
# D12.4 - Orchestration of applications packaged as containers

| | |
|---|---|
| Deliverable No.: | D12.4 |
| Deliverable Name: | Orchestration of applications packaged as containers |
| Contractual Submission Date: | 30/04/2019 |
| Actual Submission Date: | 30/04/2019 |
| Version: | v1.0 |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730897.

| *COVER AND CONTROL PAGE OF DOCUMENT* | |
|---|---|
| Project Acronym: | **HPC-Europa3** |
| Project Full Name: | Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing |
| Deliverable No.: | D12.4 |
| Document name: | Orchestration of applications packaged as containers |
| Nature (R, P, D, O): | R |
| Dissemination Level (PU, PP, RE, CO): | PU |
| Version: | v1.0 |
| Actual Submission Date: | 30/04/2019 |
| Author, Institution:<br>E-Mail: | Gadrat Sébastien, CNRS<br>sebastien.gadrat@cc.in2p3.fr |
| Other contributors | Justin Bussery, Guillaume Cochard: CNRS<br>Giuseppa Muscianisi, Simone Marocchi: CINECA<br>Raül Sirvent: BSC |

**ABSTRACT:**
This deliverable is providing a status report of the use of Singularity containers within a workflow manager. The main goal will be to highlight potential constraints when running a workflow of containerized applications. A few workflow managers will be selected and evaluated, then a real use case will be fully implemented.

**KEYWORD LIST:**
High Performance Computing, HPC, Workflow Managers, Orchestration, Containers, Singularity

| *MODIFICATION CONTROL* | | | |
|---|---|---|---|
| **Version** | **Date** | **Status** | **Author** |
| 0.1 | 25/04/2019 | Draft | Gadrat Sébastien (version for internal review) |
| 0.2 | 17/04/2019 | Draft | Gadrat Sébastien (revision including comments from reviewer) |
| 0.3 | 23/04/2019 | Draft | Gadrat Sébastien (minor updates) |
| 1.0 | 26/04/2019 | Final | PMT (consolidated document) |

*TABLE OF CONTENTS*

## Executive summary

Scientific communities require more and more computing and storage resources to fulfil their missions. They should optimize and automate as much as possible the data processing, ensuring the higher computing resources usage efficiency. They also have to ensure reproducibility and portability of this processing, especially when using several HPC centres.

Automation can be achieved using so called orchestration tools, and more specifically workflow managers. Reproducibility and portability can be achieved with the use of containers. Workflow managers are software tools which have been used in the HPC communities for a while, and several tools already exist. Some are old and mature, some others are younger and still under active development. Container technology is still young, and is not yet widely spread among HPC centres.

In this document we will evaluate the possibility of using containers in order to satisfy reproducibility and portability within workflow managers, which allow the data processing automation.

We will focus on Singularity as the container tool, as it is more dedicated and adapted for computing processing than other container technology such as Docker. In order to evaluate the ease and the potential constraints of managing Singularity containerized applications, we will test four workflow managers: FireWorks, Makeflow, Parsl and Pegasus.

We will show that managing Singularity containers using workflow managers is relatively straightforward, even if the workflow manager doesn't provide native support. Singularity support can indeed be easily implemented within the workflow managers thanks to the use of wrappers.

Any workflow managers can then be chosen, and this choice will only be driven by the features required by the users (and more specifically by the data processing constraints).

To go further and complete this study, we will test and implement two real scientific use cases: the first one is based on Solid State Physics, and the second one on Astronomy. The first one has been fully implemented, using a Singularity container containing all the required softwares, and has been executed up to the final physical result.

# 1 Introduction: Containers Orchestration and Workflow Management

There are multiple software tools one could think of to "manage" containerized applications. In this introduction, we will briefly discuss them, point out pros and cons, in order to define the required features for such tools in a HPC environment.

Looking for tools to manage containers (let's take Docker and Singularity containers as references [1]), one could think of, at least, these three categories of applications: containers orchestrators (such like the very well-known Google tool Kubernetes [2] or Apache Mesos [3]), workflow managers (one of the most known being Pegasus, but a lot of them actually exist), and even a more recent kind of tool usually referred to as function-as-a-service (there are a few of them, like Apache OpenWhisk [4]).

Containers orchestration is commonly misinterpreted as workflow management. Workflow managers allow to run a chain of various applications taking into account constraints at each step of that chain (just like input and output files, output files from one step being the input ones of a further step). They should also ensure that all tasks that can be run in parallel are done so, to be able to use computing power efficiently. Containers orchestration, on the other hand, usually refers to the possibility provided by a system to spawn on the fly more containerized applications, usually the same ones, in order to scale up and meet the current demand. It is also used to manage a container life cycle. This is typically of great use in micro-services architecture. In HPC constraints have to be satisfied at each step of the workflow, which means that containers will have to communicate. Function-as-a-service tools could also deal with the constraints being discussed here. They are

actually made for that: when a giving set of conditions are met, the system applies a given processing (the so-called function). If this processing part can be translated into a "function" (as required by the tool) then the tool will satisfy our requirement.

Another point one has to take into account for choosing the right tool is that HPC sites usually rely on a batch system to submit jobs on the computing cluster (currently the batch system being the most used in the HPC world appear to be Slurm, but Torque or PBS for instance are other good options). This adds a new requirement on the kind of tools we are looking for. These tools should indeed be able to communicate with the batch system. Because of that, only the workflow manager tools fulfil the needs. These are the tools, or at least a few of them, that we will study later on.

It could be noted that, conceptually, the combination of an orchestration tool together with a function-as-a-service tool (let say Kubernetes with Apache Open Whisk) could work and meet the needs as discussed above on a cloud-based infrastructure. However, one other thing one should deal with is the proper translation of a processing step (as in HPC) into a function. For a function-as-a-service tool, the function to be defined has to satisfy some constraints. This kind of tool being commonly used in micro-services architecture, it is not obvious one could translate all kind of HPC processing tasks into proper functions.

In the following part, part 2, we will discuss the tools evaluation process and the few tools that we have evaluated. Not being able to cover them all in the time available for this work, we will also present and discuss in that part the reasons of the tools selection we have done. Then, in part 3, we will complete the evaluation of the chosen tools by implementing one real workflow: a Solid State physics workflow based on commonly used software in that scientific area. We also planned to look to the Large Survey Synoptic Telescope one, but due to constraints on the software stack being currently developed, we could not implement it. We will then just discuss the potential difficulties that may arise from the description given in the Conceptual Design Report [5]. In the last part of this document, we will summarize the evaluation of the tools discussed in this report, and give some perspectives. It is worth noticing that this report gives a kind of state-of-the-art status of the various tools presented at the moment of writing it. Some perspectives will be given to illustrate that some incoming or expected features could modify the evaluation of these tools. Most of these tools are still under development and improvements are foreseen, new interesting features being, or would be, added.

# 2 Workflow Managers

## 2.1 Tools Evaluation

As discussed above, in the current state-of-the-art, the tools we have to use to manage containerized applications in HPC are workflow managers. In order to evaluate each tool, and to be able to compare one to each other, we will have to strictly follow the same evaluation procedure. This procedure will consist of testing and evaluating the most important and relevant aspects of the tool, taking into account the HPC infrastructure constraints.

We have then defined a set of important aspects that we will evaluate, and each of them will serve as a metric. We have grouped these metrics into 3 categories:

- Tool Description
- Tool Utilisation
- Main Features

The 'Tool description' will describe the tool and discuss its general characteristics. It will include some technical aspects such as the programming languages used to develop it, how easy one can install it, especially taking into account possible constraints due to dependencies, and its interoperability with batch systems (in particular the one commonly used in the HPC world). A second part will focus on the tool users and developers' community, documentation and the available discussion channels. Finally, we will also take into account the

state of the development, the probability to see new features added and the long-term support of the tool. All those aspects are indeed important to consider when selecting a tool.

The 'Tool Utilisation' will discuss the various ways provided by the tool to define a workflow, how to manage its complexity and to handle the data. This part will then give some hints about the tool flexibility and its capabilities to manage complex workflow and to handle constraints. The way the tool handles the workflow definition and execution, the easier, the better.

The 'Main Features' part will go deeper into technical, though crucial, aspects. We will show how the tool handles errors, and which monitoring features are available. These aspects are important to detect errors, fix them, and eventually resume the workflow (which assumes that nothing which has been previously done is lost). We will also discuss the support of containers. In our case, we will focus on Singularity, as it is currently the preferred solution for computing. This part is quite important as one of the main objectives of this task is to define whether or not the use of containers add some new constraints in managing workflows of (containerized) applications. All workflow managers do not provide native Singularity support, but a simple workaround will be to write a Singularity wrapper, which will add the container support to the tool (see part 2.7 for more details).

## 2.2 Tools to be Evaluated

There are a lot of such tools. Most of them have initially been developed at a HPC centre to provide a satisfactory solution for their users' demands and needs. In our scope, the final objective is to fully implement two real workflows using different workflow managers in order to get a real experience with these tools.

As stated earlier, in the Description of Work for this task, the LSST workflow is one of the real use cases that we will implement. The LSST collaboration has done a survey about these tools [6]. We decided to start from that survey, to add a few other well-known tools, and then select a few of them according to relevant criteria. The main criteria we took into account are the following:

- no (or few) dependency(-ies), easy to install;
- tools based on commonly used language and thus easy to use and to learn;
- tool maturity, well-known and frequently used among HPC centres.

Although these criteria may look reasonable for such a tool, none was found satisfying them all. One of the tools satisfying the most the first criterion is Makeflow. Makeflow is a set of C code to compile that basically requires nothing else to work. For the second criterion, we chose FireWorks and Parsl. They are both quite similar: they are written in Python, and are easy to install via pip install in a Python installation. As Python is a commonly used language, a lot of people are used to it which ensure that the tools could be quite easy to use. FireWorks however also requires a MongoDB database for the monitoring part. FireWorks is well-known of researchers in Genomics for instance, and Parsl is used by some US HPC centres, and appears to be a good tool candidate for LSST. Finally, none of these three tools could be said mature enough and frequently used in the HPC around the world, so we also chose to evaluate Pegasus which is a long-standing tool, which looks very powerful, frequently used, with dedicated developers and an important community. However, this tool fully relies on HTCondor, is written in Java, and seems more complex and exhibit a steeper learning curve compared to the others.

At the end, none of these workflow managers will satisfy all these criteria, but evaluating several will allow us to get an overall good overview of which features such a tool can provide.

## 2.3 Makeflow

### 2.3.1 Tool description

Presentation

Makeflow is a part of a set of tools, named Cooperative Computing Tools or cctools written by the Department of Computer Science and Engineering (or more specifically the Cooperative Computing Lab) at University Notre Dame, Indiana, USA [7]. These tools help to build a full computing environment for high performance computing. There are 10 main tools, from storage virtualization to access various file systems in a transparent way (Parrot and Chirp), resources monitoring (Resource Monitor), workflow and work queue systems (Makeflow and Work Queue), and more. These tools can be used on all the different computing environment: HPC clusters, grids and clouds.

This set of tools is open source, and can be downloaded from the website of the Cooperative Computing Lab [8] (to get stable releases) or from GitHub in one wants to contribute to the development [9]. Once the tools (written in C code) are downloaded, one will need to compile them (the classic triplet configure / make / make install). If necessary one can also compile the code with optional libraries in order to add more features (such as support for dedicated storage).

Once compiled, Makeflow will be available through command lines.

Community

As said above, these tools are developed and maintained by the people of the Cooperative Computing Lab of University of Notre Dame. They are currently 10 people working on these tools (plus other collaborators from that University). Makeflow main page shows papers discussing the tool [10]. But one can find many more papers about the other tools as well. There are also regular cctools and cc Lab Workshops [11]. The documentation is good and up-to-date, and includes a Beginner Manual. The documentation also provides workflow examples to get started quickly. However more advanced features or recent ones are less documented.

