MAESTRO
DATA ORCHESTRATION

D4.3
Initial Specification of Workflow Management and Optimisation

Work package	WP4 High-level Middleware	
Author(s)	Jayesh Badwaik	JUELICH
	Salem El Sayed	JUELICH
	John Biddiscombe	ETH Zurich / CSCS
	Maxime Martinasso	ETH Zurich / CSCS
Reviewer #1	Simon Smart	ECMWF
Reviewer #2	Dirk Pleiter	JUELICH
Dissemination Level	Public	
Nature	Report	

Date	Author	Comments	Version	Status
05.01.2021	Jayesh Badwaik	Initial Draft	V0.01	Draft
28.04.2021	Jayesh Badwaik	Improved Outline	V0.02	Draft
12.05.2021	Jayesh Badwaik	Initial list of optimisations	V0.03	Draft
20.05.2021	Jayesh Badwaik	More information on Montage	V0.04	Draft
21.05.2021	Salem El Sayed	Added executive Summary, introduction and bibliography	V0.05	Draft
20.05.2021	Jayesh Badwaik	Multiple edits and readded storage layer	V0.06	Draft
26.05.2021	Jayesh Badwaik	Added resource awareness and edited Montage section	V0.07	Draft

09.06.2021	Jayesh Badwaik	Added more information	V0.08	Draft
11.06.2021	John Biddiscombe	Added workflow execution framework contents	V0.09	Draft
14.06.2021	John Biddiscombe	CDO section last sentences, montage workflow and resource handling edits	V0.10	Draft
15.06.2021	Jayesh Badwaik	Added resource aware workflows and resource description, along with Montage DAX and optimisations	V0.11	Draft
15.06.2021	John Biddiscombe	More on resource management, caching and general descriptions	V0.12	Draft
15.06.2021	Jayesh Badwaik	Added sections on resource description and optimization problems	V0.13	Draft
16.06.2021	Jayesh Badwaik	Reorganized writing of resource description and added some images	V0.14	Draft
17.06.2021	John Biddiscombe	Resource annotations, other cleanups, DAG handling	V0.15	Draft
17.06.2021	Christopher Haine	Added some comments	V0.16	Draft
18.06.2021	Ali Mohammed	Added more comments	V0.17	Draft
18.06.2021	Maxime Martinasso	Changes in the dynamic provisioning description, overall minor improvements.	V0.18	Draft
22.06.2021	Salem El Sayed	Added concluding remarks	v0.19	Draft
22.06.2021	Christopher Haine	Added some comments	V0.20	Draft
22.06.2021	Jayesh Badwaik	Addressed and responded to comments	V0.21	Draft
22.06.2021	Jayesh Badwaik	+ more responsed to comments	V0.22	Draft
23.06.2021	Christopher Haine	Corrected author list	V0.23	Draft

23.06.2021	Jayesh Badwaik	Listing requirements that should not be missed	V0.24	Draft
25.06.2021	Ali Mohammed	Replied to comment	V0.25	Draft
25.06.2021	John Biddiscombe	Addressed some of the comments and add more material on watchers/caches	V0.26	Draft
08.07.2021	Christopher Haine	Further replies to comments	V0.27	Draft
12.07.2021	Jayesh Badwaik	Final Draft for Internal Review	V0.28	Draft
12.07.2021	Jayesh Badwaik	Typo Fixes for Internal Review	V0.29	Draft
25.08.2021	John Biddiscombe	Remove root node CDO objects for Internal Review	V0.30	Draft
30.08.2021	Salem El Sayed	Addressed comments in the executive summary, introduction and conclusion	V0.31	Draft
29.11.2021	Jayesh Badwaik	Final Version	V1.0	Final Version

DATA ORCHESTRATION IN HIGH PERFORMANCE COMPUTING
This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

Executive Summary

As part of the work done in Task 4.2 and Task 4.3 of the Maestro project an execution framework prototype was developed and was documented in D4.2. This report is part of Task 4.3 and 4.4 and continues that development effort. The target is to improve on the execution of a given workflow by introducing the Maestro framework. To achieve this, this deliverable lists the initial specifications of workflow management and the possible optimisations to the execution of a workflow. The final version and demonstrators will be released in D4.5 and as part of work done in WP6.

Contents

Glossary

CDO	. . . Core Data Object
MIO	. . . Maestro I/O interface
PM	. . . Pool Manager

1 Introduction

A workflow describes a set of applications or tools dependent on each other's data output. The relevant decisions on executing each of these steps of a workflow is typically delegated to a workflow manager. The workflow itself is usually described in a language that is dictated by the used workflow manager. The data between workflow steps or workflow components are typically transferred and stored on the external file systems. It is worth noting that we distinguish between workflows and workloads, the latter describing a combination of a workflow, configurations and a specific data set and can be thought of as a usage scenario of a workflow.

Since Maestro possesses data and memory awareness it can aid in improving the workflow execution. In a previous deliverable D4.2 [1], we dealt with creating an execution framework prototype. As a proof of concept, we showed how the execution framework can switch the data movements of a workflow executed by a workflow manager called Pegasus from files in a file system to Maestro CDOs. Specifically, the previous deliverable D4.2 [1] formulated the data flows with graphs and allowed us to solve the correctness problems in the workflows. However, since then we have added resource awareness (compute, memory, storage, network, ... etc.) to the data flows, which requires revisiting some of the previous problems. Adding resource awareness and smartness to the execution framework, it is possible to improve the execution and resource utilisation of a workflow. In other words, an updated formulation of the graph with resources will go into the flow section of formulation. This would make it possible to formulate the problem of workload scheduling and optimisation into that of a bin-packing [1].

In this deliverable, we discuss the initial design of introducing resource awareness into the execution framework and better workflow management. We also introduce some of the possible optimisations. To show and test these changes, we will introduce a workflow use case called Montage, which uses Pegasus as a workflow manager.

The remaining document is organised as follows, in Section 2 we introduce the Montage workflow which will act as our use-case scenario, in Section 3 we introduce resource awareness into the workflows, in Section 3.2 we introduce an example optimization problem, in Section 4 we discuss handling resource descriptions and related optimization and then in Section 6 we reintroduce the execution framework designed in D4.2 [1] along with some necessary changes. Finally in Section 7 we list the set of optimisations planned for development. We finish the report with some concluding remarks in Section 8.

[1]The bin packing problem is a known optimisation problem referring to the packing of different items into a finite number of bins or containers. The target is to minimize the number of bins used.

1.1 Relation to other parts of the project

This report builds on the results of Task 4.2 and Task 4.3, which were documented in D4.2 [1]. The work highly depends on the efforts of WP3 in designing and developing the Maestro-core, which was documented in D3.1 [2] and D3.3 [3]. Additionally, WP3 has initiated the need for resource description and awareness as part of the Maestro framework, which is documented in D3.4 [4]. The demonstration of the execution framework and related components will be released as part of the efforts in WP6. The final results and software will be released in D4.5.

2 Workflow Use Case: Montage

Montage is a portable toolkit for constructing custom, science-grade mosaics by composing multiple astronomical images. The mosaics constructed by Montage preserve the astrometry (position) and photometry (intensity) of the sources in the input images. The mosaic to be constructed is specified by the user in terms of a set of parameters, including dataset and wavelength to be used, location and size on the sky, coordinate system and projection, and spatial sampling rate. Many astronomical datasets are massive, and are stored in distributed archives that are, in most cases, remote with respect to the available computational resources.

Montage is an example of a class of well-specified deterministic workflows that are common in science. These workflows usually consist of a series of codes (i.e. components) connected together to perform large-scale analysis routines. Other examples of this class of workflows include: seismic hazard analysis for earthquake forecasts, analysis of large-scale social networks, analysis of the epigenomic properties of DNA sequences, searching for gravitational waves in interferometer data, and many others [5].

