D4.5
Workflow Management and Optimisation Final Release

Work package	WP4 High-level Middleware	
Author(s)	Jayesh Badwaik John Biddiscombe Alejandro Dabin	JUELICH ETH Zurich / CSCS ETH Zurich / CSCS
Reviewer #1	Maxime Martinasso	ETH Zurich / CSCS
Reviewer #2	Dirk Pleiter	JUELICH
Dissemination Level	Public	
Nature	Demonstrator	

Date	Author	Comments	Version	Status
19.11.2021	All	Initial Draft	V0.1	Draft
30.11.2021	All	Addressed Comments in Internal Review	V1.0	Final

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

Maestro is a FETHPC-2018 funded project that will design a data- and memory-aware middleware framework for HPC applications and workflows. Work Package 4 (WP4) of the Maestro project develops the high-level tools of the framework. This deliverable details the demonstrators for higher-level toolsets for workflow management and optimisation, and follows D4.3 where the initial specification of workflow management and optimisation was detailed.

Contents

1 Introduction **4**

2 Mocktage : A Mock Montage Workflow **4**

3 Workflow Execution Framework **5**
3.1 Maestro's Role and Usage 6
3.1.1 Event Subscription for CDO Transfer and Execution Flow Control 6
3.1.2 Locality-Based Optimization of CDO Transfers 7

4 Demonstrators **8**
4.1 Dynamic Provisioning 8
4.1.1 Principle 8
4.1.2 Target workflow and setup 10
4.1.3 Maestro's role and usage 11
4.1.4 Success criteria and performance metrics 12
4.1.5 Results 12
4.2 Locality-Aware Job Scheduling Optimiser 13
4.2.1 Principle 13
4.2.2 Target Workflow and Setup 14
4.2.3 Success criteria and performance metrics 15
4.2.4 Test Specifications 15
4.2.5 Results 16
4.3 Large workflow benchmarking with Mocktage 17
4.3.1 Principle 17
4.3.2 Results 17

5 Summary and Concluding Remarks **18**

A Intermediate Workflow Description Language **20**
A.1 Specification 20
A.1.1 Tasks 20
A.1.2 Data 21
A.2 A Simple Example 22

1 Introduction

In deliverable D4.2, we laid out a design of workflow translator, which takes in an already generated workflow and then uses a workflow translator to convert the workflow to a Maestro enabled workflow. In deliverable D4.3, we outlined the initial specification of workflow management and optimization. Specifically, we designed a model and a workflow manager independent language for describing the resource aware workflow. We then designed a workflow execution framework to create an optimized workflow from the original workflow, generate a workflow description for the optimized workflow which is understandable by the targeted workflow manager and run the optimized workflow through a workflow manager.

The workflow execution framework looks at the machine description to create a map of the available resources, and then allocates the resources to the different tasks so that the workflow is optimized. In the process, the workflow execution framework might create additional tasks (such as watcher and caching process) and insert them in the workflow with the appropriate dependencies.

In this document, we lay out the final design of the workflow execution framework and test it on workflows based on Montage use case. In Section2, we describe a mock use case which has been implemented in order to replicate the I/O characteristics of the Montage application. In Section 3, we describe the workflow execution framework. Then in Section 4, we test a demonstrator for each of the optimization use cases as outline in D4.3. And finally, we have the conclusions in Section5. At the end of the document, we describe the Intermediate Workflow Description Language (IWDL) in the appendix.

2 Mocktage : A Mock Montage Workflow

In deliverable D4.3, we have introduced Montage as the real life workflow to evaluate the workflow execution framework's effect. The Montage application internally uses a FITS file format to load and store image data. The FITS file format has some complexities which make it infeasible to create a serialized version of FITS data without redesigning and reimplementing the algorithms of Montage as well. For example, the FITS File format stores a pointer to the `FILE*` descriptor and majority of algorithms directly read data and write data to the descriptor instead of writing data to memory first. This prevents us from keeping all the data in the memory, since the functions often directly write data to the file. In order to get around this, a synthetic benchmark named Mocktage has been implemented, which aims to reproduce the computational and I/O workloads of Montage workflow while avoiding the difficulties of FITS file format.

