UNIVERSITY OF TARTU

Institute of Computer Science
Software Engineering Curriculum

Jaagup Viil

# Framework for Automated Partitioning of Scientific Workflows on the Cloud

Master's Thesis (30 ECTS)

Supervisor: Satish Narayana Srirama, PhD

Tartu 2017

## Raamistik teaduslike töövoogude automaatseks partitisioneerimiseks pilves

**Lühikokkuvõte:** Teaduslikud töövood on saanud populaarseks standardiks, et lihtsal viisil esitada ning lahendada erinevaid teaduslikke ülesandeid. Üldiselt koosnevad need töövood suurtest hulkadest ülesannetest, mis nõuavad tihti palju erinevaid arvuti ressursse, mistõttu jooksutatakse neid kas pilvearvutust, hajustöötlust või superarvuteid kasutades. Varem on tõestatud, et kui rakendada pilves töövoo erinevate osade jagamiseks k-way partitsioneerimis algoritmi, siis üleüldine kommunikatsioon pilves väheneb. Antud magistritöös programmeeriti raamistik, et seda protsessi automatiseerida. Loodud raamistik võimaldab automaatselt partitsioneerida igasugusegi töövoo, mis on mõeldud Pegasuse programmiga jooksutamiseks. Raamistik, kasutades CloudML'i, seab automaatselt pilves üles klastri masinaid, konfigureerib ning sätestab kõik vajaliku tarkvara ning jooksutab ja partitsioneerib etteantud töövoo. Lisaks, kuvatakse pärast töövoo lõpetamist ka ajalise kalkulatsiooni visualisatsioon. Seda kasutades saab lõppkasutaja aimu, mitu tuuma peaks töövoo jooksutamiseks kasutama, et lõpetada eksperiment mingis kindlas ajavahemikus.

**Võtmesõnad:** Teaduslikud töövood, pilvearvutus, partitsioneerimine, METIS, CloudML, Pegasus

**CERCS:** Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria) (P170)

## Framework for Automated Partitioning of Scientific Workflows on the Cloud

**Abstract:** Scientific workflows have become a standardized way for scientists to represent a set of tasks to overcome or solve a certain problem. Usually these workflows consist of numerous amount of jobs that are both CPU heavy and I/O intensive that are executed using some kind of workflow management system either on clouds, grids, supercomputers, etc. Previously, it has been shown that using k-way partitioning algorithm to distribute a workflow's tasks between multiple machines in the cloud reduces the overall data communication and therefore lowers the cost of the bandwidth usage. In this thesis, a framework was built in order to automate this process - partition any workflow submitted by a scientist that is meant to be run on Pegasus workflow management system in the cloud with ease. The framework provisions the instances in the cloud using CloudML, configures and installs all the software needed for the execution, runs and partitions the scientific workflow and finally shows the time estimation of the workflow, so that the user would have an approximate guidelines on, how many resources one should provision in order to finish an experiment under a certain time-frame.

**Keywords:** Scientific workflows, cloud computing, partitioning, Pegasus, METIS, CloudML

**CERCS:** Computer science, numerical analysis, systems, control (P170)

# Contents

# List of Figures

# List of Tables

# 1 Introduction

In recent years there has emerged a lot of applications that rely heavily on resources and computing power. It is known, that maintaining an own infrastructure is hard due to system administration, electric power bills, hardware maintenance etc. Cloud has several benefits like elasticity, scalability and the promise of providing virtually unlimited resources, so therefore, more and more users are migrating their applications to the cloud. While cloud is generally known more to be used for different enterprise applications, the scientific community has also been turning towards cloud computing for the additional benefits.

Scientific workflows have become in the technical domain the primary way of representing a solution, usually consisting of numerous amount of jobs/steps, to a certain scientific problem [GDE+07]. After the modeling of a workflow, a scientist just has to submit it to a SWfMS (Scientific Workflow-Management System) like Pegasus [DVJ+15] or Kepler [ABJ+04] and wait for the desired results, without having to think about task scheduling, data allocation, fault tolerance, etc. While running computations in the cloud has many benefits, there are also several hindrances such as communication latencies, cost and set-up to name a few. Also, finding an ideal deployment for a specific case can be hard due to different scaling requirements, platform and vendor dependability, etc.

Previously it has been shown, that the communication latencies can be reduced using existing graph partitioning techniques on a scientific workflow by distributing the tasks between a computational cluster in the cloud more efficiently [SV14]. While the previous work was focused on a single use case (Montage), in this thesis, the process of partitioning and scheduling the tasks and the execution of any non-hierarchical workflow with Pegasus SWfMS is automated. This is achieved by analyzing the task logs produced by the Pegasus system and constructing a weighted graph, partitioning the graph using METIS [KK98] and remapping the results back to the original workflow. With this approach, a user can easily partition any workflow, and gain several benefits such as overall bandwidth reduction, occasional improvement of the execution time and the decrease of the cost of running an experiment in the cloud.

As stated earlier, some of the problems of executing scientific workflows in the cloud are for example the setting up of the system itself and vendor dependability. These problems can be overcome using multi-cloud deployment tools like JClouds [JCl] or CloudML [GES+11]. In this thesis, a framework is built, to easily setup a cluster of machines in the cloud (OpenStack or Amazon), using CloudML and Java Spring. The toolkit configures and sets up everything for the user to easily run partitioned workflows in the cloud.

Another problem this thesis also tries to solve, is trying to estimate the runtime of a scientific workflow, based on the number of CPU cores. With this approach, after the initial run, a user can easily see the time estimation of the workflow and select the preferred amount of cores for their problem. This was achieved by modifying the makespan algorithm produced by I.Pietri et al. [PJDS14] and adding a scaling factor based on the initial run. Also, to ease the choosing of the resources, a cost parameter was added to the selection process, so that a user can see the estimated cost of running a workflow on a certain setup. All the functionality listed, i.e. the time estimation, partitioning and automatic setup are included in the built framework.

## 1.1 Motivation

Grid computing, while having many hindrances, has been popular for many years among scientists to run their scientific workflows on. According to C.Hoffa et al., *"One of the primary obstacles Grid users face today is that while a Grid offers access to many heterogeneous resources, a user typically needs a very specific environment that is customized to support a legacy application, remains consistent across multiple executions, or supports the idiosyncrasies of many service configurations."* [HMF+08].
Cloud with its many advantages is a good alternative in this case but has its own problems. One of the key disadvantage for a single scientist is the set up of the overall system. For example, assuming a user has modeled a scientific workflow to be run with Pegasus SWfMS in the cloud, they have to have a running and configured system to execute that workflow on top of. For instance, in the grid scenario, usually the SWfMS is already installed for the scientist and ready to be used. In the cloud case, every time instances are either provisioned or deprovisioned, they have to be configured again, which is time consuming and tedious.

Over the years there has emerged numerous tools to ease the automation of configuration of the instances in the cloud, e.g. Salt [SAL], Ansible [ANS], etc. Because some of these tools are proprietary, can force vendor lock-in or cost money, one should look for some other open-source alternatives, that can be modified for a specific use case. For example, JCloud or CloudML, that are based on Java, suit perfectly to build the basis of a framework, in order to set up a SWfMS, in this case Pegasus, with all its dependencies in the cloud easily.

Let's say that a scientist has a tool to easily setup and configure the Pegasus SWfMS system in the cloud, and provision as many instances before the execution for a certain workflow as needed. Still, a few problems remain unsolved, for example, which is a good configuration for a certain problem considering task scheduling and how many cores to provision for a specific execution? The solution to the former problem is using k-way graph partitioning to distribute the tasks between the virtual machines, to increase the intra-instance communication while decreasing the inter-instance communication. The problem of finding how many cores to provision for an execution can be solved using some kind of time estimation for the workflow, that is based on the number of CPU cores available. An algorithm, that takes into consideration the runtimes of each separate task of the workflow is a good way for a runtime approximation [PJDS14]. To fine-tune the estimation even more, a scale factor based on the first run of the workflow can be introduced to the initial algorithm. Taking into account, that for each number of cores provisioned, one gets a runtime estimation of the workflow, it is rather easy to plug in also a cost factor. With this approach, a user can easily see at what cost and how many cores suit their specific case, when considering the execution time.

All in all, in this thesis a framework is built, in order to ease the execution, configuration, and deployment of a scientific workflow with Pegasus SWfMS in an optimal and automated way in the cloud.

## 1.2 Thesis outline

The structure of the rest of the thesis is as follows. Chapter 2 provides a background to the main paradigms of this thesis. Chapter 3 introduces the related works done regarding partitioning, time estimation and multi-cloud deployment. Chapter 4 describes the architecture and the technical details of the built framework. Chapter 5 shows the results and gains of the partitioning, accuracy of the time estimation and the duration of the deployment, and finally, chapter 6 concludes the thesis and goes briefly over the future work to be done.

# 2 Background

## 2.1 Cloud computing

The cloud computing paradigm has well matured over the years, but the underlying characterization of its core idea has remained unchanged. According to National Institute of Standards and Technology:
*"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction "* [MG$^+$11]. In other words, cloud computing offers a convenient and easy way for the public to access various computing resources over a certain network. Much like water or electricity, the resources are provisioned as a utility, where the cost is calculated by the amount of usage.

Cloud computing can be categorized into three main services [ZCB10]. Infrastructure as a service (IaaS), Platform as a Service (PaaS) and Software as a Service (Saas). Infrastructure as a Service can be described as a way of provisioning different types of resources, usually virtual machines, to the users. For example, Amazon EC2 [Ama] or Google Cloud [Pla] are well known IaaS providers. PaaS refers to a service, where usually a pre-configured platform is provisioned to the customers. This allows the users to manage and develop several applications with an ease, without the need to maintain the underlying infrastructure. For example, PaaS providers like Heroku or Google App Engine are popular among developers because they are easy to use and configure. SaaS can be characterized as a service, where a certain software that uses the benefits of cloud computing is provisioned to the customers. Some of the well known SaaS applications are for example Dropbox or Salesforce. Figure 1 depicts the different services of cloud computing and its layers.

Cloud computing paradigm bears many different advantages, for example, scalability, utility computing, on-demand provisioning, etc. Scalability is one of the key features of cloud computing. It forwards the idea, that services run on the cloud are flexible and can be scaled real-time. For example load balancing is a popular way of utilizing the full power of scalability [SIV16], i.e. when the load of some application increases, spawn more resources and vice versa. Utility computing, as stated before, is the idea of "leasing" different computing resources, while charging for their usage. This is perfect, because companies can cut out the hassle of maintaining and managing their own infrastructure and migrate their applications to the cloud, while reducing the overall costs. One can not also forget the ease of deployment when using cloud computing, as users can launch several different resources quite easily and quickly. With the help of virtual machine images,

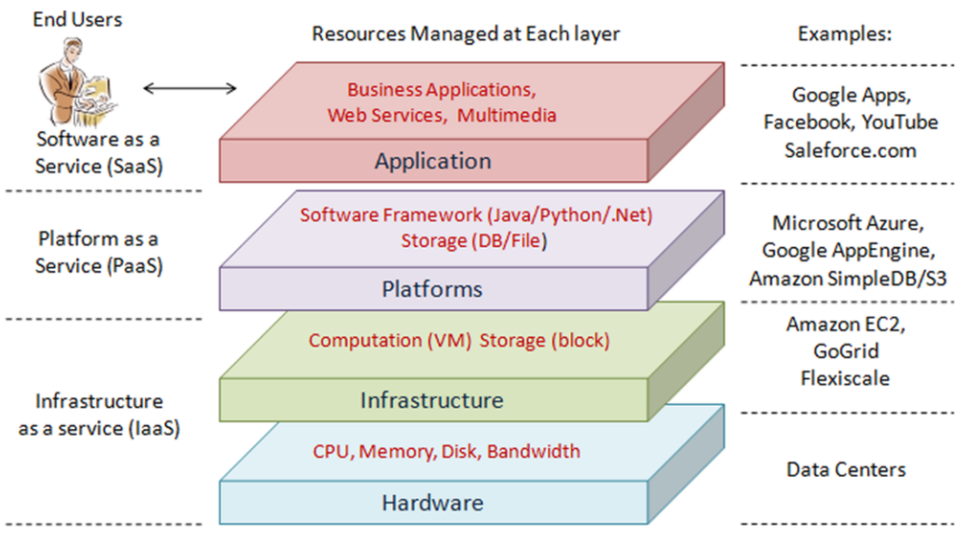

End Users
Resources Managed at Each layer
Examples:
Software as a Service (SaaS)
Business Applications, Web Services, Multimedia
Application
Google Apps, Facebook, YouTube Saleforce.com
Platform as a Service (PaaS)
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Computation (VM) Storage (block)
Infrastructure
Amazon EC2, GoGrid Flexiscale
Infrastructure as a service (IaaS)
CPU, Memory, Disk, Bandwidth
Hardware
Data Centers

Figure 1: Different cloud services and its layers [ZCB10]

it is also rather easy to create a template from a machine, that can be later used across an enterprise system as a base building block.

Like every coin has two sides, cloud computing has also its downsides. The underlying virtualization technology, that enables the running of multiple independent virtual machines on one platform causes extra communication latencies, which hinders the execution of different scientific applications [SBJV11]. Also when considering that cloud infrastructures usually use commodity computers, one has to take into consideration that they fail after a certain time, which means data replication and fault tolerance are a necessity when thinking of the execution of different applications, especially scientific workflows.