Finally, the team also provides support to people using these tools thanks to a Google Group [12]. This Google group is active and the developers are quite present.

Makeflow supports a large number of batch systems and cloud based systems:

- batch systems supported: HTCondor, SGE and relative systems, PBS, Torque, SLURM;
- cloud based systems: Mesos, Kubernetes, Amazon (EC2 and Lambda);
- we can also submit either on a local machine or configure a generic cluster, for the latter solution, there is another tool that can be helpful: Work Queue [13].

In this document, we will focus on batch support only, and more precisely to SGE. But it is quite interesting to note that Makeflow can natively submit tasks to both batch systems and cloud based ones, including kubernetes support.

### 2.3.2 Tool Utilisation

Workflow definition

Basically, a workflow definition will be written in a Makefile like syntax [14]. The definition will then consist of a set of rules. Each rule specifies a set of output files to create, a set of input files needed to create them, and a command that generates the target files from the source files. For tasks that do not need to be run on a compute node, one can add the keyword LOCAL before the command specifies in the rule: this will then run the task where Makeflow is running.

For instance, a simple “hello, world!” example will be executed as follow. The content of the hello_world.mf file will be as follow (this file stands for defining the workflow, which here is reduced to one simple task):

```
hello.out:
	echo "hello, world!" > hello.out
```

The first line gives the output files that will be produced by the task on the left, and the input files required on the right, both separated by a column ‘:’. Note that in the file above there is no input file required by the task, i.e. the command. Another point which should be highlighted is that the command should be indented (at least one space is required) in order to be correctly parsed by Makeflow.

```
$ makeflow -T sge hello_world.mf
```

In this quite simple case, we have submitted to a SGE like computing cluster (which is our local cluster).

This Make like syntax is however not well suited to define more complex tasks which will require additional operations. It will also become quite verbose when one will have to deal with a lot of input and output files. For such cases, Makeflow also supports the JX language [15], a JSON eXtended language which allows to define more complex workflow in a programmable way.

Workflow execution

Once the workflow has been defined, it can be submitted as described above.

From the machine the workflow is started, some useful information will be displayed.

For instance, if we keep the previous example, here is what we get:

```
parsing hello_world.mf...
local resources: 32 cores, 128653 MB memory, 134988 MB disk
max running remote jobs: 100
max running local jobs: 32
checking hello_world.mf for consistency...
hello_world.mf has 1 rules.
starting workflow....
submitting job: echo "hello, world!" > hello.out
submitted job 3071514
job 3071514 completed
nothing left to do.
```

This display allows to follow step-by-step what Makeflow is doing. One interesting information is that the job number given by SGE is given for each job submitted, which allows to gather more information by checking the status of the jobs directly from the batch system (which can also help to debug).

Makeflow also provides more tools to follow the workflow running, tools which will be discussed later on in the ‘Monitoring tools’ part.

Data management

By default, Makeflow copies all the required input files to the worker nodes where the task will run, and copy back the output files. That’s the main reason why it is crucial to correctly define all input and output files in the workflow definition (see the discussion above). Without the correct definition, the workflow will indeed fail because one or more input files will be missing.

It is also usual to have a shared file system mounted in the computing cluster (such as NFS or HDFS like file systems). In this case, copying the data may not be relevant, and Makeflow provide a simple way to handle that, via a command line option.

A last useful advanced feature which should be reported is the garbage collector provided by Makeflow [16] which will take care of deleting all temporary files created by a task. It will also ensure that the input and output files are preserved, thus allowing the system to resume the task in the case it failed.

### 2.3.3 Main features

Singularity support

Makeflow natively supports containers such as Singularity and docker [17]. One only has to add the right option at the command line, in our case a --singularity (with the full path to the container) since we will focus on that container.

One should notice that, since we pass a container through the command line, that implies that the same container will be used for each step of the workflow, which is not necessarily what we want.

Moreover, as we will discuss it later on with the Solid State use case (see 3.1), with this option Makeflow creates a Singularity wrapper (close to what we wrote for Parsl and FireWorks, see 2.7). But this wrapper will always be the first to get executed, and thus we could not run first an environment wrapper we needed to set up the proper working environment [18]. This issue has been discussed with the developers who are aware of that constraint, which should be fixed in a next release.

Error handling

Makeflow writes a transaction log file in the current directory to keep track of all jobs submitted (start time, end time, status). In case of failure, a directory will be created, containing relevant files to debug. The transaction log file, usually named [name of the definition file].makeflowlog will also allow Makeflow to resume the workflow once the error has been identified and fixed. The information displayed in the shell or that one can find in the various log can usually point out the problem. If needed, since Makeflow returns the job number of each job submitted one can also retrieve relevant information from the batch system.

Monitoring tools

Makeflow does not provide real time monitoring tools. However, it provides some command line tools that can convert a workflow definition into a graph or plot the number of jobs according to their states over time (mainly ready, running and completed). Although their features are useful, the only real time monitoring will be the various messages that Makeflow will display on screen (just as seen on the previous page).

## 2.4 FireWorks

### 2.4.1 Tool description

Presentation

The official FireWorks presentation states that:

FireWorks is a free, open-source code for defining, managing, and executing workflows. Complex workflows can be defined using Python, JSON, or YAML, are stored using MongoDB, and can be monitored through a built-in web interface. Workflow execution can be automated over arbitrary computing resources, including those that have a queueing system. FireWorks has been used to run millions of workflows encompassing tens of millions of CPU-hours across various application areas and in long-term production projects over the span of multiple years. An academic paper on FireWorks is also available. For details, see [19] for the tool homepage and for the reference paper [20].

Some features that distinguish FireWorks are dynamic workflows, failure-detection routines, and built-in tools and execution modes for running high-throughput computations at large computing centres.

This graphical interface is really what makes FireWorks stands out from the others small, specialized workflow managers. Indeed, it allows to easily monitor and survey the workflows.

FireWorks is a simple and light tool. There is no real strong dependencies and deployment constraints to handle with.

The requirements are only python (2 or 3) and mongoDB. During the setup, a lot of Python dependencies will be installed (Flask, matplotlib, etc.). Regarding MongoDB, it can be installed on a separate server than the one running FireWorks. It is also easy to configure, and is required for the monitoring part.

Development and community

The Github repository is still active and reported bugs are fixed pretty quick. Despite that, it seems that only a small team (1 or 2 people) are working on the project; we are not really sure if, apart from the bug fixing, the development of the product itself is still active or not.

There is a Google Group [21]. Answers are globally fast but there is only one person answering.

The documentation is not as good as you would expect. Lot of difficult cases are left unexplained and the learning curve is pretty steep. The syntax we had to use for FireWorks is quite dense and it is required to understand the product logic before being able to use it efficiently. Some of the features are less or not documented and it takes some time before we can implement them.

Tests have been made on the CC-IN2P3 computation farm which uses SGE. The following systems are currently supported:

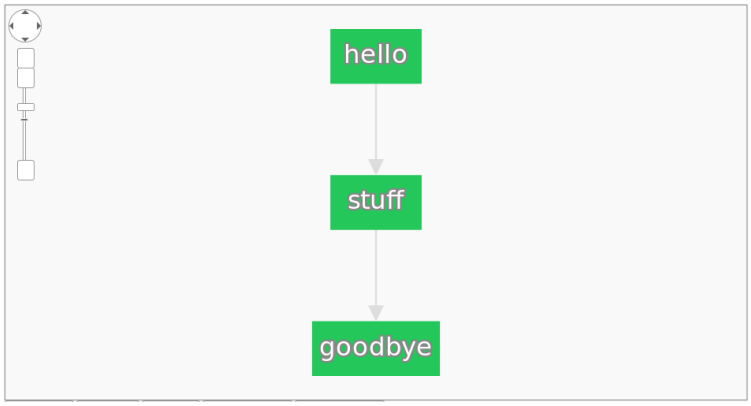
- PBS/Torque
- Sun Grid Engine
- SLURM
- IBM LoadLeveler

FireWorks is working well with SGE. There is, however, a lack of documentation regarding the setup of FireWorks to be able to use it on the computation grid.

### 2.4.2 Tool Utilisation

Workflow definition

The most basic workflow is a linear one. Here is an example of one of those:




The corresponding FireWorks code is the following:

```
from fireworks import Firework, Workflow, LaunchPad, ScriptTask, PyTask
from fireworks.core.rocket_launcher import rapidfire

# set up the LaunchPad and reset it
launchpad = LaunchPad()

# create the individual FireWorks and Workflow
fw1 = Firework(ScriptTask.from_str('echo hello'), name="hello")
fw2 = Firework(ScriptTask.from_str('echo stuff'), name="stuff")
fw3 = Firework(ScriptTask.from_str('echo goodbye'), name="goodbye")
wf = Workflow([fw1, fw2, fw3], {fw1: fw2, fw2: fw3}, name="Basic pipeline workflow")

# store workflow and launch it locally
launchpad.add_wf(wf)
rapidfire(launchpad)
```

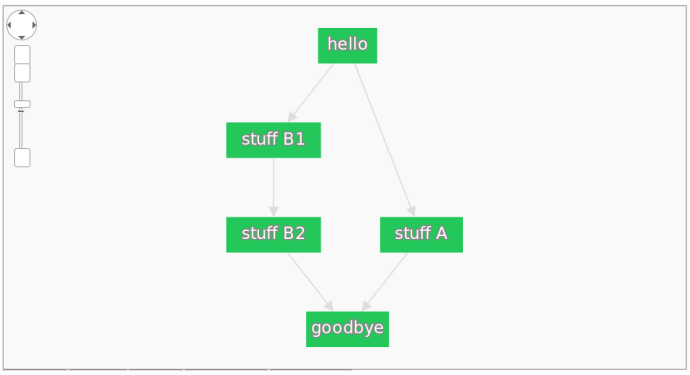
This code is divided into the following parts:

- Imports: as FireWorks is a simple Python library, one has to import it at the beginning of the workflow
- Instantiation of FireWorks launchpad: this is a mongo instance containing all the information concerning the current workflow.
- The different Firework (named fw*) contain the tasks and the information associated to each task. In that case, those are very basic.
- wf is the actual workflow. It contains the workflow's information:
    - list of the Firework (tasks)
    - dependencies between the tasks, represented as a dictionary {parent: children}
    - other information, like the workflow's name
- Finally, the workflow is added to the launchpad and executed.

By changing the dependencies representation in the dictionary, it is easy to create a more complex workflow. For example, the following dictionary:

```
wf = Workflow([fw1, fwa, fwb1, fwb2, fw3],
          {fw1: [fwa, fwb1], fwa: fw3, fwb1: fwb2, fwb2: fw3},
          name="Basic diamond workflow")
```

will create a diamond workflow:



Custom function

The previous examples were using simple bash command. It is of course possible to use custom function or Python object. Using the latter is the most powerful way to use FireWorks, but it adds a lot of complexity and the former is sufficient for the majority of the cases. Then, a Firework (task) will be defined like this:

```
fw1 = Firework(PyTask(func='my_functions.func1',
                      args=["First arg"]),
               name="First")
```

Data transfer

It is often need to transmit data between task. The easiest way to do so is to use the keywords inputs and outputs:

```
fw1 = Firework(PyTask(func='my_functions.func1', outputs=["out1"]), name="First")
fw2 = Firework(PyTask(func='my_functions.func2',
                      args=["arg1", [1, 2]],
                      inputs=["out3"]),
               name="Second"))
```

Because of the keyword outputs, the standard output of func1 is stored in the metadata of this task under the key out1. When creating the second task, the keyword inputs with the same key indicates that the data will be reused as input. The input(s) is/are concatenate at the end of the list of arguments:

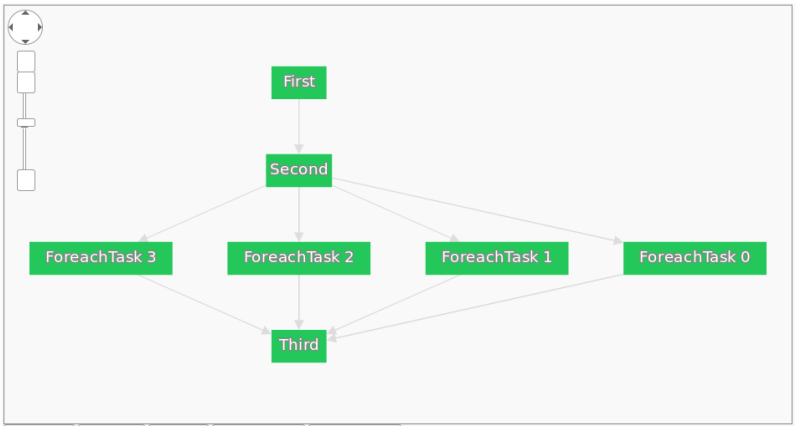
```
def func2(*args):
    print("SECOND", args)

INFO Task started: PyTask.
FOURTH ('arg1', [1, 2], 'Result 3')
INFO Task completed: PyTask
```

If data need to be passed between lots of steps, using a file might be a good idea. It is also possible to use the stored_data functionality of FireWorks in the FWAction object (see next part).

