficient use of available resources. Our scheduling algorithm also implements optimisations to achieve a high scalability. We perform a thorough evaluation based on four experiments and test if our approach meets the challenges mentioned above. The paper finishes with a discussion, conclusions, and future research opportunities. An implementation of our algorithm and software architecture is publicly available with the open-source workflow management system "Steep".

Keywords: Scientific Workflow Management Systems · Cloud Computing · Distributed Systems · Task Scheduling.

1 Introduction

With the growing amount of global data, it becomes more and more necessary to automate data processing and analysis. Specialised task automation systems are used in areas such as Bioinformatics [31], Geology [18], Geoinformatics [25], and Astronomy [5] to transform data and to extract or derive knowledge. A special kind of those task automation systems are scientific workflow management

systems. They focus on data-driven scientific workflows, which are typically represented by directed acyclic graphs that describe in what order processing tasks need to be applied to a given input data set to produce a desired outcome. Scientific workflows can become very large with *hundreds up to several thousands of tasks* processing data volumes ranging from gigabytes to terabytes.

Modern scientific workflow management systems operate in a distributed manner. They can utilize resources of computing infrastructures such as the Grid [14] or the Cloud [29] to horizontally scale out. This not only speeds up workflow execution but also allows data sets of arbitrary size exceeding the storage capabilities of single computers (Big Data) to be processed. To accomplish this, distributed infrastructures combine the computational power and storage resources of a large number of independent machines. This imposes a challenge on scientific workflow management systems: *How can workflow tasks be assigned to these machines in a smart way to make best use of available resources?*

The general task scheduling problem is known to be NP-complete [38,23] and of high interest to the research community. Several approaches with varying aims and requirements have been published to find practical solutions for the Grid and the Cloud [20,34]. In this paper, we present a *distributed task scheduling algorithm* and a *corresponding software architecture* for a scientific workflow management system that specifically targets the Cloud. The main challenges that need to be covered in this respect are, on the one hand, that machines are highly heterogeneous in terms of hardware, number of virtual CPUs, main memory, and available storage, but also with regard to installed software, drivers, and operating systems. On the other hand, the different tasks in a scientific workflow also have requirements. A compute-intensive task might need a minimum number of CPUs or even a graphics processing unit (GPU), whereas another task might require a large amount of main memory, and a third one needs a specific software to be installed. In other words, *machines have certain capabilities* and *tasks have requirements regarding these capabilities* (or *required capabilities*). This has to be considered during task scheduling. As shown in Section 2, this concept has not been fully covered by existing approaches yet.

In addition to the heterogeneity of machines, the topology of a Cloud is *highly dynamic*. New compute and storage resources can be added on demand and removed at any time. This property is often used to scale a distributed application up when needed (e.g. to manage peak load or to speed up processing) and later down again to save resources and, in consequence, money. Of course, scaling up only makes sense if work can actually be distributed, which is typically the case when a workflow is very large and contains many tasks that could potentially be executed in parallel.

As mentioned above, workflow tasks should be assigned to machines in a smart way in order to optimise resource usage, reduce the total time it takes to complete a workflow, and, in consequence, save money by freeing up resources as soon as possible. However, in a distributed environment whose topology changes dynamically and that is used by multiple tenants at the same time, it is impossible to calculate an optimal task-to-machine mapping in advance to achieve the

"perfect" run time. Instead, task scheduling has to be performed during workflow execution and needs to be able to adapt dynamically to changing conditions.

It is further known that in a distributed environment (and in a Cloud in particular), *faults* such as crashed machines, network timeouts, or missing messages can happen at any time [10]. This highly affects the execution of scientific workflows, which often take several hours or even days to complete.

1.1 Challenges and requirements

To summarise the above, a scientific workflow management system running in the Cloud has to deal with at least the following major challenges:

Capability-based scheduling
Workflow tasks require different capabilities from the machines but, in contrast, the infrastructure is highly heterogeneous.

Dynamic environment
The execution environment is highly dynamic and new compute resources can be added and removed on demand.

Scalability
Scientific workflows can become very large and may contain hundreds of thousands of tasks.

Fault tolerance
In a distributed system, faults such as crashes or network errors can occur at any time.

From these challenges, we derive specific requirements that our scheduling algorithm and software architecture should meet:

REQ 1. The algorithm should be able to assign tasks to heterogeneous machines, while matching the capabilities the tasks need with the capabilities the machines provide.

REQ 2. Tasks should be assigned to parallel machines in an optimised manner so that the overall run time of the workflow is reduced.

REQ 3. Our system should not assume a static number of machines. It should horizontally scale the workflow execution to new machines added to the cluster and be able to handle machines being removed (be it because a user or a service destroyed the machine or because of a fault).

REQ 4. If necessary, the execution of workflow tasks that require capabilities currently not available in the cluster should be postponed. The overall workflow execution should not be blocked. The algorithm should continue with the remaining tasks and reschedule the postponed ones as soon as machines with the required capabilities become available.

REQ 5. The system should support rapid elasticity. This means it should automatically trigger the acquisition of new machines on demand (e.g. during peak load or when capabilities are missing).

REQ 6. The system should be scalable so it can manage both a large number of tasks as well as a large number of machines.

REQ 7. As faults can happen at any time in a distributed environment, our system should be able to recover from those faults and automatically continue executing workflows.

1.2 Contributions

In the scientific community, dynamically changing environments, very large workflows, and fault tolerance are considered major challenges for modern distributed scientific workflow management systems, but they have not been fully covered by existing approaches yet and therefore offer many research opportunities [11]. In the previous section, we discussed these challenges and added another major one, namely that tasks in a scientific workflow need certain capabilities from the machines but the Cloud is highly heterogeneous.

A system that addresses all four of these challenges needs to be designed from the ground up with them in mind. To the best of our knowledge, none of the existing approaches, algorithms, and systems cover all of them in one design (see also our comparison with related work in Section 2). *In this paper, we present such an algorithm as well as the software architecture of a scientific workflow management system in which the algorithm is embedded.*

Our scheduling algorithm is able to assign workflow tasks to heterogeneous machines in the Cloud based on required capability sets. The software architecture consists of a set of components that communicate with each other through an event bus and a database to perform task scheduling in an efficient, scalable, and fault-tolerant manner.

