Capability-based Scheduling of Scientific Workflows in the Cloud

Michel Krämer[1,2][a]
[1]Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283 Darmstadt, Germany
[2]Technical University of Darmstadt, 64289 Darmstadt, Germany

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

Abstract: We present a distributed task scheduling algorithm and a software architecture for a system executing scientific workflows in the Cloud. The main challenges we address are (i) capability-based scheduling, which means that individual workflow tasks may require specific capabilities from highly heterogeneous compute machines in the Cloud, (ii) a dynamic environment where resources can be added and removed on demand, (iii) scalability in terms of scientific workflows consisting of hundreds of thousands of tasks, and (iv) fault tolerance because in the Cloud, faults can happen at any time. Our software architecture consists of loosely coupled components communicating with each other through an event bus and a shared database. Workflow graphs are converted to process chains that can be scheduled independently. Our scheduling algorithm collects distinct required capability sets for the process chains, asks the agents which of these sets they can manage, and then assigns process chains accordingly. We present the results of four experiments we conducted to evaluate if our approach meets the aforementioned challenges. We finish the paper 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”.

1 INTRODUCTION

Scientific workflow management systems are used in a wide range of areas including (but not limited to) Bioinformatics (Oinn et al., 2004), Geology (Graves et al., 2011), Geoinformatics (Krämer, 2018), and Astronomy (Berriman et al., 2004) to automate the processing of very large data sets. A scientific workflow is typically represented by a directed acyclic graph that describes how an input data set is processed by certain tasks in a given order to produce a desired outcome. Such workflows can become very large with *hundreds up to several thousands of tasks* processing data volumes ranging from gigabytes to terabytes.

Distributed computing infrastructures such as the Grid (Foster and Kesselman, 1998), or the Cloud (Mell and Grance, 2011) combine the computational power and storage resources of a large number of independent machines. They provide a well-suited environment for the execution of scientific workflows. In order to make best use of available resources, the workflow tasks have to be assigned to the machines in a smart way. While the general task scheduling problem is known to be NP-complete (Ullman, 1975; Johnson and Garey, 1979), the topic is of high interest to the research community and several approaches with varying aims and requirements have been published to find practical solutions for the Grid and the Cloud (Hemamalini, 2012; Singh and Chana, 2016).

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 challenge here is that, on the one hand, 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, the *machines have certain capabilities* and the tasks have *requirements regarding these capabilities* (or *required capabilities*). This has to be considered during task scheduling. As we show in Section 2, this concept has not been fully covered by existing approaches yet.

[a] https://orcid.org/0000-0003-2775-5844

Krämer, M.
Capability-based Scheduling of Scientific Workflows in the Cloud.
DOI: 10.5220/0009805400430054
In *Proceedings of the 9th International Conference on Data Science, Technology and Applications (DATA 2020)*, pages 43-54
ISBN: 978-989-758-440-4

In addition to the heterogeneity of machines, the topology of a Cloud is *highly dynamic* as 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.

Also, it is 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 (Chircu, 2018). 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 can occur at any time.

From these challenges, we derive specific requirements that our scheduling algorithm and the software architecture of our scientific workflow management system should satisfy:

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. Our system should not assume a static number of machines. Instead, 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 3. 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 4. 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 5. The system should be scalable so it can manage workflows with a large number of tasks.

REQ 6. 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

The main contribution of this paper is our *scheduling algorithm* that is able to assign workflow tasks to heterogeneous machines in the Cloud based on required capability sets.

In addition, we present a *software architecture* of a scientific workflow management system our algorithm is embedded in. We describe a set of components that communicate with each other through an event bus and a database to perform task scheduling in a scalable and fault-tolerant manner.

The remainder of this paper is structured as follows. We first analyse the state of the art in Section 2 and describe the research gap our work bridges. In Section 3, 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. After this, we present the software architecture in Section 4 and finally our main contribution, the scheduling algorithm, in Section 5. In Section 6, we also present the results of four experiments we conducted to evaluate if our approach meets the challenges and requirements defined above. We finish the paper in Section 7 with conclusions and future research opportunities.

An implementation of our scheduling algorithm and the software architecture is publicly available with the *Steep Workflow Management System*, which has recently been released under an open-source licence on GitHub: https://steep-wms.github.io/

2 RELATED WORK

Task scheduling can be performed in various ways. Many algorithms employ heuristics 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. Min-Min and Max-Min (Ibarra and Kim, 1977; Freund et al., 1998), 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.