Commonly, when scientific applications are developed, they are run on high performance super computers or grid infrastructures. While performing applications, that require a lot of computational resources, run quite efficiently on these platforms, the lack of control and the dependency on the availability of these underlying resources are the main reasons why a scientist should consider using cloud computing. With the help of commercial cloud services like Amazon EC2 or Microsoft Azure [Azu], the accessibility to spawn customizable computer clusters has been significantly simplified.

Whether scientists are using cloud environments like Amazon EC2 or OpenStack, cloud computing can be divided into these three types [MG$^{+}$11]:

- Public cloud – A model, where the provisioned resources are available to the public. The customer can utilize as many resources as needed in exchange for money. Usually, these clouds are owned by businesses or government organizations.

- Private cloud – A model where a cloud environment, e.g. Nebula or OpenStack has been set up within an organization and is unavailable to the public. This setup provides more fine grained control over the resources available and can be configured for advanced security.

- Hybrid cloud – A combination of both of the models described beforehand. For example, when an application that is running in a private cloud suddenly becomes under heavy communication load, one can burst it to a public cloud for more resources to satisfy the increased demand.

While Amazon Elastic Compute Cloud, or in short, EC2, has been a popular IaaS provider for a long time, over the years, there has emerged several other massive cloud infrastructures like Google Compute Engine or Microsoft Azure. By using these services, customers can create, start, destroy instances according to their need and they only have to pay for the usage of the resources. While these infrastructures share all of the main benefits of cloud computing like elasticity and scalability, it falls on the shoulders of the end user to choose the most suitable vendor. For example, when considering billing, EC2 charges users hourly, while Google Cloud supports per-minute billing. Also, the price rates vary for different regions, each vendor offers different type of machines with different storage constraints, etc., making the choosing of the provider a tedious process.

The previously listed cloud infrastructures, as stated, have a lot of advantages, however the main problem using their services is the cost. For example, users can not test whether their applications are running correctly on the cloud beforehand and they have to pay even when they are migrating and testing the application. For these reasons there has emerged open source cloud platforms like Nebula or OpenStack. In essence, they are quite similar to the commercial cloud providers – features like elasticity, scalability and virtualization are also part of the system and just a matter of implementation. Scientists can deploy their applications, test them under different conditions and if they want to, migrate also it to the commercial cloud.

Albeit open source private clouds may seem a tempting alternative for scientists, it is actually not that simple. It takes a lot of work, skills and time to set up a private cloud on your own infrastructure. One has to carefully consider which kind of infrastructure to buy and set up in the first place. Additionally one has to rely a lot on the documentation, which can be sometimes insufficient, and if there is not a large community behind the open source project, finding answers for a particular problem will be hard. These problems does not exist with commercial cloud services because a feature like client support is already paid for.

## 2.2 Scientific workflows

Commonly, when a scientist is working on a problem, they have numerous steps to take in order to verify and validate their results, such as repeating experiments with new data sets, reproducing previous results, and so forth. Often both the observations gotten and the different tasks of the experiments are needed to be shared with other scientists, which can be cumbersome, if there is no standardized way of representing these experiments. Fortunately, scientific workflows have enabled scientists to model and describe different problems in an easy and a standardized way, in order to automate the overall scientific process. It is a paradigm for representing, orchestrating and managing complicated scientific computations [GDE+07].

Workflows are composed of several different basic structures: processes, data distributions, pipelines, data aggregations and data redistributions [BCD+08]. For example, data aggregation is a process, where multiple inputs are aggregated into one output, while data distribution does the exact opposite as a single input of data is divided into multiple subsets in order to increase parallelism in the next step of the workflow's execution.

Scientific workflows are being used in various different fields such as cosmology, biology, physics, etc. For example, the Laser Interferometer Gravitational Wave Observatory has used Pegasus SWfMS to execute and create their scientific workflows in order to detect gravitational waves produced by different types of events in the universe [Blu16]. Scientific workflows are used even in the fields like seismology, for instance, the CyberShake workflow has been intensively utilized by the Southern California Earthquake Center to characterize different earthquake hazards [GJC+11].

There are numerous ways to represent scientific workflows. In this thesis, all the workflows are created in the directed acyclic graph in XML format, known as DAX, which is compatible with Pegasus SWfMS. For example, the Taverna SWfMS [Tav] uses a SCUFL (Simple Conceptual Unified Flow Language) to specify their workflows.

### 2.2.1 Overview of the workflows used

This subsection describes the multiple workflows that were used in the thesis, in order to test different aspects regarding data scheduling, time estimation, setup of the workflows themselves, etc.

**Montage**

Montage is an open source toolkit, designed by the NASA/IPAC Infrared Science archive, to generate different mosaics from the sky using FITS images as an input [Mon]. The toolkits workflow representation is quite popular among scientists to be used for testing of different hypothesis, ranging from scheduling of tasks to workflow cost estimations. It is a data intensive application and 95% of the time, it is waiting on I/O operations and only 5% of the time on computation [AJD12], which makes the workflow perfect for trying various data scheduling techniques.

It consists of 9 different levels, where each level has various types of jobs. The *mProjectPP* jobs are loosely correlated to the number of input images (fetched from the NASA/IPAC database) and perform on each input a plane-to-plane transform. The *mDiffFit* jobs will calculate the difference of the overlapping pair images gotten from the previous jobs and its outputs are merged by the *mConcatFit* job [Mon]. The *mBgModel* determines what corrections to apply on each image and the *mBackGround* applies these corrections on the images. The next job, *mImgTbl*, extracts geometry information and creates image metadata table, that will be used in the upcoming tasks. Finally the *mAdd* job, which is the most compute intensive, co-adds the reprojected images to produce an output mosaic, which is reduced by the *mShrink* job and transformed into a JPEG format by the *mJPEG* job [BCD+08].

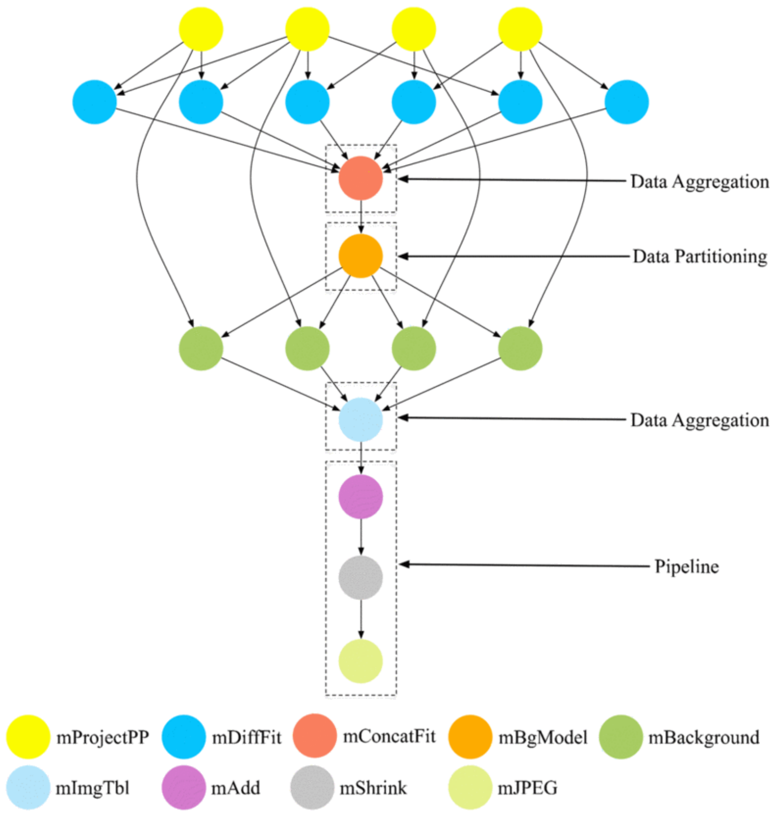


Figure 2: Montage workflow [BCD+08]

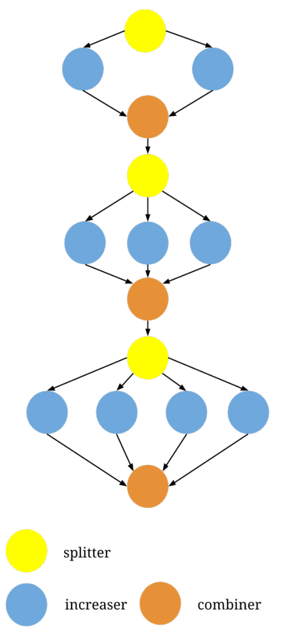
The workflow's size increases with the number of FITS input images. For example, a 2.0-degree Montage workflow consists of 312 input images and results in a total of 1442 jobs, while a 5.0- degree workflow already has 1718 input images and a total of 5657 jobs. Figure 2, shows the structure of the Montage workflow and its jobs.

**PGen**

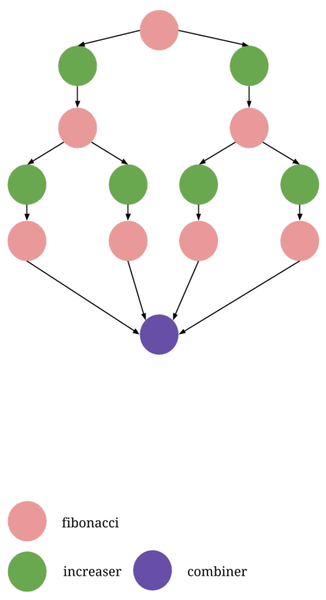
The PGen - large-scale genomic variation analysis workflow was built in order to perform efficient analysis on soybean data [WWF$^+$08]. It was modeled in order to study the genomic variations of plants using next generation re-sequencing data. It consists of 12 different levels, which is composed of numerous types of jobs. In contrast to the Montage workflow, the PGen is a lot more CPU demanding, although it can be scaled up to handle terabytes of data. The workflow takes for input a reference genome, chromosome names and multiple FASTQ files and outputs various sequence alignment data. Due to the fact, that using real input files, the PGen workflow can execute for days or even weeks, synthetic inputs were used to fasten up the overall execution.

**Custom workflows**

In this thesis, two custom workflows with different structures were also created, in order to see that not only the total communication is decreased when partitioning a workflow, but also if the overall execution time is reduced.



(a) Increaser workflow



(b) Fibonaci workflow

Figure 3: Custom workflows

Figure 3 shows the structures and the jobs of the created custom workflows.

The first workflow, called "fileIncreaser", is a simple benchmark workflow, that consists of as many number of levels as specified by the user. It takes a custom file for an input. The workflow splits the file into $n$ parts, depending on the level of the workflow, increases the split files by the specified size gotten from the user and combines them into a larger file. This workflow was modeled considering only data-transfer in mind and the time spent on computing is close to none.

The second "fibonacci" workflow, is a bit different than the "fileIncreaser". It takes a text file for input, that contains two numbers. The Fibonacci job adds these two numbers together, calculates the Fibonacci number from the result and outputs two files: a new text file consisting the next numbers to calculate and a dummy file, that has the size of the logarithm of the Fibonacci number calculated. The dummy file is created in order to create additional data communication between the nodes of the workflow. Each Fibonacci job is followed by two increaser jobs. The leftmost increaser job increases the numbers gotten from the Fibonacci job by *2/5* and the rightmost by *2/8.* These fractions were chosen randomly, in order to make one branch of the tree more computationally demanding than the other. The final job, combiner, just takes all the outputs from the last calculated Fibonacci jobs and outputs a dummy file of 50mb.

Both workflows are scalable, which means that with different parameters from the user, they produce different size workflows. For example, the Fibonacci workflow with 6 iterations consists of 126 jobs and produces only 749 Mb of data, while 10 iterations already has 2045 jobs and produces in total of 39GB amount of data.

## 2.3 Scientific Workflow management systems

As stated, there has emerged a paradigm to represent and model different scientific computations. In order to fully take advantage of it, i.e. ease the modeling, monitoring, and execution of a workflow, there has been built many scientific workflow management systems (SWfMS) over the years.

Most SWfMS are similar in their architecture and are composed of five layers: presentation layer, user services layer, workflow execution plan (WEP) generation layer, WEP execution layer and infrastructure layer [LPVM15]. According to M.Bux and U.Leser, SWfMS can be further divided to textual language based, graphical standalone systems and domain-specific web portals [BL13].

Textual language based SWfMS are systems, where the modeling, monitoring and execution of workflows are based on textual languages. They are greatly modifiable and meant to be run on different heterogeneous compute resources. For example, the Pegasus SWfMS, which is the main "execution engine" in this thesis, is a textual language based SWfMS. Workflows are usually described in a textual representation like XML, which can be written manually or modeled with the help of the SWfMS itself. Pegasus, for example, provides both Java, Python and Perl libraries in order to ease the modeling of scientific workflows. The orchestration and management of the workflow is also modeled and represented by various XML and text-based configuration files. The essence of textual language based SWfMS lies in highly efficient computing of high workloads [BL13]. In most cases, using textual language based SWfMS need a lot more configuration, set-up and understanding of the system itself in order to run a scientific workflow. For these reasons there has emerged graphical SWfMS.