The inputs to the workflow include a "template header file" that specifies the mosaic to be constructed, and several input images in standard FITS format (a file format used throughout the astronomy community) . Input images are taken from archives such as 2MASS. The input images are first re-projected to the coordinate space of the output mosaic. The re-projected images are then background rectified and co-added to create the final output mosaic. Figure 1 shows the structure of a small Montage workflow using vertices to represent tasks and edges to represent data dependencies between tasks. Montage workflows typically contain a large number of tasks that process large amounts of data. For example, a workflow to generate the 2 degree square mosaic of 2MASS images centered around the celestial object M17 would contain approximately 1,000 individual tasks.

Montage is considered to be I/O-bound because on large workloads, if often spends 70-95% of its time waiting on I/O operations [6, 7]. In typical workflow, Montage consumes around 2 GiB of data and outputs around 4 GiB of data per file. The characteristic of Montage that seems to have the most significant impact on its performance is the large number (29,000) of it accesses.

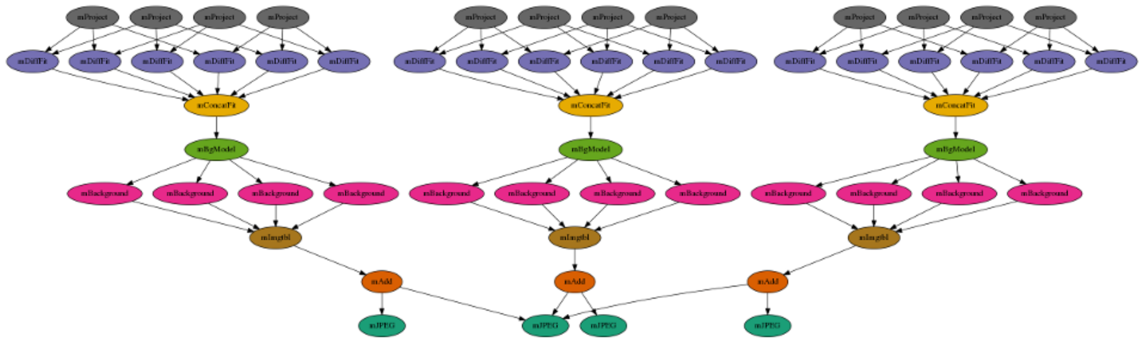

Figure 1: Montage Workflow Example 1

One can see this in the improved performance on systems when the file system can cache both reads and writes in the memory [6]. On large workflows on distributed nodes, studies have also shown that prefetching at least part of data on the destination node increases performance.

The most important operation is reading and writing files, which is an explicit feature of the Montage. However, the transfer of data between nodes is another important feature which impacts tha performance of the workflows. This is evinced by the observation that filesystems like MemFS provide better performance for Montage workflows [8]. Specifically, the design of uniformly distributing file stripes across the storage nodes belonging to an application allow better balance of the data across the storage servers, leading to better performance for Montage workflow.

2.1 Specifying the Workload

For our experiments with Montage, we will consider a Montage prototype workflow[2]. The prototype creates a Pegasus workflow which then executes tasks to create a montage of astronomical images. The workflow generator provides a configurable parameter named the degree. By configuring the degree of the workflow, one could either widen or narrow the workflow graph. For example, for a 2 degree workflow, a simplified version of the workflow will be similar to that shown in Figure 1.

2.2 Maestro Enabled Workflow

As noted above, Montage takes files as input and outputs images as files. A major task in porting Montage based workflow to Maestro would be to port the file read/write functions to Maestro based CDO operations. Currently, Montage uses Pegasus workflow which in turn uses the files generated by Montage components as dependencies in the workflow diagram. Maestro, in

[2]https://github.com/pegasus-isi/montage-workflow-v3

contrast, uses CDOs to transfer data between applications. This will require changes to the input and output functions of Montage components.

In the aspect of workflows, Maestro-enabled Montage components can take advantage of the fact that the data transfer (in terms of CDOs) can happen through the memory instead of through the global file system. This can allow for certain optimizations, for example, With these things in account, we are expecting the following steps to port Montage to Maestro:

- Port the individual components to input CDOs instead of files. This will require creating a fork of Montage which is Maestro enabled. In particular, this would require replacing file read/write functions in Montage with Maestro enabled functions.
- Use a workflow execution environment to generated optimized Pegasus workflows for Montage. In particular, this would require using the knowledge of the dependency graph and the systems to schedule jobs on certain nodes so that the communication between nodes is minimized while at the same time, making maximal use of all the available nodes.

2.3 Evaluation Metrics

In this use case, Maestro is supposed to use the data awareness of workflows to schedule tasks in a manner such that data locality is improved. This along with faster CDO transfers (as opposed to filesystem read and writes) should provide for an improved performance.

2.4 Licensing Information

The Montage workflow consists of three different software.

1. Montage v6.0 `https://montage.ipac.caltech.edu`

   License: BSD 3-clause License

2. Pegasus v5.0 `https://pegasus.isi.edu/`

   License: Apache 2.0

3. Astropy v4.2 `https://www.astropy.org/`

   License: BSD 3-clause License

3 Resource Aware Workflows

In deliverable D4.2, we analyzed dataflows in context of workflows and provided techniques to ensure the correctness of workflows. The analysis assumed that the resources are not a constraint on the data transfer. In this section, we introduce resource awareness to the dataflows and try to formulate it in a manner which will allow us to make intelligent decisions about both workflows and allocation of resources to the applications. Due to significant amount of changes to the older model, for the benefit of the reader we describe the complete model here.

As in the deliverable D4.2, the motivation to analyze the workflows is to be able to make decisions with respect to the tasks of the workflows, and do so in a manner which does not require any workflow manager specific modification in the tasks itself. Any strategy for optimizing workflows should take into account both static and dynamic nature of data dependencies in the workflow. One of the ways to do so would be to only consider information that is available in dynamic workflows, namely the workflow that has already been scheduled and the next task which has to be scheduled. However, such a strategy would be suboptimal when applied to static workflows where much more information is available.

In order to avoid pitfalls of both the methods when modelling the workflows, we consider an `a posteriori` view of the workflow. That is, the workflows are analyzed as if the complete information about the workflow is available during analysis just like in a static workflow. However, it is during the analysis and optimization part, where we differentiate between the static and dynamic workflows. For dynamic workflows, the optimization algorithm can only use information from the dependencies to make a decision for scheduling a task, while for static workflow, the whole workflow can be used for making scheduling decision.

3.1 The Execution View and the Data View of a Workflow

Consider a workflow consisting of three applications A, B and C such that the application A produces CDO 1 and 3, application B produces $2, 4$ and 5 while application C produces 6. Further, assume that application A requires CDO 2 (which is produced by application B) to produce CDO 3. In the similar manner, assume that application B requires CDO 3 to produce 4 and 5, and finally that application C requires CDO 5 to produce CDO 6. Let us now define the term CDO dependency.

Definition 1 (CDO Dependency)**.** *We say that a CDO x depends on a CDO y through an application P if an application P requires CDO x in order to produce CDO y.*

With this definition of CDO dependency and the data flow information from $D4.2$, we can define the above workflow as a graph as shown in 2.

Let us now define some terms in order to be able to speak of the above workflow in more concrete terms. To improve the expository quality of the description, some of the terms from deliverable D4.2 are redefined here. Please refer to D4.2 for more details on them.