In order to do so, the Montage workflow itself was studied to determine the I/O and computational characteristics of the various Montage components. Experiments in past have shown that the Montage workflow is dominated by I/O consuming approximately $70 - 90\%$ of compute time in the workflow [1, 2]. Furthermore, the components of Montage are primarily single threaded. Using this information, mock applications resembling various stages of the Montage workflow were written where the mock applications would consume CDOs of sizes equal to the corresponding size of FITS files in Montage, sleep for the corresponding compute time of the application and then finally output a CDO which was equal to the corresponding size of FITS files. The single threaded and I/O

dominated nature of Montage application make such a mockup a good representation of the actual workload [3, 4, 5]. In particular, we look at CDOs of sizes from 1GiB to 4 GiB for benchmarking purposes.

3 Workflow Execution Framework

The workflow execution framework for Maestro-aware workflows is called splinter [6]. It is a collection of tools written in python which allows one to take a resource aware workflow (which have been introduced in [7]) and convert it into an optimized workflow. The workflow execution itself is a four stage pipeline with the following stages:

1. Generate a resource aware workflow in the Intermediate Workflow Description Language (IWDL).

   For this stage, we write a workflow generator which generates a description of the desired workflow in IWDL.

2. Allocating Resources for Tasks in the Workflow and Optimizing Task Placement

   The workflow generator written in IWDL is then passed onto the resource allocator tool of splinter. The resource allocator tool perform the two optimization stages (Dynamic Provisoning and Scheduling) on the workflow, allocates resources to the tasks and adds the appropriate helper tasks such as watchers and stager jobs. It then generates a resource-allocated optimized workflow.

   This is the stage at which the workflow execution engine might introduce any additional tasks or perform any optimizations that might benefit the workflow performance.

3. If required, convert the optimized workflow into the target format for the specified workflow manager.

   Finally, the translator module of splinter takes in the resource-allocated optimized workflow and converts it into a workflow description which is understandable by the workflow executor.

   As of now, we have two different workflow translators, a simple one written in python (splinter-native), and a second one which is adapted from Pegasus (splinter-pegasus). The splinter-native workflow generator directly generates an IWDL file from a custom written python script, while the splinter-pegasus generates a Pegasus workflow description which can then be consumed by Pegasus to execute the workflow. All the optimization passes have been implemented for splinter-native.

   Pegasus, as of now, does not have a way of manually fixing the resources which are to be used for execution of the tasks. It also does not allow for putting constraints on task resources, wherein, one can say that two given tasks have to be executed on the same node. Due to the limitations of Pegasus, the optimization passes have not yet been implemented for Pegasus backend (splinter-pegasus).

4. Run the optimized workflow.

3.1 Maestro's Role and Usage

Maestro-enabled workflows also use optimizations coming from the Maestro middleware itself. However, to combine both the optimizations and to ensure the correctness of the workflow, the execution framework should understand and account for the Maestro middleware optimizations and features. Two of these features are locality based optimization of CDO transfer and event subscription.

3.1.1 Event Subscription for CDO Transfer and Execution Flow Control

In a classical workflow manager like Pegasus, a new task is started when all its dependency tasks have ended and when all the filesystem dependencies have been satisfied. With a Maestro enabled workflow, the filesystem dependency has been converted to a CDO dependency. This requires new methods to keep track of CDOs and to ensure that the workflow manager is notified when a task is ready to start (on account of all CDOs being available), and to ensure that the CDOs are not disposed of before they can be consumed by the expected consumer.

For this, we use the event subscription model which has been implemented in Maestro core. Event subscription model allows one to keep a track of CDOs that are being declared, offered, requested and withdrawn among others. This allows us to perform the following operations to ensure the correctness of the workflow:

1. Starting of Tasks whose Data Dependencies are Satisfied

   By monitoring which CDOs have been offered, a monitor process can keep track of which tasks are ready to be started, on account of all their data dependencies being satisfied. This information can be signalled back to the workflow manager, which then allows us to start the corresponding task. See for example figures 1 and 2 which show a simple workflow consisting of a fork and join operation on data. The original workflow generates two datasets that are fed into the two branches, but both branches must wait until both datasets are ready before executing, in the maestro enabled version, the dependency chain shifts to the watcher objects which can now signal that data for each branch is ready independently.