Graphical workflow management systems focus on the ease of usage and graphical illustration of a workflow. Commonly these systems are installed on a server or a local client, ready to be used by a scientist. Workflows are typically constructed by a collection of supported components with drag and drop functionality, making the modeling rather easy. Due to the reason, that it is tedious to portray a large workflow graphically and to ease the overall modeling, most systems enforce their users to nest their workflows with sub-workflows. There exists several graphical SWfMS like Taverna [Tav], KNIME [KNI] or Kepler [Kep]. For example in Taverna, users can easily create, edit and run their workflows using Taverna Workbench. Just like Pegasus, it also supports different execution sites like grids and clouds.

The last category, domain specific web portals, consists of different online applications, where users can create, execute and share their workflows. As the class name suggests, these SWfMS are explicitly domain specific and do not support the execution of cross domain workflows. For example, portals like Mobyle [Mob] or Galaxy [BKC+10] are well known frameworks in the bioinformatics domain. The main strength of this class of SWfMS is the ease of sharing both of the results and the designed workflows in the community.

There are several core functionalities when speaking of SWfMS like fault tolerance, monitoring, data and task scheduling, etc. For instance, the reliability of a workflow's execution in the Pegasus SWfMS is achieved by reactive fault tolerance, i.e. after a failure, the jobs or data transfers are automatically retried. Another key functionality of a SWfMS is parallelization, i.e. the ability to exploit the resources available efficiently to execute a scientific workflow or its tasks in parallel. There are two distinct parallelism levels: fine-grained parallelism, which conveys the idea of executing different jobs in parallel and coarse-grained parallelism, which is the execution of sub-workflows in parallel [LPVM15]. Most of the named SWfMS above support some kind of parallelization, but they do not take advantage of hybrid parallelism, which is the combination of data, pipeline and independent parallelism techniques. For this reason, there exists a few SWfMS that take advantage of hybrid parallelization like Chiron [ODS+13], that allows a user to run a workflow in the most optimal way, when considering parallelization.

## 2.4 Cloud deployment

In the past, there has been developed a lot of different software to deploy an application to the cloud. While solutions of popular cloud providers like Amazon CloudFormation [Clob] or Azure Resource Manager [Man] provide better integration and with good configuration could be more efficient than multi-platform deploying systems, the deployed configurations cannot be easily migrated. Also, because platform specific deployment tools will make the customer dependent on their products, more and more scientists are interested in multi-platform deployment software, which mostly are open-source. It will help to solve the vendor lock problem and allow them to easily migrate their computing applications to different cloud providers. Although, one has to remember, that as in the cloud platform cases (OpenStack, Nebula), the lack of good customer service and the deficiency of satisfactory documentation, could make the usage of open source multi-platform deployment tools a bit of a hassle.

One of the popular multi-platform deployment tool is JClouds. It is an open source library, which is known to work with multiple of different cloud providers like Amazon or GoGrid. It has features like load balancing, easy customization etc. Because of the high configurability and accessible API, JClouds has also different frameworks built on top of them, like CloudMF (Cloud Modelling Framework) [FCR+13].

CloudMF, the multi-platform deployment framework used in this thesis (referred as CloudML in general), is a toolkit that enables users to deploy, provision and adapt multi-cloud systems and change them at run-time using Java code or JSON files. It was developed part of the EU FP7 REMICS project [REM].

The Cloud Modelling Framework is composed of two parts [FCR+13]:

*(i)* CloudML – Cloud Modelling Language, a domain-specific modelling language, developed in order to assist a user with the design of the deployment of multi-cloud systems. It allows the modelling of the machines to deploy, their external and internal relationships in the system, the set-up, etc.

*(ii)* models@run-time environment - a broker that is responsible for the migration and adaption of the described model by the user. It maps the modifications and updates of the running system back to the original model in run time.

For smaller projects, the configuration is rather easy. Users can define which kind of instances to provision on which clouds, what kind of scripts to configure after deployment, what bindings to define between the instances, etc. Listing 1 shows a snippet of the modeling of a system with CloudML with JSON 1.

While the configuration may seem easy for smaller projects, it becomes gradually harder with larger systems – one can see only during a full deployment if the configuration file is valid and does exactly what is needed.

Listing 1: Snippet of modeling a system with CloudML using JSON

```
"nodeTypes":[{
    "id":"mainNode",
    "os":"Ubuntu",
    "provider":"openstack nova",
    "compute":[2,4],
    "memory":[2048,4096],
    "storage":[40960],
    "location":"RegionOne",
    "sshKey":"big_duke_6",
    "securityGroup":"jaagup_pegasus",
    "privateKey":"jaagup_key.pem",
}],
"artefactTypes":[{
    "id":"pegasus",
    "retrieval":"wget http://../pegasus.sh",
    "deployment":"bash pegasus.sh",

}]
```

Setting up the configuration with files that are written in Java can also be a hassle, because it is more complicated to set up than with JSON and with several configurations, the code can get "ugly" really fast. Also one downside of using CloudML with Java is reliability, since if some unexpected error is thrown from the underlying library JClouds, it is hard do tap back to the current deployment model, if it is not backed up on every iteration.

Another alternative to the aforementioned framework CloudML, is an EU research project called SeaClouds [BIS+14]. It is also an open-source toolkit in order to deploy and manage multi-cloud applications. The main difference between the two is that SeaClouds is mainly focused on heterogeneous PaaS platforms, although it also supports the deployment and configuration of various IaaS infrastructures. A key functionality of SeaClouds is providing agility for a user after the initial migration. For example, SeaClouds will monitor the quality of service requirements, and in the case of a violation, the application or part of the application will be migrated to a different cloud vendor, that satisfies the initial requirements.

With the recent interest in containers, one could suggest also Docker[Doc] as an alternative to CloudML, which is an open platform application that is meant to run distributed systems on different infrastructures, including cloud. It has become popular because it is really lightweight and easily deployable. Docker packages your application into containers, which includes the application with its dependencies and shares the kernel with another containers.

Although it has gained a lot of interest because of the advantages stated before, it also has its downsides. It is not as stable as just using virtual machines for the reason that it is an ongoing project. Additionally, because containers run on the same kernel, setting up firewall rules and loading modules are difficult. One also has to remember, that using Docker alone does not provide a user with a unified way to deploy the containers to different infrastructures. In order to do that, users have to use external frameworks like Clocker [Cloa] or RightScale [Rig], which can cost money and need additional time to set up and configure.

## 2.5 Partitioning

Graphs are used in various different fields like biology, informatics, physics to convey and model different problems in a structured way. One of the fundamental algorithmic procedures, called graph partitioning, is the splitting of a graph into smaller sub-sets. The technique can bear many benefits like the overall communication minimization, reduction of the complexity of a graph, speeding up a route planning, among others. [BMS+16].

Because of these benefits, graph partitioning is utilized in numerous domains, e.g. it is extensively used in image processing, as means to partition the pixels of a picture into clusters that represent various objects. Another popular field to use graph partitioning is parallel processing, where the distribution of tasks to processors or machines can be more evenly dispensed, which in return results in a better load balance and the minimization of communication [BMS+16].

There are several approaches and algorithms to find a good partition for a graph including spectral partitioning, geometric partitioning, multilevel graph partitioning to name a few [KK98]. There are different toolkits implementing some of these algorithms, like Chaco [HL95] or PaToH [ÇA11], but in this thesis, the METIS library is used, which implements the multilevel k-way graph partitioning algorithm. It has shown to produce more quality partitions than for example different spectral partitioning schemes [KK98]. Another reason the METIS toolkit is used, is because of its speed, as graphs with millions of nodes can be partitioned in a matter of seconds, which is perfect as scientific workflows can be composed of a huge number of nodes.

The METIS library takes for an input a graph file, where on each line there is specified the nodes of the graph, its connections and weights. After execution, METIS will output a file with $n$ lines ($n$ is the number of vertices) and an integer value that shows to which group a node from the original graph belongs to.

In this thesis, in order to build a weighted graph that can be used with METIS, after the first execution, the log files of each task of a scientific workflow are analyzed. The weights of the edges in the graph are going to be the amount of data exchanged between the nodes in the workflow. For example, if task $A$ sends 1 megabyte of data to its adjacent task $B$, then the weight for the edge $e$ connecting nodes $A$ and $B$ is going to be 1. With this approach, one can build an undirected graph of the initial scientific workflow, where on the edges are the data communication sizes between the tasks of the workflow.

Figure 4 depicts the gain, that one can gain when using graph partitioning. The graphs tasks are partitioned between three machines using a random algorithm and the multilevel k-way partitioning algorithm. One can see, that the weights between the vertices in the random case will give a sum of 12 and in the METIS case only 9, meaning that the total communication between the machines in the cluster can be minimized when scheduling the tasks using graph partitioning.

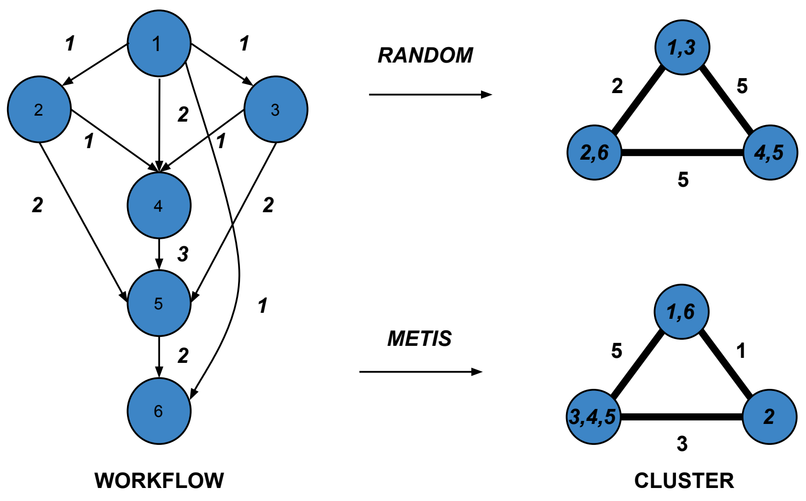


Figure 4: Partitioning gain with METIS

### 2.5.1 Multilevel k-way partitioning

The k-way partitioning problem is expressed as following.
Given a graph $G = (V, E)$ with $|V| = n$, partition $V$ into $k$ subsets $V_1, V_2, ...V_k$, such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, and $\cup_i V_i = V$, and the number of edges of $E$ whose incident vertices belong to different subsets is minimized [KK98].

The problem can be extended to graphs with weights on both of vertices and edges, which furthers the initial goal to partition the graph nodes into $k$ disjoint subsets, so that the total sum of the node weights in each subset is going to be the same, and the sum of the weights of the edges, whose shared vertices belong to a separate subsets is minimized [KK98]. Usually, the solution for a k-way partitioning of $V$ is expressed by a partition vector $P$ of length $n$, so that for $\forall v \in V$, $P[v]$ is an integer within 1 and $k$, showing which partition a vertex $v$ belongs to.

The multilevel k-way partitioning takes advantage of bisecting a graph $G$ using a multilevel algorithm. The main idea between a multilevel algorithm is pretty straightforward - decrease the size of the initial graph, calculate a bisection of it and project the results back to the original graph $G$.

In METIS, the partitioning consist of three distinct phases (Figure 5):

- Coarsening phase – The inital graph $G_0$ that is provided is coarsened down to a set of smaller graphs, i.e. it is remodelled into a sequence of graphs $G_1, G_2, ..., G_m$, such that $|V_0| > |V_1| > |V_2| > ... > |V_m|$ [KK98].

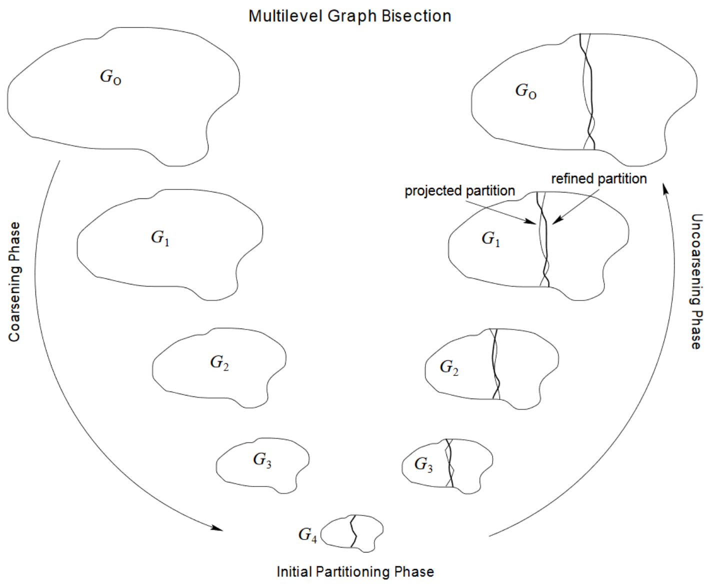


Figure 5: The different phases of multilevel k-way partitioning [KK98]

- Initial partitioning phase – After the graph is coarsened down, the k-way partitioning algorithm is used to divide the graph into several different parts.
- Uncoarsening phase – The final phase, where the coarsest graph is projected back to the initial graph, i.e. the partition $P_m$ is projected back to the initial graph $G_0$ and in each uncoarsening iteration, the graph is also refined with the k-way algorithm, so that the final partition will be the most precise.