Definition 2 (Informational Attribute (I-value) for CDO)**.** *If it is a priori known that a CDO is going to be produced in a workflow, then the CDO is assigned a*

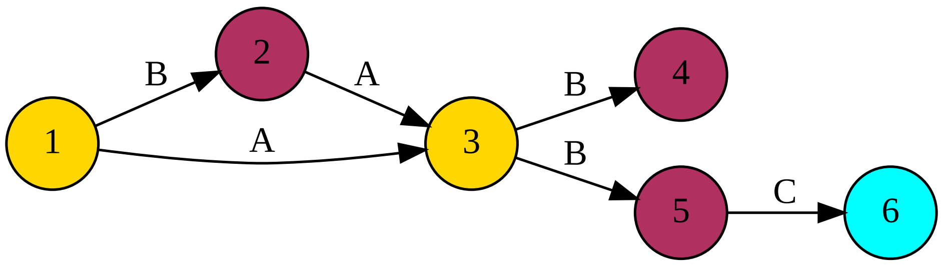

Figure 2: Data View of a Workflow

"Static" attribute. Otherwise the CDO is assigned a dynamic temporal attribute. In order to have a concise notation, we call this attribute the I value for a CDO and the I value can either be S for Static or D for Dynamic. For a CDO x, the I value can be denoted by $I(x) = S$ or $I(x) = D$. For more information on these attributes, please refer to D4.2.

As an example of a dynamic CDO, consider an iterative solver which outputs a CDO at every iteration. In such an iterative solver, the number of iterations are often not known apriori, and therefore, it is not clear whether the 100-th iteration of the solver will be executed. Therefore, the 100-th CDO (if it is produced) will be labeled with a dynamic attribute. However, assume that the iterative solver has been programmed in such a manner that it is guaranteed to execute at least 4 iterations of the solver. Then we know for sure, that the 3-rd CDO will be produced even before the workflow has been executed, and therefore the 3-rd CDO will be marked as Static.

Definition 3 (Execution Attributes (W-value) for CDO Dependency)**.** *If a CDO dependency requires an application to start execution, then dependency is labeled as an Execution dependency otherwise, the dependency is labeled as Normal dependency. In order to have a concise notation, we call this attribute the W value for a CDO dependency and the W value can either be E for Execution or N for Normal. For a CDO dependency x, the I value can be denoted by $W(x) = E$ or $W(x) = N$. For more information on these attributes, please refer to D4.2.*

In D4.2, we saw that, sometimes, it is an optimization for an application to start as soon as the required data becomes available, but no earlier. For example, in the graph in Figure 2, consider that the application B starts execution when CDO 1 becomes available. In this case, the production of CDO 1 can be thought of as the trigger to start application B which will then produce CDO 2. In this case, the dependency between CDO 2 and CDO 1 will be marked as an execution dependency. In contrast, if one looks at the dependency between CDO 2 and CDO 3, then both the applications are already started, and therefore there is no need to startup any application. In this case, the dependency would be labeled as Normal.

During a workflow, it can often happen that the producer application has already produced a CDO and wants to exit but the consumer application is not yet ready to receive the CDO. These can often lead to correctness concerns. In order to track these situation, we introduce the temporal label for a CDO dependency.

Definition 4 (Temporal Attributes (T-value) for CDO Dependency)**.** *If the producer which produces a CDO is ready to withdraw the CDO and the consumer is not ready to consume the CDO yet, then we denote the CDO dependency as an Awaiting dependency, otherwise, the CDO dependency is labelled as a Fulfilled dependency. In order to have a concise notation, we call this attribute the T value for a CDO dependency and the T value can either be A for Awaiting or F for Fulfilled. For a CDO dependency x, the T value can be denoted by $T(x) = A$ or $T(x) = F$.*

Finally, we introduce resource awareness in our workflow. Assume that the application A takes an inventory P of resources to produce CDO 1 and then inventory Q of resources to produce 3. An optimization problem for the workflow should be able to refer to these two different operations differently. In order to do so, we annotate the application label with the CDO that the dependency graph is producing. So, in this case, the CDO dependency between 2 and 3 and between 1 and 3 will be marked as A_3 instead of A, with A_3 denoting the inventory of resources consumed during production of CDO 3 by application A. We call the A_3 part of the application A as an application phase. In order to provide the same information for CDO 1, we introduce a ghost "Start" node at the start of workflow and introduce an "End" node at the end for symmetry. All this information can now be combined into a graph, giving us a resource-aware data view of the workflow as shown in figure 3.

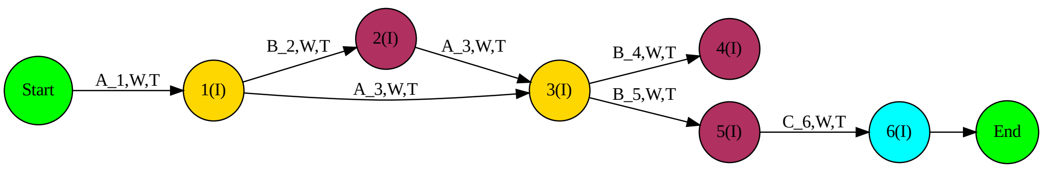

Figure 3: Data Flow View

We now need a good way to describe the resource consumption in a workflow. If we look at the resource aware data view of the workflow in 3, we will see that there are multiple edges which are associated with the same application phase. For example, the A_3, W, T from 1 to 3 and from 2 to 3 is the same application phase.

In order to be able to reason well about such resources, we consider the line graph of the above graph. This gives us the graph in 4. The $A_3(1, 3)$ refers to the edge between CDO 1 and CDO 3 in the dataflow graph, and similarly for node $A_3(2,3)$ corresponding to edge between CDO 2 to CDO 3.

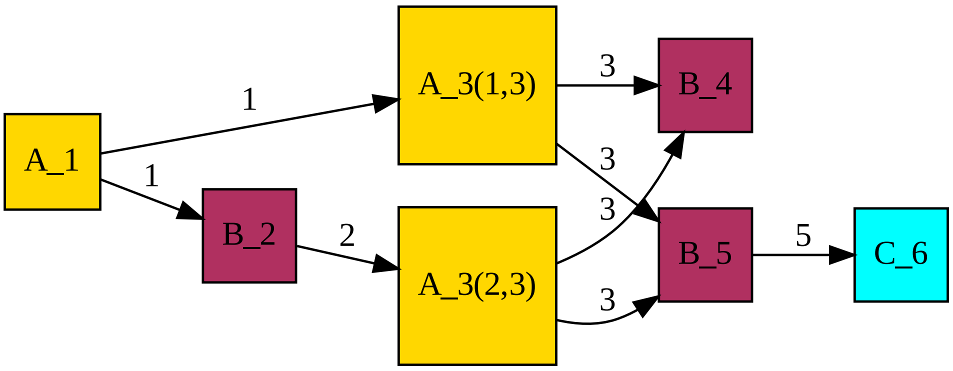

Figure 4: Line Graph of Data View

As we can see here, this line graph has a duplicate node for node A_3 which can be detected by the fact that both have the same name and both produce the same CDO 3. We merge the two nodes into a single node, which gives us an a merged line graph of the data flow. We call this the execution view of the workflow.

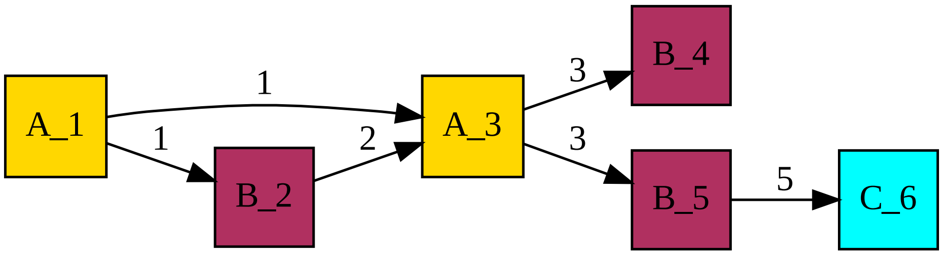

Figure 5: Execution View of the Workflow Flow

With this graph, one can now talk about an optimization problem in terms of resources. For example, one can say that resources A_3 can only be consumed after consuming B_2, while the resources B_4 and B_5 can be consumed concurrently. Furthermore, the resources B_4 and C_6 can also be consumed concurrently.