2. Keeping Producers Alive for CDO availability

   There can be situations in the workflow, when a producer has produced a CDO and is looking to exit, but the consumer is yet to start. If nothing is done in such a scenario, it is possible that the producer will dispose of the CDO and when the consumer becomes alive, there will not be any CDO for the consumer. One method to solve this is to keep at least one copy of the CDO alive by keeping one of the CDO-owning processes alive.

   To prevent this situation, a watcher process corresponding to a produced CDO can keep track of all the tasks that will need that CDO, and requires this CDO. This ensures that when a withdraw is done, the process doing the withdraw is blocked on it. Once all other consumers consume the CDO, the watcher can remove the require request for the CDO, and allow the producer to exit. Subsequently, the watcher process keeps monitoring the CDO events and keeps sending similarly appropriate requests to the pool manager to make sure that the CDO remains alive.

3. Caching CDOs on Appropriate Nodes

   The caching job subscribes to the CDO events and captures the events which make the desired CDO available to the pool. Subsequently, the caching process demands and acquires the CDO, so that it can be made locally available to the consumer. Figure 2 shows how cache objects are inserted into the workflow - they are created at the same time as the process generating the data that they are to consume and make a copy of the data produced, allowing the producer to exit and the workflow to continue. They terminate when all consumers of their data have made use of it.

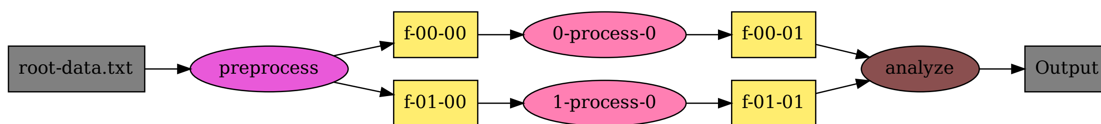

Figure 1: A simple workflow with a fork and join of tasks. The execution of each branch can only proceed when the *preprocess* step has completed and generated data for both branches of the pipeline.

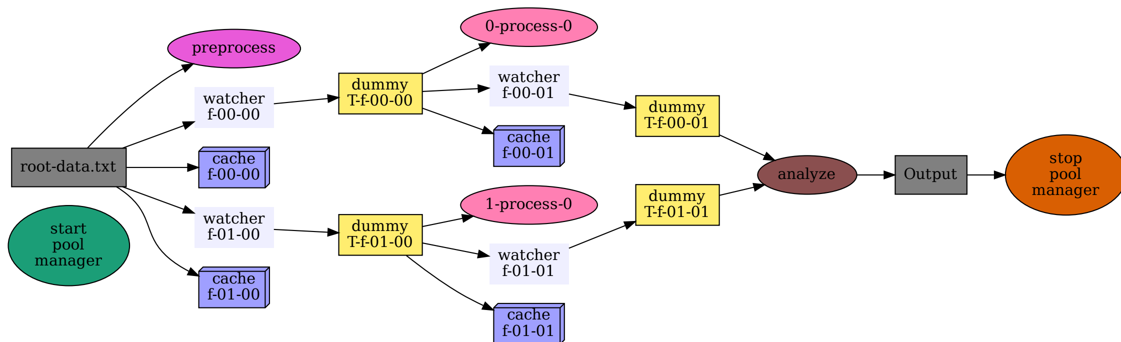

Figure 2: A simple workflow with maestro watcher and cache objects added. When watchers for each CDO signal that data is available for each branch, the execution can proceed independently on each branch. (Note that the 'dummy' file objects do not need to be generated, they exist solely as placeholders to satisfy the dependency conditions of the workflow management software and in fact the processes will output equivalent CDO datasets that are not shown explicitly, but are consumed by the cache/watcher objects).

3.1.2 Locality-Based Optimization of CDO Transfers

Given two applications, a CDO transfer between the two applications can be much faster if the two applications are on the same node rather than them being on different nodes. In Maestro, this can happen in a couple of different ways.