The algorithm Sufferage, in contrast, is able to reassign a task from machine *M* to another one if there is a second task that would achieve better performance on *M* (Maheswaran et al., 1999). Casanova et al. present a heuristic called XSufferage that extends Sufferage and also considers data transfer cost (Casanova et al., 2000). They claim that 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 (Gherega and Pupezescu, 2011). Their approach prevents the heuristic itself from becoming a bottleneck in the scheduling process.

The algorithms mentioned above are optimised for certain situations. In contrast, genetic algorithms (GA) are able to automatically adapt to changing conditions. A GA mimics the process of natural evolution. It uses historical information to select the best mapping of tasks to machines. Good results with GA, for example, were achieved by Hamad and Omara who use Tournament Selection (Hamad and Omara, 2016) or by Page and Naughton whose algorithm does not make assumptions about the characteristics of tasks or machines (Page and Naughton, 2005).

There are other approaches that apply behaviour known from nature to the task scheduling problem. Ant colony optimisation algorithms (Tawfeek et al., 2013; Li et al., 2011), for example, try 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 (Thennarasu et al., 2020).

Besides scheduling algorithms that process individual tasks, there are more complex ones that consider the interdependencies in a scientific workflow and try to find optimal solutions by analysing the directed graph in total. Blythe at al. investigate the difference between task-based approaches like the ones mentioned above and the more complex workflow-based approaches (Blythe et al., 2005). 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) (Binato et al., 2002). 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 (Topcuoglu et al., 2002). 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.

There are a number of distributed scientific workflow management systems that typically implement one or more of the algorithms mentioned above. Examples are *Pegasus* (Deelman et al., 2015), *Kepler* (Altintas et al., 2004), *Taverna* (Hull et al., 2006), *Galaxy* (Giardine et al., 2005), *Airflow* (Apache Airflow, 2020), and *Nextflow* (Di Tommaso et al., 2017). Other frameworks that can process large data sets in the Cloud are Spark (Zaharia et al., 2010) and Flink (Carbone et al., 2015). They are not workflow management systems but follow a similar approach and also need to schedule tasks from a directed graph.

2.1 Research Gap

In the scientific community, dynamically changing environments, very large workflows, and fault tolerance are considered major challenges for modern distributed scientific workflow management systems, which have not been fully covered by existing approaches yet and therefore offer many research opportunities (Deelman et al., 2018). In Section 1, 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. None of the approaches, algorithms, and systems mentioned above (and to the best of our knowledge, no other existing work) cover all of them in one algorithm design. *In this paper, we present such an algorithm as well as the software architecture it is embedded in.*

There are similarities, however, between our approach and existing ones. DXSufferage, for example, uses the multi-agent paradigm (Gherega and Pupezescu, 2011). 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 (Casanova et al., 2000), this can potentially lead to better file reuse.

Note that, in this paper, we focus on the aforementioned Cloud challenges. Evaluating different scheduling heuristics to achieve the shortest makespan is beyond the scope of this paper and remains for future work.

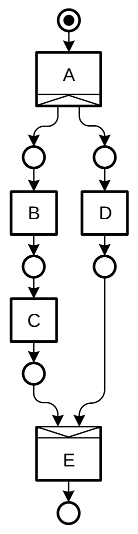
Also note that our approach is not directly comparable to workflow-based scheduling algorithms that consider the graph in total. Instead, we employ a hybrid strategy that first splits the graph into process chains and then schedules these instead of individual tasks.

3 WORKFLOW SCHEDULING

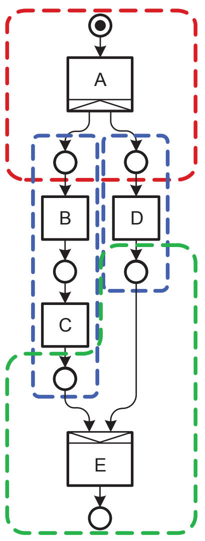
As described above, a scientific workflow is typically represented by a directed graph that describes in which order certain tasks need to be applied to an input data set to produce a desired outcome. Figure 1a shows a simple example of such a workflow in the extended Petri Net notation proposed by van der Aalst and van Hee (2004). In this example, an input file is first processed by a task A. This task produces two results. The first one is processed by task B whose result is in turn sent to C. The second result of A is processed by D. The outcomes of C and D are finally processed by task E.