3.2 An Example Resource Optimization Problem

With the resource aware workflows introduces, we now consider a sample optimization problem and show how resource awareness can improve performance.

Assume that CDO 1 is on node 1 of the system while application A is supposed to run on node 2. In this case, the data on node 1 will first have to be copied to node 2 before application A can transform it to CDO 2. Assume that the Q resources are required to copy the data from node 1 to node 2 and R resources are required to transform CDO 1 to CDO 2. For the purpose of the example, assume that Q is a 10 GiB/s network link over a period of 120 seconds while R is 256 cores over a period of 120 seconds.

One thing we can observe for the workflow is that resources R can only be consumed after resources Q have been consumed. As noted, in a classical scenario, due to the way workflow managers work, one can only specify the resource requirements at the start of the job and a time for which they are required. This will result in the workflow manager requesting access to a 10 GiB/s network and 256 cores for 240 seconds for application A.

With a data and a resource aware workflow execution framework however, the workflow execution framework can see that the CDO 1 is being produced on node 1 and being consumed on node 2. With this observation, the translator can add an additional job M called a cache worker that will run on node 2 and copy the CDO 1 from node 1 to node 2. And now, the workflow manager can assign a $120GiB/s$ network for 120 seconds to task M and a 256 core machine for 120 seconds to task A, thereby providing an optimization to the workflow.

4 Resource Description and Optimization

In Section 3, we introduced resource aware workflows and were able to formulate a language to describe resource requirement of various tasks in a workflow graph. Next, we need to be able to reason about performances of various workflow graphs in order to be able to make optimizations decisions for the workflow. For this purpose, we need a description of the available resources and their characteristics. The description of resources together with a resource requirement can allow us to formulate a resource optimization problem.

There are two aspects one needs to consider when one is trying to describe the resources from the purpose of the optimizing your workflows. The first aspect is to ensure that it is feasible to allocate a given configuration of resources to a certain task. There can be many different factors in the feasibility. For example, in case of computing cores, it is desirable for the cores to be allocated exclusively to one task. In contrast, for disk storage, it is quite possible to allocate access to multiple tasks at once, but there is a limit on the total allowed storage. Furthermore, a lot of sites have a policy of only allocating complete nodes. This means that you cannot have one task using GPUs from a node whose CPUs are already being consumed by a different node. This aspect of resource description is generally only dependent on the site, and is quite independent of the workflow.

The second aspect one needs to consider are variances in the performance of different resources. For example, certain filesystems might have a better performance profile for small random files while other might have better performance profiles for large sequential files. Depending on the performance profile of the workflow, the first characteristics might be a better indicator to predict performance than the second indicator. As we can see, the performance characteristics of the system are not as independent of the workflow as the topological information. Furthermore, it is not feasible to design a set of performance metrics which will be useful for all workflows to the detail that the workflow might care about.

In order to solve this situation, we separate our resource description into two parts. The first part, which we call radle, contains a topological description of the resources and the workflow-independent information for those resources. We next plan to provide a performance metric map `mapi`, which take maps subset of resources from the resource description to custom user-specified information . For example, a developer can attach the transfer speeds of 1 KiB packets between two network interfaces as the user level performance metrics to the subset consisting of the two nodes.

5 Radle : A Topological Resource Description

Optimizations in a Maestro-enable workflow can require the workflow execution framework to manipulate the resources for multiple different purposes. Consider the following example.

Assume a system architecture with two nodes X and Y and the following features:

1. Both nodes have a local disk
2. Both nodes are directly connected via a network link
3. Both nodes are connected to a shared storage

Task A is scheduled on node X and produces a CDO, which is consumed by a task B scheduled on node Y. There are the following options to move the CDO from node X to Y and each of the method has a different cost associated with it.

1. Task A communicates the CDO directly via the network to Task B.
2. Task A writes the CDO into the local store of node X and task B reads the CDO from the local store of node X.
3. Task A writes the CDO into the local store of node Y and task B reads the CDO from the local store of node Y.

4. Task A writes the CDO to the shared storage and Task B reads the CDO from the shared storage.

In terms of number of copies, directly transferring over the network is the cheapest method of all, because there is no intermediate copy. However, there can be a situation where the task A wants to withdraw the CDO but task B is not yet ready to consume the CDO. In such cases, you can either block task A from exiting, thereby increasing the consumption of compute time , or you can store the CDO in a temporary storage in one of the other three methods. For each of the other three methods there will be different costs which will be important. For example, for the third method in the list, if the write into local store of Y can be hidden away behind some other computation, then the performance of the third method can rival the first method in terms of compute time, even though there is an extra storage cost. Furthermore, it doesn't suffer from the limitations of the first method. However, if local storage is an expensive quantity in the workflow, then may be the third method also is not very attractive.

When solving such optimization problems, it is important that all resources provide a minimum level of information about themselves based on which optimization algorithms could be designed. However, it is also important that the approach does not prevent the resource from advertising its complete functionality, if an optimization algorithm wishes to take advantage of it in the future.

To that end, we have a two level classification of the resources, with the first level being called categories and the second level being called drivers. The category provides the basic information on which the general optimization algorithms can be design and the drivers provide all the rest of the information which is specific to the resource in question. In our analysis, we have identified five major category of functionality that a resource is supposed to provide to a workflow, namely compute, memory, storage, network, and generic.

One issue here is that, even though technically there are two different resources in the GPU, both the resources have to be allocated together. The resource description language solves this issue by specifying a new category of resources called "Meta" resources. These resources do not have any resources of their own but own other resources in the topology tree. For example, a node can be a meta resource which owns the main memory, the compute power and all the other devices in the node. Similarly, we can have meta resources like a NUMA domain or a package. This brings the total category of resources to six.

1. Compute
2. Memory
3. Storage
4. Network
5. Meta

6. Generic

While the categories are bounded in the description, the drivers, by the nature of their purpose, are unbounded. For example, an example of a driver in the storage category would be SSD, while another would be the HDD. For compute category, you could have CPUs and GPU compute units.

Definition 5 (Resource Description)**.** *A resource description D is a tree of resources where a resource R on a system is an entity identified by a dictionary as shown below.*

```
resource_name:
  category: category_name
  driver: driver_name
  is_indivisible: true | false
  exclusivity:
    is_exclusive: true | false
    shared_parameters: [array_of_shared_parameters]
  category_parameters:
    ...
  driver_parameters:
    ...
  availability:
    type: [fixed-time-after-request,fixed-time,non-contiguous,alltime]
    info:
      ...
```

The resource has a name, and then there are mandatory fields which specify the category and the driver. The parameters which are required by the category are specified in `category_parameters` while all the other resource specific parameters are specified in the `driver_parameters`. In case the resource is a meta category resource, the `category_parameters` would contain the name of the children resources. The `availability` specifies the time interval during which the resource is available. The `is_indivisible` field specifies if the child resources (if any) can be assigned separately to different tasks or do they have to be assigned to the same task. All resources with no children have their `is_indivisible` parameter as true. The `is_exclusive` parameter indicates if the resource can be shared. If the resource can be shared, then `shared_parameters` is the array of parameters which are exclusive and therefore determine the upper bounds on the sharing limits.

Having described resource description, we should also describe a language for the task to ask for resources. We call that resource requirement. A task can request multiple resource requirements.

Definition 6 (Resource Requirement)**.** *A resource requirement an entity identified by a dictionary as shown below.*

```
requirement:
  category: <category_name>
  driver: driver_name
  is_indivisible: true | false
  exclusivity:
    is_exclusive: true | false
    shared_parameters: [array_of_shared_parameters]
  category_parameters:
    ...
  driver_parameters:
    ...
  availability:
    duration:
      ...
```

As you can see the requirements have a structure very similar to the description with the main difference being that all the fields in the requirement apart from the category field are optional.

Having formulated a language for resource description and requirement, we can now use the workflow execution framework to allocate resources to the workflows in an optimized manner.