1. For some hardware, the RDMA data transfer over a loopback interface can be much faster than the RDMA data transfer over the network.
2. A shared memory transport can be used to transfer the CDO from producer to consumer. A shared memory transfer is faster than a transfer over a loopback network and the implementation work for a shared memory transport in Maestro is currently in progress.

In that context, scheduling a task consuming a particular CDO on a computational node where the CDO already exists will reduce the time spent in transfer of the CDO. Maestro automatically detects and selects the fastest method of transfer, and therefore, there is no action required on behalf of the application to enable this optimization. This is a case where a workflow can take advantage of Maestro's internal optimization features to optimize its workflow.

We would like to note that we are running multiple jobs on the same node. In general, schedulers avoid scheduling multiple jobs on the same node because of the possible performance degradation due to contention. However, in this particular case, the two different jobs are not really arbitrary, but are actually connected to each other with the cache and the watcher tasks being helper jobs to the primary task.

Specifically, the cache and the watcher jobs are not consuming any compute resources and are more like background jobs in the same spirit as the filesystem workers. Furthermore, the watcher jobs are shut down as soon the task has started, while the cache jobs are shut down as soon as the task has made the appropriate data request to the caching.

All these factors can justify running a corresponding helper process on the same node as the main task process.

4 Demonstrators

4.1 Dynamic Provisioning

4.1.1 Principle

Being able to dynamically provision more efficient and dedicated data managers (e.g BeeGFS or MinIO) will improve performance in workflows with data intensive tasks not suited for shared system-wide storage. For example, a dedicated and exclusive parallel filesystem can perform better for Montage than a global and shared one.

Dynamic Storage Resource Provisioning (DSRP)[1] uses a YAML file to describe storage nodes available on a cluster, including capacity, features and mount points for the NVME/SSD devices. This information can be read by Radle[2], a topological resource description developed for Maestro and described in detail on D4.3[7], as a "storage" category so other components can be aware of them. Then, the workflow manager can start a data manager as one of its tasks by calling DSRP with a list of storage nodes to use. Currently, DSRP uses all the devices described within a storage node and does not support using a subset of them. This could be a limitation if more than then one data manager must run in the same storage node.

[1] https://github.com/eth-cscs/dynamic-resource-provisioning
[2] https://gitlab.jsc.fz-juelich.de/maestro/radle

By using containers, there is no requirement for extra software installation nor admin privileges for deploying data managers (with the exception for BeeGFS noted later) because all process and directories structures reside in its own isolated Linux namespace. Also new data managers could easily be added. Currently DSRP can deploy three data models: filesystem (BeeGFS), object storage (MinIO) and NoSQL database (Apache Cassandra). The deployment is done with Ansible[3], a popular automation and configuration management tool for Unix-like systems, and parameters can be set for each instance, allowing specific tuning. Regarding system requirements, DSRP expects storage devices to have a local filesystem (i.e. Ext4, XFS, BTRFS, etc) and be already mounted at a standard POSIX directory where the user can write to. Inside them, appropriate directories are created and mapped for a data manager instance. Users do not need raw access to the devices which also simplifies management.

On most HPC systems, both compute and near-compute storage nodes are granted exclusive usage while a job is running, implying that by default storage resources are only accessible by the running user. Data isolation works differently for filesystems and the other data models. In the first case, they fall under the POSIX-standard permissions and ACL infrastructure, so the user can set permissions or ACL to restrict or share data. Also for sharing data, the user has to provide the information necessary to mount the filesystem. This mounting step can be done with DSRP or manually. For the other data models, a client or library establishes a network connection to a server for granting access to the data model. In this case, the user would have to provide host, port, and a secret or credentials to allow other users to connect.

All data managers' root directory is a dedicated user directory on the storage nodes, providing an extra separation among users if the data needs to be persistent on these storage nodes.