In order to be able to schedule such a workflow in a distributed environment, the graph has to be transformed to individual executable units. Our scientific workflow management system follows a hybrid scheduling approach that applies heuristics on the level of the workflow graph and later on the level of individual executable units. We assume that tasks that access the same data should be executed on the same machine to reduce the communication overhead and to improve file reuse. We therefore group tasks into so-called *process chains*, which are linear sequential lists (without branches and loops).

While grouping the tasks, we also need to take the capabilities into account that they require from the machines in the Cloud. In our implementation, capabilities are user-defined strings. For example, the set {"*Ubuntu*", "*GPU*"} might mean that a task depends



(a) Workflow graph.



(b) Generated process chains.

Figure 1: A workflow is split into four individual process chains in three iterations.

on the Linux distribution Ubuntu as well as the presence of a graphics processing unit. We call the union of the required capabilities of all tasks in a process chain a *required capability set*.

Our system transforms workflows to process chains in an iterative manner. In each iteration, it finds the longest linear sequences of tasks with similar required capabilities and groups them to process chains (see Figure 1b). For our example workflow, task A will be put into a process chain in iteration 1. Our system then schedules the execution of this process chain according to the algorithm from Section 5. After the execution has finished, the system uses the results to produce a process chain containing B and C (assuming they require similar capabilities) and another one containing D. These process chains are then scheduled to be executed in parallel. The results are finally used to generate the fourth process chain containing task E, which is also scheduled for execution.

4 SOFTWARE ARCHITECTURE

Figure 2 shows the main components of our scientific workflow management system: the HTTP server, the controller, the scheduler, the agent, and the cloud manager. Together, they form an instance of our system. In practice, a single instance typically runs on a separate virtual machine, but multiple instances can also be started on the same machine. Each component

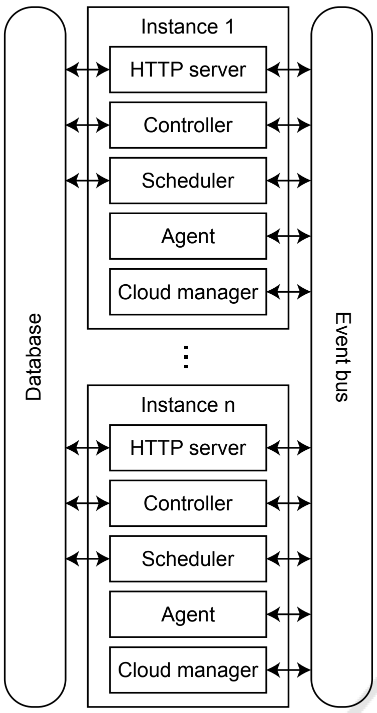


Figure 2: An overview of the components in our scientific workflow management system and how they communicate with each other.

can be enabled or disabled in a given instance. That means, in a cluster, there can be instances that have all five components enabled, and others that have only an agent, for example.

All components of all instances communicate with each other through messages sent over an event bus. Further, the HTTP server, the controller, and the scheduler are able to connect to a shared database. In the following, we describe the roles and responsibilities of each component.

The HTTP server provides information about scheduled, running, and finished workflows to clients. Clients can also upload a new workflow. In this case, the HTTP server puts the workflow into the database and sends a message to one of the instances of the controller.

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. The schedulers load the process chains from the database, send them via the event bus to the selected agents for execution, and finally 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 (a component that interacts with the API of the Cloud infrastructure) and asks it to create a new agent.

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 practice, but in our implementation, this is configurable. This approach decouples the components from each other and increases fault tolerance.

5 CAPABILITY-BASED SCHEDULING ALGORITHM

In this section, we present the capability-based scheduling algorithm that is executed in our scheduler component. We first describe the main scheduling function (Section 5.1) and then discuss how our algorithm selects candidate agents (Section 5.2).

5.1 Main Scheduling Function

Listing 1 shows the main function of our scheduling algorithm. As mentioned above, the scheduler calls this function at regular intervals and immediately after new process chains have been added to the database.

At the beginning, our algorithm calls *findRequiredCapabilitySets()*. This function 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 function will try to schedule as many process chains as possible until it reaches a *break* statement. There is only one in line 12 that is reached when there is no agent available anymore (see below).