In other words, the partitioning in the METIS toolkit of a graph $G$ into $k$ parts is done using the multilevel k-way partitioning, which consists of three different phases in order to refine the initial partitioning and produce higher quality results.

## 2.6 Summary

This chapter described the state of the art techniques used in this thesis. To recap, the cloud computing paradigm has allowed scientists to migrate and run their applications on top of a large number of resources in different parts of the world with an ease. Due to the rise of multi-cloud deployment tools, users are not constrained any longer by vendors and can switch seamlessly between different cloud providers. Furthermore, with the help of partitioning, the communication overhead of a scientific application running in the cloud can be reduced significantly. This allows the execution of various scientific applications on different SWfMS, across multiple clouds in an optimized way.

# 3 Related work

This chapter covers the related work that has been done regarding the scheduling of scientific workflows, the automatic configuration and set-up of a SWfMS and the execution time estimation of workflows.

## 3.1 Scheduling of scientific workflows

Extensive research about scheduling the tasks of scientific workflows in the cloud has been done, when considering numerous amount of different parameters: data transfer, cost, execution time, etc. For example, C. Lin and S. Lu proposed a SHEFT [LL11] scheduling algorithm, that takes into consideration the elasticity of the cloud. Their preliminary experiments showed that the SHEFT algorithm is better when considering execution time than a well known HEFT [THW02] algorithm, that is often used to reduce the makespan of a workflow. Their algorithm also enables resources to scale elastically during the execution, which the HEFT does not support. In this thesis, the elasticity of cloud is not considered, when scheduling tasks between virtual machines. Firstly because the SWfMS Pegasus does not support adding new execution sites (machines) dynamically during run time and secondly, because at the moment, the framework supports only a homogeneous set-up of virtual machines. When considering robust scheduling, that takes into account the cost and deadline constraints, then D. Poola et al., proposed a resource allocation model, that outperformed in most cases the similar ICPCP and GA algorithms [PGB$^+$14]. This thesis on the other hand focuses the scheduling of the workflow tasks by considering the data movement between the nodes and not cost, heterogeneity and neither deadline constraints.

There has been done a lot of research about scheduling workflows, when taking into account the data movement. J.Zhang et al., adopted a k-means clustering placement with a multilevel task replication strategy, in order to optimize the overall data transfer of a workflow between multiple data centers in the cloud [ZWL$^+$15]. Their simulations showed that both the data transfer time and the data transfer volume was reduced when using the proposed method. This thesis on the other hand focuses the scheduling of the tasks in a single data center, and does not take into account the locality of the data centers.

Using graph partitioning as the main strategy of a workflow scheduling has been done in [cKU11], [GHKS14] and [TT12]. In [cKU11] a multi-constraint hypergraph partitioning tool called PaToH [ÇA11] was enhanced with a heuristic called DPTA. They achieved up to 38% of improvement in the communication cost on the workflows gotten from Pegasus. In this thesis a multilevel k-way graph partitioning algorithm is used, that is implemented in the partitioning toolkit named METIS. In [GHKS14] the METIS toolkit was also used. It was compared to CPLEX and their own data placement heuristic, that does not use graph partitioning. It was shown that METIS is a good, fast and scalable toolkit for reducing the data communication of a workflow. While in the computational cluster they used the Hadoop distributed file system, in this thesis, each machine has its own files system and the sharing of data between the machines is done in a peer-to-peer manner.

In [TT12], M. Tanaka and O. Tatebe showed, that using multi-constraint graph partitioning for task assignation, can reduce the execution time up to 31% when compared to other scheduling techniques like round-robin or immediate task scheduling. In this thesis however, I show that the runtime of a workflow can be reduced occasionally, when using k-way partitioning as it is heavily dependent on the characteristics of a workflow. Furthermore, for their evaluation, a Pwrake parallel SWfMS was used, but in this thesis, the Pegasus SWfMS is the main system for workflow execution.

## 3.2 Time estimation

Estimating the runtime of workflows in different environments like grids or clouds has been a research topic for several years. In [HWW13] a framework was built to predict the performance of workflows in cloud environments. The work was mainly carried out using e-Science platform, but the authors state that the methodology can be applied to different systems as well. The time estimations were based on the amount of data consumed and produced by a component and the success of the predictions were dependent on the data relationships. In [MM12] T.Miu and P.Missier estimated the makespan of scientific workflows on their input features. They used C4.5 machine learning algorithm that took the data inputs of past experiments and showed when sufficient amount of historical data is present, the relative absolute error can be below 20%.

However in this thesis the time estimation is based on the work of [PJDS14], which calculates the estimation based on the structure of the workflow and does not need a lot of historical data. The makespan algorithm gotten from [PJDS14] is left unchanged, but a scaling factor based on the first actual run is introduced, in order to increase the accuracy of the estimation even further.

## 3.3 Configuration and set-up

In the past years, there has been proposed and developed many cluster management and deployment systems. While there are cloud configuration and automation systems like Chef [CHE], Ansible [ANS] or Salt [SAL], that do not have any restrictions on what kind of system or software to configure and deploy to the cloud, there are architectures that specifically focus on the deployment of WfMS and the execution of scientific workflows like [VHHLR15], [HWWC13] or [JD11]. In

[VHHLR15] K. Vukojevic-Haupt et al., present a multistep bootstrapping process, that allows a user to start a complex workflow middleware with minimal effort. They designed the "bootware" with multiple requirements like: "be lightweight", "be generic", "be robust", etc. The main feature of the approach is the independence of specific WfMS [VHHLR15]. This thesis however focuses only on the configuration and deployment of the Pegasus SWfMS on a cluster of machines using CloudML [GES$^+$11]. For validation in [VHHLR15] the SimTech SWfMS was set up and the provisioning of three virtual machines and the required middleware took approximately 17 minutes. In contrast, the deployment of six machines and their configuration with the built framework takes about 19 minutes on average or less ($\approx$ 7 minutes with a snapshot), depending on the Internet connection and the workflow specified by the user. Both frameworks described have support for different cloud environments like Amazon-EC2 or OpenStack.

In [JD11] an automated deployer for cloud environments called Wrangler was developed. Robustly speaking, it is similar to the software that was developed by K. Vukojevic-Haupt et al., as it supports multiple cloud environments, different WfMS, plugins, etc. On the other hand, both systems still need some kind of configuration input by the user, e.g, which kind of WfMS to install, its dependencies, the configuration between the machines using the default WfMS, etc. In this thesis, the user just has to select a vendor, virtual machines, provide a workflow for the framework, and the rest is done by the click of a button. One disadvantage also in [JD11], is that the agent software has to be pre-installed in the VM image beforehand, while no such requirement is needed for the framework developed in this thesis. Also the availability of the software of both K. Vukojevic-Haupt et al., [VHHLR15] and G.Juve and Ewa Deelman [JD11] is unknown, as no reference to the described software is provided in either paper. In contrast, the open-source framework developed in this thesis is available in [Vii17].

## 3.4 Summary

This chapter covered the related works performed regarding the topics of scheduling of scientific workflows, configuring and bootstrapping SWfMS and estimating the makespan of workflows.

All of the techniques used in the framework have been extensively researched before, but combining the different parts, i.e. partitioning, automatic set-up with CloudML and time-estimation [PJDS14] into an easy to use framework, has not been addressed before. The thesis tried to address this gap in the related work, which is discussed further in the next chapters.

# 4 Architecture overview

This chapter describes the underlying frameworks executing the workflows, modifications made to them and finally the system and its components built in this thesis.

## 4.1 Pegasus and Condor

As stated earlier, the SWfMS used in this thesis is Pegasus, which is widely used framework in the scientific community for the execution of workflows. Pegasus has a lot of features like reliability, error recovery, scalability, provenance to name a few [DVJ+15].

It consists of four main parts: *mapper, execution engine, job scheduler* and a *monitoring* component. The *mapper* takes an abstract workflow file in the format of DAX and maps the tasks onto the execution environment, while taking into account the needs specified by the user like the computational resources required or the software desired. The *execution engine* is self-explanatory and it is responsible for the running of the tasks in the workflow on both local and remote execution sites. The *job scheduler* is the manager of each task, i.e. it tracks their states, submits them to the queue, etc. The last subsystem, *monitoring component*, is responsible for the logging and monitoring of the executing workflows, allowing users to see the status of the current executing workflow. The component also saves valuable information to a workflow's database, e.g. performance and provenance information in order to debug failures more easier.

To execute a workflow with Pegasus, users have to have defined these four components:

*(i)* Abstract workflow file - the DAX file that consists of all the tasks that are needed to be executed. It can contain the next three components listed on its own, but to enforce readability and ease the management of a workflow, it is suggested to divide them into different parts.

*(ii)* Sites catalog file - Describes the different compute environment resources (clouds, grids, HPC, etc.) that the user wants to execute their workflow upon.

*(iii)* Transformation catalog - This component has to have all the software specified that are needed to run the tasks for each execution site specified in the site's catalog.

*(iv)* Replica catalog - Keeps a mapping of all the files that are needed to execute the workflow. For example, in the Montage workflow case, the replica catalog would contain all the initial FITS input images.

Without Condor, however, there would not be Pegasus. Condor DAGMan (Direct Acyclic Graph Manager) and most of the Condor stack is used to run the submitted workflows by Pegasus. Condor, the underlying engine, is a workload management system, that was built in order to execute compute-intensive tasks [TWML02]. Some of the key functionality of Pegasus is actually gained already from the Condor framework itself, for example, the support of heterogeneous systems or execution reliability.

Condor uses a powerful concept, called ClassAds, to match different tasks with the available resources based on requirements and preferences. According to T.Tannenbaum et al., it can be compared to the advertising in newspapers, because buyers define what and for how much they are willing to buy and sellers advertise what they have got for sale [TWML02]. In computing terms, machines in a Condor pool advertise their resources, for instance, CPU load, disk size, RAM, etc. and each job executed in Condor includes a ClassAd, containing the requirements to run the job, which helps to find a match in the pool of resources available to Condor.

The Condor architecture of a pool is quite straightforward (Figure 6) - a single machine, called the central manager, is responsible for the orchestration of jobs between rest of the machines. Each machine has different Condor daemons running, depending on their role in the pool. The different daemons are: *condor_master* (keep rest of the daemons running), *condor_startd* (advertises a machines ClassAd), *condor_starter* (starts a job on a machine), *condor_schedd* (stores jobs to a queue), *condor_shadow* (responsible for file transfers, logging, statistics), *condor_collector* (collects all information about a Condor pool), *condor_negotiator* (responsible for the negotiating of jobs within the pool) [TWML02].

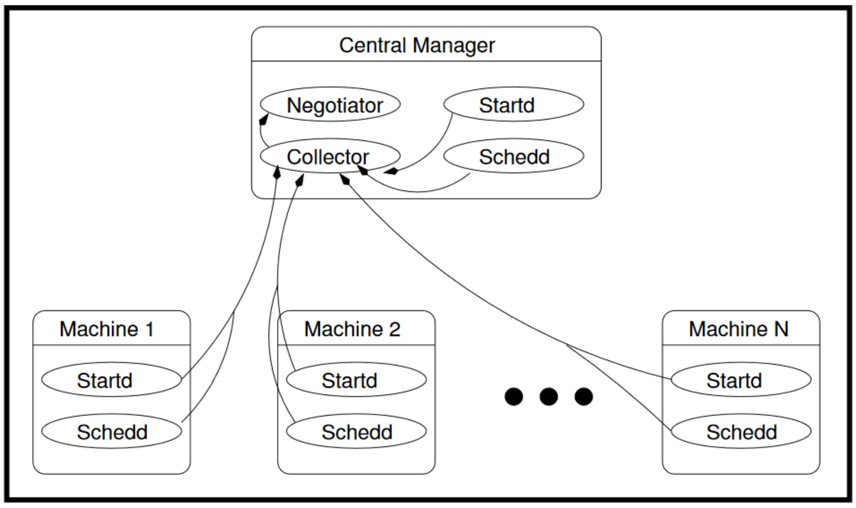


Figure 6: Layout of a Condor pool [TWML02]

## 4.2 Peer-to-peer approach with Mule

In Pegasus, there are different ways to share data between the execution sites, such as a shared file system, non shared file system (NFS), GlusterFS, etc. The most common solution in cloud is using a non-shared file system, meaning that every execution site that Pegasus has access to, has its own means of storing data.

One problem using this set-up, is that a Condor pool consists of a central manager. In this scenario, every bit of communication is sent through the manager, which is a bottle-neck in the overall system. When extending this problem to a scheduling context when using partitioning, one can see, that there will be no benefit gained in this scenario as it does not matter if a job is run on machine *1* or *2*, all the data transfers go through the central manager anyway (Figure 6).

To overcome this problem, the same solution as in [SV14] was used - enable the sending of data in the computational cluster in a peer-to-peer manner. This was done using Mule [AJD12], which was written as an additional library for Pegasus to allow machines to send data in a peer-to-peer way.

Mule is composed of three components: a replica index server, a cache daemon and a client [AJD12]. The replica index server stores all mappings of the files used by Mule. The cache daemon is run on every machine that runs a job with Pegasus, and stores all the data produced to the local disk. The client is an interface for the aforementioned replica server and cache daemons.