DSRP allows users to run different data managers at the same time on different storage nodes, and jobs running on compute nodes can consume them together as required. Independent instances of the same data manager can be run simultaneously, allowing specific tasks to benefit from dedicated storage backends or specific data manager configurations. For instance, one workflow could have a dedicated BeeGFS optimized for writing data (e.g. simulation output), another one optimized for reading and metadata access and finally an object storage for providing web access of the data to other workflows/portals of selected simulation outputs. This capability provides a great flexibility for complex workflows that may benefit from using different data managers with varying configurations.

To provide storage customization, DSRP uses configuration templates that are specialized at deployment time, allowing each instance of a data model to run with a particular set of parameters. For filesystem data managers, it provides an option to stage-in data by decompressing a set of archived files (using the command 'tar'). In a similar way, it can stage-out files from the dynamic filesystem by creating archive files on a permanent filesystem. The 'tar' format was chosen for its simplicity and little CPU usage, with the goal of reducing large quantity of files to one single file, and in particular, to avoid copying many small files for which copying or moving often offers poor performance on large shared filesystems.

[3]https://www.ansible.com

The use of containers provides a fast and easy way to deploy the data managers without requiring system wide or user installation, and using containers also simplifies adding new data managers. DSRP uses Sarus[4], which is an OCI-compatible container engine specialized for HPC usage, but other engines could also be used as we are only considering portable container (Docker-like container). Container images can be fetched from public and private repositories, enabling easy use of artifacts provided by third parties as well as specific system-tuned or custom artifacts. Moreover, users don't require extra permissions (with the exception noted next for kernel-space filesystems) and all process are run within the same user id, avoiding privilege-escalation security risks by not impersonating users.

Although containers are used to run the data managers, avoiding extra software installation, for kernel-space filesystems, Linux requires a specific filesystem kernel module to be loaded (kernel configuration and modules are not part of a container). As BeeGFS is not installed by default on most systems, a BeeGFS kernel module is required to be installed on the compute nodes, and optionally (but recommended) on storage nodes to allow direct file copy. The BeeGFS inside the container should be compatible with the kernel module loaded on the host. Moreover, administrators have to setup higher permissions for users to enable them to mount a kernel-space filesystem (like BeeGFS), as users should be able to mount multiple filesystems to different paths.

For the other currently available data managers (MinIO for object storage and Apache Cassandra for databases) there are no extra requirement because both servers and clients run completely on user space, and are accessed via an application or library.

4.1.2 Target workflow and setup

Montage[5] is a powerful and complex toolkit for assembling Flexible Image Transport System (FITS) images. For DSRP testing purposes, the simpler Mocktage (presented on Section 2) application was chosen because it provides a Maestro-enabled workflow used to mock up Montage, and Splinter (introduced on Section 3) is used as a Maestro-enabled workflow manager.

The target workflows consist of a set of graphs with several iterations and forks that produce and consume objects. No additional tasks that require computational resources were added, as the objective is to benchmark I/O intensive operations. These workflows are run with different object sizes over a shared filesystem and a dedicated dynamically provisioned one. Finally, total amount of transferred data is summed and timed to give the overall bandwidth, which allows to compare and measure the potential benefits if this optimization.

To use DSRP, a job is added at the beginning of the workflow graph to deploy a filesystem data manager (BeeGFS), and another job at the end to unmount and stop it. There is no need to modify any other application or component. Figure 3 shows an example of such a workflow where dynamic provisioning tasks have been added to start/stop and mount/unmount the filesystem. The simple example uses 5 forks with each fork having a length of 3 processes (or iterations).

[4]https://sarus.readthedocs.io
[5]http://montage.ipac.caltech.edu/

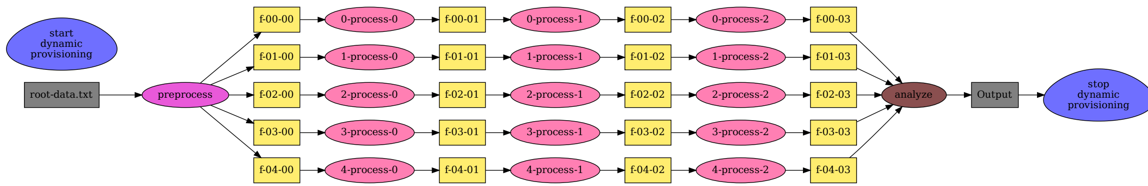

Figure 3: Workflow example used to demonstrate dynamic provisioning using BeeGFS.