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

 3    for $i \in [0, maxLookups)$:
 4      candidates = selectCandidates(S)

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

13      for $(candidate, R) \in candidates$:
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: 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.

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). The function *selectCandidates()* is described in detail in Section 5.2. 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 $R \in S$. In this case, the function iterates over all required capability sets (line 8) and checks if there actually is a corresponding registered 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 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

```
function selectCandidates($S$):
  $candidates = \emptyset$

  for $a \in Agents$:
    send $S$ to $a$ and wait for $response$
    if $a$ is available:
      get best $R_i \in S$ from $response$
      $P = (a, R_i)$
      $candidates = candidates \cup \{P\}$

  $L$ = all $P \in candidates$ with best $a$ for each $R_i$

  return $L$
```

Listing 2: Pseudo code of the function that selects agents based on their capabilities.

list $L = \{P_1, ..., P_m\}$ of pairs $P = (a, R_i)$ of an agent a and the matching required capability set R_i. Listing 2 shows the pseudo code.

The function sends all required capability sets to each agent via the event bus. The agents respond whether they are available and which required capability set they support best. The function collects all responses in a set of *candidates*. It finally selects exactly one agent for each required capability set.

The decision of which available agent to select can be implemented based on certain heuristics (e.g. the expected earliest completion time of process chains with the given required capability set). In our implementation, we select the agent that was idle for the longest time. In practice, this has proven to be a heuristic that achieves good throughput and, at the same time, prevents starvation because every agent will be selected eventually.

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.

6 EVALUATION

In Section 1.1, we defined four major challenges for the management of scientific workflows in the Cloud and derived six requirements for our system. In order to evaluate if our scheduling algorithm and our software architecture meet these challenges and requirements, we conducted four practical experiments (one for each challenge). In this section, we present the results of these experiments and discuss benefits and drawbacks of our approach.

Table 1: This caption has more than one line so it has to be justified.

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

6.1 Setup

All experiments were performed in the same environment. We set up our system in a private OpenStack Cloud and configured it so that it could access the Cloud's API and create further virtual machines on demand. We deployed the full stack of components presented in Section 4 to each virtual machine. These components communicated with each other through a distributed event bus. They also shared a MongoDB database deployed to a separate virtual machine. In the following, according to our system architecture, we use the term *agent* for a virtual machine running our system and capable of executing process chains.

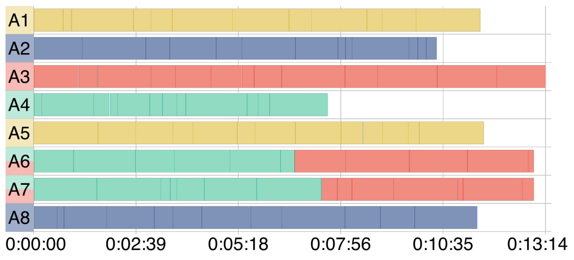
We defined four types of agents with the capability sets $R1$, $R2$, $R3$, $R4$, as well as fifth type offering both capability sets $R3$ and $R4$. To simulate a heterogeneous environment, we configured a maximum number of agents that our system was allowed to create per required capability set (see Table 1).

For each experiment, we collected all log files of all instances of our system and converted them to graphs. Figure 3 shows the results. Each of the subfigures—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. Each required capability set has a different colour (see legend in Figure 3d). 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.

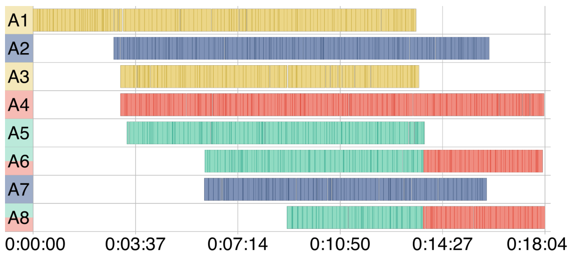
6.2 Experiments

Experiment 1: Capability-based Scheduling
(Requirement covered: REQ 1)

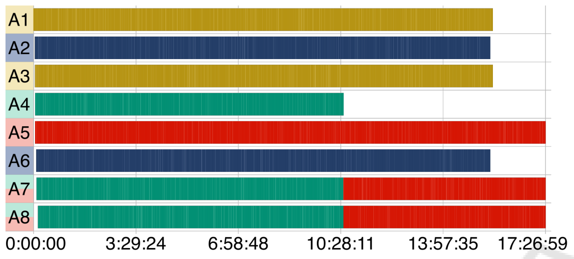
Our primary goal in this paper was to create a scheduling algorithm that is able to assign workflow task to



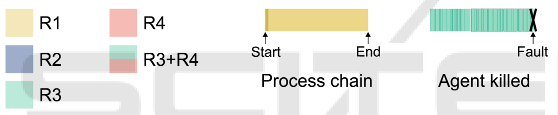
(a) 100 process chains are distributed to the agents with the correct capabilities.