The peer-to-peer data sharing with Mule is achieved by storing every outputted file from a job's execution to the local cache daemon. If some other job, needs some files that is not available in its local cache, it will query the replica index server for a list of machines that will have that particular data. If a match is found, a machine is selected and the files are downloaded to the local cache daemon from the selected machine.

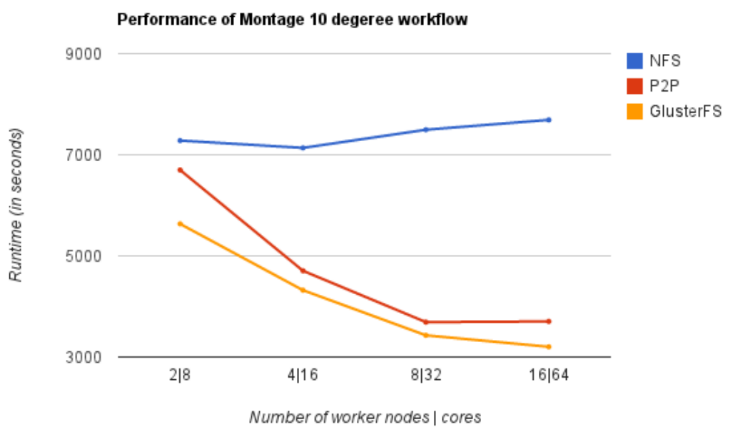


Figure 7: Execution times of Montage on different storage systems [AJD12]

After implementing Mule, the data communication was reduced in the cluster already by ≈ 50% [SV14]. Also, from the figure 7 one can see that the runtime of a workflow is also faster, when using peer-to-peer mode compared to simple NFS mode with a central manager.

Because of the reason, that Mule is not supported natively by Pegasus, the source code of *pegasus-transfer*, which is responsible for the sending of data between the machines, had to be modified, by adding a new handler called *MuleHandler*. After intertwining everything together, the set-up to execute scientific workflows on the cloud can be seen in figure 8.

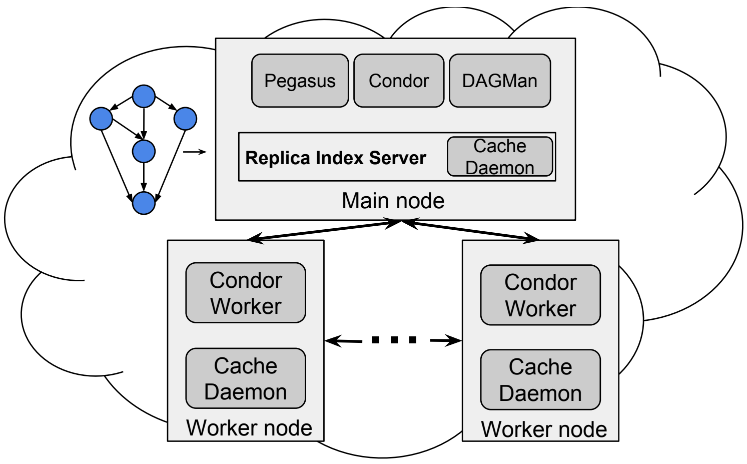


Figure 8: The current setup to execute a workflow in the cloud

## 4.3 CloudML additions

JClouds, the underlying library used by CloudML is evolving and developed rapidly. For this reason, some of the internal code of CloudML had to be tweaked, modified and updated, in order for it to work properly and efficiently again.

Some of the changes introduced to CloudML are the following:

- Execution of commands with responses - For the reason to execute commands via SSH and get the results back as a string, a function named *execCommandWithResponse()* was added. For example, one of the uses of this function were to periodically get the status of a workflow, using the output of *pegasus-status* and showing the result to the end user.

- Downloading of files - Additional function called *getFileStream()* was added in order to receive different log files to build the time estimation from Pegasus runs.

- Multiple amazon regions - Several functions regarding Amazon EC2 were changed to support different regions, as previously the region named ”eu-west-1” was hard coded in to CloudML.

- Private address to external component - Previously, the class *ExternalComponentInstance* only had fields for public IP addresses. For the overall system to work properly, the additional functionality to receive private IP addresses was added as it was necessary for the setup of Condor.

The cycle of modification, building and validation, were a bit cumbersome and slow, because the building of the CloudML project takes over $\approx$ 5 minutes on average, plus the additional time of provisioning and configuring the instances to verify that the changes that were made, were correct indeed.

## 4.4 The framework

The overall architecture of the framework built, can be divided into three main components: frontend, backend and the cloud (Figure 9). Frontend serves all the necessary templates for a scientist to start the overall process of setting up the machines, running the workflow, seeing current status, etc. The backend is responsible for all the communication between the frontend and the cloud, and finally, the cloud is used to execute the submitted workflow.

The backend can further be divided into three main parts: *deployer, workflow manager* and *estimator* (Figure 9).

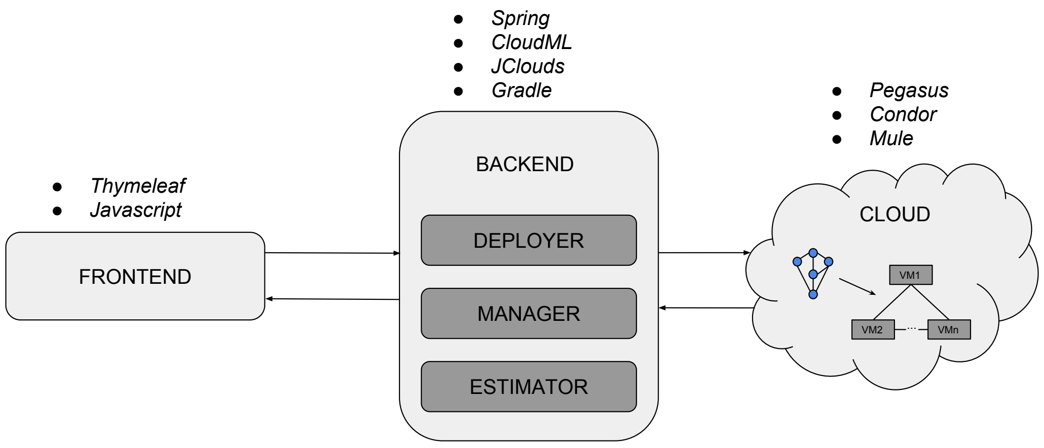


Figure 9: The architecture of the framework built

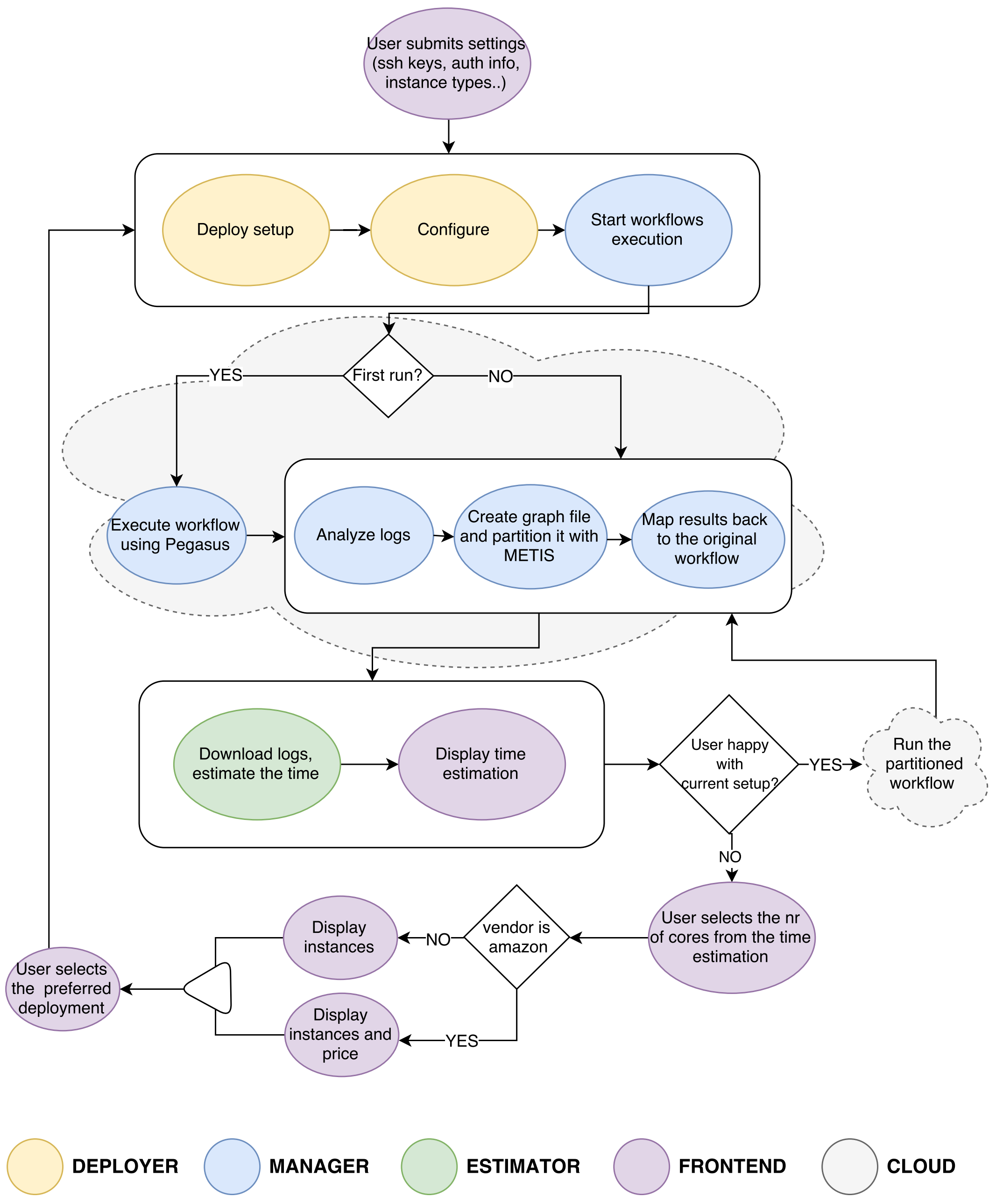
The *deployer* is the main engine that sets up the instances specified by the user either in Amazon EC2 or OpenStack. It works as following: spawn one instance and install the needed software (Condor, Mule, Pegasus, ..), run the user specified script on it, which installs the software needed to run their workflow and generates the initial workflow itself. Afterwards, clone the first instance and spawn the rest of the machines from the clone. If the user already has an image of the first instance, the initial installation steps are skipped and the machines are spawned right away. After the provisioning of the instances, they are configured, which involves changes in Condor configuration files, starting of the Mule cache daemons, etc. Finally, a JSON file is created with the information of the current set-up. The file can be used later to attach the deployment and continue the overall cycle (Figure 10).

The *workflow manager* is responsible for the execution, termination, restarting and partitioning of the scientific workflow. After deploying all of the instances, the workflow is automatically executed. When the workflow finishes the first run, it is partitioned between the amount of instances in the current cluster. This involves analyzing the logs from the first execution to collect the data transfers between the different jobs and the times taken to execute each individual job in the workflow. From the analyzed logs a graph file is constructed, which is fed into the METIS library. The output from METIS is re-mapped back to the original workflow, so that for each job an execution site where to run the task in the cluster is specified.

The last component, *estimator*, is used to calculate the time estimation of the scientific workflow. The estimation is computed also from the logs, using the makespan algorithm gotten from [PJDS14]. The results, algorithm and the additional scaling factor that was introduced based on the first execution are further discussed in subsection 5.2.

The overall cycle of using the framework can be seen in figure 10 - first a user submits the initial settings, which includes all the information about the deployment (SSH keys, number of instances, vendor, etc.) and the workflow to be executed. Next, the instances are provisioned using the *deployer*. After the set-up, the *manager* starts the execution of the workflow. If the workflow has not been run before, it is run for the first time, otherwise it skips straight to the partitioning. After partitioning the workflow, the *estimator* gets the log files from the cloud and calculates the time estimation.

In the frontend, the time estimation plot of the current workflow is displayed, where the $x$-axis shows the number of cores and the $y$-axis shows the amount of time in minutes it takes to execute the workflow using $x$ cores (Figure 13). Users can select various points from the plot and get price information regarding their selection (Amazon EC2 only). For example, if a user selects from the estimation plot a point $(x = 1, y = 120)$, all the Amazon instances are shown, which has one core. With each instance shown, the price for the experiment is also displayed, e.g. when selecting a t2.nano instance for the aforementioned point, the cost calculated would be \$0.0059*2hours=\$0.0118



User submits settings (ssh keys, auth info, instance types..)
Deploy setup
Configure
Start workflows execution
First run?
YES
NO
Execute workflow using Pegasus
Analyze logs
Create graph file and partition it with METIS
Map results back to the original workflow
Download logs, estimate the time
Display time estimation
User happy with current setup?
YES
Run the partitioned workflow
NO
User selects the nr of cores from the time estimation
vendor is amazon
NO
Display instances
YES
Display instances and price
User selects the preferred deployment
DEPLOYER
MANAGER
ESTIMATOR
FRONTEND
CLOUD

Figure 10: The overall cycle of the process

After selecting the instances that seem suitable to run the workflow under a certain amount of time and the cost is reasonable, the overall cycle is started again, i.e. deploying the new machines, configuring them, executing the workflow, etc.

The screen-shots of the framework in some of its different cycles can be seen in the figures 11, 12 and 13.