A full implementation of our approach is publicly available with the *Steep Workflow Management System*, which has been released under an open-source licence on GitHub: https://steep-wms.github.io/

1.3 Differences to the conference paper

This paper is a significant extension of our conference paper presented at DATA 2020 [26]. In the previous work, we introduced a first version of our algorithm and software architecture. In the meantime, we were able to explore new research aspects and, as a result of this, to significantly improve our approach. In summary, the extended paper covers the following additional topics:

- We *improved our scheduling algorithm to use heuristics* (Section 5.3) in order to distribute tasks more evenly to machines. Our old approach did not fully use available resources. We therefore added a new requirement regarding optimised allocation of tasks to machines (REQ 2). Our new approach *significantly reduces the overall run time of workflows.*
- In addition, we implemented several optimisations to improve the scalability of our approach, not only in terms of amount of work it can handle but also *to support thousands of machines running in parallel* (Sections 5.4 and 5.5).
- We conducted a *completely new evaluation* to test our new approach and to show how it compares with our old one (Section 7). The evaluation now also *measures the performance of our scheduling algorithm* (Section 7.2).
- We improved our software architecture so *multiple agents can be started on a single virtual machine* (Section 4). We make use of this new feature in our scalability experiment.
- Section 4.1 now describes *virtual machine setups*, which are an integral part of our architecture to create virtual machines with given capabilities.
- We added more details about our scheduling algorithm (Section 5) and an illustrative example (Section 6).

1.4 Structure of the paper

The remainder of this paper is structured as follows. We first analyse the state of the art in Section 2. Then, we introduce an approach to map scientific workflow graphs dynamically to individual *process chains* (i.e. linear sequences of workflow tasks), which can be treated independently by our scheduling algorithm (Section 3). We describe the software architecture in Section 4 and finally our main contribution, the scheduling algorithm, in Section 5. An illustrative example in Section 6 demonstrates how our system works in practise. We also present the results of four experiments we conducted to evaluate if our approach meets the challenges and requirements defined above (Section 7). We finish the paper in Section 8 with conclusions and future research opportunities.

2 Related Work

There are various algorithms performing task scheduling. Their aims vary from each other but most of them try to optimise resource usage and to reduce the *makespan*, i.e. the time passed between the start of the first task in a sequence and the end of the last one. For this, they implement heuristics. Min-Min and

Max-Min [22,15], for example, iterate through all tasks in the sequence and calculate their earliest completion time on all machines. Min-Min schedules the task with the minimum earliest completion time while Max-Min selects the task with the maximum one. This process continues until all tasks have been processed.

In contrast, the Sufferage algorithm reassigns a task from machine M to another one if there is a second task that would achieve better performance on M [28]. As an extension to this approach, Casanova et al. present a heuristic called XSufferage, which also considers data transfer costs [9]. The authors claim their approach leads to a shorter makespan because of possible file reuse. Gherega and Pupezescu improve this algorithm even further and present DXSufferage, which is based on the multi-agent paradigm [16]. Their approach prevents the heuristic itself from becoming a bottleneck in the scheduling process. For increased flexibility, Nayak and Padhi present an approach that first analyses all tasks to be scheduled and then, based on the current situation, selects from different heuristics to achieve the best performance [30].

Our approach is also based on heuristics. Since it is very hard to analyse tasks and to predict their earliest completion time on heterogeneous virtual machines in a dynamic environment like the Cloud, our approach uses the remaining number of workflow tasks for a certain set of required capabilities to evenly distribute work to machines and to adapt to changing conditions during run time.

Other dynamic approaches are based on genetic algorithms (GA), which mimics the process of natural evolution by using historical information. A GA selects the best mapping of tasks to machines. Good results with this type of algorithms were achieved by Hamad and Omara who use Tournament Selection [19] or by Page and Naughton whose algorithm does not make assumptions about the characteristics of tasks or machines [32].

Applying behaviour known from nature to task scheduling is an idea that has lead to other noteworthy approaches: Ant colony optimisation [35,27] tries to dynamically adapt scheduling strategies to changing environments. Thennarasu et al. present a scheduler that mimics the behaviour of humpback whales to maximize work completion and to meet deadline and budget constraints [36].

The algorithms mentioned above can be used to schedule individual tasks. In contrast, there are more complex approaches that consider the interdependencies in the directed acyclic graphs of scientific workflows. Blythe at al. investigate the difference between task-based and workflow-based approaches [7]. They conclude that data-intensive applications benefit from workflow-based approaches because the workflow system can start to transfer data before it is used by the tasks, which leads to optimised resource usage. Binato et al. present such a workflow-based approach using a greedy randomized adaptive search procedure (GRASP) [6]. Their algorithm creates multiple scheduling solutions iteratively and then selects the one that is expected to perform best. Topcuoglu et al. present two algorithms: HEFT and CPOP [37]. HEFT traverses the complete workflow graph and calculates priorities for individual tasks based on the number of successors, average communication costs, and average computation costs. CPOP extends this and prioritises critical paths in workflow graphs.

Scientific workflow management systems such as *Pegasus* [12], *Kepler* [1], *Taverna* [21], *Galaxy* [17], *Argo* [3], *Airflow* [2], and *Nextflow* [13] typically implement one or more of the algorithms mentioned above. There are other frameworks to process large data sets in the Cloud. Most noteworthy systems are Spark [39] and Flink [8]. They are not workflow management systems but follow a similar approach and also need to schedule tasks from a directed graph.

There are some similarities between our approach and existing ones. DX-Sufferage, for example, uses the multi-agent paradigm [16]. Similar to agents, our components are independent and communicate with each other through an event bus. There can be multiple schedulers sharing work and processing the same workflow. In addition, we convert workflow graphs to process chains, which group tasks with the same required capabilities and common input/output data. Just like in XSufferage [9], this can potentially lead to better file reuse.

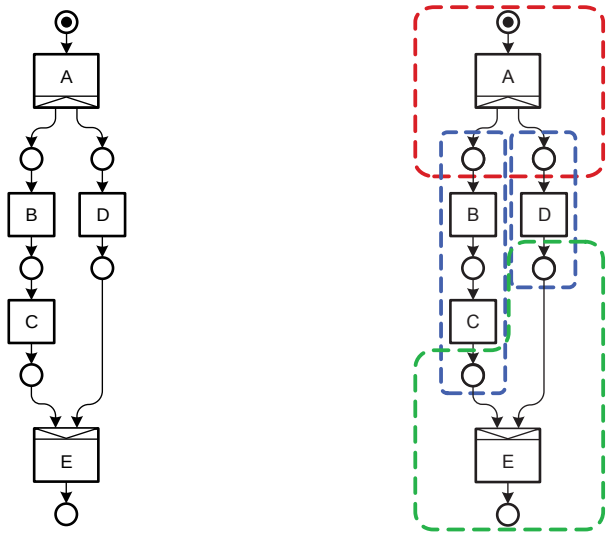
Note that our approach is not directly comparable to workflow-based scheduling algorithms that consider the graph in total. Instead, we employ a hybrid strategy that first splits the graph into process chains and then schedules these instead of individual tasks.

3 Traversing scientific workflow graphs

Figure 1a shows an example of a scientific workflow represented by a directed graph. An input file (the circle with the dot) is first consumed by task A, which produces two output files. These files are then processed independently (and possibly in parallel) by tasks B and D. The result of B is further transformed by task C. The results of C and D are consumed by a task E, which produces the final outcome of the workflow. The figure uses the extended Petri Net notation proposed by van der Aalst and van Hee (2004).