4.1.3 Maestro's role and usage

DSRP allows a Maestro-enabled workflow to seamlessly use dedicated data managers on required stages, allowing performance benefits comparing to cluster-wide shared resources. It also simplifies the usage of specific or custom data managers not provided by default on HPC systems.

Execution framework developed for Maestro is being used to insert tasks into an already existing workflow. It decides where it would be required to start a dedicated data manager, such that it can improve performance or provide specialized data model for a section of the workflow task graph. By using these tasks, a dynamic storage is started or stopped, and, for a filesystem, mounted and unmounted on selected nodes.

As DSRP performs storage deployments before other Maestro components are initiated, it does not impose restrictions on Maestro components. After a data manager is made available, the Maestro-enabled workflow can decide on how to use the new storage resource, to improve either data movement or any another data intensive I/O activity. This provides great flexibility to easily customize, incorporate and test data managers within complex workflows by just modifying DSRP related tasks.

On D4.3 [7], it is shown how a Pegasus WMS workflow can be generated and optimized. To incorporate DSRP into a workflow, a job is added to require a dynamic data manager. For example, this code shows how to use the Pegasus Python API to add a job requesting a BeeGFS filesystem with a capacity of at least 10 TB:

```
dynpro = Job("dsrp_deploy.py", node_label="start\ndynamic\nprovisioning")\
              .add_metadata(maestro_workflow_provisioning_backend="beegfs",
                            maestro_workflow_provisioning_backend_size="10T",
                            maestro_dynpro='true')
super().add_jobs(dynpro)
```

In the same way, a job should be added at the end to stop the data manager gracefully. Then, the workflow manager translates it to tasks to execute them with the adequate parameters. For Splinter, the task to start DSRP is created as

```
1     "dynpro1" : {
2       "id" : 1,
3       "command" : "dsrp_deploy.py start beegfs -t CLUSTER_NAME
            -c COMPUTE_NODELIST -s STORAGE_NODELIST -m
          CLIENT_MOUNT_PATH",
```

```
4        "dependency_info" :  {
5          "task" : [1],
6        },
7      }
```

where the variables in capital indicate nodes for the storage client and servers, and mount point on compute nodes. In the current Splinter implementation, this variables are passed as environmental variables when the job scheduler starts the job.

4.1.4 Success criteria and performance metrics

The relevant metric for DSRP is the time taken to run a workflow with or without it. As mentioned, a dedicated dynamically provisioned filesystem data manager should reduce execution time, so the same workflows are tested with DSRP enabled and disabled, and time spent is measured.

4.1.5 Results

All tests were performed on Piz Daint[6] at CSCS. The baseline benchmark uses a shared cluster-wide 8.8 PB Lustre filesystem, therefore results can have small variations related to usage at the moment.

The Pegasus API was used to generate graphs with 2, 4, 6 and 8 forks and with 5, 10, 15 and 20 iterations, resulting in 16 different graphs of the same topology as the smaller one shown in figure 3. As described previously, when testing DSRP a job was added at the beginning to start a dynamically provided BeeGFS filesystem. Then, each graph was run with object sizes of 1, 2 and 4 GB, so the total amount of data transferred by a workflow ranges from 24 GB ((1 read + 1 write) * 2 forks * 5 iterations (1 GB each), + 2 initial generate writes and 2 final consumption reads) to 1344 GB (8 branches, 20 iterations, 4GB size, 8 generate, 8 consume). These are a comprehensive set of 48 cases that show how the filesystem can affect performance. We also include the standard memory transfer for CDOs to compare with the file-based transports. The benchmark results show the aggregated bandwidth achieved for the entire graph execution - or more specifically, total memory transferred divided by total time taken, including all start/stop times of each process in the graph.

The workflows ran on 6 nodes, using a round-robin job allocation. For the tests with BeeGFS, the filesystem data manager was deployed with default options on two storage nodes with three 6 TB NVME devices each, providing a total capacity of 36 TB.