6 Workflow Execution Framework

As noted above in Section 3, the workflows in this section, unless specified otherwise, are now application workflows which are those as seen by workflow managers. In these workflows, mechanisms for triggering applications and monitoring dependencies between them become more important.

We can see from the workflow description in D4.2 [1] and Section 3 that the application workflows as available from the workflow managers are not generally data aware. Triggers for process/job execution are taken not from data availability, but from process/job termination (under the assumption that if Job-B requires data produced by Job-A, then when Job-A terminates, then the data must be available and execution may proceed). This makes it difficult for workflow managers to take advantage of the extra knowledge that data awareness brings and optimize the application workflow accordingly. In order to attempt to address this deficiency, we have introduced a workflow execution framework in D4.2 which can complement Maestro core middleware, making application workflow more efficient in the existing workflow managers by exposing data dependencies as well as process dependencies to the workflow management.

Since then, we have introduced resource awareness into our semantic workflow model. The resource awareness allows us to optimize the workflow even more by moving some of the jobs away time-sensitive parts of the workflow, thereby reducing the bottlenecks. As described in Section 3, resource awareness requires changes to both the semantic and application workflow execution, these changes must be introduced via workflow translations/transformations to adapt the execution graph to fit the hardware upon which it will run.

6.1 An Algorithmic Overview of Workflow Execution Framework

There are multiple different components which come together to form an optimized Maestro-enabled application workflow. In this subsection, we layout an algorithm to describe the complete process of the converting a workflow to a Maestro-enabled optimized workflow:

1. Insert resource requirement annotations in the semantic workflow based on the performance profiles of the workflow. The resultant workflow is called a resource-aware semantic workflow. We remark that this part of the translation process requires support for decision making process from outside of the workflow execution process.
2. Use the steps in workflow translator to generate a Maestro aware semantic workflow from the resource-aware semantic workflow.
3. If required, port the application code to use Maestro for input and output data objects.
4. Generate a resource-dependency graph from the semantic workflow as described in Section 3. Use radle to generate resource description of the computing site. Combine the two descriptions to formulate the resource optimization problem as described in Section 4.
5. Solve the resource optimization problem with the resource dependency graph. Use the results of the resource optimization problems to generate a resource-annotated semantic workflow.
6. Translate the resource-annotated semantic workflow into the desired application workflow for the specified workflow manager.

6.2 Workflow Translator

In the initial design of the workflow translator, the central idea was to take an existing pegasus workflow (which is an application workflow) and add annotations to it. After adding those annotations, the workflow would then be passed through the workflow translator to generate additional CDO related pool-monitor/watcher/cache processes that would provide information to the system that enabled Maestro optimizations. The steps involved using a workflow generator such as Pegasus would typically be

1. Create a catalog of pre-defined input/output files representing data that will be touched by the workflow during execution (known as the *Replica Catalog*)

2. Create a catalog of tasks that represent processes/transformations that will operate on data/files during workflow execution (known as the *Transformation Catalog*)

3. Create a catalog of hardware resources that the workflow is targeted at (known as the *Site Catalog*), a different site catalog exists for each cluster/HPC machine that might be used

4. Create the workflow using Pegasus API by creating a graph of jobs (nodes) and edge dependencies (files). The workflow itself is now represented by a DAG

5. Save the workflow

6. Annotate the saved workflow to add Maestro specific hints/support/features

7. Transform the workflow according to the annotations to add Maestro specific processes/jobs and alter the execution flow, replacing files with CDOs where applicable and adding monitor/watcher tasks

8. Load the modified workflow back into Pegasus and execute it on the selected hardware, the Pegasus execution step now uses the Site Catalog to transform the workflow into the final execution tree that is then passed to the underlying HTCondor framework that is responsible for actually running the jobs.

Steps 1-5,8 all use the Pegasus python API to define the catalogs and create the graph. The annotation of the graph can be done by hand, or, more realistically, it will be performed programmatically and integrated as part of the transformation step, so that steps 6,7 become a single operation.

However, the Pegasus API supports adding arbitrary annotations to nodes/edges in step 4 and it is in fact technically simpler and easier to annotate the DAG during creation than afterwards (when it has been saved to a flat file). The reason for this is that the workflow DAG is created programmatically as a series of loops and connections where iteration numbers, dataset names (and by implications their CDO equivalents) are known in advance but once the DAG is saved out as a flattened file, the relationships must be rebuilt by reconstructing the DAG and there may be decision making ambiguities in which nodes to transform that were not present when the DAG was initially constructed.

In fact it turns out that nearly all workflow execution frameworks follow a similar pattern with a similar API (essentially providing the equivalents of `add_job`, `add_dependency`) as well as supporting metadata annotations during construction of the DAG.

6.2.1 Generating annotated Pegasus workflow

Given the above observations, we therefore decided to experiment with replacing part of steps 5,6,7 with annotations directly into the DAG creation during step 4. In essence, we modify the workflow generation by allowing the maestro workflow object to intercept initialization and job creation calls by overriding those functions and calling the inherited ones with modified data (outlined as follows)

```
class Maestro_Workflow(Workflow):
  # create pool manager at start of workflow
  def __init__(self, maestro_flags, name, infer_depends):
    # setup pool manager etc

  # override job create and insert CDO related objects
  def add_jobs(self, *jobs):
    # perform substitution of filenames for CDOs etc

# Original unmodified workflow creation step
wfp = Workflow(name="demo-orig.yml")

# Maestro enabled workflow creation
wfm = Maestro_Workflow(maestro_flags, name="demo-maestro.yml")
```

This may at first glance appear to introduce a hard dependency on the workflow manager chosen for the project (in this case Pegasus), but in fact, most of the workflow managers provide a similar python API to create jobs and it is in many ways simpler to modify the job creation step by adjusting the DAG creation code directly in python, than it is to create a translator for every workflow DAG format generated by all the different workflow managers.

There are other advantages to applying transformations 'in place' at DAG design time because nodes may be annotated directly with CDO related tags which are easy to understand within the context of the code that generates the DAG, whereas if the DAG were loaded from disk and parsed or annotated by hand, much of the context around a node is lost and it becomes a much harder job to identify which nodes need Maestro changes and which do not

```
# Disable CDO generation - output not Maestro enabled
job1 = Job("Example").add_metadata(cdo_disabled='true')
# Mark this job as generating the final CDO
job2 = Job("Last").add_metadata(final_cdo='true')
```

These arbitrary metadata tags can be read directly in the `add_jobs` call and used to apply transformations directly to the node as it is inserted. In addition, the metadata tags persist throughout the lifetime of steps 4-8 outlined above and so may also contain resource-aware annotations that affect transformations necessary to handle allocation according to the Site Catalog or our own Maestro specific translations/transformations derived from radle.

6.2.2 CDO related annotations

The generation of CDO enhanced workflows using the strategy outlined proves to be straightforward as these may be applied using simple node substitution. By way of example, consider a simple case where we create data, pass it through 2 stages/iterations of some processing with a left and right branch, then combine the two branches and post-process the data, Figure 6.