Foreach

Foreach is the method allowing to parallelize execution. Given a list, FireWorks will split the execution between all the items, each one being treated by one function/node.




The corresponding part of the workflow is:

```
fw1 = Firework(PyTask(func='my_functions.func1', outputs=["out1"]), name="First")
fw2 = Firework(ForeachTask(task={"_fw_name": "PyTask",
                                 "func": "my_functions.func2",
                                 "inputs": ["out1"],
                                 "outputs": ["out2"]},
                           split="out1"),
               name="Second")
```

In that workflow, the output of fw1 (out1) is a list. fw2 uses that list to split the task into several tasks, each of those treating one element of the list.

FWAction

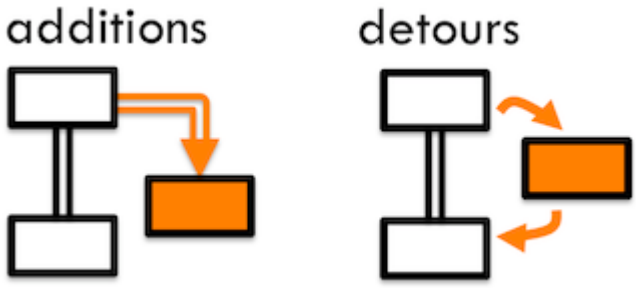
All the metadata for a task are stored in a FWAction object. If a function returns such an object, FireWorks will replace or append the new metadata to the stored FWAction. For example, it is possible to store the output this way:

```
def func3(s):
    return FWAction(mod_spec=[{'_push_all': {'out3': "Result 3"}}])
```

This allow to use other FireWorks features, like adding steps.

Adding steps

Using FWAction, it is possible to dynamically add steps. There is two ways to do so: addition or detour.



An addition adds a step without linking it back to the workflow; the detour links it back. For both of them, it is possible to add a simple step or a whole workflow.

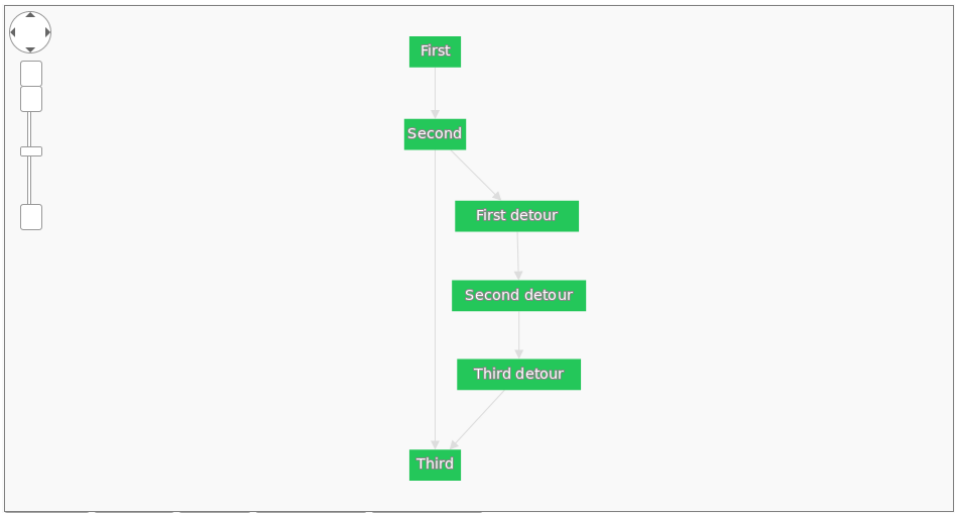
Here are examples of addition and detours:

```
### Addition ###
def func2(s):
    new_fw = Firework(PyTask(func='my_functions.func4', args=[1]), name="Fourth 1")
    return FWAction(additions=new_fw, mod_spec=[{'_push_all': {'out1': ["Result 2"]}}])

### Detour ###
def func2(s):
    new_fw = Firework(PyTask(func='my_functions.func4', args=[1]), name="Fourth 1")
    return FWAction(detours=new_fw, mod_spec=[{'_push_all': {'out1': ["Result 2"]}}])

### Detour with a workflow ###
def func2(s):
    fw1 = Firework(PyTask(func='my_functions.func1', outputs=["out1"]), name="First detour")
    fw2 = Firework(PyTask(func='my_functions.func3', inputs=["out1"], outputs=["out2"]), name="Second detour")
    fw3 = Firework(PyTask(func='my_functions.func3', inputs=["out2"]), name="Third detour")
    wf = Workflow([fw1, fw2, fw3], {fw1: fw2, fw2: fw3}, name="test detours workflow")
    return FWAction(detours=wf, mod_spec=[{'_push_all': {'out1': ["Result 2"]}}])
```

The last one will create the following workflow:




It is also possible to append one or more workflows to a workflow. This is particularly useful if one needs to run the same workflow several times (for example with different inputs).

This is done like this:

```
fw1 = Firework(ScriptTask.from_str('echo "Hello"'), name="Hello", fw_id=1)
fw2 = Firework(ScriptTask.from_str('echo "Goodbye"'), name="Goodbye", fw_id=2)
main_wf = Workflow([fw1, fw2], {fw1: fw2}, name="Main workflow")

fw1 = Firework(ScriptTask.from  str('echo hello'), name="hello")
```

```
fw2 = Firework(ScriptTask.from_str('echo stuff'), name="stuff")
fw3 = Firework(ScriptTask.from_str('echo goodbye'), name="goodbye")
wf = Workflow([fw1, fw2, fw3], {fw1: fw2, fw2: fw3}, name="Basic pipeline workflow")

main_wf.append_wf(wf, [1], detour=True)
```

### 2.4.3 Main features

Singularity support

FireWorks offer no support for Singularity or any other container solution. However, it is quite easy to design the task to execute the actual calculation in a container. The container and the execution can be controlled by a wrapper (see 2.7).

This is transparent for the user: the execution is similar and the potential errors will be treated like if the wrapper overhead was not there.

Web GUI

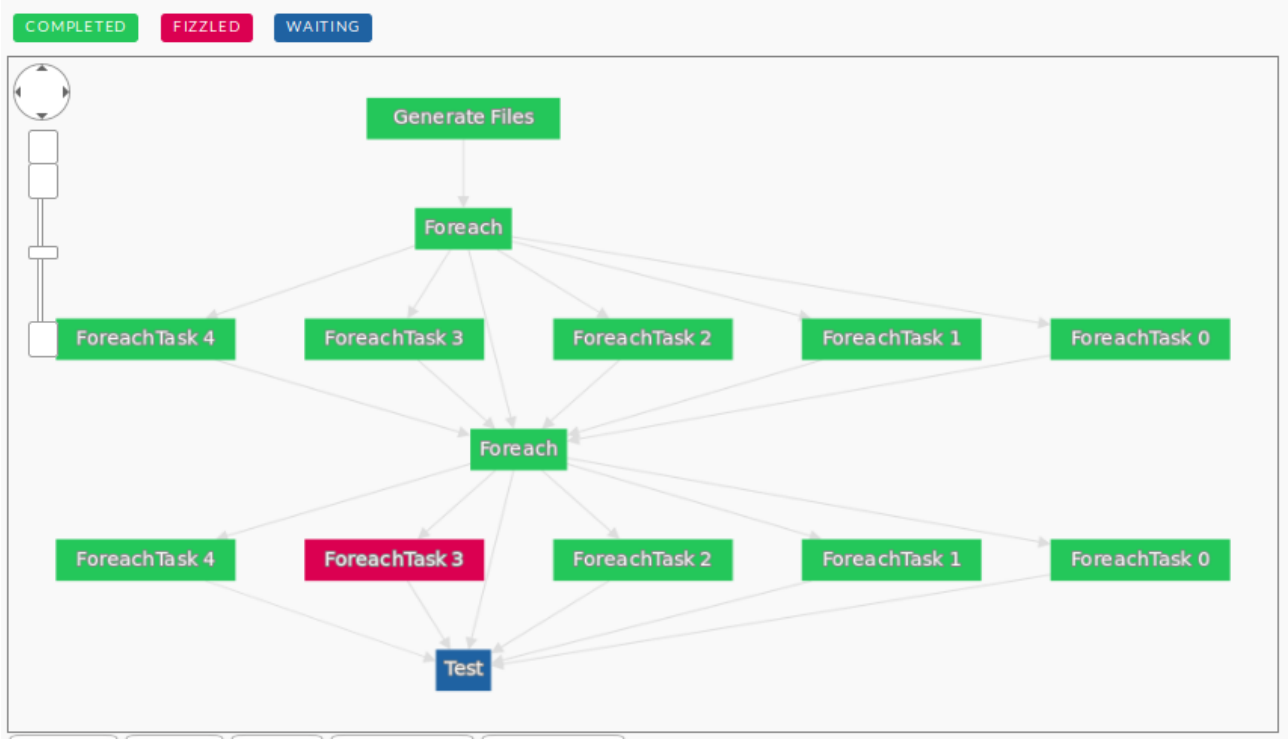
The web GUI is a powerful tool that can be used to display workflow graphs, errors, statistics and to monitor the running workflows. It uses Flask and is launched as a service. The graphs are the one used to show the previous examples.

In the next picture, the left column shows the workflows (with the state of the tasks); the right column shows the statistics:


The MongoDB database can be easily accessed through network and can be run on a separate cluster than the compute one. Thanks to that, the web GUI can be launched from any machine which has a network access to the MongoDB database, and not only the machine from which the workflow had been launched. Moreover, a given user will only be able to access its own workflows as the database requires the users' credentials.

Error handling

FireWorks use Python error handling system. It will catch error raised by task but will not stop. Indeed, FireWorks implements the lazy fail system, meaning that it will do all the tasks that do not depend on the task in error.




After an error, it might be needed to execute the workflow from the stopping point instead of running the entire workflow from scratch. To do so, one has the add the Dupefinder keyword. This will indicate to FireWorks that it has to look for duplicates (i.e. tasks with same inputs) and not run them again. Hence, Dupefinder can be used only if the task is deterministic (the same inputs will always yield the same outputs).

```
fw1 = Firework(ScriptTask.from_str('echo hello'), name="hello", spec={"_dupefinder": DupeFinderExact()})
```

## 2.5 Parsl

### 2.5.1 Tool description

Presentation

Parsl is a young project officially started in 2017 (first public commit on Github on October 2016). It is developed by the University of Chicago following the award of a NSF grant for "Integrating Parallel Scripted Workflow into the Scientific Software Ecosystem" [22] of $2,749,446.00.