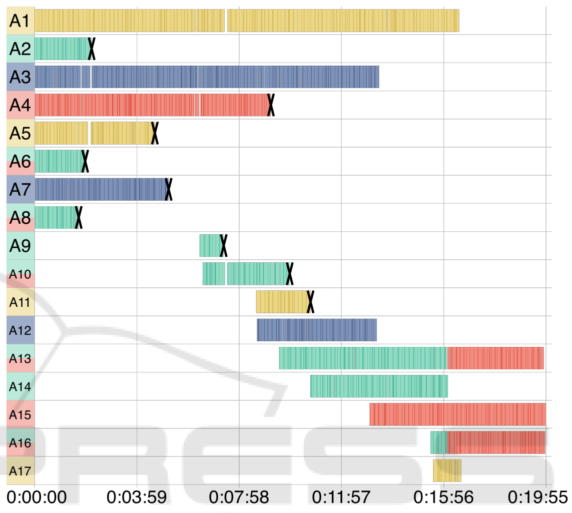
(b) The system creates agents with capabilities required by 1 000 process chains on demand.



(c) The system is able to handle 150 000 process chains.



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



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

Figure 3: Results of our experiments.

distributed machines based on required and offered capabilities. Figure 3a shows we reached this goal.

We deployed a static number of eight agents with different capability sets. We then 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. The workflow took 13 minutes and 14 seconds to complete in total.

Note that our algorithm is designed to ask the agents which required capability set they would like to execute before assigning process chains to them (see Section 5.2). Agents $A6$ and $A7$ were able to execute process chains requiring both $R3$ and $R4$ but they obviously preferred to accept the ones with $R3$ first before they continued with $R4$.

Experiment 2: Dynamic Environment
(Requirements covered: REQ 1–4)

For our second experiment, we deployed only one agent with capability set $R1$. We then executed a workflow with 1 000 process chains. Figure 3b shows the timeline of the workflow run.

As soon as the first scheduler instance found all required capability sets in the database, it asked the cloud manager component of our system to create new agents. 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 in our implementation, the cloud manager only creates one agent of a certain capability set at a time. Also, as described earlier, we configured a maximum number of agents per capability set. These are the reasons why only four new agents appear at about minute 3, two others at minute 6, and a final one between minutes 8 and 9.

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.

Experiment 3: Scalability
(Requirements covered: REQ 1–5)

In order to show the scalability of our system, we launched a workflow with 150 000 process chains. Similar to the second experiment, we started with one agent. The other ones were automatically created by our system on demand.

Figure 3c shows the timeline of the workflow over more than 17 hours. The darker colour of the graph stems from the fact that there are a lot more start events in this workflow than in the other ones.

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.

Experiment 4: Fault Tolerance
(Requirements covered: all)

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

We killed nine 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 2 and 6, approximately, there was no agent that was able to execute process chains with $R3$. Similarly, between minutes 9 and 13, approximately, $R4$ was not processed by any agent ($A13$ would have been able to, but as mentioned earlier, it preferred to execute $R3$ first). The execution of these process chains was postponed and resumed later.

6.3 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.

As described in Section 5.2, our current implementation of the scheduler chooses between multiple available agents by comparing their idle time. Although this works very well in practice, our experiments have shown that this approach leaves room for improvement. For example, in experiment 1 (Figure 3a), if the schedulers had assigned process chains with $R4$ to agents $A6$ and $A7$ earlier, then the makespan could have been reduced by approximately one or two minutes (about 5–10%). Agents $A3$, $A6$, and $A7$ would have finished earlier with $R4$ while agent $A4$ could have processed $R3$ until the end.

Nevertheless, our approach allows various scheduling strategies to be implemented since our scheduler has a complete overview of the required capability sets and the available agents. Different heuristics such as Min-Min or Max-Min can be implemented in the *selectCandidates* function without changing the general structure of the algorithm. Note that these heuristics would work on the level of required capability sets and not on the individual process chains, which differentiates our approach from existing works. However, investigating different heuristics and optimising the makespan was not a goal of this paper and remains for future work.