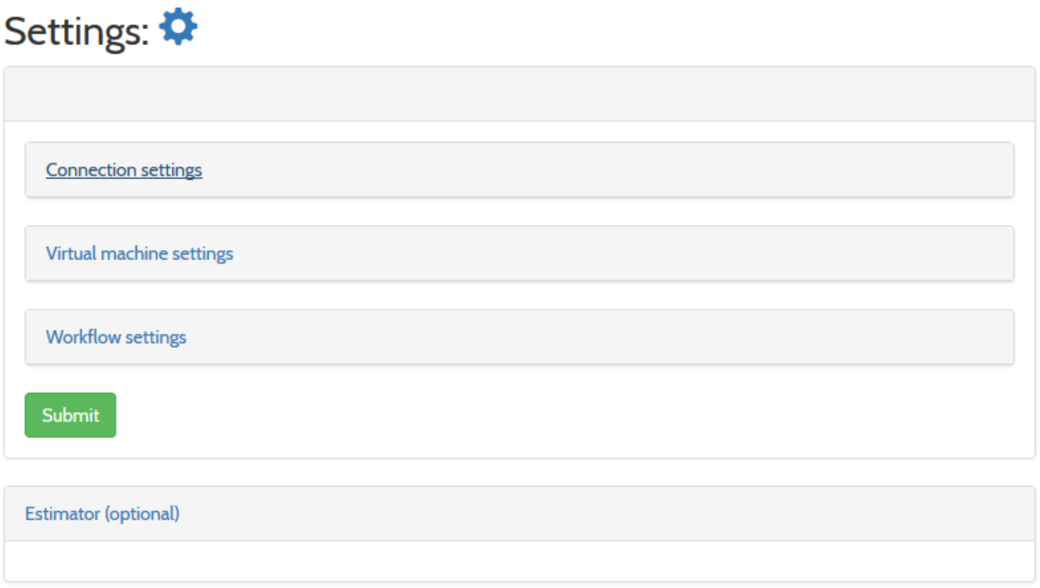
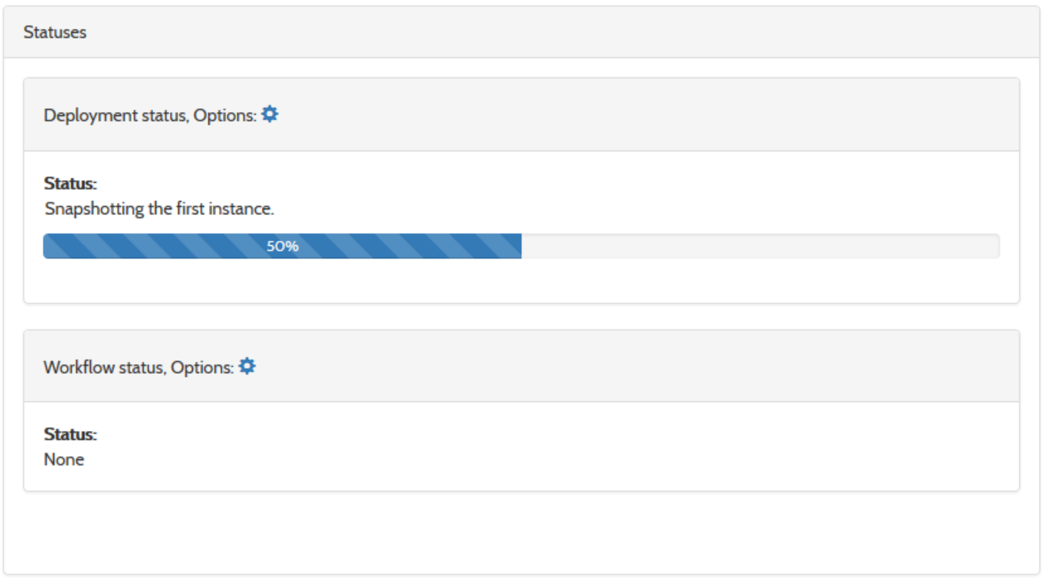


Figure 11: The settings page


Statuses

Deployment status, Options:

**Status:**
Snapshotting the first instance.

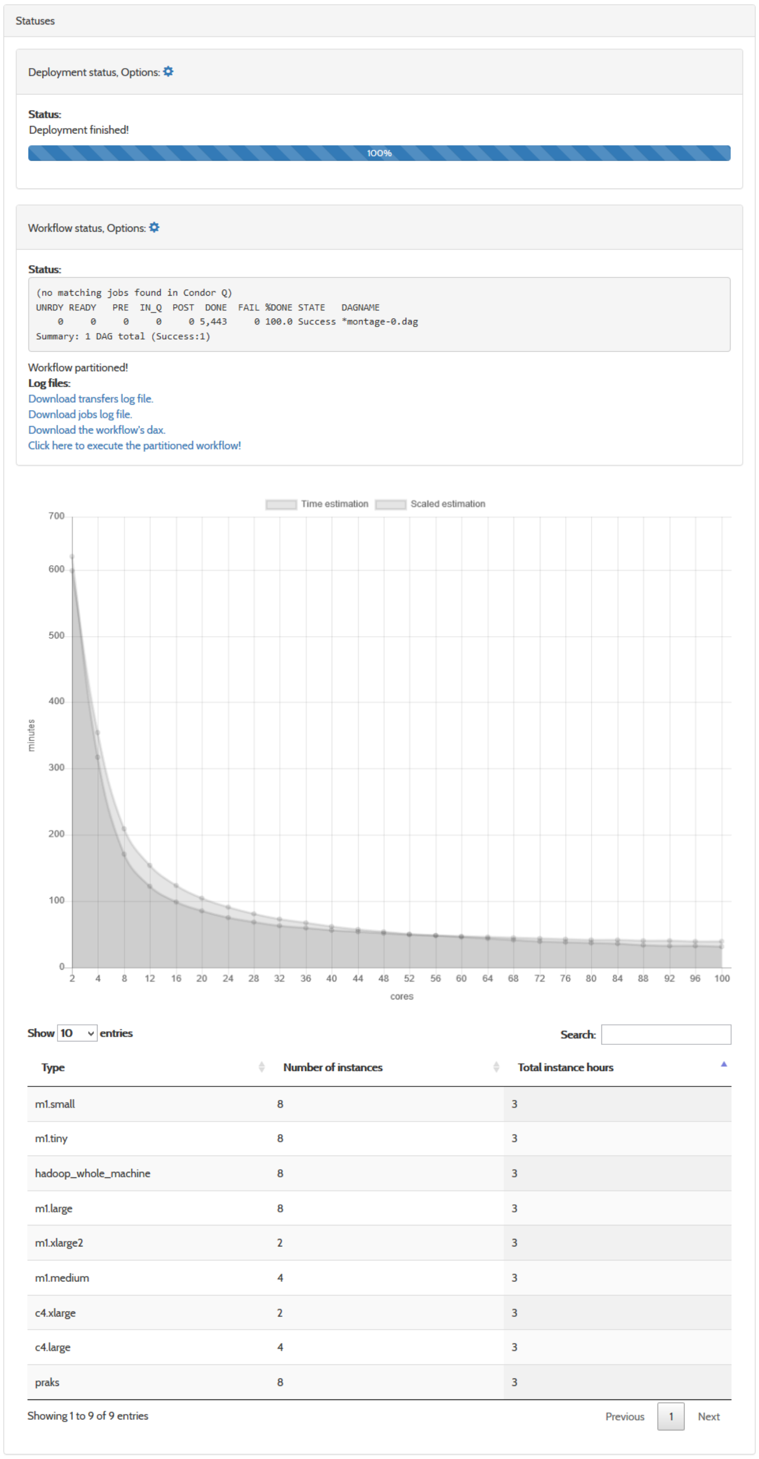
50%

Workflow status, Options:

**Status:**
None

Figure 12: The deployment of a system

Figure 11 shows the different sub-settings a user has to select to start the initial deployment. Figure 12 illustrates the set-up of a cluster and figure 13 shows the state of the overall cycle after the deployment, initial execution of the workflow and clicking the point *cores=8* on the time estimation.



Deployment status, Options:

**Status:**
Deployment finished!

100%

Workflow status, Options:

**Status:**

```
(no matching jobs found in Condor Q)
UNRDY READY   PRE  IN_Q  POST  DONE  FAIL %DONE STATE   DAGNAME
    0     0     0     0     0 5,443     0 100.0 Success *montage-0.dag
Summary: 1 DAG total (Success:1)
```

Workflow partitioned!
**Log files:**
Download transfers log file.
Download jobs log file.
Download the workflow's dax.
Click here to execute the partitioned workflow!

**Show** 10 **entries** **Search:**

| Type | Number of instances | Total instance hours |
|---|---|---|
| m1.small | 8 | 3 |
| m1.tiny | 8 | 3 |
| hadoop_whole_machine | 8 | 3 |
| m1.large | 8 | 3 |
| m1.xlarge2 | 2 | 3 |
| m1.medium | 4 | 3 |
| c4.xlarge | 2 | 3 |
| c4.large | 4 | 3 |
| praks | 8 | 3 |

Showing 1 to 9 of 9 entries    Previous 1 Next

Figure 13: The state after clicking the point *cores=8* on an estimation

## 4.5 Summary

This chapter described the framework built in this thesis, which uses the Pegasus SWfMS with a peer-to-peer library in order to execute the scientific workflows in the cloud. The built framework consists of three main parts: deployer, manager and the estimator. The deployer sets up the machines in the cloud, the manager executes the workflow with Pegasus and partitions it, while the estimator estimates the execution time of the workflow. After the initial execution, users can run the partitioned workflow, or if not satisfied, spawn a new deployment consisting of different virtual machines and run the already partitioned workflow on top of it.

# 5 Evaluation

This chapter shows the benefits gained when scheduling a scientific workflows tasks using multilevel k-way partitioning, the accuracy and the algorithm of the time estimation and the amount of time it takes to deploy a whole system to different clouds.

## 5.1 Scheduling gains

As shown in the illustrative figure 4, the main benefit of scheduling of the tasks in the cluster using METIS is the decrease of the inter-instance communication while increasing intra-instance communication, i.e. the reduction of data transfers between the machines.

Previously, it has been shown that the data transfer of a workflow can be reduced up to ≈ 70% [SV14], using the aforementioned technique. While the previous work was focused on a single case (Montage), in this thesis, several other workflows have been used (Section subsubsection 2.2.1).

Figure 14 and 15 show the data transfer differences of two custom workflows, when running them in a cluster consisting of eight c4.xlarge machines (4 cores, 7Gb RAM), when using a random algorithm and METIS for task scheduling.

One can see, that the communication is decreased significantly, when using graph partitioning. For example, in the Fibonacci workflows case, the data transfer is reduced by astonishing ≈ 95%, which means that the cost of running the experiment is also reduced. For instance, Amazon charges $0.01 per GB of data transfer in and out to EC2 when using a public IPv4 address.

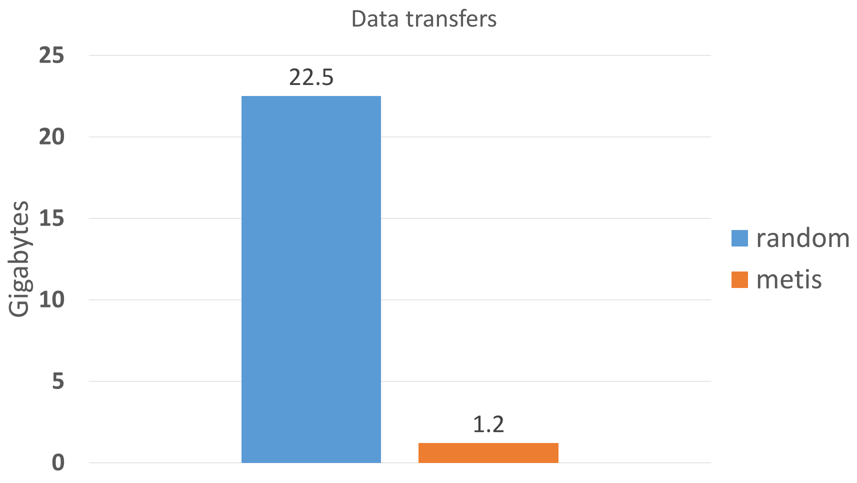


Figure 14: Fibonacci workflow data transfers

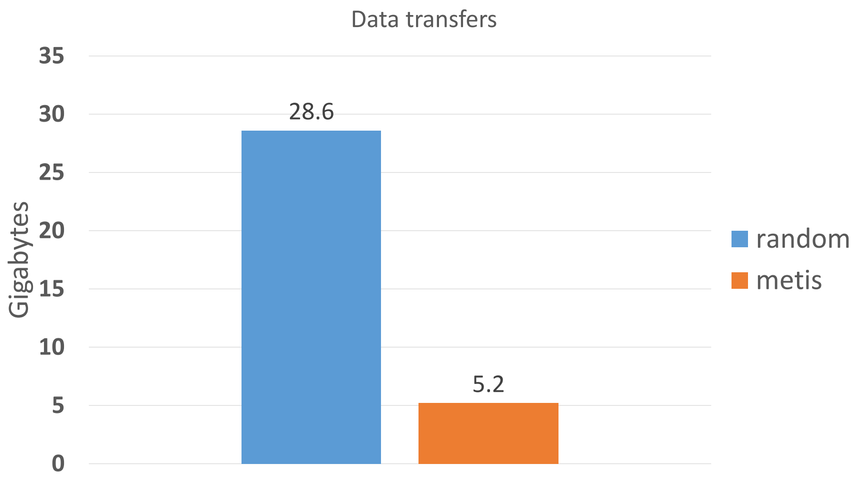


Figure 15: Increaser workflow data transfers

An additional goal was to find out, if the *k-way* partitioning could also reduce the overall runtime of an experiment, meaning that the cost associated with it, could be reduced even further. To evaluate this, two different setups were used: six m1.medium (2core, 4GB RAM) instances for the SoyBean workflow (two 153 MB synthetic inputs with a synthetic 1GB reference file) and eight c4.xlarge instances for the Montage workflow (5.0-degree 2mass).