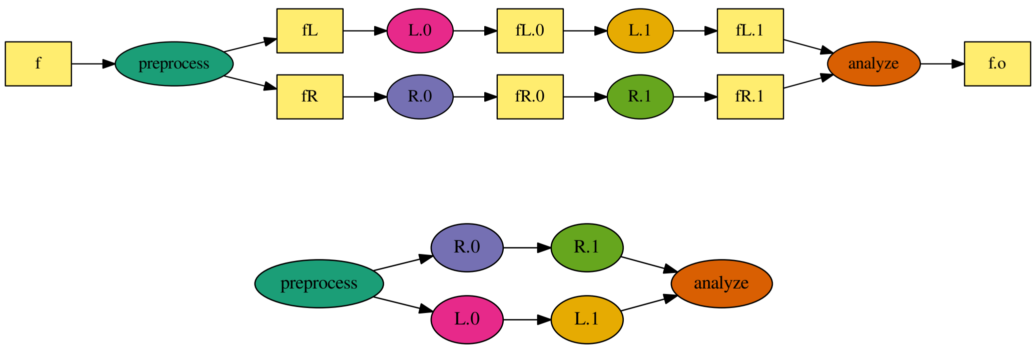

Figure 6: Standard DAG representing a trivial workflow where work is split into two paths and then recombined. Top: Dependencies are inferred via file input/output; Bottom: Execution is defined by process lifetime

As has been noted in previous documents, the execution flow of Left and Right paths are synchronized by the mutual dependency on the preprocess step. When the processes generating files have been modified to instead generate CDOs then, by intercepting calls to `add_jobs` we can insert CDO nodes replace files and change the dependency chain so that the Left and Right paths instead depend on a CDO *watcher* that can be terminated when data is generated, allowing the two paths to execute independently. Unfortunately, a trivial substitution of this kind does not solve the larger problem, as watcher processes are now tied to the previous generation steps and so unwanted synchronization still exists (the watcher will only start when the process generating the data terminates)

We can solve the problem of watcher objects depending on their previous process for execution by breaking the dependency chain between them and the process creating the CDO that they consume/watch. We make watchers instead depend on the parent process of the one producing the data they consume. This triggers the creation of watchers at the same time as the process that is creating the data they watch for. Figure 8 shows how the DAG appears with the modified dependencies. The tasks/processes that produce data files now produce CDO objects that have no onward dependency chain, but watcher objects are created that will consume the CDOs and produce a dummy output that drives the next part of the execution flow. Note that the dummy outputs are present only to satisfy the workflow API and do not need

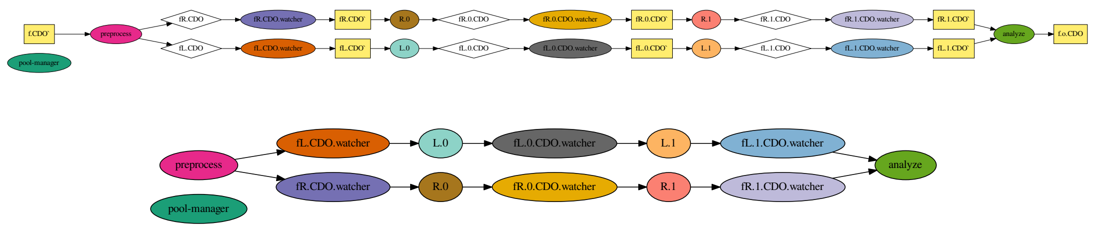

Figure 7: Maestro enabled DAG. Top: Files are replaced by CDOs and extra watcher objects become triggers; Bottom: Watcher objects are still tied to previous processes as they are only executed when then previous step completes

to be created in practice, when the watcher terminates, the dependent processes will be launched.

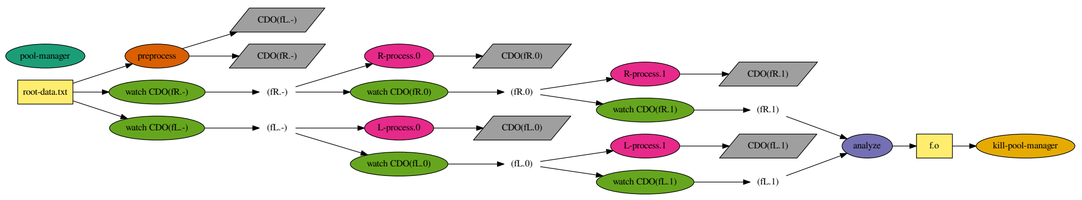

Figure 8: Maestro enabled data-driven DAG with modified watcher dependencies. CDO objects (gray boxes) do not have any direct downstream dependents, watchers are spawned to wait until the CDOs are present and then trigger the next execution. (Dummy watcher outputs (unboxed) are present only to satisfy the Pegasus dependency API).

When optimizing the execution flow for resource usage based on affinity or task placement, we will ideally want to place watcher objects (or their data caching equivalents) in the same location as the process(es) that depend upon them. The existence of a large number of watcher objects (though they are short lived) will cause unwanted pressure on compute resources since the watcher objects must be allocated prior to data generation, for this reason it is essential that watcher objects are as lightweight as possible and ideally will be run on otherwise unused hyperthreads using oversubscription of compute nodes to reduce the need for additional compute resources.

For reference, we note that once we have enabled CDO generation in Pegasus, the montage workflow can be transformed automatically from Figure 9 to that shown in Figure 10. The graph demonstrates that our ‘in place’ translation of workflows works as well as a stand-alone translator would do and results in workflows that produce identical result.

Our introduction of CDO objects introduces lifetime related problems. A watcher process, may be started at prior to the generation of data, but because the watcher terminates when the CDO becomes available – which will

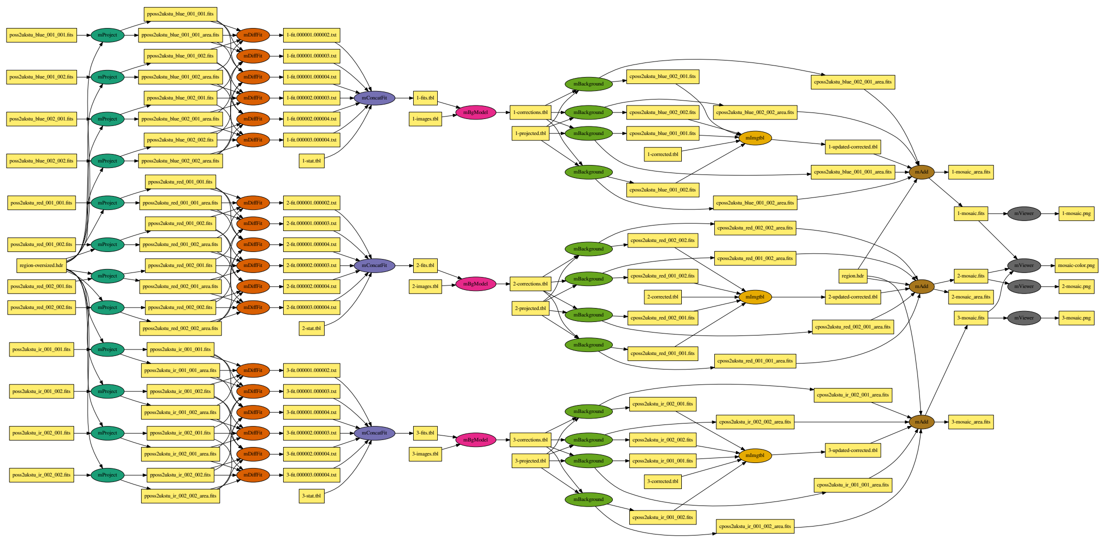

Figure 9: Unmodified workflow DAG for small montage-v3 (0.5°) graph

frequently be when the process generating the CDO is itself about to terminate – the watcher cannot be used as a storage holder for the CDO. The watcher serves only as a trigger indicating that a CDO has been generated. If the process generating the CDO terminates, then an additional CDO cache is required to keep the CDO alive. In Figure 11 we show an example of a DAG with additional cache nodes added. The cache can be created by annotating an input/output using `input.add_metadata(cdo_cache='true')` and the process is automatically added to the DAG (though by default, we can assume that any process consuming a CDO object via a watcher will require a cache). Note that in Figure 11 the links between watcher nodes and cache nodes do not indicate execution dependencies and are only included to show the linked relationship between the two. It is important to note that the need for watcher and cache objects is due to the restricted scheduling capabilities of the workflow execution system that launches jobs when others terminate, the watcher object only needs to wait for a signal from the pool manager to know when a CDO has been generated, then signal the CDO cache and terminate. The CDO cache object needs only to copy the data (keep it alive) as the CDO producer terminates and the next process (a CDO consumer) is started (either immediately, or when the job scheduling engine decides). Once the cached data is consumed, it too can shut down. In practice, a single CDO cache object may hold multiple CDOs from producers on the same node and supply them to consumers on the same node and thus have a lifetime that goes beyond a single task and the number of cache objects can be correspondingly reduced. Testing will reveal whether the cost of of watcher/cache objects mitigates the cost using the filesystem for data transfers. The job scheduler itself (Slurm for example), offers some features that may allow the cost of monitoring CDOs via watchers to be reduced using gang-scheduling to over subscribe nodes