Our scientific workflow management system transforms such workflow graphs into individual executable units called *process chains*. These are linear sequential lists of tasks (without branches and loops) that can be scheduled independently on virtual machines in the Cloud. In order to find process chains, our system traverses the graph and looks for tasks that require the same capabilities from the machines. On each junction (i.e. when a task creates more outputs than it consumes inputs; or the other way around), the system creates a new process chain. For the example in Figure 1a, the system creates a process chain for task A, then two chains (one containing B and C, and another one containing only D), and a final one for task E. The chains with B/C and D can be assigned to separate virtual machines and executed in parallel according to our algorithm presented in Section 5.

In our implementation, capabilities are user-defined strings. For example, the set $\{Ubuntu, GPU\}$ might mean that a task depends on the Linux distribution Ubuntu as well as the presence of a graphics processing unit. In the following, we call the union of the required capabilities of all tasks in a process chain a *required capability set*.



(a) Workflow graph (b) Generated process chains

Fig. 1: A workflow is split into four individual process chains [26].

4 Software architecture

Our scientific workflow management system consists of four main components: the HTTP server, the controller, the scheduler, the agent, and the cloud manager (see Figure 2). Typically, one instance of our system will be deployed to exactly one virtual machine in the Cloud. If necessary, it is possible to run multiple instances on the same machine.

Each component can be enabled or disabled in a given instance. In a cluster, there can be one primary instance, for example, that has only the controller and scheduler enabled, and multiple secondary instances each running one agent. In addition to that, the agent can be spawned more than once inside a single instance. This allows this instance to run multiple workflow tasks in parallel and to make best use of available resources (for example, if each workflow task only requires one CPU core or a limited amount of memory).

The system contains an event bus that is used by all components of all instances to communicate with each other. Moreover, the HTTP server, the controller, and the scheduler are connected to a shared database where they manage workflows and process chains. In the following, we describe the roles and responsibilities of each component.

The HTTP server is the main entry point to our system. It provides information about scheduled, running, and finished workflows to clients. If the HTTP server receives a new workflow from a client, it stores the workflow in the database and sends a message to one of the instances of the controller.

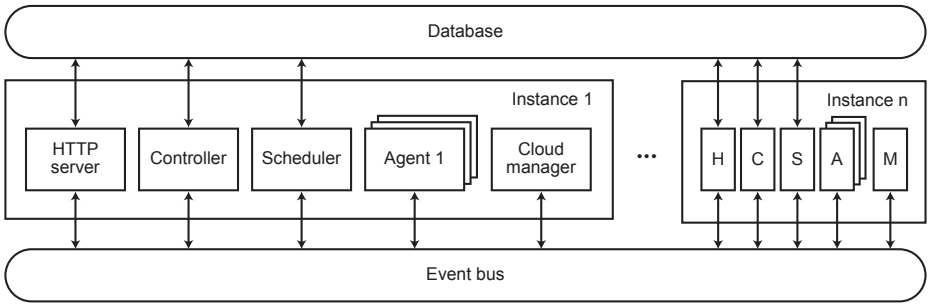


Fig. 2: An overview of the components in our scientific workflow management system and how they communicate with each other.

The controller receives this message, loads the workflow from the database, and starts transforming it iteratively to process chains as described in Section 3. Whenever it has generated new process chains, it puts them into the database and sends a message to all instances of the scheduler.

The schedulers then apply our algorithm (see Section 5) and select agents to execute the process chains. They load the process chains from the database, send them via the event bus to the selected agents for execution. Upon completion, they write the results into the database. The schedulers also send a message back to the controller so it can continue with the next iteration and generate more process chains until the workflow has been completely transformed.

In case a scheduler does not find an agent suitable for the execution of a process chain, it sends a message to the cloud manager. This component interacts with the API of the Cloud infrastructure, creates new virtual machines on demand, and deploys agents to them. This is based on so-called *virtual machine setups*, which are described in Section 4.1.

Note that messages between the HTTP server, the controller, and scheduler may get lost (e.g. because of network failures). Due to this, the controller and the scheduler also check the database for new workflows and process chains respectively at a regular interval. We found 20 seconds to be a sensible value in practise, but in our implementation, this is configurable. This approach decouples the components from each other and increases fault tolerance.

4.1 Virtual machine setups

The cloud manager component creates virtual machines (VMs) on demand and deploys software to it including at least one instance of our workflow management system with one or more enabled agents. The process of deploying software is called *provisioning*. The kind of VM to create as well as the actual software to be deployed depend on the capabilities that the process chains to be executed on this VM require. For example, for a process chain that needs a graphics

processing unit (GPU), the cloud manager will create a VM with such a device and deploy the necessary drivers to it.

The behaviour of the cloud manager is configurable. The mapping between required capabilities and VM types (sometimes called *flavors* or *instance types*; depending on the Cloud provider) is specified in a configuration file, and the software is deployed by provisioning templates. These templates are shell scripts that the cloud manager executes on the virtual machines right after they have been created. The last step in each set of provisioning templates always starts an instance of our system, so the cloud manager knows when the provisioning process has completed and the new agents can be used for scheduling.

Each set of provisioning templates and the corresponding mapping from required capabilities to VM types is called a *virtual machine setup* in our system.

5 Capability-based scheduling algorithm

This section introduces the capability-based scheduling algorithm that is executed in our scheduler component. First, the main scheduling function (Section 5.1) is described as well as how our algorithm selects candidate agents (Section 5.2) based on heuristics (Section 5.3). After that, database queries (Section 5.4) and optimisations for improved scalability (Section 5.5) are discussed.

5.1 Main scheduling function

As mentioned above, the scheduler runs at regular intervals and immediately after new process chains have been added to the database. Listing 1.1 shows the main function of our algorithm that assigns process chains to agents.

Our algorithm first calls the function *findRequiredCapabilitySets()*, which performs a database query to retrieve all distinct sets of capabilities required to execute the process chains not scheduled yet. In other words, given a capability set $R_i = \{c_1, ..., c_n\}$ for a process chain pc_i, the result of *findRequiredCapabilitySets()* is a set $S = \{R_1, ..., R_m\}$ of distinct required capability sets.

From line 3 on, our algorithm performs up to *maxLookups* scheduling operations. After the regular interval or when new process chains have been added, the function will be called with *maxLookups* set to infinity. The main idea is that the algorithm will try to schedule as many process chains as possible until it reaches a *break* statement. There is only one of these statements in line 12. It is reached when all agents indicate they are not available (see details below).

Inside the main for loop, the function first selects a set of candidate agents that are able to execute at least one of the given required capability sets from S (line 4) by calling the function *selectCandidates()*. This function is described in detail in Section 5.2. In short, it returns a list of pairs of a candidate agent and the required capability set R it can execute.