As anticipated, the partitioning can not guarantee every time the reduction of the makespan of a scientific workflow. Figures 16, 17 illustrate this clearly, where

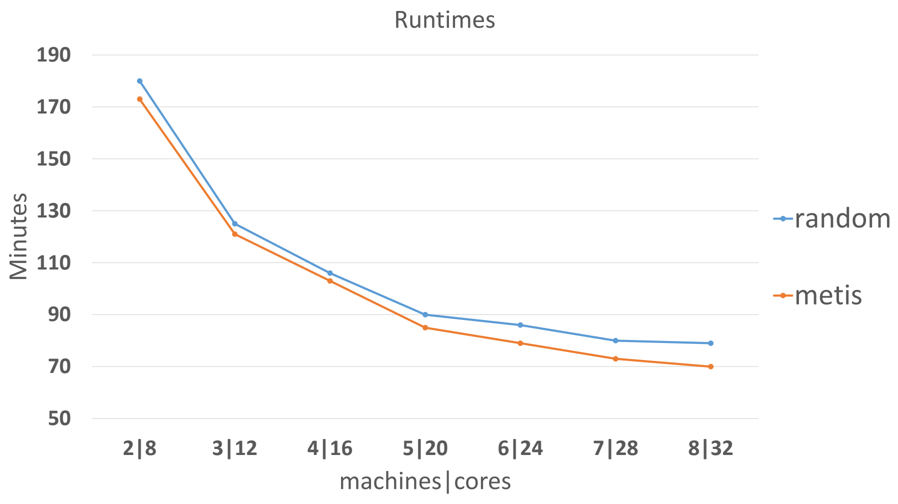


Figure 16: Montage workflow runtimes

in the Montage workflows case, the runtime is faster when using METIS, but in the Soybeans case, it is indeterminable, meaning that the reduction of the execution time is heavily dependent on the characteristics of the workflow.

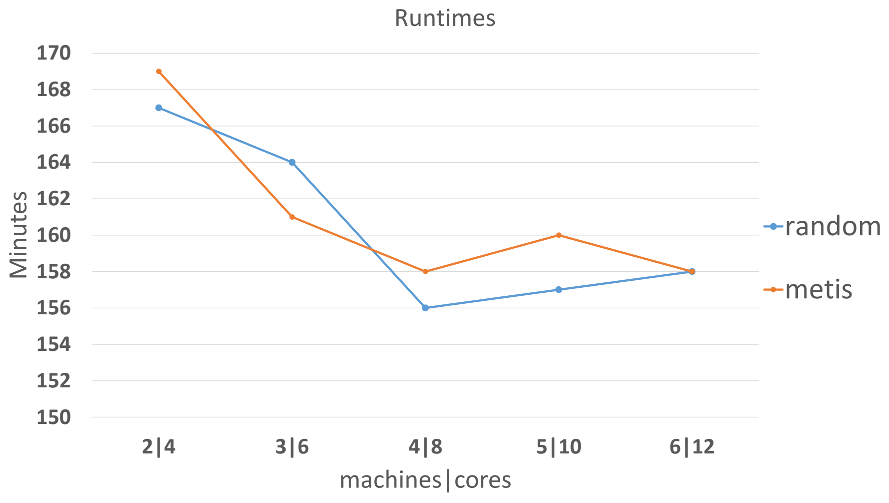


Figure 17: Soybean workflow runtimes

The Soybean workflow is a lot more CPU intensive than Montage, meaning that the scheduling of tasks, whilst taking into account data transfers of the workflow, can not ensure the decrease of the execution time (the same observation was made on the Fibonacci workflow case). Although it is clear, that when a workflow is I/O heavy, like Montage, which means it does not spend a lot of time on each tasks CPU-wise, but its data, the makespan of the workflow can be reduced indeed, as seen in figure 16.

Another gain of partitioning, is the decrease of the overall input and output operations in the computational cluster. This is directly correlated to the reduction of the data transfers, i.e. the fewer data transfers are made between different machines, the fewer unnecessary I/O operations in the cluster are executed.

In order to verify this, the Fibonacci and the 5.0-degree Montage workflow were run in a cluster consisting of eight c4.xlarge machines, while measuring the I/O operations of the experiments. In the Fibonacci experiment, the total I/O operations made without the partitioning was *494137*, while with METIS it was *395814*, which is a difference of *98323* (19%) operations. In the Montage workflows case, the numbers were respectively *710271* and *525250*, resulting in *185021* (26%) lesser I/O operations. To illustrate, Amazon EC2 [Ama] charges $0.05 per 1 million I/O requests. These expenses can grow significantly, when using workflows that run for weeks or even months, but with the help of partitioning, these costs can be easily reduced.

## 5.2 Time estimation

As discussed, one of the key functionalities of the framework is the runtime estimation of a workflow. This gives a scientist a base estimate, how many resources one has to provision in order to finish an experiment under a certain amount of time.

The pseudo-code to calculate the runtime of a workflow based on the number of cores provisioned, is as following [PJDS14]:

Listing 2: Time estimation calculation pseudocode

```
function calculateMakeSpan(cores, levels, jobs) {
   total_makespan = 0;
   for (level : levels) {
      jobs = level.getJobs()
      total_runtime, max_runtime = 0
      max_cores = jobs.size()
      for (job : jobs) {
         job_runtime = job.getRuntime()
         total_runtime += job_runtime
         if (job_runtime > max_runtime) {
            max_runtime = job_runtime
         }
      }
      makespan = max((total_runtime/min(cores,max_cores)),max_runtime)
      total_makespan += makespan;
   }
   return total_makespan;
}
```

The workflow jobs are grouped into levels, using a top-down approach, when traversing the graph of the workflow. For instance, the Montage workflow consists only of 9 different levels, even though it could have tens of thousands of jobs. For each level, the makespan is calculated separately with the equation:

$$makespan_{level} = max(\frac{total\_runtime_{level}}{min(cores, max\_cores)}, max\_runtime) \qquad (1)$$

In essence, the makespan of a level, is either the amount of time it takes to execute the longest job ($max\_runtime$) or the sum of all the job execution times in the level ($total\_runtime$) divided by the amount of cores provisioned.

The algorithm produces quite good results on its own, e.g, in 92 out of 95 experiments conducted by Ilia.Petri et al., the error was less than 20% [PJDS14]. To fine-tune the results even more, I introduced a scaling factor, which uses the execution time of the first initial run.

In order to scale up the estimation, a function to represent the line had to be found. I plotted the time estimations gotten from the makespan algorithm and drew log-log plots, which resulted more or less in a straight line, indicating that the estimation lines could be represented by a function: $log(y) = a * log(x) + b$, which simplifies to a polynomial $y = x^a * e^b$, where $a$ is the slope and $b$ is the intercept of the curve. After finding the function of an estimation, the slope was modified so that the estimation curve scales up and passes through the initial runtime point.

I conducted two experiments, one in Amazon, using c4.xlarge instances and the other in OpenStack, using m1.xlarge (4core, 4GB RAM) machines to see the accuracy of the makespan algorithm and the scale up.

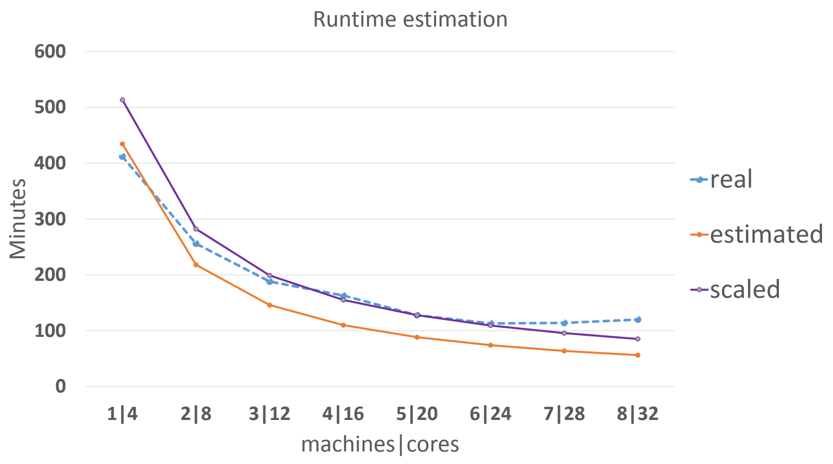


Figure 18: Makespan estimation of Fibonacci workflow

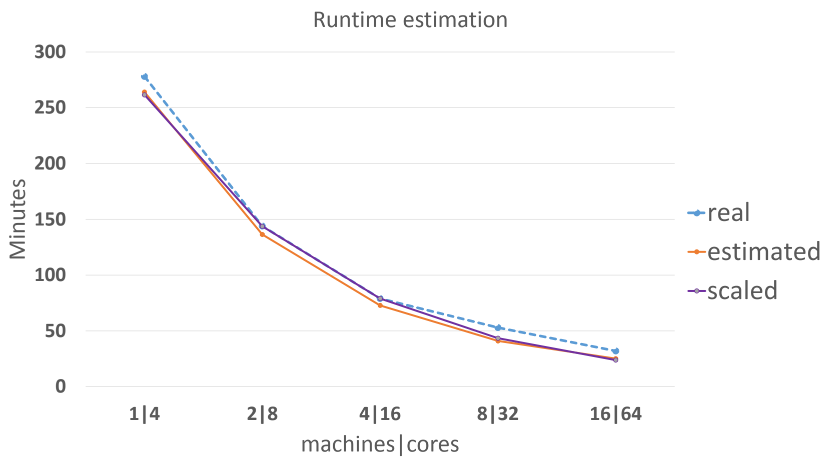


Figure 19: Makespan estimation of 5.0-deg. Montage workflow

The results of the experiments can be seen in the figures 18 and 19, where in both cases, the estimation accuracy of the scaled runtime was better than the baseline estimation. The error % for each point, when compared to the real run for both of the experiments can be seen in the following tables:

Table 1: Error % for the Montage runtime estimations

| **machines—cores** | **Base estimate error %** | **Scaled error %** |
|---|---|---|
| 1—4 | 5 | 5.9 |
| 2—8 | 5.2 | 0.18 |
| 4—16 | 7.8 | 0 |
| 8—32 | 22.6 | 18 |
| 16—64 | 21.5 | 25.4 |
| **Average error** | **12.4%** | **9.9%** |

Table 2: Error % for the Fibonacci runtime estimations

| **machines—cores** | **Base estimate error %** | **Scaled error %** |
|---|---|---|
| 1—4 | 5.4 | 24.5 |
| 2—8 | 14.8 | 10.2 |
| 3—12 | 22.3 | 5.8 |
| 4—16 | 32.5 | 4.7 |
| 5—20 | 30.9 | 0 |
| 6—24 | 34.4 | 3.2 |
| 7—28 | 43.9 | 16 |
| 8—32 | 29.7 | 28 |
| **Average error** | **29.7%** | **11.6%** |

In these experiments, the results were scaled using the center point of the graph, which resulted in the most accurate result. For example, when scaling the Fibonacci workflow up, using the point $x = 1|4$ (Figure 18), the average error is 29.2%, which is basically the same as the baseline estimation. During the real deployment, the point that is going to be used for the scaling is taken from the last execution of the workflow.

It is quite obvious, that these numbers can vary a lot due to various reasons, such as cloud latencies, the choice of the scaling point, the workflow logs used for the baseline estimation, to name a few. This means that the estimation results should be taken as a rough guideline for finding the number of cores to provision to a workflow.

## 5.3 Deployment times

Another interest was to find out the total duration of the deployment, i.e. the time span of "bootstrapping", and see if it can be sped up. For example, the deployment of 8 c4.xlarge instances from a pre-made image in Amazon EC2 took in total approximately 24 minutes with the framework, which was too long.

After a quick investigation, the problem lied in the configuration. Out of the 24 minutes, it took about 13 minutes to configure the eight instances. This was due to the reason, that for every separate configuration command, a new SSH connection was established. While this approach improves reliability and debugging, it took too long to execute these commands sequentially. The solution for this problem was executing the configuration commands all at once, for every instance. This approach reduced the configuration time in this case from 13 minutes to approximately 4 minutes.