Figure 4 shows the results grouped by the quantity of iterations. The BeeGFS filesystem performs better than the baseline in almost all cases, proving it is useful both on small and big object sizes. It also shows its performance is stable for larger jobs. For objects of 1 GB, BeeGFS seems to perform better than memory-based CDOs. We assume it is related to both BeeGFS local caching (by keeping the object on the node's memory) and asynchronous writing (the actual write operation is delayed if possible). However, an important point to note is that for the smaller graphs, the runtime is to some extent determined by the slurm job launcher that might take a few seconds to launch a task,

[6] https://www.cscs.ch/computers/piz-daint/

which in turn, only needs a few seconds to to run. The larger graphs give a reliable indicator of performance under heavier loads. Further analysis of these results appears in section 4.3.2 where the CDO implementation is discussed.

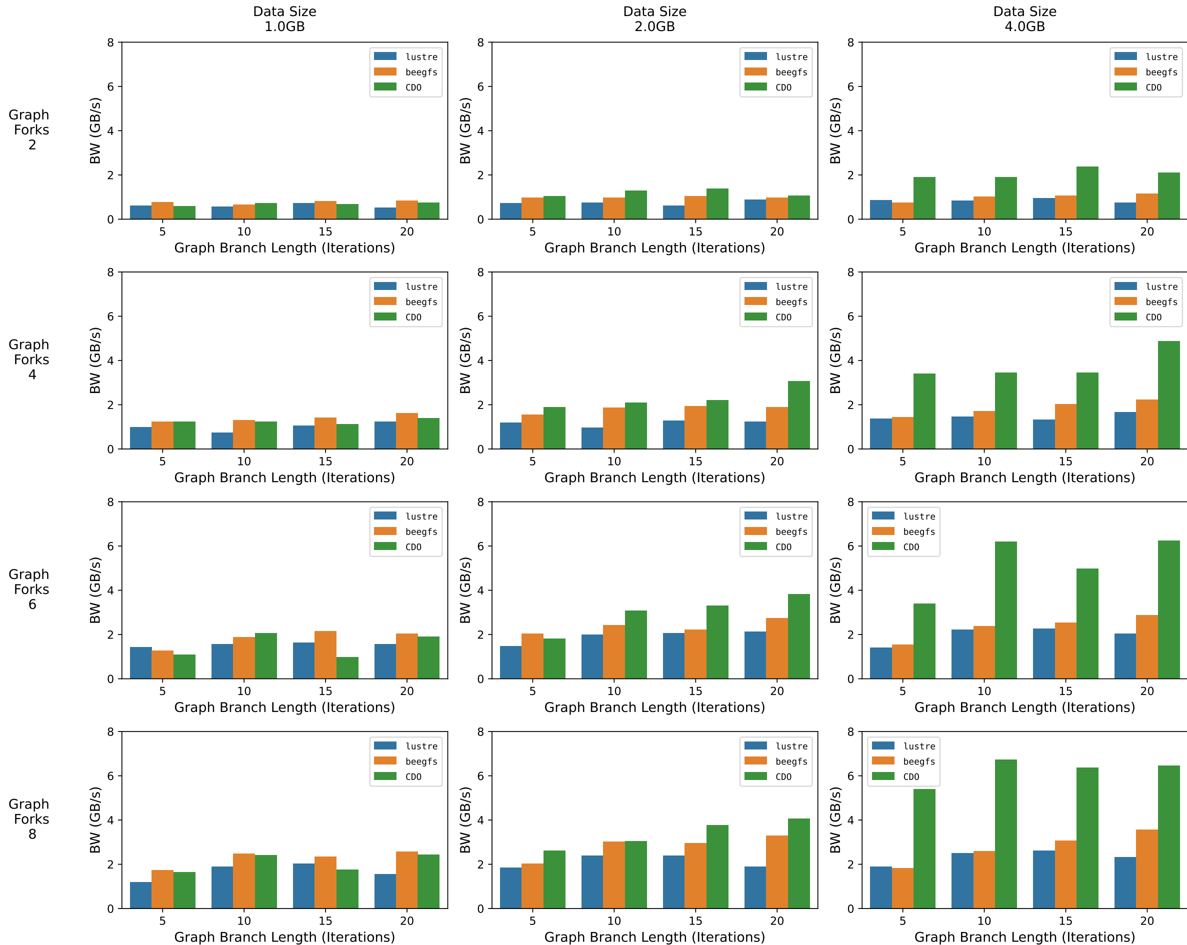

Figure 4: Comparison for workflows running over Lustre, BeeGFS and memory CDOs.