If this list is empty (line 5), all agents are currently busy or there is no agent that would be able to execute at least one $R \in S$ (i.e. none of them is available). In this case, the function iterates over all required capability sets (line 8) and

```
1  function lookup(maxLookups):
2    S = findRequiredCapabilitySets()

3    for i ∈ [0, maxLookups):
4      candidates = selectCandidates(S)

5      if candidates == ∅:
6        /* All agents are busy or none of them
7           have the required capabilities. */
8        for R ∈ S:
9          if existsProcessChain(R):
10           launch:
11             requestAgent(R)
12       break

13     for (candidate, R) ∈ candidates:
14       pc = findProcessChain(R)
15       if pc == undefined:
16         /* All process chains with R were
17            executed in the meantime. */
18         continue

19       agent = allocate(candidate)
20       if agent == undefined:
21         /* Agent is not available any more. */
22         continue

23       /* Execute process chain
24          asynchronously. */
25       launch:
26         executeProcessChain(pc, agent)
27         deallocate(agent)

28         /* Agent is has become available.
29            Trigger next lookup. */
30         lookup(1)
```

Listing 1.1: The main function of our algorithm checks what capabilities are required at the moment and if there are available agents that can execute process chains with these capabilities. If so, it retrieves such process chains from the database and schedules their execution [26].

checks if there actually is a corresponding process chain in the database (line 9). This is necessary because all process chains with a certain required capability set may have already been processed since *findRequiredCapabilitySets()* was called (e.g. by another scheduler instance or in a preceding iteration of the outer for loop). If there is a process chain, the function *requestAgent* will be called, which asks the cloud manager component (see Section 4) to create a new VM with

an agent that has the given required capabilities (line 11). We use the keyword *launch* here to indicate that the call to *requestAgent* is asynchronous, meaning the algorithm does not wait for an answer.

The algorithm then leaves the outer for loop because it is unnecessary to perform any more scheduling operations while none of the agents can execute process chains (line 12). Process chains with required capabilities none of the agents can provide will essentially be postponed. As soon as the cloud manager has created a new agent with the missing capabilities, the *lookup* function will be called again and any postponed process chains can be scheduled.

If there are agents available that can execute process chains with any of the required capability sets from S, the algorithm iterates over the result of *selectCandidates()* in line 13. For each pair of a candidate agent and the corresponding required capability set R it can execute, the algorithm tries to find a matching registered process chain with R in the database. If there is none, it assumes that all process chains with this required capability set have already been executed in the meantime (line 15). Otherwise, it tries to allocate the candidate agent, which means it asks it to prepare itself for the execution of a process chain and to not accept other requests anymore (line 19). If the agent cannot be allocated, it was probably allocated by another scheduler instance in the meantime since *selectCandidates* was called (line 20).

Otherwise, the algorithm launches the execution of the process chain in the background and continues with the next scheduling operation. The code block from line 25 to line 30 runs asynchronously in a separate thread and does not block the outer for loop. As soon as the process chain has been executed completely in this thread, our algorithm deallocates the agent in line 27 so it becomes available again. It then calls the *lookup* function and passes 1 for *maxLookups* because exactly one agent has become available and therefore only one process chain has to be scheduled.

5.2 Selecting candidate agents

The function *selectCandidates* takes a set $S = \{R_1, ..., R_n\}$ of required capability sets and returns a list $L = \{P_1, ..., P_m\}$ of pairs $P = (a, R_i)$ of an agent a and matching required capability set R_i. Listing 1.2 shows the pseudo code.

The function uses the event bus to send all required capability sets to each agent. The agents analyse the required capability sets based on defined heuristics (see Section 5.3) and then reply whether they are available and which set they support best. The function collects all responses in a set of *candidates*. Finally, it selects exactly one agent for each required capability set.

Note that some or all agents might not be available, in which case the result of *selectCandidates* contains less required capability sets than S or is even empty.

5.3 Scheduling heuristics

An agent receives required capability sets from the scheduler and matches them against the capabilities it actually has. For example, let us assume one of these

```
1  function selectCandidates(S):
2    candidates = ∅

3    for a ∈ Agents:
4      send S to a and wait for response
5      if a is available:
6        get best R_i ∈ S from response
7        P = (a, R_i)
8        candidates = candidates ∪ {P}

9    L = all P∈candidates with best a for each R_i

10   return L
```

Listing 1.2: Pseudo code of the function that selects agents based on their capabilities [26].

sets is $\{Ubuntu,\ GPU\}$ (as described in Section 3, capabilities are user-defined strings). The agent will only consider this set if it actually runs on Ubuntu and has a GPU (as specified in its VM setup; see Section 4.1).

After the agent has selected all required capability sets it generally supports, it chooses one set for which it would like to receive process chains to execute. In Listing 1.2, this is called the "best" $R_i \in S$.

In our previous work, we only had one heuristic that selected the best agent for a certain required capability set based on the longest idle time [26]. In practise, this has proven to achieve good throughput and, at the same time, to prevent starvation because every agent was selected eventually.

Our new approach extends the existing heuristic. When selecting the best capability set, the agent now also considers the number of remaining process chains in the database for each $R_i \in S$. If two or more capability sets are supported similarly well, the agent selects the one with the highest number of remaining process chains. This makes sure process chains are distributed more evenly to agents supporting similar capabilities (see our evaluation results in Section 7).

5.4 Caching database queries

The function *findRequiredCapabilitySets* performs a database query to look for distinct required capability sets. Queries for distinct values are complex and can take quite some time, especially for very large collections. The only way to find all distinct values is to perform a sequential scan on the collection. Most DBMS even have to sort the collection first. There are approaches to find approximations of distinct values [24,4] but our algorithm needs exact results.

Such a query should not be performed too often as it can drastically impact the throughout of the scheduler. As an optimisation, we use a cache to keep distinct required capability sets in memory until the next regular scheduling in-

terval. This can lead to inconsistencies if multiple schedulers access the database and their caches become outdated. In the worst case, two things can happen:

- The cache may still contain required capability sets of process chains that have already been executed. In this case, the algorithm will not find a matching process chain in the database (line 15) and just continue with the next candidate agent and required capability set. This means, there may be unnecessary queries per scheduling operation, but these queries are negligibly fast because they can be implemented with simple SELECT-WHERE statements that make use of an index.
- One or more required capability sets may be missing from the cache. This can only happen if new process chains are added to the database while a scheduling operation is currently running. Adding new process chains will, however, trigger a cache update, so the next scheduling operation will use all required capability sets.

In any case, the cache will be updated at the regular interval, which will eliminate unnecessary queries and trigger the scheduling of remaining process chains.