The scalability and fault tolerance of our approach mostly stems from the fact that we use a database to store process chains. First, this out-of-core approach reduces the required main memory. Only a few process chains (plus all distinct required capability sets) need to be kept in memory at a time, which allows hundreds of thousands of process chains to be scheduled without any issue. Second, since the database holds the remaining process chains to execute, it essentially keeps the current state of the overall workflow execution. 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.

The database can considered a single point of failure. If it becomes unavailable, workflow execution cannot continue. In pratice, this is not a problem because as soon as it is up again, our scheduling algorithm can proceed and no information will be lost. However, the database has an impact on scheduling performance. Our algorithm as it is presented in Section 5 needs to fetch the set of distinct required capability sets each time the *lookup* function is called. Queries for distinct values are known to be time-consuming, especially for large collections like in our case. The only way to find all distinct values is to perform a sequential scan. Most DBMS even have to sort the collection first. There are approaches to find approximations of distinct values (Kane et al., 2010; Bar-Yossef et al., 2002) but our algorithm needs exact results. Our open-source implementation therefore caches the set of required capability sets for a configurable amount of time. Our algorithm is robust, and even when this cache becomes inconsistent with the database, it will still work correctly. It will either not find a process chain for a given required capability set (Listing 1, line 15), or one or more required capability sets will be missing from the cache. In the first case, the algorithm will just continue with the next candidate agent and required capability set (line 18). Regarding the second case, adding new process chains will trigger a cache update, so the next scheduling operation will be able to use all required capability sets. In any case, after the configurable amount of time has passed, the cache will be updated anyhow.

Our experiments have revealed other places where our system could be improved. At the moment, our cloud manager creates only one agent per capability set at a time. This could be parallelised in the future to further reduce the makespan. Also, there are small gaps visible between process chain executions in our evaluation timelines. In experiment 4, they are due to the fact that we killed schedulers that had just allocated an agent and did not get the chance to deallocate it. In our implementation, agents automatically deallocate themselves if they do not receive a message from a scheduler for a certain amount of time, but until then, they do not execute anything. The gaps in the other experiments are most likely due to long database operations but need further investigation. As can be seen in Figure 3, these gaps are very small (only a few milliseconds) and do not happen very often.

7 CONCLUSION

In Section 1.1, we identified four major challenges regarding the execution of scientific workflows in the Cloud: capability-based scheduling, a dynamic environment, scalability, and fault tolerance. These challenges, and in particular capability-based scheduling, have not been fully covered in the scientific community yet. In this paper, we presented a scheduling algorithm and a software architecture for a scientific workflow management system that address these challenges. In four experiments, we have shown that our approach works and meets our requirements. We also discussed benefits and drawbacks.

We implemented our algorithm and the software architecture in the *Steep Workflow Management System*, which has recently been released under an open-source licence (https://steep-wms.github.io/). We are using Steep 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 our system are fundamental in this project.

In addition, the AI service requires a graphics processing unit (GPU), which is a limited resource in the Cloud and particularly expensive. Our capability-based scheduling algorithm helps in this respect to distribute the workflow tasks to the correct machines (i.e. to only use GPU machines when necessary). As our system supports elasticity and a dynamic number of machines, we can also scale up and down on demand, only create GPU machines when needed, and release them as soon as possible. In other words, in this project, our system saves time and money.

Based on the work presented in this paper, there are a number of opportunities for future research. For example, the iterative approach to transform workflow graphs to process chains presented in Section 3 allows very complex workflows to be processed. It supports workflows *without a priori design-time knowledge* (Russell et al., 2016), which means the system does not need to know the complete structure of the workflow before the execution starts. This enables us to dynamically change the structure of the workflow during the execution as the number of instances of a process chain can depend on the results of a preceding one. The approach is also suitable to execute workflows *without a priori runtime 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 we are currently working on.

Our experiments have revealed areas where our approach and implementation can be further improved. In the future, we will investigate different scheduling heuristics such as Min-Min or Max-Min to improve the makespan of workflows. As mentioned above, we also discovered small gaps in the results of our experiments. They are most likely due to the way we implemented our system. We will investigate them and try to find optimisations. Further, we will improve our cloud manager so it creates multiple agents with a given capability set in parallel.