The official Parsl presentation states that:

“Parsl is a Python library for programming and executing data-oriented workflows (dataflows) in parallel. Parsl scripts allow selected Python functions and external applications (called apps) to be connected by shared input/output data objects into flexible parallel workflows. Rather than explicitly defining a dependency graph and/or modifying data structures, instead developers simply annotate Python functions and Parsl constructs a dynamic, parallel execution graph derived from the implicit linkage between apps based on shared input/output data objects. Parsl then executes apps when dependencies are met. Parsl is resource-independent, that is, the same Parsl script can be executed on a laptop, cluster, cloud, or supercomputer.”

One of the main points to notice is that, unlike most of the workflow manager, Parsl builds dynamically the workflow during execution. We will see later what this different philosophy implies when using Parsl.

Parsl has no other dependency than Python3. The installation is made like every other Python library, and only on the controlling node (not on all the worker nodes). Parsl is quite easy to use, but one should have Python knowledge to ease the use of Parsl.

Parsl supports the following batch systems:

- Cobalt
- Slurm
- Condor
- GridEngine
- Torque
- AWS
- GoogleCloud
- Jetstream
- Kubernetes [23]

Development and Community

Parsl is still under development and there are weekly commits. The development team has between 3 to 4 full time people on the project. The developers' team is on Slack and answer quickly to either bug report, feature requests or any question.

The documentation is quite short, but enough to use Parsl. As it is almost pure Python, the use is actually quite simple for basic use. For more "exotic" use, Parsl is more difficult to get around as it is quite new and some kind of use not yet implemented.

The high-level road-map can be found here [24].

### 2.5.2 Tool Utilisation

Workflow definition

As previously said, the workflow is not defined; instead, it is dynamically built during execution. For example, if we take a very simple, two-steps, linear workflow:

```
@python_app
def hello():
        import time
        time.sleep(2)
        return "Hello"

@python_app
def goodbye():
        import time
        time.sleep(2)
        return "Goodbye"

if __name__ == '__main__':
        print(hello().result())
        bye = goodbye()
        print(bye)
```

```
        print(bye.result())
        print(bye)
```

This workflow has the following output:

```
Hello
<AppFuture at 0x7f7ee909e358 state=running>
Goodbye
<AppFuture at 0x7effb7392358 state=finished returned str>
```

The result() blocks the execution until hello() is done. Then, the script print bye, which is the Parsl object containing the function goodbye(). This function is running (because of the 2 seconds sleep); the following line prints "Goodbye" after the 2 seconds sleep. Finally, we see that the object state is now "finished".

Parsl execute the whole function on the designated worker node. Hence, the function has to have all the necessary imports inside the function, which is contrary to what is advised by Python convention.

This example was using Python function; Parsl also offers to use @bash_app:

```
@bash_app
def echo_hello(stdout='hello.out', stderr='hello.err'):
        return 'echo "hello"'
```

Parallel workflow

The previous workflow was linear. What if we need to execute the two functions simultaneously?

One way to to it would be to instantiate bye() before waiting for the hello() result:

```
if __name__ == '__main__':
        bye = goodbye()
        print(hello().result())
        print(bye.result())
```

Another way would be to instantiate all the steps together:

```
if __name__ == '__main__':
        steps = []
        for i in range(5):
        steps.append(hello())

        for s in [s.result() for s in steps]:
                print(s)
```

The five strings "Hello" will be printed at the same time. This is the way usually used in a scientific workflow situation.

Data management

We have seen how Parsl can wait for the result of functions; it can do the same with folder. Indeed, Parsl allow to give file(s) as input and/or output and then wait for the completion of the function to pass the files between functions. This is a simple way to pass data while respecting the dependencies.

When using files, one can use the paths to said files, but it is also possible to use Parsl built-in file abstraction:

```
# Create Parsl file objects
parsl_infile = File('https://example.com/infile.txt')
parsl_outfile = File(os.path.join(os.getcwd(), 'out.txt'),)

# Call the copy app with the Parsl file
copy_future = copy(inputs=[parsl_infile], outputs=[parsl_outfile])
```

Workflow execution

To execute locally, the configuration is quite simple:

```
import parsl
from parsl.app.app import python_app, bash_app
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

#parsl.set_stream_logger() # <-- log everything to stdout
config = Config(
        executors=[ThreadPoolExecutor(max_threads=20)],
        lazy_errors=True,
)
parsl.load(config)
```

To execute on an HPC, the configuration is bigger. For example, here is the configuration to use Parsl in the CC-IN2P3 HPC centre:

```
import parsl
from parsl.app.app import python_app
from parsl.channels import LocalChannel
from parsl.providers import GridEngineProvider
from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor

config = Config(
        executors=[
        IPyParallelExecutor(
        label='cc_in2p3_local_single_node',
        provider=GridEngineProvider(
        channel=LocalChannel(),
        nodes_per_block=1,
        init_blocks=1,
        max_blocks=1,
        walltime="10:20:00",
        scheduler_options='',         # Input your scheduler_options if needed
        worker_init='',     # Input your worker_init if needed
        ),
        engine_debug_level='DEBUG',
        )
        ],
)
parsl.load(config)
```

In both cases, it is very easy to execute a workflow: one only need to execute the Python script, like every other Python script.

### 2.5.3 Main Features

Singularity support

Parsl supports container; the documentation only shows how to use Docker, but also quote Singularity as supported.

However, the support seems quite limited: Parsl only use an image (no build nor pull from a registry) to execute all the steps. The feature is in experimental state. Hence, it seems than using a custom wrapper might offer more functionalities (see part 2.7). The presence of the native support might however indicate that it will offer more functionalities in the future.

Error handling

Parsl execute Python functions, so error handling is close (if not identical) to what is usual in Python. Hence, while developing a Parsl workflow, one should try to catch and raise the errors that might happen. Parsl allows to retry steps in error, and offers lazy fail: the workflow will continue after an error until one dependency is not met.

Parsl also allow to cache functions outputs to not uselessly rerun the same steps. It is also possible to create checkpoints, where Parsl while save all the data from a workflow and from where it can start again.

Monitoring tools

Parsl has a GUI, but the tool is in alpha and does not offer, for now, any useful functionality. The development of this feature should be one of the priorities for the next major release.

## 2.6 Pegasus

### 2.6.1 Tool description

Presentation

Pegasus project has started in 2001 and since frequent release are published. The system is developed by the Science Automation Technologies team at the USC Information Sciences Institute. The current release is 4.9.1 (06/04/19). The full development timeline is available on the website [25]. The project is well-known in the workflow managers world as it has been used in many various scientific domains such as astronomy, bioinformatics, earthquake science, gravitational wave physics, ocean science, limnology, and others.

The official Pegasus presentation states that:

“The Pegasus project encompasses a set of technologies that help workflow-based applications execute in a number of different environments including desktops, campus clusters, grids, and clouds. Pegasus bridges the scientific domain and the execution environment by automatically mapping high-level workflow descriptions onto distributed resources. It automatically locates the necessary input data and computational resources necessary for workflow execution. Pegasus enables scientists to construct workflows in abstract terms without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). Pegasus also bridges the current cyber infrastructure by effectively coordinating multiple distributed resources.”

Community

The community behind Pegasus is pretty wide as it has been used in many different projects. However, it’s not that easy to reach this community because every project has its small group of user and they don’t usually

interact with each other. There is also no real communication tool but it's really easy to contact Pegasus support and its developers. They gave an important point to carry the user and analyse its case.

Because the project as started 18 years ago, the documentation and the features available are pretty huge. It's well organized but it lacks of some clarity sometimes, especially when dealing with workflows creation. Lot of the examples are not fully documented. On the other hand, as said above, Pegasus team is really open to question and answers quite fast.

### 2.6.2 Tool Utilisation

Workflow definition

Pegasus uses the DAX method to write workflows. The DAX is a description of an abstract workflow in XML format and can be generated using the Pegasus API in Java, Perl, or Python (Perl is deprecated in 4.9.0 and will no longer be supported in version 5).

Documentation about the API interface can be found here [26].

A lot of example have been created and are available in the main Pegasus Github repository.

Some of theses examples are explained in the documentation as well [27].

A tutorial is also available with a dedicated VM where you can try Pegasus [28].

Pegasus can be executed on multiple platforms and architectures (localhost, cloud, etc...) but always required Condor to be installed on theses sites. Here is the list of the supported execution environments:

- Localhost
- Condor Pool
- Cloud (Amazon EC2/S3, Google Cloud, ...)
- Amazon AWS Batch
- Remote Cluster using PyGlidein
- Remote Cluster using Globus GRAM
- Remote Cluster using CREAMCE
- Local Campus Cluster Using Glite
- SDSC Comet with BOSCO glideins
- Remote PBS Cluster using BOSCO and SSH
- Campus Cluster
- XSEDE
- Titan Using Glite

Here is an example of a python script which uses the DAX API to generate a simple workflow:

```
#!/usr/bin/env python

import os
import pwd
import sys
import time
from Pegasus.DAX3 import *


daxfile = "split.dax"
USER = pwd.getpwuid(os.getuid())[0]
dax = ADAG("split")
```

```
dax.metadata("creator", "%s@%s" % (USER, os.uname()[1]))
dax.metadata("created", time.ctime())

webpage = File("pegasus.html")

# First job that splits the webpage into smaller chunks
split = Job("split")
split.addArguments("-l","100","-a","1",webpage,"part.")
split.uses(webpage, link=Link.INPUT)
dax.addJob(split)

# we do a parameter sweep on the first 4 chunks created
for c in "abcd":
        part = File("part.%s" % c)
        split.uses(part, link=Link.OUTPUT, transfer=False, register=False)

        count = File("count.txt.%s" % c)


    # Defining a job using wc binary
        wc = Job("wc")
        wc.addArguments("-l",part)

    # Set job output to a file to be able to get the wc command output
        wc.setStdout(count)
        wc.uses(part, link=Link.INPUT)
        wc.uses(count, link=Link.OUTPUT, transfer=True, register=True)
        dax.addJob(wc)

        # Adding dependency to be able to run wc job only after split job finished
        dax.depends(wc, split)

f = open(daxfile, "w")
dax.writeXML(f)
f.close()
print "Generated dax %s" %daxfile
```

The generated DAX file looks like this:

```
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX http://pegasus.isi.edu/schema/dax-3.6.xsd" version="3.6"
name="split">
        <metadata key="created">Thu Jan 31 13:43:51 2019</metadata>
        <metadata key="dax.api">python</metadata>
        <metadata key="creator">jbussery@cctbcondor09</metadata>
        <job id="ID0000001" name="split">
        <argument>-l 100 -a 1 <file name="pegasus.html"/> part.</argument>
        <profile namespace="pegasus" key="label">p1</profile>
        <uses name="pegasus.html" link="input"/>
        <uses name="part.d" link="output" register="false" transfer="false"/>
        <uses name="part.b" link="output" register="false" transfer="false"/>
        <uses name="part.a" link="output" register="false" transfer="false"/>
        <uses name="part.c" link="output" register="false" transfer="false"/>
```

```
        </job>
        <job id="ID0000002" name="wc">
        <argument>-l <file name="part.a"/></argument>
        <profile namespace="pegasus" key="label">p1</profile>
        <stdout name="count.txt.a" link="output"/>
        <uses name="part.a" link="input"/>
        <uses name="count.txt.a" link="output" register="true" transfer="true"/>
        </job>
        <job id="ID0000003" name="wc">
        <argument>-l <file name="part.b"/></argument>
        <profile namespace="pegasus" key="label">p1</profile>
        <stdout name="count.txt.b" link="output"/>
        <uses name="part.b" link="input"/>
        <uses name="count.txt.b" link="output" register="true" transfer="true"/>
        </job>
        <job id="ID0000004" name="wc">
        <argument>-l <file name="part.c"/></argument>
        <profile namespace="pegasus" key="label">p1</profile>
        <stdout name="count.txt.c" link="output"/>
        <uses name="part.c" link="input"/>
        <uses name="count.txt.c" link="output" register="true" transfer="true"/>
        </job>
        <job id="ID0000005" name="wc">
        <argument>-l <file name="part.d"/></argument>
        <profile namespace="pegasus" key="label">p1</profile>
        <stdout name="count.txt.d" link="output"/>
        <uses name="part.d" link="input"/>
        <uses name="count.txt.d" link="output" register="true" transfer="true"/>
        </job>
        <child ref="ID0000002">
        <parent ref="ID0000001"/>
        </child>
        <child ref="ID0000003">
        <parent ref="ID0000001"/>
        </child>
        <child ref="ID0000004">
        <parent ref="ID0000001"/>
        </child>
        <child ref="ID0000005">
        <parent ref="ID0000001"/>
        </child>
</adag>
```

This workflow takes a file as input and split its content in different sub file delimited by a number of characters.