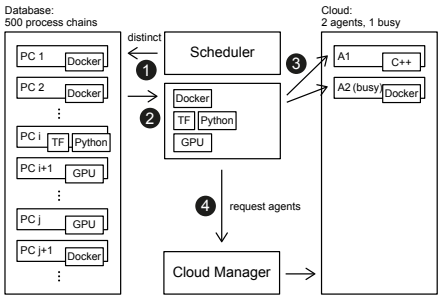
5.5 Optimisations for improved scalability

Another possible bottleneck of our algorithm besides database queries is the *selectCandidates* function, which sends required capability sets to all agents. Even though all capability sets can be sent together in one message, in a setup with n process chains and m agents, $n \times m$ messages need to be sent per workflow execution. Since these messages are sent over the event bus and possibly through a slow network, the execution of *selectCandidates* can significantly affect the time it takes to schedule a single process chain and, as a consequence, the run time of the whole workflow.

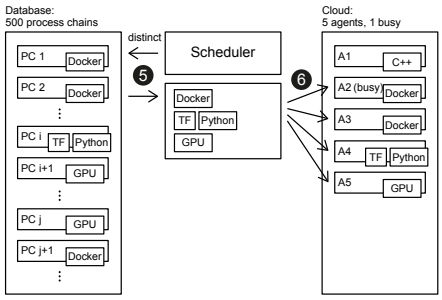
In addition to caching database queries, we therefore implemented two other optimisations. First, each scheduler instance caches the capabilities supported by the individual agents. This allows it to skip those agents that definitely do not support a certain required capability set. Second, the scheduler also keeps track of which agents are currently executing process chains (i.e. to which of them it has sent a process chain and has not received a result yet). Those agents can also be skipped as they would respond that they are unavailable anyhow. In the best case, when all agents are busy except one, the scheduler only needs to ask this agent and can skip the others.

6 Illustrative example

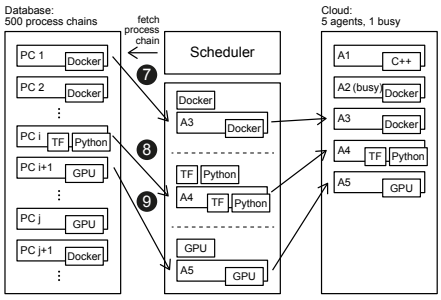
Figure 3 shows an example of how our scheduling algorithm works in practise. Note that it represents one specific case (other control flows are possible) but covers almost all aspects of our algorithm and architecture.



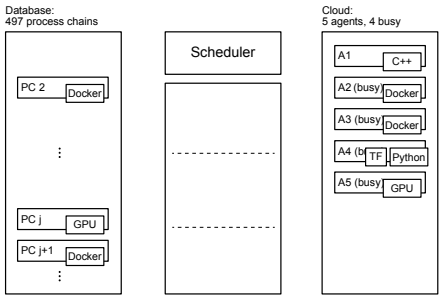
(a) The scheduler accesses the database, the agents, and the cloud manager.



(b) The second scheduling step utilises the newly created agents.



(c) Process chains are assigned to the new agents one by one.



(d) At the end, the new agents are busy executing process chains.

Fig. 3: Illustrative scheduling example

At the beginning, there are 500 process chains in the database to be scheduled. Some of them require "Docker" to be installed on the virtual machine on which they will be executed, others require "TF" (TensorFlow) and "Python", and a third group requires the VM to have a graphics processing unit ("GPU"). Two agents are currently running in the Cloud. One supports "C++" applications and the other one has "Docker" installed but is currently busy.

In the first scheduling step (Figure 3a), the scheduler fetches distinct required capability sets from the database ❶. It gets $\{Docker\}$, $\{TF, Python\}$, and $\{GPU\}$ ❷. The scheduler then sends these sets to all agents ❸. Agent $A1$ responds that it does not support any of these capabilities, and agent $A2$ indicates that it is currently busy. Since there are no agents available, the scheduler sends the required capabilities to the cloud manager and tells it to create new VMs ❹. The first scheduling step stops at this point.

After the cloud manager has created new VMs and deployed agents to it (Figure 3b), the next scheduling step starts. The scheduler fetches distinct required capability sets from the database again ❺ and sends them to all agents ❻ (except $A1$ because it knows it does not support the capabilities). This time, the

new agents $A3$, $A4$, and $A5$ respond that they are available and which capability sets they support.

Once the scheduler has received all answers (Figure 3c), it fetches a process chain for the capability set $\{Docker\}$ from the database and assigns it to agent $A3$ ❼. This process repeats for $\{TF, Python\}$ and $\{GPU\}$, which are assigned to $A4$ and $A5$, respectively ❽❾.

At the end of the second scheduling step (Figure 3d), all new agents are busy. As soon as one of them becomes available again, a next step will be triggered to schedule the remaining process chains in the database.

7 Evaluation

In order to evaluate if our scheduling algorithm and our software architecture meet the challenges and requirements defined in Section 1.1, we conducted four practical experiments (one for each challenge). This section presents the results of these experiments and discusses benefits and drawbacks of our approach.

All experiments were performed in the same environment (a private OpenStack Cloud). We set up our system so that it had access to the API of the OpenStack Cloud and was able to create virtual machines on demand. We deployed the full stack of components presented in Section 4 to each virtual machine. All components communicated with each other through a distributed event bus. We also deployed a MongoDB database on a separate virtual machine to which the components had access.

We defined four VM setups for virtual machines and agents with the capability sets $R1$, $R2$, $R3$, and $R4$ respectively, as well as fifth one with both capability sets $R3$ and $R4$. To simulate a heterogeneous environment, we configured a maximum number of virtual machines that our system was allowed to create per required capability set. Table 1 shows the settings we chose for the individual experiments. Note that in experiment 1, 2, and 4, we deployed one agent per virtual machine. In experiment 3 where we tested the scalability of our system, we deployed 125 agents per virtual machine resulting in a total number of 1 000 agents.

Table 1: Maximum number of virtual machines configured for each required capability set.

Required capability set	**Maximum number of virtual machines**
$R1$	2
$R2$	2
$R3$	1
$R4$	1
$R3 + R4$	2
Total	**8**

For each experiment, we collected all log files of all instances of our system and converted them to graphs. Figure 4 shows the results of experiments 1 and 2 including a legend. Each sub-figure—which we discuss in detail in the following sections—depicts a timeline of a workflow run. The lanes (from left to right) represent individual agents and indicate when they were busy executing process chains. All required capability sets have different colours (see legend in Figure 4e). The colour of the agents and the process chains specifies what capabilities they offered or required respectively. A process chain has a start (emphasized by a darker shade of the colour) and an end. In experiment 4, we also killed agents on purpose. The point in time when the fault was induced is marked by a black X (see Figure 5b).

7.1 Experiments

Experiment 1: Capability-based scheduling
(Requirements covered: REQ 1–2)

Figure 4a shows the results of our first experiment. One of our goals in this paper was to create a scheduling algorithm that is able to assign workflow tasks to distributed machines based on required and offered capabilities. The results show that this goal was reached.