REFERENCES

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. (2004). Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 423–424. IEEE.

Apache Airflow (2020). Apache Airflow Website. https://airflow.apache.org/. Last accessed: 2020-04-14.

Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., and Trevisan, L. (2002). Counting distinct elements in a data stream. In Rolim, J. D. P. and Vadhan, S., editors, *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer Berlin Heidelberg.

Berriman, G. B., Deelman, E., Good, J. C., Jacob, J. C., Katz, D. S., Kesselman, C., Laity, A. C., Prince, T. A., Singh, G., and Su, M.-H. (2004). Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Optimizing Scientific Return for Astronomy through Information Technologies*, volume 5493, pages 221–233. International Society for Optics and Photonics.

Binato, S., Hery, W. J., Loewenstern, D. M., and Resende, M. G. C. (2002). *A Grasp for Job Shop Scheduling*, pages 59–79. Springer US.

Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., and Kennedy, K. (2005). Task scheduling strategies for workflow-based applications in grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 759–767.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). 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.

Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. (2000). Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop HCW*, pages 349–363.

Chircu, V. (2018). Understanding the 8 fallacies of distributed systems. https://dzone.com/articles/understanding-the-8-fallacies-of-distributed-syste. Last accessed: 2020-02-18.

Deelman, E., Peterka, T., Altintas, I., Carothers, C. D., van Dam, K. K., Moreland, K., Parashar, M., Ramakrishnan, L., Taufer, M., and Vetter, J. (2018). The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., and Wenger, K. (2015). Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.

Di Tommaso, P., Chatzou, M., Floden, E. W., Barja, P. P., Palumbo, E., and Notredame, C. (2017). Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319.

Foster, I. and Kesselman, C., editors (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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., and Siegel, H. J. (1998). Scheduling resources in multi-user heterogeneous computing environments with SmartNet. *Calhoun: The NPS Institutional Archive*.

Gherega, A. and Pupezescu, V. (2011). 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*, pages 313–320.

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., and Nekrutenko, A. (2005). Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455.

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., and Vahi, K. (2011). Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics*, 168(3):367–381.

Hamad, S. A. and Omara, F. A. (2016). Genetic-based task scheduling algorithm in cloud computing environment. *International Journal of Advanced Computer Science and Applications*, 7(4):550–556.

Hemamalini, M. (2012). Review on grid task scheduling in distributed heterogeneous environment. *International Journal of Computer Applications*, 40(2):24–30.

Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P., and Oinn, T. (2006). Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34:W729–W732.

Ibarra, O. H. and Kim, C. E. (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289.

Johnson, D. S. and Garey, M. R. (1979). *Computers and Intractability: A guide to the theory of NP-completeness*. WH Freeman.

Kane, D. M., Nelson, J., and Woodruff, D. P. (2010). An optimal algorithm for the distinct elements problem. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 41–52. Association for Computing Machinery.

Krämer, M. (2018). *A Microservice Architecture for the Processing of Large Geospatial Data in the Cloud*. PhD thesis, Technische Universität Darmstadt.

Li, K., Xu, G., Zhao, G., Dong, Y., and Wang, D. (2011). Cloud task scheduling based on load balancing ant colony optimization. In *Proceedings of the 6th Annual Chinagrid Conference*, pages 3–9.

Maheswaran, M., Ali, S., Siegal, H. J., Hensgen, D., and Freund, R. F. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pages 30–44.

Mell, P. M. and Grance, T. (2011). The NIST definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, USA.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. (2004). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054.

Page, A. J. and Naughton, T. J. (2005). Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*.

Russell, N., van van der Aalst, W. M., and ter Hofstede, A. H. M. (2016). *Workflow Patterns: The Definitive Guide*. MIT Press.

Singh, S. and Chana, I. (2016). A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of Grid Computing*, 14:217–264.

Tawfeek, M. A., El-Sisi, A., Keshk, A. E., and Torkey, F. A. (2013). Cloud task scheduling based on ant colony optimization. In *Proceedings of the 8th International Conference on Computer Engineering Systems (ICCES)*, pages 64–69.

Thennarasu, S., Selvam, M., and Srihari, K. (2020). A new whale optimizer for workflow scheduling in cloud computing environment. *Journal of Ambient Intelligence and Humanized Computing*.

Topcuoglu, H., Hariri, S., and Min-You Wu (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

Ullman, J. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association.