Table 3 and 4 show the deployment times of six t2.micro instances (1core, 1GB RAM) in Amazon and OpenStack, with and without a snapshot. From the tables, one can see that there are big differences with the deployment to private and public clouds. For example, the software setup takes an astounding 10 minutes longer in our local SciCloud [SBV10] than in Amazon EC2. This is due to the fact, that the downloading of the required software in Amazon is a lot faster. On the flip-side, some of the deployment steps, like image creation or configuration is faster in the SciCloud. Nevertheless, on average, the deployments took approximately the same amount of time, i.e, in Amazon it took without and with an image $(18+9)/2 \approx 13$ minutes to deploy, which is roughly the same as in OpenStack, $(22+5)/2 \approx 13$ minutes.

The overall deployment time can vary significantly because it is mostly dependent on the amount of instances launched and the workflow script provided. For example, in the Montage workflow configuration script, one could download all the input files to the local disk, increasing the total duration of deployment remarkably.

Table 3: Deployment times without snapshot

| **Deployment step** | **Amazon (time)** | **OpenStack (time)** |
|---|---|---|
| First instance deployment | 0:01:57 | 0:00:51 |
| File uploads | 0:01:15 | 0:00:55 |
| Software setup | 0:02:37 | 0:12:37 |
| User script setup | 0:00:05 | 0:00:05 |
| Image creation | 0:03:59 | 0:01:18 |
| Rest of the instances creation | 0:05:08 | 0:05:18 |
| Configuration | 0:03:17 | 0:00:59 |
| **Total time** | **0:18:18** | **0:22:03** |

Table 4: Deployment times with snapshot

| **Deployment step** | **Amazon (time)** | **OpenStack (time)** |
|---|---|---|
| First instance deployment | 0:00:00 | 0:00:00 |
| File uploads | 0:00:00 | 0:00:00 |
| Software setup | 0:00:00 | 0:00:00 |
| User script setup | 0:00:00 | 0:00:00 |
| Image creation | 0:00:00 | 0:00:00 |
| Rest of the instances creation | 0:05:58 | 0:04:27 |
| Configuration | 0:03:31 | 0:01:13 |
| **Total time** | **0:09:29** | **0:05:40** |

## 5.4 Summary

This chapter outlined the various experiments that were conducted in different settings, in order to see the benefits of partitioning, the accuracy of the time estimation and the configuration and set-up times of the framework. As found, partitioning has many benefits like the reduction of I/O operations, the decrease of data communication in the cluster and the occasional minimization of the execution times of a workflow.

The workflows makespan estimation algorithm of [PJDS14] was found to be quite accurate on its own, but after scaling it up using an existing execution point, the results were even more accurate. To bring an example, the error percent of the runtime estimation of the ”fibonacci” workflow was reduced from 29.7% to 11.6%.

Finally, as anticipated, the configuration and set-up times of a deployment with the built framework, were different on different clouds. For example, the software setup in Amazon took only $\approx 3$ minutes, while the same step took approximately 10 minutes longer in our local SciCloud. Even with some major differences in the configuration steps, on average, the deployment times (without and with a snapshot) in OpenStack and Amazon were roughly the same (about 13.7 minutes).

# 6 Conclusion

In this thesis, an open-source framework was developed in order to ease the process of deploying, running and partitioning a scientific workflow in the cloud with Pegasus SWfMS.

The framework provisions and configures the selected resources automatically into multi-cloud environments (OpenStack or Amazon) using CloudML. After the initial execution of the workflow, it is partitioned using the METIS toolkit.

Partitioning helps to decrease the data communication in the cluster and occasionally shortens the execution time, reducing the overall cost of the experiment. For example, the thesis observed that in some cases, the data transfers in the computational cluster were reduced up to $\approx 95\%$. Furthermore, the total I/O operations made in the cluster, was also reduced. To illustrate, running a 5.0-degree Montage workflow with partitioning, the amount of operations made was decreased by 26%, which is hugely beneficial, considering that Amazon EC2 charges $0.05 per 1 million of I/O transfers made.

Additionally, a workflows time-estimation algorithm gotten from [PJDS14], was introduced to the framework, in order for a scientist to have an approximate idea how many resources to provision for a certain scientific workflow. The outcome from the algorithm was scaled up using the initial execution point, in order to produce even more accurate results. For instance, the initial time estimation was improved in some cases by 18.1%. Also, a cost factor based on the time estimation was implemented into the framework, so that the user has a better overview on how many machines and at what cost to provide to a certain workflow, when taking into account the execution time.

The overall time of deploying and configuring a system with the framework was heavily dependent on the amount of machines specified, the cloud environment used and the workflow provided. To illustrate, the overall process to provision and configure six t2.micro instances without a snapshot took 18 minutes in Amazon, but 22 minutes in OpenStack.

All in all, the framework helps a scientist to deploy and configure the needed resources to the cloud with an ease, execute the workflow in an optimal manner and provide a time/cost based guideline for the future executions.

## 6.1 Future research directions

Regarding the future work, there is much to be done and researched. The major part of this thesis is based on the multilevel *k-way* partitioning, which has shown promising results. One area of interest is to try different partitioning techniques to see if there are big differences in the end results. Also, right now the partitioning is done while taking into consideration only the data transfers of the jobs. Another approach could be adding also the CPU load of the tasks, to distribute the jobs among the cluster even more accurately.

Development wise, the framework could benefit from several modifications like the support of additional cloud vendors, e.g. Google Cloud, making the overall process more reliable (Figure 10), even friendlier user interface, etc. Furthermore, one key functionality worth adding, would be the support for a heterogeneous setup among multiple clouds. Currently, all the machines deployed to the cluster are the same type, running in the same cloud.

The time-estimation of a scientific workflow could also be improved. Instead of constructing the baseline estimation from a single execution, to increase accuracy even further, multiple runs should be incorporated. Additionally, the scaling up of the workflow can be enhanced using several data points.

# References

[ABJ$^+$04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[AJD12] Rohit Agarwal, Gideon Juve, and Ewa Deelman. Peer-to-peer data sharing for scientific workflows on amazon ec2. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 82–89. IEEE, 2012.

[Ama] Amazon. Amazon elastic compute cloud (amazon ec2). Available: http://aws.amazon.com/ec2/. visited (06.04.2017).

[ANS] ANSIBLE. https://www.ansible.com/. visited (11.04.2017).

[Azu] Microsoft Azure. https://azure.microsoft.com/en-us/. visited (11.04.2017).

[BCD$^+$08] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[BIS$^+$14] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. Seaclouds: A european project on seamless management of multi-cloud applications. *SIGSOFT Softw. Eng. Notes*, 39(1):1–4, February 2014.

[BKC$^+$10] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

[BL13] Marc Bux and Ulf Leser. Parallelization in scientific workflow management systems. *arXiv preprint arXiv:1303.7195*, 2013.

[Blu16] Amy Blumenthal. How isi's pegasus helped scientists make the discovery of a century. Accessible: https://viterbi.usc.edu/news/news/2016/isi-gravitational-waves-software-pegasus.htm, February 2016. visited (22.04.2014).

[BMS$^+$16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[ÇA11] Ümit Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[CHE] CHEF. https://www.chef.io/solutions/cloud-management/. visited (11.04.2017).

[cKU11] Ümit V. Çatalyürek, Kamer Kaya, and Bora Uçar. Integrated data placement and task assignment for scientific workflows in clouds. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 45–54, New York, NY, USA, 2011. ACM.

[Cloa] Clocker. The docker cloud maker. http://www.clocker.io/. visited (02.05.2017).

[Clob] Amazon CloudFormation. https://aws.amazon.com/cloudformation/. visited (02.05.2017).

[Doc] Docker. https://www.docker.com/. visited (02.05.2017).

[DVJ$^+$15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[FCR$^+$13] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with cloudmf. In *Proceedings of the Second Nordic Symposium on Cloud Computing &#38; Internet Technologies*, NordiCloud '13, pages 38–45, New York, NY, USA, 2013. ACM.

[GDE$^+$07] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *Computer*, 40(12), 2007.

[GES$^+$11] Glauco Goncalves, Patricia Endo, Marcelo Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mangs. Cloudml: An integrated language for resource, service and request description for d-clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 399–406. IEEE, 2011.

[GHKS14] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. Distributed data placement to minimize communication costs via graph partitioning. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, page 20. ACM, 2014.

[GJC$^+$11] Robert Graves, Thomas H Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, et al. Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics*, 168(3-4):367–381, 2011.

[HL95] Bruce Hendrickson and Robert Leland. The chaco users guide: Version 2.0. Technical report, Technical Report SAND95-2344, Sandia National Laboratories, 1995.

[HMF$^+$08] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 640–645. IEEE, 2008.

[HWW13] Hugo Hiden, Simon Woodman, and Paul Watson. A framework for dynamically generating predictive models of workflow execution. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, pages 77–87. ACM, 2013.

[HWWC13] Hugo Hiden, Simon Woodman, Paul Watson, and Jacek Cala. Developing cloud applications using the e-science central platform. *Phil. Trans. R. Soc. A*, 371(1983):20120085, 2013.

[JCl] Apache JClouds. https://jclouds.apache.org/. visited (22.04.2017).

[JD11] Gideon Juve and Ewa Deelman. Automating application deployment in infrastructure clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 658–665. IEEE, 2011.

[Kep] Kepler. https://kepler-project.org/. visited (08.04.2017).

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KNI] KNIME. http://www.knime.org/. visited (22.04.2017).

[LL11] Cui Lin and Shiyong Lu. Scheduling scientific workflows elastically for cloud computing. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 746–747. IEEE, 2011.

[LPVM15] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[Man] Azure Resoure Manager. https://azure.microsoft.com/en-us/features/resource-manager/. visited (02.05.2017).

[MG$^+$11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[MM12] Tudor Miu and Paolo Missier. Predicting the execution time of workflow activities based on their input features. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 64–72. IEEE, 2012.

[Mob] Mobyle. http://snap.hpc.ncsu.edu/cgi-bin/mobyle/portal.py. visited (08.04.2017).

[Mon] Montage. An astronomical image engine. http://montage.ipae.caltech.edu.

[ODS$^+$13] Eduardo Ogasawara, Jonas Dias, Vitor Silva, Fernando Chirigati, Daniel de Oliveira, Fabio Porto, Patrick Valduriez, and Marta Mattoso. Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341, 2013.

[PGB$^+$14] Deepak Poola, Saurabh Kumar Garg, Rajkumar Buyya, Yun Yang, and Kotagiri Ramamohanarao. Robust scheduling of scientific workflows with deadline and budget constraints in clouds. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 858–865. IEEE, 2014.

[PJDS14] Ilia Pietri, Gideon Juve, Ewa Deelman, and Rizos Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, WORKS '14, pages 11–19, Piscataway, NJ, USA, 2014. IEEE Press.

[Pla] Google Cloud Platform. https://cloud.google.com/. visited (06.04.2017).

[REM] REMICS. Reuse and migration of legacy applications to interoperable cloud services. Available: http://www.remics.eu/.

[Rig] RightScale. https://www.rightscale.com/. visited (02.05.2017).

[SAL] SALT. https://docs.saltstack.com/en/latest/topics/cloud/. visited (11.04.2017).

[SBJV11] Satish Narayana Srirama, Oleg Batrashev, Pelle Jakovits, and Eero Vainikko. Scalability of parallel scientific applications on the cloud. *Sci. Program.*, 19(2-3):91–105, April 2011.

[SBV10] Satish Srirama, Oleg Batrashev, and Eero Vainikko. Scicloud: scientific computing on the cloud. In *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, pages 579–580. IEEE Computer Society, 2010.

[SIV16] Satish Narayana Srirama, Tverezovskyi Iurii, and Jaagup Viil. Dynamic deployment and auto-scaling enterprise applications on the heterogeneous cloud. *memory*, 1024:2048, 2016.

[SV14] Satish Narayana Srirama and Jaagup Viil. Migrating scientific workflows to the cloud: through graph-partitioning, scheduling and peer-to-peer data sharing. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pages 1105–1112. IEEE, 2014.

[Tav] Taverna. Powerful, scalable, open source & domain independent tools for designing and executing workflows. https://taverna.incubator.apache.org/. visited (18.04.2017).

[THW02] Haluk Topcuoglu, Salim Hariri, and 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.

[TT12] Masahiro Tanaka and Osamu Tatebe. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 65–72. IEEE, 2012.

[TWML02] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Beowulf cluster computing with linux. chapter Condor: A Distributed Job Scheduler, pages 307–350. MIT Press, Cambridge, MA, USA, 2002.

[VHHLR15] Karolina Vukojevic-Haupt, Florian Haupt, Frank Leymann, and Lukas Reinfurt. Bootstrapping complex workflow middleware systems into the cloud. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pages 126–135. IEEE, 2015.

[Vii17] Jaagup Viil. Cloud partitioning tool. https://bitbucket.org/JaagupViil/cloud-partition-tool, 2017.

[WWF$^+$08] Jonathan D Wren, Dawn Wilkins, James C Fuscoe, Susan Bridges, Stephen Winters-Hilt, and Yuriy Gusev. Proceedings of the 2008 midsouth computational biology and bioinformatics society (mcbios) conference. *BMC bioinformatics*, 9(9):S1, 2008.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[ZWL$^+$15] Jinghui Zhang, Mingjun Wang, Junzhou Luo, Fang Dong, and Junxue Zhang. Towards optimized scheduling for data-intensive scientific workflow in multiple datacenter environment. *Concurrency and Computation: Practice and Experience*, 27(18):5606–5622, 2015. cpe.3601.