As shown, a simple workflow is not that simple to write. It's important to note that Pegasus is pretty bad at doing dynamic tasks: for example, listing a directory to add the contained files as resources for the workflow. It's better to handle this part with an external wrapper than to do it with a Pegasus job.

For example, in this use case we have two jobs. The first one is generating files and the second one is putting file in it. If we want to add a third one which list the generated file to be able to get them and process them later, we will have one task per file which can conduct to a very complex behaviour for such a simple task. The recommended way to do it is to list files using a for loop and file the replica catalogue which will give access to the generated files for the rest of the workflow.

```
# Job 1 : Generate files
print "Creating Job 1 : Generate files"
jobs_gen = []
gen_files = []
for i in range(10):
        gen = Job("touch")
        gen.addArguments("lsst_%d" % i)
        gen_files.append(File("lsst_%d" % i))
        gen.uses(gen_files[i], link=Link.OUTPUT, transfer=False, register=False)
        jobs_gen.append(gen)
        # Adding job to workflow
        lsst_wf.addJob(gen)


# Job 2 : Preparing files (putting content)
print "Creating Job 2 : Preparing files"
jobs_ech = []
pre_files = []
for i,gen_file in enumerate(gen_files):
        ech = Job("echo")
        jobs_ech.append(ech)
        ech.addArguments("File %d" % i)
        ech.uses(gen_file, link=Link.INPUT)
        output = File("%s.txt" % gen_file.name)
        pre_files.append(output)
        ech.setStdout(output)
        ech.uses(output, link=Link.OUTPUT, transfer=True, register=False)
        lsst_wf.addJob(ech)
```

Writing a dax file is not the only thing the user has to deal with because, he needs to write a Replica Catalogue containing static file mapping (if they are not present in the input directory), a Transform Catalogue referencing the binaries location that will be used by the workflow, and a site file describing the executing environment.

Here is the corresponding Replica Catalogue for the previous workflow:

```
pegasus.html file:///scratch/split/input/pegasus.html site="local"
count.txt.b file:///scratch/split/output/count.txt.b site="local"
count.txt.a file:///scratch/split/output/count.txt.a site="local"
count.txt.d file:///scratch/split/output/count.txt.d site="local"
count.txt.c file:///scratch/split/output/count.txt.c site="local"
```

Here is the corresponding Transform Catalog for the previous workflow:

```
tr wc {
        site condorpool {
        pfn "/usr/bin/wc"
        arch "x86_64"
        os "LINUX"
        type "INSTALLED"
        }
}
```

```
tr split {
        site condorpool {
        pfn "/usr/bin/split"
        arch "x86_64"
        os "LINUX"
        type "INSTALLED"
        }
}
```

Here is the corresponding Site file for the previous workflow:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
        xsi:schemaLocation="http://pegasus.isi.edu/schema/sitecatalog http://pegasus.isi.edu/schema/sc-4.1.xsd"
version="4.1">

        <!-- The local site contains information about the submit host -->
        <site handle="local" arch="x86_64" os="LINUX">
        <!-- This is where intermediate data will be stored -->
        <directory type="shared-scratch" path="/scratch/split/scratch">
        <file-server operation="all" url="file:///scratch/split/scratch"/>
        </directory>
        <!-- This is where output data will be stored -->
        <directory type="shared-storage" path="/home/jbussery/split/output">
        <file-server operation="all" url="file:///scratch/split/output"/>
        </directory>
        </site>

        <site handle="condorpool" arch="x86_64" os="LINUX">
        <!-- These profiles tell Pegasus that the site is a plain Condor pool -->
        <profile namespace="pegasus" key="style">condor</profile>
        <profile namespace="condor" key="universe">vanilla</profile>
        <!-- This profile tells Pegasus to create two clustered jobs
        per level of the workflow, when horizontal clustering is
        enabled -->
        <profile namespace="pegasus" key="clusters.num" >2</profile>
        </site>
```

Workflow execution

Pegasus comes with a set of tools to create plots and graphs (GANTT) to be able to have a graphic representation of what the workflow is actually doing.

After defining the workflow, the only thing remaining is to plan the execution using pegasus-plan like this:

```
pegasus-plan --conf pegasus.properties \
        --dax $DAXFILE \
        --dir $DIR/submit \
        --input-dir $DIR/input \
        --output-dir $DIR/output \
```

```
        --cleanup leaf \
        --force \
        --sites condorpool \
        --submit
```

With this command, we are able to set if the jobs have to run on the Condor farm or on the local Condor installation. We also set the submit directory where we will be able to found all the task traces and various logs regarding the execution.

Workflow dependencies

Pegasus handles by default job dependency by specifying relation between them in the workflow definition. Here is an example using the Python DAX API:

```
wf.addDependency(Dependency(parent=job1, child=job2))
```

For larger workflow, it is possible to define some Profiles to be able to specify which job has to run on which execution site. It allows the user to group small jobs into a bigger job and let it executes on a specific sites different from the one running larger jobs [29].

With Pegasus, the user is also able to create hierarchical workflows which are workflow which contains other workflows [30].

Data management

Pegasus does data management for the executable workflow.

It uses a Replica Catalogue to discord the locations of the input datasets and add data movement and registration node in the workflow to:

- **stage-in** input data to the staging sites
- **stage-out** output data generated by the workflow
- **stage-in** intermediate data between compute sites
- **data registration** nodes to catalogue the locations of the output data on the final storage site into the replica catalogue.

Pegasus also transfers user executable, which are registered in the Transform Catalogue, to the compute sites.

To be able to correctly handle these operations, it requires to choose a data staging configuration model:

Shared file system, compute jobs in the workflow run in a directory on a shared fs.

NonShared file system, compute jobs in the workflow run in a local directory on the worker node; it requires a specific node set as coordinator which will handle data transfer and distribution to the compute nodes; this setup will use a specific binary: pegasus-transfer.

Condor Pool without shared file system, all data input/output is achieved using Condor File IO instead of pegasus-transfer [31].

### 2.6.3 Main Features

Singularity support

Docker, Shifter and Singularity are natively supported in Pegasus.

Since version 4.8.0 (currently 4.9.0+), Pegasus has support for application containers in the non-shared file system.

Specifications regarding Singularity and Docker image format can be found here [32].

It's important to note that containerized Applications can only be specified in the transformation catalogue, not via the DAX API.

Example of container implementation in a workflow [33].

All the data management is also available for the container, so it’s completely transparent for the user because Pegasus handles the tasks where file have to be put in the container and retrieved from it to send them to the next instruction.

Error handling

When the user generates the workflow DAX file using the API, the linker will be able to catch a first layer of error regarding the workflow definition.

After submitting the workflow, we can follow the processing using pegasus-status.

When a workflow has finished its execution, the whole process can be analysed in more details using pegasus-analyser.

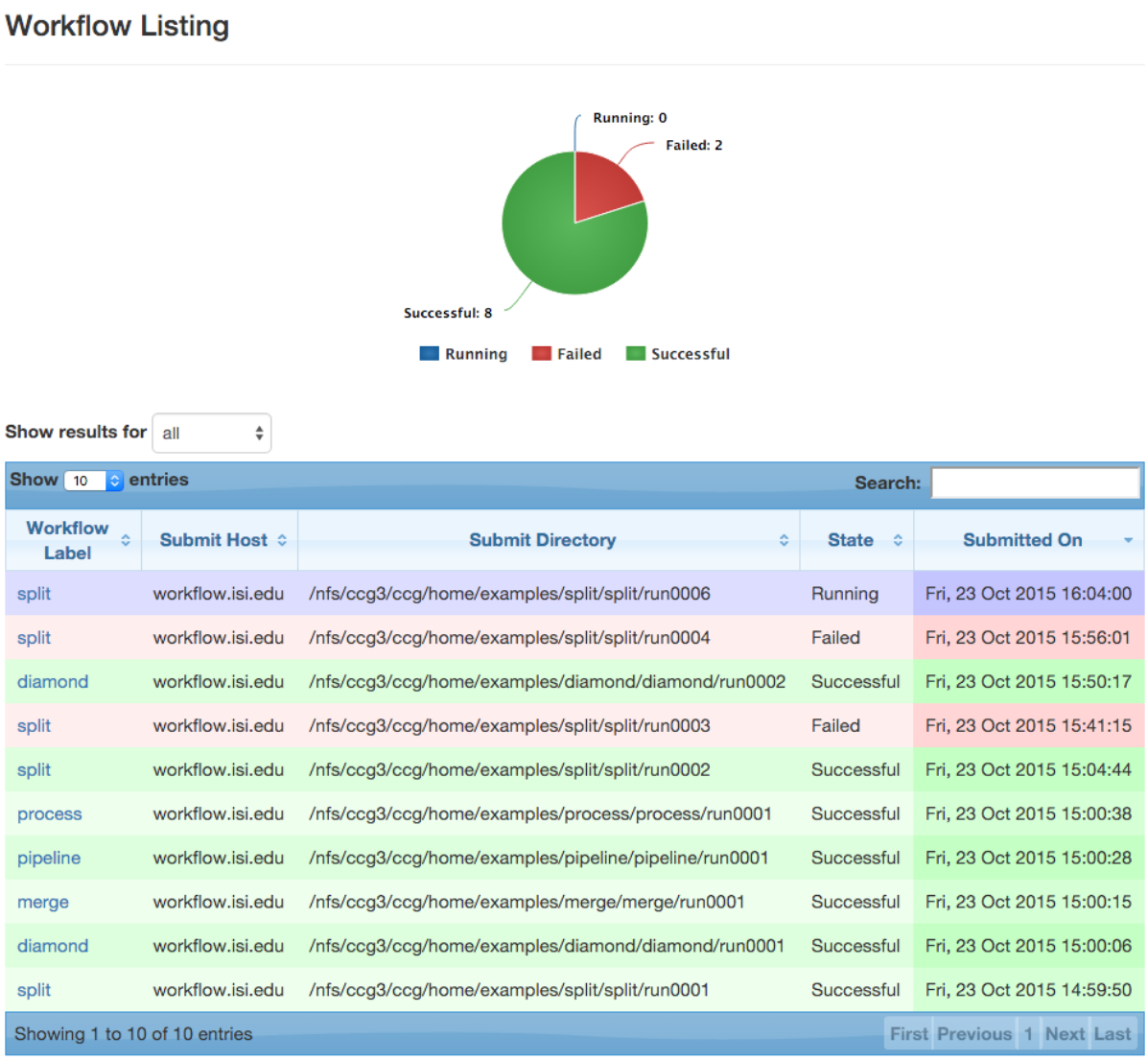
Resubmitting a failed job is also supported by the software. For example:

```
pegasus-run /Path/To/Workflow/Directory
```

The full documentation is available here [34].

Monitoring tools

Pegasus provides a dashboard where all the information regarding the workflow execution are displayed. Additionally, we can find a graph and some other statistics. At the time of writing this report, we could not properly set up the monitoring tools on our cluster, so the following information is extracted from the official website (as stated below the image), and is giving for informative purpose only.




source: https://pegasus.isi.edu/documentation/images/dashboard_home.png

From this dashboard, we can get a full detailed view of a job, its status and the files related to it. Because of some configuration issues on our site which doesn't populate our dashboard during workflow execution, the screenshot above has been taken from Pegasus user guide.

Pegasus also supports notifications which can be used to trigger some jobs or to inform the user regarding a specific action or an error.

During the workflow execution, Pegasus populates a database, which is used by the dashboard for the display, and also by many other monitoring components.

Last but not least, Pegasus supports publication of the monitoring events to an AMQP Message Server. This way, it is possible to integrate this monitoring in a larger system and create graphs and alerts [35].

## 2.7 Custom Singularity Wrapper

As we have seen, some of the workflow managers do not support Singularity. However, as they always allow to use custom Python functions, it is quite easy to execute Singularity from a function. To do so, the easiest is to use a wrapper that can be reused at each step where Singularity is needed. This allows to keep the same behaviour and functionalities between each stage. This wrapper can of course be written in any language, but using the same language than the rest of the workflow (i.e. Python for most of the case) is the easiest.

The wrapper should at least offer these functionalities:

- Choice of the Singularity image: for most of the workflows, the same image will be used at each step. However, it might be needed to use two or more different images, for example in case of different versions of the same software and/or library (the most common example is Python 2.x and 3.x: using two images might be easier than setting up two virtualenv into one image). Choosing the image does not add much more complexity, so it should be offered.
- Setting Singularity options: the options (the most common being binding) are often necessary to run the image.
- Setting the path: the software might need Singularity to be executed from a specific directory (for example in case of input files).
- Executing a command line: the most common way to use Singularity in a workflow is with singularity exec. This allows to either launch a simple command line or a script.
- Catching stdout and stderr: the standard outputs should be put into files, either to re-use the result in a next step on to debug in case of error.
- Catching errors and raise them: in case of error (an exit code different of 0), the error should be raised to the workflow manager, which should handle it.

Here is an example of a very simple wrapper (proof-of-concept type):

```
class SingularityExec:
    def __init__(self, singularity_image, script, singularity_options=""):
        self.singularity_image = singularity_image
        self.script = script
        self.singularity_options = singularity_options

def run(self, path):
    from subprocess import Popen
    command = 'singularity exec {} {} {}'.format(self.singularity_options,  self.singularity_image,self.script)
    with open(out + ".out", "w") as stdout:
        with open(out + ".err", 'w') as stderr:
            proc = Popen(command.split(), stdout=stdout, stderr=stderr, cwd=path)
    exitcode = proc.returncode
    if exitcode:
        raise ValueError(err)
    return True
```

In this wrapper, the sub-process import is included in the function as Parsl need the import to be done that way. If used with another workflow manager, the import can be done as usual.

As said, this is a very simple wrapper, which is working fine in a test environment but could be completed before being used in production. Amongst the functionalities that could be added are:

- setup the environment before executing Singularity: this could include setting environment variables or launching a script.
- add some checks: Singularity installation and versions, etc.

- pull an image from a registry.
- choose the way Singularity is used: exec, run.
- more error handling.

## 2.8 Discussion about FireWorks and Parsl

FireWorks and Parsl seem to have a lot in common: they are both two small projects, aiming at doing one thing and keeping it simple. They both use Python, internally and as a way to describe the workflows, and they mostly propose the same features.

Nonetheless, when using them it is easy to see that behind those similarities are some philosophical differences. FireWorks made the choice to need a more important overlay; one of the preferred described way to add a custom task is to describe it with Python class using FireWorks' syntax. Even though it is entirely possible to only use simple Python function to obtain almost the same result (some very specific cases might be not covered), this need a longer time to learn. The workflow description itself is written in Python using FireWorks' methods, which also leads to a very specific syntax that is not always easy to write and/or read.

This, combined with an incomplete and sometimes misleading documentation, implies that FireWorks has a steeper learning curve, or that a new user needs to rely on existing examples to understand how it works. However, once FireWorks' inner workings are understood, it offers every functionality a simple Workflow manager should offer, including a proper graphical interface (GUI). This powerful tool is ran as a web service and allows to see workflows (either already executed, currently running or waiting) and errors. This is an irreplaceable feature and probably one of the greatest strength of FireWorks.

The main drawbacks about FireWorks would probably not be really technical, but would rather concern the development cycles and team. Indeed, while the project seems to be quite mature and stable, in the recent past some updates has broken some (though minor) features, which could lead to think that internal testing is not optimum - even though the bug fixes were quick after users reported them. The team is small and with no real roadmap for the future; the documentation sparse and not up-to-date. Thus, the main concern would be about the evolution of FireWorks in the future. The product is now working and could be selected for a production use for a short project. But what about the long term?

Parsl is also in Python (both for the core and for the workflow descriptions), but works differently. To create a workflow, one only need to add Parsl decorators to Python functions. Indeed, the workflow is not defined per see, but is dynamically generated while the script is executed. Thus, Parsl's syntax is really easy for anyone who knows Python; if data need to be passed between steps, the syntax is slightly more complicated, but is still far easier than FireWorks. On the other hand, the workflow is only defined by the order in which functions are called, so it might be less easy to follow the workflow, especially for non-Python specialist.

Parsl is still young and under development. The team seems quite active and count around five people. The project is funded by the National Science Foundation (NSF), so we might except a support for, at least, the next few years. Nonetheless, its young age is also Parsl main drawback, as it lacks one crucial feature: there is no graphical interface. However, the development team is actively working on it, and it might be proposed in a near future.

# 3 Workflow implementation of real use cases

## 3.1 Solid State Physics using Yambo and Quantum Espresso

Physical context and software stack

In the spectrum of Solid State Physics available codes, it has been decided to use all open-source projects widely used by the physics and chemistry communities: Quantum ESPRESSO [36] and Yambo [37]

Quantum ESPRESSO (QE) is an integrated suite of Open-Source computer codes for electronic-structure calculations and materials modelling at the nanoscale. It is based on density-functional theory (DFT), plane waves, and pseudopotentials. It has evolved into a distribution of independent and interoperable codes in the open-source spirit.

The code consists of a set of components, and a set of plug-ins that perform more advanced tasks, plus a number of third-party packages designed to interact with the core components.

Between these third-party packages Yambo has been chosen, as an important member of the key group of ab-initio spectroscopy codes supported by the European Theoretical Spectroscopy Facility. A code for Many-Body calculations in solid state and molecular physics. Yambo relies on the Kohn-Sham wavefunctions generated by two DFT public codes: abinit [38] and Quantum Espresso.

The Singularity container built for such solid state physics workflow is mainly made by Openmpi 2.1.1 as compiler, Quantum Espresso version 6.3 and Yambo version 4.3.2.

The Singularity recipe used is:

```
Bootstrap: docker
From: centos:centos7.4.1708
IncludeCmd: yes

%post
echo "Installation software"
yum -y install wget vim which
yum -y groupinstall "Development tools"
yum -y install binutils dapl dapl-utils ibacm infiniband-diags libibverbs libibverbs-devel libibverbs-utils libmlx4 librdmacm librdmacm-utils mstflint opensm-libs perftest qperf rdma

################################
# OpenMPI 2.1.1 installation
cd /tmpdir
wget https://download.open-mpi.org/release/open-mpi/v2.1/openmpi-2.1.1.tar.gz
tar -xvf  openmpi-2.1.1.tar.gz
rm openmpi-2.1.1.tar.gz
cd openmpi-2.1.1
./configure --prefix=/usr/local/openmpi --disable-getpwuid --enable-orterun-prefix-by-default

make
make install
```

```
export PATH=/usr/local/openmpi/bin:${PATH}
export LD_LIBRARY_PATH=/usr/local/openmpi/lib:${LD_LIBRARY_PATH}
export MPICC=mpicc
export MPICXX=mpicxx
export MPIF90=mpif90
export MPIF77=mpif77
export MPIFC=mpifort

################################
# QE 6.3 installation
cd /tmpdir
wget https://gitlab.com/QEF/q-e/-/archive/qe-6.3-backports/q-e-qe-6.3-backports.tar.gz
tar -xvf q-e-qe-6.3-backports.tar.gz
rm q-e-qe-6.3-backports.tar.gz
cd q-e-qe-6.3-backports
export QE_INSTALL_DIR=/usr/local/q-e-qe-6.3-backports/
./configure --enable-openmp --enable-parallel
make all

export PATH=/tmpdir/q-e-qe-6.3-backports/bin:/usr/local/openmpi/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:$PATH

################################
# Yambo 4.3.2 installation
cd /tmpdir
wget https://github.com/yambo-code/yambo/archive/4.3.2.tar.gz
tar -xvf 4.3.2.tar.gz
rm 4.3.2.tar.gz
cd yambo-4.3.2
./configure
make yambo interfaces

################################
%environment
 export PATH=/tmpdir/yambo-4.3.2/bin:/tmpdir/q-e-qe-6.3-backports/bin:/usr/local/openmpi/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:$PATH
 export LD_LIBRARY_PATH=/usr/local/openmpi/lib:$LD_LIBRARY_PATH
```

```
export MPICC=mpicc
export MPICXX=mpicxx
export MPIF90=mpif90
export MPIF77=mpif77
export MPIFC=mpifort
```

The implemented workflow is the optical spectrum of the Silane (SiH4) molecule [39], by means of the Time-Dependent Density Functional Theory (TDDFT), in the Adiabatic Local Density approximation (ALDA) approximation, the method includes local field effects corresponding to charge oscillations. To treat localized systems we used the supercell approach. No k-point sampling is needed for the wave function, so we proceed considering only the gamma point. A big number of plane wave is needed to represent an isolated system like our SiH4 molecule.

The workflow main steps are:

- run the self-calculation on SiH4 molecule using QE. The input file can be downloaded at [40];
- then, using the obtained results, run a standard bands calculation on SiH4 molecule wave function;
- convert the QE output in Yambo input with the p2y program;
- run the Yambo calculations with ALDA approximation in order to produce the absorption spectra of the molecule as a data text, formatted in columns;
- and finally plot the data, several programs can be used, such as gnuplot, matplotlib, excel, and so on.

From a physical point of view, it is possible to change some input parameters and analyze how the system reacts to such variations. The parameters that can be changed are:

- the geometry of the input system (e.g. twisting the Si-H bond lengths);
- the polarization bands;
- in the RPA approximation, including Local Fields, both in reciprocal space and in the basis of e-h pairs check the effect of increasing the block size of the response function and evaluate the effect of the ALDA kernel with respect the local field effects.

In the following the Si-H bond length will be changed several times: 1.489, 1.589, 1.689, 1.789, 1.889 [Angström]. These values will be automatically updated in the matrix of the atomic positions in the first step of the workflow. The five results are plotted in the same graph for comparison, and a short interpretation is given.

Workflow description and implementation

The basic workflow is pretty simple as it is linear. The only dependency one has to take into account is the correct input files which are either basic parameters files or output files from the previous step. This workflow is then sequential and implementing it is straightforward.

Looking at the data management, Parsl and FireWorks use the shared file system to store all the input and output files. Whichever was the worker node on which a given step was run, the shared area was available and writable, hence allowing an easy access to the files. For Makeflow and Pegasus, on the other hand, we keep the default behaviour which is to copy the input files to the worker node, then copy the output files back to the launch directory of the workflow manager. We could also use the shared file system as it is an available functionality but since we are only dealing with small text files, there were no need to use it, as this would not improve the execution of the workflow.

One of the main goals of this task is to identify potential constraints on using containers in workflows for scientific computing. Regarding Makeflow, we could not use the native Singularity support (see also part 2.3.3).

The root of that issue is that the Singularity container which has to be used to run that workflow is in a version 3 format, which is not compatible with Singularity 2. On the CC-IN2P3 computing cluster, where we ran all the workflows the current default Singularity version is 2.6.1. One has then to update the environment in order to be able to use a more recent version. This is quite straightforward and the required environment can be set up using a dedicated script. However, the native support of Singularity in Makeflow is done by running first a Singularity wrapper in which the task will be executed. This wrapper, in the current model, is always the first to be run, and then it was impossible to run before it a dedicated environment wrapper to set up the expected environment. We could however easily fix that by moving both the environment script and the right Singularity command inside the same wrapper option in a Makeflow command line.

The "classic" way of using a Singularity container with Makeflow would be something like:

```
$ makeflow --wrapper ./setup.sh -T sge --singularity=/pbs/throng/ccin2p3/hpc-
europa3/img_yambo432_qe63_backports_openmpi211.sif qe_yambo.mf
```

The setup.sh script is required to set up a working Singularity version 3 environment. We should instead use the following command line:

```
$ makeflow --wrapper "./setup.sh; singularity exec --home $(pwd) /pbs/throng/ccin2p3/hpc-
europa3/img_yambo432_qe63_backports_openmpi211.sif -c {}" qe_yambo.mf
```

Which was basically moving both wrappers in the same one in order to ensure the environment is first set up before instantiated the Singularity container.

Though this issue was easily fixed, this showed that Makeflow is not that flexible when it comes to use it in a different way that the one proposed by the tool. This issue has been discussed with the developers who said that the use of wrappers with Makeflow will be enhanced and should be more flexible in a future release.

For Pegasus, the support for the containers (docker, singularity or shifter) is fully integrated in the workflow management and, as for a classic workflow, we can use the pegasus data management process to copy the container and the files from the node to the worker and inside the container. All the data created in the container can be transferred back to the node as well.

Coming back to the workflow itself, running it would only allow to compute the optical spectrum of the Silane molecule in one configuration. In order to compute it for the five realistically relevant parameters discussed in the previous part one may want to run that sequential workflow concurrently, each of them starting with a different parameter for the Si-H bond length. We had to add a starting step which will prepare the 5 independent workflows (with the right configuration), and a final step which will do the analysis, here only to graph the 5 results on the same graph: it only consists of superimposing the 5 plots made using matplotlib in a short python script.

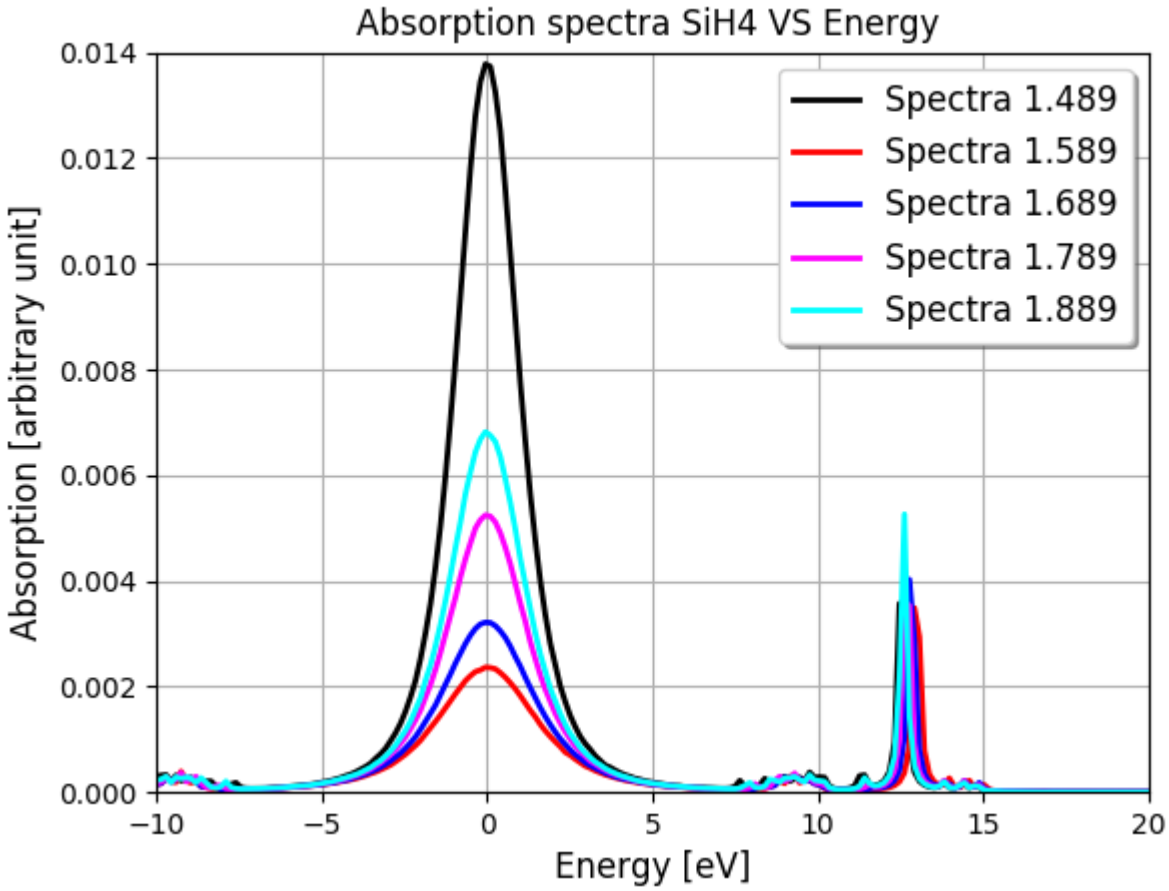
Using FireWorks GUI, this looks like the following. It is difficult to read the name of each step in the plot below, but the names are irrelevant. This graph is only here to highlight the workflow structure as described above.


Physical interpretation

The graph obtained from the workflow is given below. One can draw some preliminary observations from it.

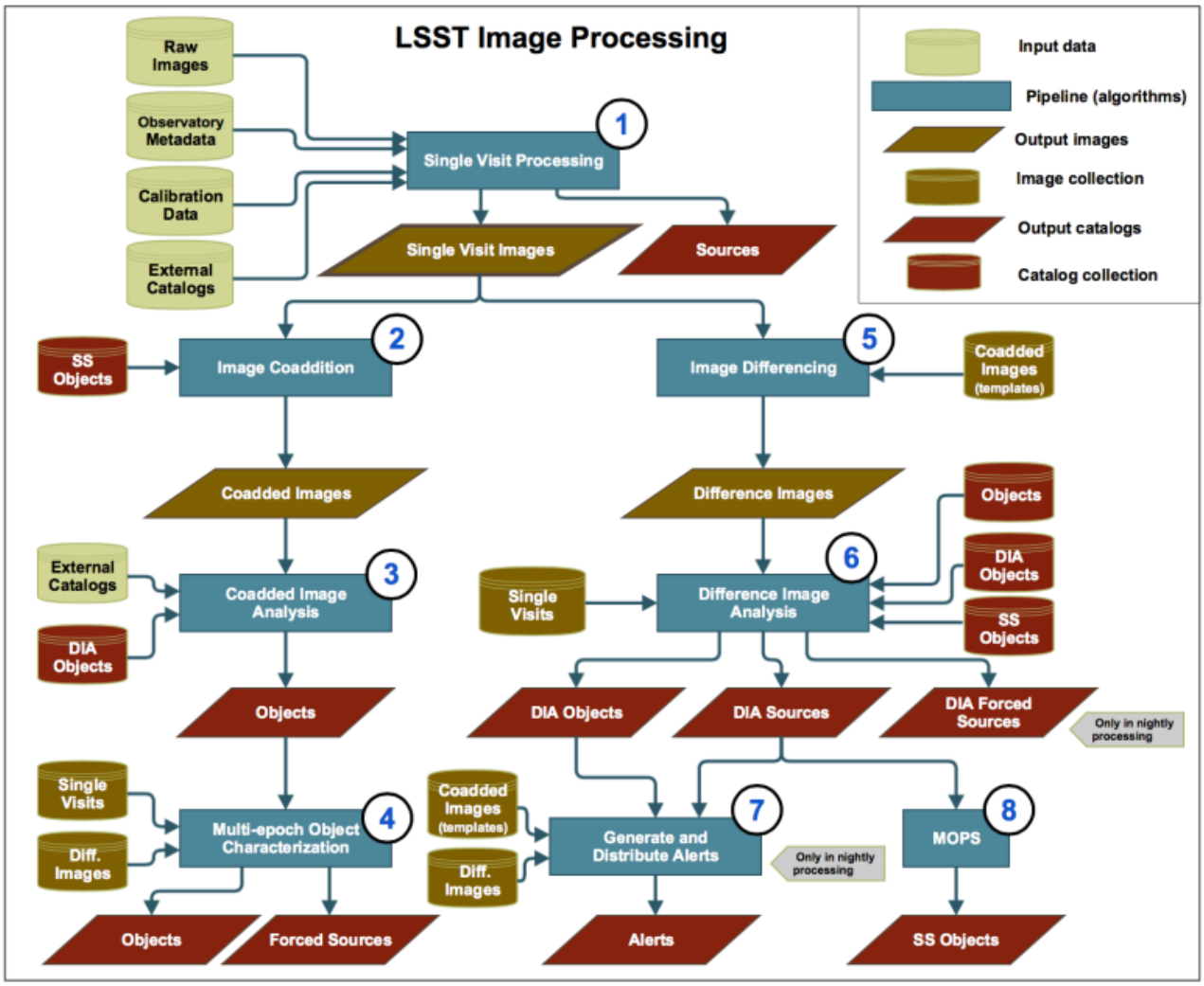
The absorption spectra are connected with the symmetry of the molecule. The starting position of the molecule (which corresponds to the relaxed geometry “Spectra 1.489”) has every hydrogen in the 4 vertex corners of a regular tetrahedron. By changing the Si-H bond distance of 1 hydrogen we increasingly distort this regular symmetry. In turn the excited states and the probability of transition (from the ground state to the excited states and back) are modified accordingly.

The geometry distortion we imposed modifies the magnitude of the main peak (on the left) and produce a small shift in the frequency of the secondary peak (on the right), as one can see in the graph below.




## 3.2 The Large Survey Synoptic Telescope

We also looked to the expected workflow of LSST, the Large Survey Synoptic Telescope [41]. LSST is a wide-field survey reflecting telescope with an 8.4-meter primary mirror currently under construction, that will photograph the entire available sky every few nights. The computing model is also being designed, but one can find some useful description of it and the expected workflow in the Data Products Definition Document [5] which describes the conceptual design of the computing model. The appendix A of that document discusses the conceptual pipeline design, which can be illustrated by the following schema (extracted from that document).




The LSST workflow is relatively simple with 2 almost independent branches, as seen in the above schema. The main difficulty is the data management (and more specifically the data each step required), as it was for the previous use case we studied. LSST deals with sky images, but in various formats. There are especially three formats that will impose some constraints on the workflow: the single visit images, which are created at step 1, what LSST called coadded images (several images have been added in order to put the emphasis on low intensity objects in the picture), created at step 2, and finally the difference images created at step 5. One can note that the left branch construct the coadded images and thus studies the low intensity objects in the sky, where the right branch "subtract" images one to the other (creates the difference images) in order to exhibit temporary object in the sky. It is also important to notice that the workflow refers to collection of such images, which we will interpret as a set of such images (set of a given, more or less high but defined, number of images).

Following this interpretation of the current model design, the main difficulties lie in step 4 and 6 and 7 which require these images collections. We then first need to build these collections of images before being able to run these steps.

The trick to implement such a workflow is to ensure the collection of images are properly created before the steps which require them are triggered. The workflow managers we evaluated provide some features which could help meeting these conditions. For Parsl and FireWorks, since one can directly modify the Python code,

we could add a simple loop over the steps which create these collections. For Makeflow one could take advantage of the JX language which offers a programmable approach for defining a workflow (which would simply result in a kind of loop for the steps which create the collections).

However, the LSST software stack, and in particular the data management part is still under development, and it is not possible, at the time of writing this report, to fully implement each step of the workflow. We then implemented the workflow without implementing the computing application and data access of each step. Implementing such a workflow was not difficult, but of course more problems may arise when dealing with the real applications and constraints from the LSST data management system.

## 3.3 Summary

We have evaluated the functionalities of four workflow managers: Parsl, FireWorks, Makeflow and Pegasus. We also use them to implement two real use cases taken from the scientific community: a Solid State Physics and an astronomy one.

From this work, we can draw some preliminary observations and rough conclusions about these tools. We should indeed emphasize that these tools are still under active development and these conclusions could only be considered valid at the time of writing this report. Moreover, we confronted these tools against two real use cases, which is of course not enough to really gain a good overview of the capabilities each tool provides.

The first criteria that might be used is the learning curve. On that point, there are two kinds of managers: the simple ones, that are quite recent and aim to do only one (more or less specific) thing; and the more complex ones, often older and with more functionalities. FireWorks and Parsl belong to the first ones: as they describe the workflow in Python, they are quite easy to use at first while offering the basic features we expect from such tools. Of course, this ease of use comes with a previous knowledge of Python; without it, those tools might lose in attractiveness. Even knowing Python, the learning of the specific Python syntax (each tool implements the workflow definition in a different way) can be confusing and making specific workflows can be quite time consuming. However, as those managers tend to be either recent or quite confidential, they lack public information other than the documentation. It is hard to find working examples, tutorials or active community other than the ones maintained by the developers.

On the other hand, managers like Makeflow and Pegasus are more complex as they need specific syntax and are less flexible. Of course, those downsides are counterbalanced by the powerfulness of those tools, which offer many advanced features. As they are well established products, they have large communities on the internet and it is easier to find resources online or at events (workshops, papers, meetings with developers).

It should be noted that the installation of FireWorks, Parsl or Makeflow is pretty straightforward (Python module for the two firsts; classical compilation for the latter), whereas Pegasus needs HTCondor as a prerequisite.

In term of features, they all offer the basic set that will be needed to orchestrate a workflow: dependencies between tasks, different kind of workflows (linear, diamond, dynamic depending on input set), errors management, etc. In our case, they also all work well with Singularity, and this specific point was not the main question nor the main difficulty we could have.

But, even if they offer the same features (or very similar ones), one will not implement the same workflow in the same way with each of these tools. The simpler tools offer more flexibility; FireWorks in particular seems to be the more flexible and the easiest workflow manager we evaluated. Indeed, once the specific syntax is learnt and the principles of the manager are understood, it is easy to create custom workflow that can be quite complex. Once again, the problem will probably be the lack of documentation and the time needed to discover all the features offered (which will require to dig into the code or discuss with the developers).

Parsl is the same kind of product, but is still quite green. As it is still under development, we can expect Parsl to be mature in a few years (the current version is 0.7, so not even 1, i.e. an expected stable version). Basic