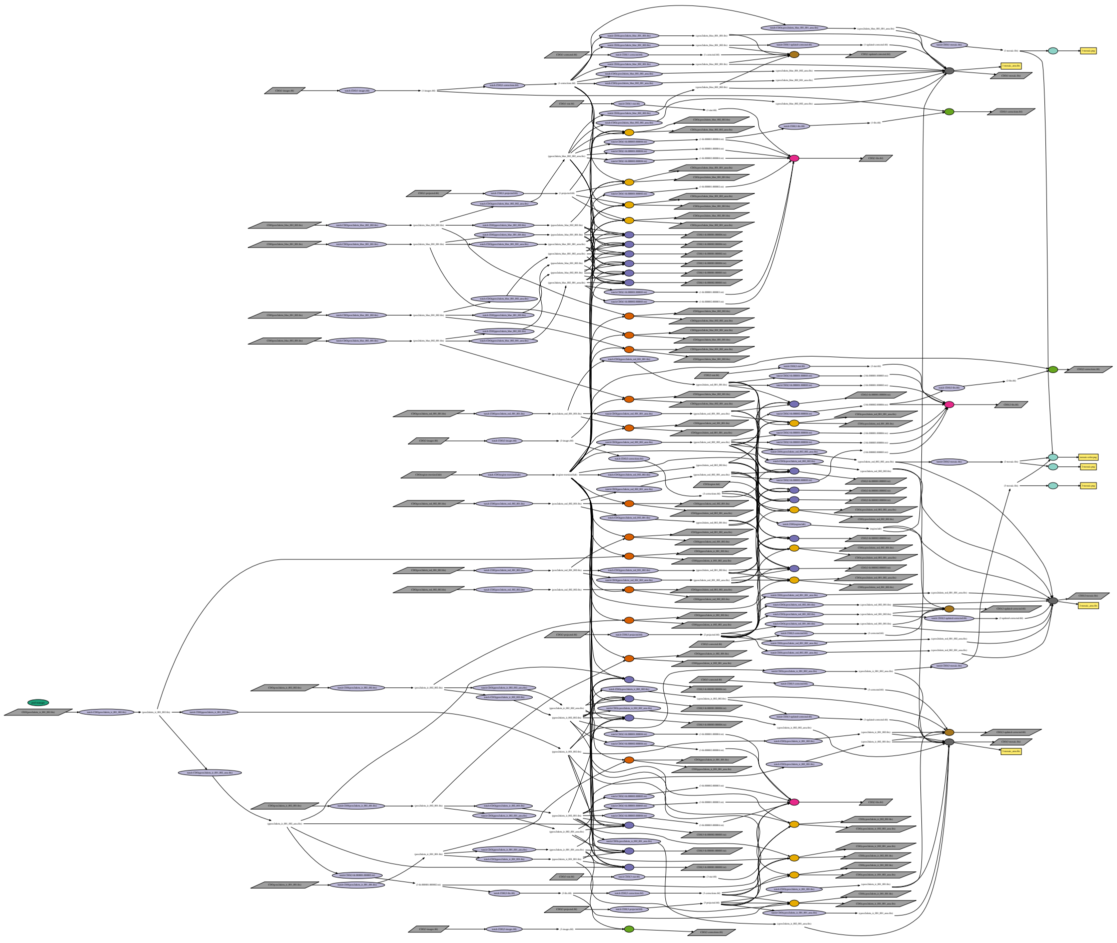

Figure 10: DAG with CDO watcher nodes for small montage-v3 (0.5°) graph. Note that in this graph, all data inputs have been converted to CDO objects, in the final implementation, some input files will remain as disk based files.

and place multiple of them on a single core. Ultimately, the developments made for Maestro compatibility should be fed back to the workflow management projects to improve the triggering mechanisms used so that data awareness rather than job dependencies alone may be used for workflow execution.

6.2.3 Resource based annotations

The Pegasus workflow engine supports *clustering* of jobs in order to improve placement of them on hardware resources. Clustering may be controlled by the depth of a job in the DAG, or may be specified by labeling/annotating tasks that are to be clustered together. The granularity of clustering may also be controlled by specifying the maximum number of jobs to combine as well as the estimated runtime of jobs – it may not be beneficial to cluster short running jobs with longer ones. The clustering capabilities of Pegasus are unfortunately limited by the underlying HTCondor framework that actually executes

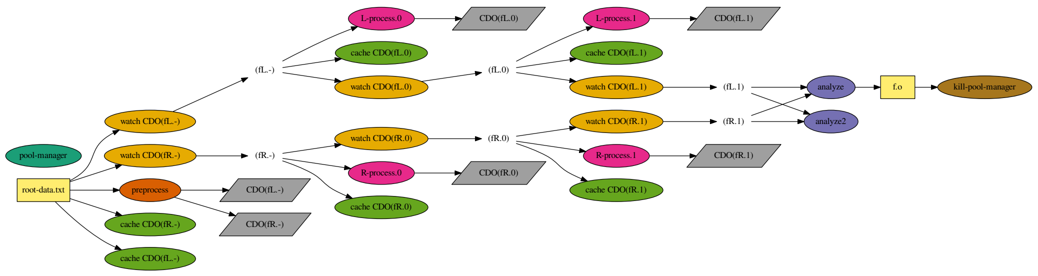

Figure 11: DAG from Figure 8 with extra cache objects to keep data alive

the DAG (a package known as DAGMan), however when required, it should be possible to pass additional Slurm specific annotations through from the workflow description to the execution engine and control placement more finely without direct adjustments to the DAG. Currently the granularity of how tasks can be assigned to individual NUMA nodes or sockets has not been tested – this remains a future task. Within the project, radle will be used to generate descriptions of hardware resources. Radle represents a lower level (or more detailed) view of the resources than the site catalog that is used internally by the workflow management tool. The challenge in the next part of the project will be to map the workflow requirements implied by radle onto the Site Catalog that is used by the execution framework and at the same time perform transformations on the DAG in such a way that

- The job placement/affinity/clustering requirements can be mapped to those supported by the site catalog
- The subsequent transformation made by the workflow environment (step 8) do not undo or clobber any transformations made by our translator to support resource management

Note that the transformations made to the DAG for Figures 7, **??** and 10 were made by simple substitution of nodes and deletion of dependencies (files/CDOs) alone, this can be done because these trivial changes to the DAG do not require knowledge of nodes at distant parts of the graph, only a modification of immmediate parent/child relationships and replacements of files with CDOs. For more sophisticated resource placement, clustering, affinity/NUMA awareness it will be necessary to traverse the graph after construction in line with the original concept of the workflow translator.

6.2.4 Dynamic provisioning annotations

Dynamic provisioning provides the capability to create on-demand data managers that can be used by the workflows or Maestro core. Startup and shutdown of the dynamic resources can then be inserted as an additional dependency tasks within the graph. The pool manager itself may be annotated using metadata such as `add_metadata(maestro_workflow_core_backend="minio")` which can be used to tell the pool-manager what sort of data manager to initialize and use. Other annotations may be inserted as need arises to instantiate new data managers.

If the services managing the resource depend upon optimizations that are made based on the resource based annotations, then they will need to be generated in concert with the other graph transformations taking place. For example, if the transformations/optimizations squeeze jobs into a smaller node count than initially expected, then any temporary filesystem brought up needs only to cover those nodes actually required. Exploration of these possibilities is a goal of the next phase of the project.