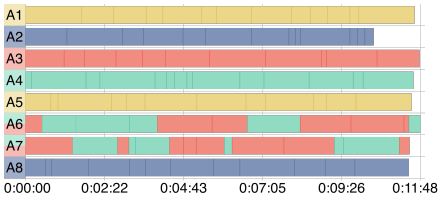
We deployed a static number of eight agents with different capability sets and sent a workflow consisting of 100 process chains to one of the instances of our system. As soon as the workflow was saved in the database, all scheduler instances started assigning process chains to the individual agents. The colours in the figure show that all process chains were correctly assigned. Agents $A6$ and $A7$ were able to execute process chains requiring both $R3$ and $R4$, which is indicated by alternating colours. The workflow took 11 minutes and 48 seconds to complete in total.

Note that these results also show a significant improvement of our algorithm compared to our earlier work [26]. This is mostly due to the new scheduling heuristics presented in Section 5.3. Previously, as shown in Figure 4b, agents $A6$ and $A7$ preferred to accept process chains with $R3$ first before they continued with $R4$. This resulted in an inefficient use of resources. Agent $A4$ was not used completely during the workflow run and too much work was allocated to $A3$, $A6$, and $A7$. The workflow took 13 minutes and 14 seconds. Our improved algorithm is about one and a half minutes faster and distributes work much more evenly.

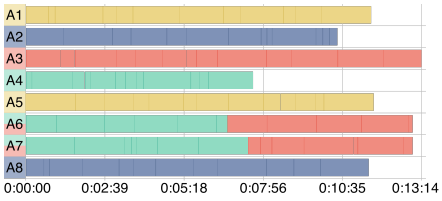
Experiment 2: Dynamic environment
(Requirements covered: REQ 1–5)

Our second experiment started with only one agent supporting capability set $R1$. We executed a workflow with 1 000 process chains. Figure 4c shows the timeline of the workflow run.

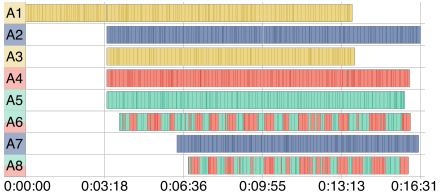
According to our algorithm, the scheduler first looked for available agents to which to assign process chains. Since there was only one agent running, it assigned process chains to it but also asked the cloud manager component to



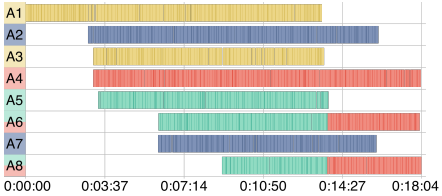
(a) 100 process chains are distributed to agents with the correct capabilities.



(b) Results of experiment 1 from our earlier work for comparison [26]



(c) The system creates agents with capabilities required by 1 000 process chains.



(d) Results of experiment 2 from our earlier work for comparison [26]

(e) Legend: colours for required capability sets, start and end of a process chain, and time when an agent was killed.

Fig. 4: Results of experiments 1 and 2

create new agents for the other capability sets. Starting a virtual machine and deploying itself to it took our system almost three minutes. Process chains requiring missing capabilities were postponed but the scheduler continued assigning the ones with $R1$. As soon as the new agents had started, process chains were assigned to them.

Note that we configured our system to only create one virtual machine of a certain capability set at a time. Also, as described earlier, we limited the number of virtual machines per capability set. These are the reasons why only four new agents appear at about minute 3, one more between minutes 3 and 4, and two other ones around 6:30.

The experiment shows that our system can create new virtual machines on demand and that the schedulers make use of new resources as soon as they become available. Similarly to experiment 1, we compared the results with those from our earlier work [26]. The workflow now took 16 minutes and 31 seconds and was more than one and a half minutes faster than the previous 18 minutes and 4 seconds. Workflow tasks were again distributed much more evenly to the agents and resources were used reasonably.

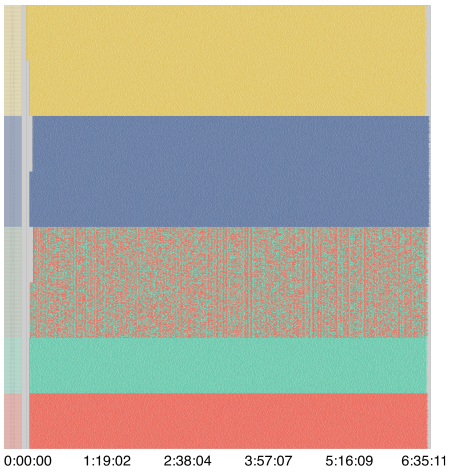
Experiment 3: Scalability

(Requirements covered: REQ 1–6)

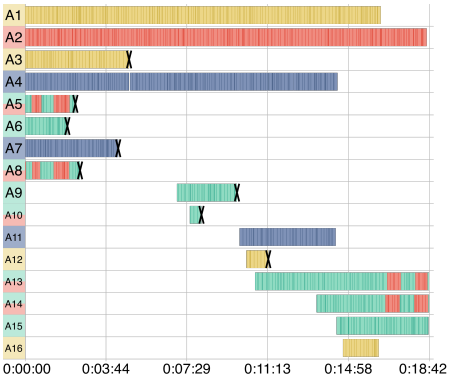
In order to show the scalability of our system, we launched a workflow with 300 000 process chains. Similar to the second experiment, we started with one virtual machine. The other ones were automatically created by our system on demand. However, this time we increased the number of agents per virtual machine to 125 resulting in a total number of 1 000 agents.

Figure 5a shows the timeline of the workflow over more than six and a half hours. Note that we sorted the agents in this graph by capability set for better legibility. Again, all process chains were assigned to the correct machines. Although the number of process chains the system had to manage was very large, it did not crash and kept being responsive the whole time. Also, the system was able to handle the large number of agents without interruptions. There are no gaps in the graph, which means new process chains were scheduled immediately when an agent became available. This shows that our optimisations described in Section 5.5 worked as expected. All agents also finished their work almost at the same time, which means our scheduler distributed the work evenly (based on the heuristics described in Section 5.3).

In our previous work, we performed a similar experiment but with only 150 000 process chains and 8 agents [26]. At that time, our system was not able to manage 1 000 agents. Our improved approach is much more scalable now.



(a) The system is able to handle 300 000 process chains running on 1 000 agents (sorted by capability set for better legibility).



(b) The system is able to recover from faults and to still finish all 1 000 process chains from the current workflow.

Fig. 5: Results of experiments 3 and 4

Experiment 4: Fault tolerance

(Requirements covered: all)

In our final experiment, we tested if our system can manage faults during a workflow run. Figure 5b shows the timeline. We started with eight agents and executed the same workflow as in experiment 2 with 1 000 process chains. Between minutes 2 and 3, we started to randomly kill agents by sending them a SIGKILL signal (indicated in the figure by a black X).