workflows are easy to design, but some more advanced workflows can be tricky to create. Parsl is also very different in its philosophy, as the workflows are dynamically created and not fixed before execution as it is the case for every other workflow manager tested. This tool is definitely more suited for people who have a previous knowledge of Python.

Makeflow runs workflow from a makefile like definition file. This file is quite easy to write and implementing a simple workflow will be done quickly. However, this simple syntax will make it not suited for more complicated workflows, or when dealing with numerous input and output files. For such cases, we will have to define the workflow using the JX language. As discussed above, the way Makeflow supports Singularity and deals with wrappers, is not versatile. As Makeflow is compiled, one also has to interact with the system using the command line. In particular, Makeflow does not provide a GUI or a user-friendly way to follow the workflow running. This could only be done by requesting Makeflow, or directly the batch system, by command lines. Makeflow is a mature product (and seem quite reliable), with a very active and responsive development team which will help the users to solve any issues they could have.

Pegasus is quite different from the others as it uses a DAG process to execute the workflow. The user needs to have knowledge in python to be able to write the workflow description using their API but after that, Pegasus handles the whole process by providing data management functionalities and cleaning tasks from the workflow. It's a quite powerful tool but examples are not that easy to find and the document is pretty dense. However, Pegasus developers are really easy to contact and can offer support for each specific use case.

Once the workflows are running, FireWorks really stands out with its GUI. This web interface offers a live monitoring of the workflows and allows to understand graphically what's going on. This is a really powerful feature and one of the biggest positive point for FireWorks.

For Parsl and Makeflow, following workflows execution has to be done using command line and log files analysis. Makeflow also offers a non-dynamic graphical interface showing the workflow and the status of the tasks. Pegasus provides a bunch of command line tools to debug workflows and give statistic information on the workflow execution and the different tasks. It also provides tools to generate a schema and a GANTT chart. All this information can also be found in the web interface which makes it really powerful to follow, resume, cancel and debug workflows.

# 4 Conclusions and Outlooks

The main goal of this deliverable was to evaluate workflow managers and highlight potential constraints when running containerized applications. For this work, we focus on Singularity as the container technology. Parsl and FireWorks don't provide native Singularity support, whereas Makeflow and Pegasus do. However, it quickly appears that providing a native Singularity support was not necessary, as one can always use a wrapper to enable the functionality.

The evaluation eventually came back to the workflow manager itself, especially when it comes to implement a real workflow. The features offer by the tool to satisfy at best the constraints of the workflow, and the easy use of the tool were the most important points.

At the end, it seems to be no workflow manager that rises above all others and stands out as the best that should be used in all cases. Instead, the choice of the workflow manager will depend on the use case and on the team that will use it: if some of the team have previous knowledge of one of the tools or of Python, that will probably impact the choice.

Over all, FireWorks and Parsl seem to be well suited for simple workflow. These tools are quite easy to install, and it should be easy to implement such workflows, especially if some people of the team have a good knowledge of Python. FireWorks, thanks to its nice GUI, is probably the best choice in this case.

Makeflow is also easy to install, and require no specific knowledge. It will however demand more efforts in order to implement the workflow, especially if one has to use the JX language or any other advanced features

the tool provides. It also provides a better portability as the workflow definition does not depend on any language version (which is the case of Parsl and FireWorks). The (slightly) steeper learning curve will be worth it for complex workflows.

Pegasus is a very mature product offering a wide range of advanced features. The cost of this quite powerful tool is a really steep learning curve compared to the other tools. Pegasus installation and configuration are more complex, and the tool strongly relies on HTCondor. This system should probably be favoured for really complex and big workflows, for which the benefits will be worth the time investment.

To go further with this study, one should probably focus on data management. In this work, the data we have to dealt with were small files (text files), and copying them or accessing them from a shared file system was easy and reliable. Dealing with much bigger files could add more constraints, and the way the data are accessed and read may impact the overall performances.

# 5 References


[1] Docker home page: https://docs.docker.com/
Singularity home page: https://www.sylabs.io/docs/
[2] Kubernetes home page: https://kubernetes.io/fr/
[3] Apache Mesos home page: http://mesos.apache.org/
[4] Apache OpenWhisk home page: https://openwhisk.apache.org/
[5] "LSST/LSE-163" GitHub. Accessed April 8, 2019. https://github.com/lsst/LSE-163
[6] "DMTN-025: A Survey of Workflow Management Systems." Accessed March 29, 2019. https://dmtn-025.lsst.io/
[7] "Research - Cooperative Computing Lab."Accessed April 9, 2019. http://ccl.cse.nd.edu/research/
[8] "Cooperative Computing Lab." Accessed April 9, 2019. http://ccl.cse.nd.edu/software/downloadfiles.php
[9] "Cooperative-Computing-Lab/cctools." GitHub. Accessed April 9, 2019. https://github.com/cooperative-computing-lab/cctools
[10] "The Makeflow Workflow System - Cooperative Computing Lab." Accessed April 9, 2019. http://ccl.cse.nd.edu/software/makeflow/
[11] "CCL Workshops, Tutorials, and Meetings - Cooperative Computing Lab." Accessed April 9, 2019. http://ccl.cse.nd.edu/workshop/
[12] "CCT Google Group." Accessed April 9, 2019. https://groups.google.com/forum/#!forum/cctools-nd
[13] "Work Queue: A Flexible Master/Worker Framework - Cooperative Computing Lab." Accessed April 9, 2019. http://ccl.cse.nd.edu/software/workqueue/
[14] "Makeflow User's Manual." Accessed April 9, 2019. http://ccl.cse.nd.edu/software/manuals/makeflow.html#basic
[15] "JX Makeflow." Accessed April 9, 2019. http://ccl.cse.nd.edu/software/manuals/makeflow.html#jx
[16] "Garbage Makeflow." Accessed April 10, 2019. http://ccl.cse.nd.edu/software/manuals/makeflow.html#garbage
[17] "Container Makeflow." Accessed April 9, 2019. https://ccl.cse.nd.edu/software/manuals/makeflow.html#container
[18] "Wrapper Makeflow". Accessed April 9, 2019. https://ccl.cse.nd.edu/software/manuals/makeflow.html#wrapper
[19] "Introduction to FireWorks (workflow Software) - FireWorks 1.8.7 Documentation." Accessed April 4, 2019. https://materialsproject.github.io/fireworks/
[20] Jain, Anubhav, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, et al. 2015. "FireWorks: A Dynamic Workflow System Designed for High-Throughput Applications." Concurrency and Computation: Practice and Experience. https://doi.org/10.1002/cpe.3505
[21] "FireWorks Google Group." Accessed April 9, 2019. https://groups.google.com/forum/#!forum/fireworkflows
[22] "NSF Award Search: Award#1550588 - Collaborative Research: SI2-SSI: Swift/E: Integrating Parallel Scripted Workflow into the Scientific Software Ecosystem." Accessed April 4, 2019. https://www.nsf.gov/awardsearch/showAward?AWD_ID=1550588&HistoricalAwards=false
[23] "Container Support - Parsl 0.7.2 Documentation." Accessed April 9, 2019. https://parsl.readthedocs.io/en/latest/userguide/containers.html
[24] "Roadmap - Parsl 0.7.2 Documentation." Accessed April 4, 2019. https://parsl.readthedocs.io/en/latest/devguide/roadmap.html
[25] https://pegasus.isi.edu/pegasus-timeline/
[26] https://pegasus.isi.edu/documentation/api.php

[27] https://pegasus.isi.edu/documentation/example_workflows.php
[28] https://pegasus.isi.edu/documentation/tutorial.php
[29] https://pegasus.isi.edu/documentation/job_clustering.php
[30] https://pegasus.isi.edu/documentation/hierarchial_workflows.php
[31] https://pegasus.isi.edu/documentation/transfer.php#ref_data_staging_configuration
[32] https://pegasus.isi.edu/documentation/container-transfers.php
[33] https://pegasus.isi.edu/documentation/container_examples.php
[34] https://pegasus.isi.edu/documentation/monitoring_debugging_stats.php
[35] https://pegasus.isi.edu/documentation/monitoring_amqp.php
[36] https://www.quantum-espresso.org/
[37] http://www.yambo-code.org/
[38] https://www.abinit.org/
[39] MAL Marques, A. Castro and A Rubio, J. Chem. Phys. 115,3006 (2001).
U. Itoh, Y. Toyoshima, H. Onuki, N. Washida, and T. Ibuki3, J. Chem. Phys. 85, 4867 (1986).
J. C. Grossman, M. Rohlfing, L. Mitas, S. G. Louie, and M. L. Cohen, Phys. Rev. Lett. 86, 472–475 (2001).
R. J. Needs, P. R. C. Kent, A. R. Porter, M. D. Towler, G. Rajagopal, Int. Journal of Quantum Chem.,86, 218 (2002).
M. Grüning, A. Marini and X. Gonze, Nano Letters,9, 2820 (2009). (Grüning, Marini, and Gonze 2009).
[40] "SiH4 Ground State." Accessed April 10, 2019. http://www.yambo-code.org/tutorials/files/SiH4_ground_state.zip
[41] "Welcome - The Large Synoptic Survey Telescope." Accessed April 10, 2019. https://www.lsst.org/