6.2.5 Extending workflow manager capabilities

We have seen in the previous discussion, that the workflow manager itself is limited in its capabilities and does not support data-driven execution. The solutions proposed here involve making CDO watcher objects root node objects with a much longer lifetime than is strictly necessary. Whilst it will be possible to reduce this problem by traversing the graph and starting watcher objects later on such that they become dependent on jobs that execute just before the one that needs them. A more robust solution may be possible by treating each sub-DAG as a separate workflow in its own right. Pegasus supports the handling of multiple DAGs for the treatment of ensemble runs of workflows when data may arrive from a new run and trigger the execution of a new workflow. The drawbacks to this approach will be that each workflow would need knowledge of previous/other workflow jobs in order to connect to them and at the level of the CDO where each file is replaced by a CDO object, in fact every node of the graph becomes its own workflow. An alternative strategy might be to maintain the DAG state 'ourselves' and submit each node/task by effectively implementing a trigger based on CDO availability directly.

The Pegasus workflow manager is not able to handle dynamic graphs or execution flows go well beyond what it is capable of representing. In fact Pegasus can only represent 1 out of the 16 possibilities and all node connections are of the form (S, S, E, A) - meaning that both producer and consumer nodes are static, they form an execution dependency and the data is awaiting consumption by being written out to disk. The annotations we are adding and transformations we are making to the DAG should allow us to represent other relationships (or connection types) inside the workflow. Future work will explore the possibilities of enhanced DAG execution, as well as other DAG types for demonstration.

7 Planned Optimizations

With the framework for optimization of workflows now in place, we now list optimizations which we are planning to implement using this framework, in order to provide better performance of the workflows. We use Montage as the experimental test case for the optimizations.

7.1 Improved Concurrency

In a workflow manager, there are different triggers for starting a task in an application workflow. However, if we want the applications to be workflow manager agnostic, then the choice generally reduces to two options, namely finishing of a task in an application workflow and generation of file system artifacts. In some workflow managers, like Pegasus for example, a trigger of a task is only initiated when all the prerequisite tasks end.

This can introduce pessimisation in a application workflow. Consider for example, a task which produces a group of files over a long time and a host of dependent tasks, one for each of the files. In a normal application workflow, the dependent tasks cannot start until the first task has ended even though the data is already available for the tasks to start. This serializes the workflow even though the data dependency allows for more concurrency.

In regards to this, the Workflow Execution Framework can improve concurrency by taking advantage of Maestro CDOs. The workflow execution framework goes over the dependencies between tasks and puts a watcher for every CDO that is generated. The watcher looks for the CDO and shuts down as soon as the CDO is generated. The workflow execution framework also shifts the dependency of dependent task from the required task to the watcher. This allows the dependent task to start as soon as the CDO is available, instead of waiting for the task to finish.

7.1.1 Demonstration in Montage

Provenance records is an important part of a lot of scientific workflows like Montage. One of the ways to facilitate provenance is to store all data products and records from all steps on analysis. In a classical Montage workflow, this is automatically achieved, since the output is stored in terms of a file which is then used by the next stage.

In Maestro-enabled Montage, this step is not automatic. CDOs by default are destroyed at the end of the application workflow. Therefore, it is important for the process to not only make the CDO available to the pool manager, but to also write the resulting output file to a persistent storage. As noted above, in a traditional workflow execution environment, this would mean that the next task won't start until the previous task has produced the CDO and written the file to a persistent storage. In order to optimize this part of the application workflow, we can use a watcher process to track the creation of CDOs and then start the dependent task immediately. In the mean time, the original task continues on with its writing of the file to the persistent storage.

7.2 Dynamic Provisioning

Experiments with Montage have shown that a file system with uniformly distributed caching can improve the performance of the Montage workflows. One way to get this optimization would be to dynamically provision a file system for the workflow. In the workflow execution, we can do this by inserting a task before the task that writes a file and make that task dynamically provision the required file system. In a more general setup, this feature can be used to make available feature-specific file systems on demand to application workflows.

7.2.1 Demonstration for Montage

At the begining of the workflow, a task is added to instantiate a cache-based filesystem. This filesystem instantiation is achieved by the dynamic provisionning component. We use the file system BeeGFS that provides the corresponding caching feature. The workflow translator will go over the workflow and check for all tasks that create an output files. Then, it will augment the task by starting the dynamicaly-provisioned filesystem service to configure and enable access of the dynamically-provisionned file system from the node running the task. The file is then written in this newly-provsioned BeeGFS file system. Applied independently of other optimizations, this optimization should be able to demonstrate the performance benefits of using a distributed cache-based file system over a traditional file system.

7.3 Pre-Fetch CDOs to Nodes

Given two tasks connected by the data dependency, due to constraints of the workflow management it is not always possible to ensure that the two tasks will run on the same node. In such scenario, if the workflow is aware of the nodes on which the dependent tasks are going to run, then the application workflow can pre-fetch the CDO from the source node to the destination node before the dependent task has started. This will ensure that when the dependent task starts, the data required would be present on the same node as the node of the dependent task. This would then kick in the Smart Memory Transport and reduce the time needed for the task to fetch the data, thereby improving the performance of the CDO.

7.3.1 Demonstration in Montage

In Montage, the workflow execution environment will insert cache worker tasks as a dependency to the task which requires the CDO. These cache workers can fetch the CDO to the node on which the task is going to run. This is all done while the app is still waiting to start and makes sure that all the CDOs that the tasks require are present on the same node as the task when the task starts.

7.4 Workflow Scheduling

The resource description of a system can help in making informed decisions on where to execute certain tasks which can then lead to better performance of the workflow. For example, by knowing the dependencies of data between tasks, tasks having a dependency of data between them can be assigned to the same node or at least the same subnetwork. This can reduce the communication cost of transferring data between tasks and reduce pressure on the network bandwidth.

7.4.1 Demonstration for Montage

In the Montage workflow, the tree has pretty large disjoint subgraphs. The workflow execution framework will attempt to put all the tasks in a single subgraphs on a dedicated node. This way, all the CDOs generated in that subgraph will be available on the same node for the dependent asks, leading to a more performant workflow.

8 Concluding Remarks

This report documented the initial specifications to optimised workflow management in Maestro. By extending, the originally proposed (D4.2 [1]), workflow execution framework using resource awareness, it is possible to improve on the workflow management and execution. In turn this also provides the opportunity for further optimizations including Improved Concurrency, Dynamic Provisioning, Pre-Fetch CDOs to Nodes and Workflow Scheduling. The initial specifications for all these extensions are documented here. Additionally we have introduced a new use-case called Montage, which will be used to showcase the planned optimisations. Future work as part of WP4 and WP6, will implement these changes as well as test and quantify their impact.

References

[1] F. Tessier, J. Badwaik, and S. El Sayed, “D4.2 execution framework prototype,” 2020.

[2] C. Haine, U. Haus, and A. Tate, “D3.1 Initial Core Middleware Specification, API Document,” 2019.

[3] C. Haine and U. Haus, “D3.3 full core middleware release,” 2020.

[4] S. El Sayed, J. Badwaik, U. Haus, C. Haine, J. Novo, M. Arenaz, G. Umanesan, S. Narasimhamurthy, and S. Wu, “D3.4 transformation and map optimisation release,” 2021.

[5] P. Groth, E. Deelman, G. Juve, G. Mehta, and B. Berriman, "Pipeline-centric provenance model," in *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pp. 1–8, 2009.

[6] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," in *2009 5th IEEE International Conference on E-Science Workshops*, pp. 59–66, 2009.

[7] "Montage performance profile."

[8] A. Uta, A. Sandu, and T. Kielmann, "Poster: Memfs: An in-memory runtime file system with symmetrical data distribution," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 272–273, IEEE, 2014.