We killed eight agents during the workflow run. The figure shows that each time, the system was able to recover from the faults. It created new agents with the missing required capabilities and started assigning process chains to them as soon as they became available. Between minutes 3 and 7, approximately, there was no agent able to execute process chains with $R3$. The execution of these process chains was postponed and resumed later.

We performed the same experiment in our earlier work [26]. The results are quite similar and our improved approach still works as expected.

7.2 Scheduling performance

In order to evaluate the performance of our approach, we measured the time it took to schedule a single process chain (including asking all agents and fetching the process chain from the database). We did this for all experiments and calculated the averages, medians, and standard deviations (see Table 2).

The results indicate that scheduling is in general very fast and does not produce much overhead per process chain. They also show that our system is scalable. Even in experiment 3 where we had to distribute 300 000 process chains to 1 000 agents, the performance was not slower than in experiment 100. It also stayed consistent throughout the whole workflow run. This is mostly due to the optimisations described in Sections 5.4 and 5.5. Note that the performance in experiment 3 was actually slightly better than in experiment 1 (on average by about 1–1.5 ms), which is most likely due to the fact that our implementation runs on the Java Virtual Machine, and the longer it runs, the better the just-in-time compiler can optimise the code.

Table 2: Measured performance of the scheduler component for each experiment (values are per process chain)

Experiment	**Average**	**Median**	**Std. deviation**
1	16.3 ms	16 ms	2.1 ms
2	16.8 ms	16 ms	4.9 ms
3	15.1 ms	15 ms	3.9 ms
4	16.5 ms	16 ms	6.9 ms

7.3 Discussion

The results of our experiments show that our system meets all of the challenges and requirements for the management of scientific workflows in the Cloud defined in Section 1.1.

In order to assign process chains to the correct machines with matching capabilities, our scheduler asks each agent whether it wants to execute a process chain with a given required capability set or not. An alternative approach would be to let the agents fetch the process chains themselves whenever they are ready to execute something. However, in this case, it would not be possible to create agents on demand. If there is no agent fetching process chains, nothing can be executed. Our scheduler, on the other hand, has an overview of all required capability sets and can acquire new resources when necessary.

In our previous work [26], the scheduler only chose between multiple available agents by comparing their idle time. This was not very efficient and led to unnecessary long workflow runs and unused resources. Our new approach with improved heuristics and caching not only reduces the overall time of workflow runs but also makes the system more scalable. Our previous experiments also revealed small gaps in the workflow run, which could be traced back to the unoptimised scheduling algorithm. We did not observe these gaps anymore with the new approach.

According to our evaluation results, the heuristics presented in Section 5.3 are quite effective. Nevertheless, more sophisticated approaches also considering the expected run time of individual process chains [22,15] or even the dependencies between process chains in the workflow graph [6,37] are conceivable. However, in a distributed environment, it is very hard to predict the run time of a single task without additional knowledge, so this topic remains for future work.

The fact that we use a database to store process chains has many benefits. First, this out-of-core approach reduces the number of process chains that need to be kept in memory at a time. This allows the scheduler to process hundreds of thousands of process chains without any issue. Second, multiple scheduler instances can access the database and work in parallel on different virtual machines. Due to this, the total time it takes to schedule all process chains of a workflow can be reduced and the values presented in Section 7.2 become negligible compared to the whole workflow run. Lastly, since the database holds the remaining process chains to execute, it essentially keeps the current state of the overall workflow execution. This enables fault tolerance: if one scheduler instance crashes, another one can take over. Our open-source implementation even supports resuming workflows if all schedulers have crashed after a restart of the whole cluster.

Nevertheless, the database can be considered a single point of failure. If it becomes unavailable, workflow execution cannot continue. In practise, this is, however, not a problem because as soon as it is up again, our scheduling algorithm can proceed and no information will be lost.

There are still places where our system could be improved. At the moment, our cloud manager creates only one virtual machine per capability set at a time.

This could be parallelised in the future to further reduce the overall run time of workflows. However, this requires a clever heuristic so that the cloud manager does not create more virtual machines than actually necessary. This heuristic should be based on the number of remaining process chains in the database for a given capability set, which can change while the virtual machines are being created. This optimisation remains for future work.

8 Conclusion

Distributed scientific workflow management systems running in the Cloud face the challenges of capability-based scheduling, a dynamic environment, scalability, and fault tolerance (see Section 1.1). In this paper, we presented a software architecture and a scheduling algorithm addressing these challenges. Our work contributes to the scientific community as the challenges have not been fully addressed in literature yet.

The *Steep Workflow Management System*, which has been released under an open-source licence, implements our approach. Steep is used in various projects. One of them, for example, deals with the processing of large point clouds and panorama images that have been acquired with a mobile mapping system in urban environments. The data often covers whole cities, which makes the workflows particularly large with thousands of process chains. The point clouds are processed by a service using artificial intelligence (AI) to classify points and to detect façades, street surfaces, etc. Since this is a time-consuming task and the workflows often take several days to execute, the scalability and fault tolerance of Steep are fundamental in this project.

The AI service requires a GPU, which is a limited resource in the Cloud and particularly expensive. The capability-based scheduling algorithm we proposed in this paper helps in this respect by distributing workflow tasks to the correct machines. As our system supports elasticity and a dynamic number of machines, it can scale up and down on demand. It only creates GPU machines when needed and releases them as soon as possible. In other words, in this project, our approach saves time and money.

There is a range of opportunities for future research. For example, the approach to transform workflow graphs to process chains introduced in Section 3 is implemented in an iterative way and allows very complex workflows to be processed. It supports workflows *without a priori design-time knowledge* [33], which means the system does not need to know the complete workflow structure before the execution starts. As the number of instances of a process chain can depend on the results of a preceding one, this property allows the system to dynamically change the workflow structure during run time. This approach is also suitable to execute workflows *without a priori run-time knowledge*, meaning that the number of instances of a certain process chain may even change *while* the process chain is running. This enables cycles and recursion. Details on this will be the subject of a future publication on which we are currently working.

References

1. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. pp. 423–424. IEEE (2004)
2. Apache Airflow: Apache Airflow Website. https://airflow.apache.org/ (2020), last accessed: 2020-04-14
3. Argo Workflows: Argo Website. https://argoproj.github.io/ (2020), last accessed: 2020-10-21
4. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L.: Counting distinct elements in a data stream. In: Rolim, J.D.P., Vadhan, S. (eds.) Randomization and Approximation Techniques in Computer Science. pp. 1–10. Springer Berlin Heidelberg (2002)
5. Berriman, G.B., Deelman, E., Good, J.C., Jacob, J.C., Katz, D.S., Kesselman, C., Laity, A.C., Prince, T.A., Singh, G., Su, M.H.: Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In: Optimizing Scientific Return for Astronomy through Information Technologies. vol. 5493, pp. 221–233. International Society for Optics and Photonics (2004)
6. Binato, S., Hery, W.J., Loewenstern, D.M., Resende, M.G.C.: A Grasp for Job Shop Scheduling, pp. 59–79. Springer US (2002). https://doi.org/10.1007/978-1-4615-1507-4_3
7. Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., Kennedy, K.: Task scheduling strategies for workflow-based applications in grids. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid). pp. 759–767 (2005). https://doi.org/10.1109/CCGRID.2005.1558639
8. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4), 28–38 (2015)
9. Casanova, H., Legrand, A., Zagorodnov, D., Berman, F.: Heuristics for scheduling parameter sweep applications in grid environments. In: Proceedings of the 9th Heterogeneous Computing Workshop HCW. pp. 349–363 (2000). https://doi.org/10.1109/HCW.2000.843757
10. Chircu, V.: Understanding the 8 fallacies of distributed systems. https://dzone.com/articles/understanding-the-8-fallacies-of-distributed-syste (2018), last accessed: 2020-02-18
11. Deelman, E., Peterka, T., Altintas, I., Carothers, C.D., van Dam, K.K., Moreland, K., Parashar, M., Ramakrishnan, L., Taufer, M., Vetter, J.: The future of scientific workflows. The International Journal of High Performance Computing Applications **32**(1), 159–175 (2018)
12. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., Wenger, K.: Pegasus: a workflow management system for science automation. Future Generation Computer Systems **46**, 17–35 (2015). https://doi.org/10.1016/j.future.2014.10.008
13. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P.P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nature biotechnology **35**(4), 316–319 (2017). https://doi.org/10.1038/nbt.3820
14. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)

15. Freund, R.F., Gherrity, M., Ambrosius, S., Campbell, M., Halderman, M., Hensgen, D.Z., Keith, E., Kidd, T., Kussow, M., Lima, J.D., Mirabile, F., Lantz, M., Rust, B., Siegel, H.J.: Scheduling resources in multi-user heterogeneous computing environments with SmartNet. Calhoun: The NPS Institutional Archive (1998)
16. Gherega, A., Pupezescu, V.: Multi-agent resource allocation algorithm based on the xsufferage heuristic for distributed systems. In: Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 313–320 (2011). https://doi.org/10.1109/SYNASC.2011.37
17. Giardine, B., Riemer, C., Hardison, R.C., Burhans, R., Elnitski, L., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Taylor, J., Miller, W., Kent, W.J., Nekrutenko, A.: Galaxy: a platform for interactive large-scale genome analysis. Genome research **15**(10), 1451–1455 (2005). https://doi.org/10.1101/gr.4086505
18. Graves, R., Jordan, T.H., Callaghan, S., Deelman, E., Field, E., Juve, G., Kesselman, C., Maechling, P., Mehta, G., Milner, K., Okaya, D., Small, P., Vahi, K.: Cybershake: A physics-based seismic hazard model for southern california. Pure and Applied Geophysics **168**(3), 367–381 (2011). https://doi.org/10.1007/s00024-010-0161-6
19. Hamad, S.A., Omara, F.A.: Genetic-based task scheduling algorithm in cloud computing environment. International Journal of Advanced Computer Science and Applications **7**(4), 550–556 (2016). https://doi.org/10.14569/IJACSA.2016.070471
20. Hemamalini, M.: Review on grid task scheduling in distributed heterogeneous environment. International Journal of Computer Applications **40**(2), 24–30 (2012)
21. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic acids research **34**, W729–W732 (2006)
22. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. Journal of the ACM **24**(2), 280–289 (1977). https://doi.org/10.1145/322003.322011
23. Johnson, D.S., Garey, M.R.: Computers and Intractability: A guide to the theory of NP-completeness. WH Freeman (1979)
24. Kane, D.M., Nelson, J., Woodruff, D.P.: An optimal algorithm for the distinct elements problem. In: Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. pp. 41–52. PODS '10, Association for Computing Machinery (2010). https://doi.org/10.1145/1807085.1807094
25. Krämer, M.: A Microservice Architecture for the Processing of Large Geospatial Data in the Cloud. Ph.D. thesis, Technische Universität Darmstadt (2018). https://doi.org/10.13140/RG.2.2.30034.66248
26. Krämer, M.: Capability-based scheduling of scientific workflows in the cloud. In: Proceedings of the 9th International Conference on Data Science, Technology, and Applications DATA. pp. 43–54. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009805400430054
27. Li, K., Xu, G., Zhao, G., Dong, Y., Wang, D.: Cloud task scheduling based on load balancing ant colony optimization. In: Proceedings of the 6th Annual Chinagrid Conference. pp. 3–9 (2011). https://doi.org/10.1109/ChinaGrid.2011.17
28. Maheswaran, M., Ali, S., Siegal, H.J., Hensgen, D., Freund, R.F.: Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In: Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99). pp. 30–44 (April 1999). https://doi.org/10.1109/HCW.1999.765094
29. Mell, P.M., Grance, T.: The NIST definition of cloud computing. Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, USA (2011)

30. Nayak, B., Padhi, S.K.: Mapping of independent tasks in the cloud computing environment. International Journal of Advanced Computer Science and Applications **10**(8) (2019)
31. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004). https://doi.org/10.1093/bioinformatics/bth361
32. Page, A.J., Naughton, T.J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (2005). https://doi.org/10.1109/IPDPS.2005.184
33. Russell, N., van van der Aalst, W.M., ter Hofstede, A.H.M.: Workflow Patterns: The Definitive Guide. MIT Press (2016)
34. Singh, S., Chana, I.: A survey on resource scheduling in cloud computing: Issues and challenges. Journal of Grid Computing **14**, 217–264 (2016). https://doi.org/10.1007/s10723-015-9359-2
35. Tawfeek, M.A., El-Sisi, A., Keshk, A.E., Torkey, F.A.: Cloud task scheduling based on ant colony optimization. In: Proceedings of the 8th International Conference on Computer Engineering Systems (ICCES). pp. 64–69 (2013). https://doi.org/10.1109/ICCES.2013.6707172
36. Thennarasu, S., Selvam, M., Srihari, K.: A new whale optimizer for workflow scheduling in cloud computing environment. Journal of Ambient Intelligence and Humanized Computing (2020). https://doi.org/10.1007/s12652-020-01678-9
37. Topcuoglu, H., Hariri, S., Min-You Wu: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems **13**(3), 260–274 (2002). https://doi.org/10.1109/71.993206
38. Ullman, J.: NP-complete scheduling problems. Journal of Computer and System Sciences **10**(3), 384–393 (1975). https://doi.org/https://doi.org/10.1016/S0022-0000(75)80008-0
39. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. USENIX Association (2010)