DSRP can improve performance of Maestro-enabled workflows by adding dedicated, dynamically provided data managers. In particular, it was shown how a filesystem data manager reduces execution time on several cases. As this component is easy to integrate, it can be incorporated into existing workflows with minimal modifications.

4.2 Locality-Aware Job Scheduling Optimiser

4.2.1 Principle

By scheduling tasks which have data dependency between them on the same node, we will reduce the amount of data being transferred over the network. Node-local transfers are more performant than global network transfer and are not subject to the contention of sharing network bandwidth. Furthermore, the local transfers reduce the pressure on the network.

The scheduling optimiser runs before the resource allocator runs and marks tasks which should run on the same node due to data dependency between them. The way it does so is by assigning indices to tasks such that all tasks which have been assigned the same index will run on the same node. The resource allocator then takes this index marking into account while assigning resources. An algorithm which works fairly well for a Mocktage like workflow (single threaded tasks with large memory requirement) is given below by the following python code.

In the code below, the `n_node_total` is the total number of nodes available for execution. The `node_threshold` is the number of tasks that can be run on a single node. The dependencies between the tasks are given as a direct acyclic graph. For each task `t` in the direct acyclic graph, we assign a value `level(t)` as follows: If no task depends on a task `t`, then `level(t) = 0`. For any other task `t`, `level(t) = max(level(t_i))` where `t_i` are all the tasks which depend on task `t`.

The `node_sched[t]` variable defines the preferred node to place the task on. The algorithm starts with an arbitrary node and keeps on placing tasks on that node until the node is full, and then moves onto the next tasks. By using the dependency information, the tasks which depend on each other are placed on the same node.

For each task with `level(t) =0`, we execute the following algorithm:

Listing 1: Locality Aware Scheduling Algorithm

```
# Set the next node to 0
next_node = 0

# Set the Number of Total Nodes Available
n_total_node = <total number of nodes>

# Set the number of maximum threads possible on a node
node_capacity = <number of mocktage tasks that can go on a node>

# Set node_sched variable as an array with size equal to number of tasks
node_sched = [-1] * n_total_task

# Recursive Function to Schedule Tasks on Nodes
def scheduler(next_node, remaining_node_capacity, task):
    # Determine the size of the child tree
    ct_size = Size of Child Tree
    # Check if all the tasks in the child tree fit on a node
    if ct_size < remaining_node_capacity:
        # If yes, schedule them on the available node
        for child in child tree:
            node_sched[child] = next_node
            remaining_node_capacity = remaining_node_capacity - 1
    else
        # If they do not fit on the same node, schedule the immediate children and then
            call the scheduler function recursively
        for ic in immediate_child_list:
            node_sched[ic] = next_node
            remaining_node_capacity = remaining_node_capacity-1
            if remaining_node_capacity = 0:
                next_node = next_node+1
                remaining_node_capacity = node_capacity
            scheduler(next_node, remaining_node_capacity, ic)
```

4.2.2 Target Workflow and Setup

The target workflow used for locality aware scheduling is Mocktage which has been described in 2.

4.2.3 Success criteria and performance metrics

There are two relevant metrics when measuring the effect of the locality-aware scheduling optimiser:

1. Total Run Time
2. Wait Time for a CDO

   Time spent by an application waiting for the data transfer of a CDO.

We test these parameters on a workflow twice, one with a locality-aware scheduling optimizer enabled (scheduled workflow) and another with the locality-aware scheduling optimizer disabled (unscheduled workflow), and we measure the parameters for the two situations.

4.2.4 Test Specifications

We use two workflows in the demonstrator, a small one for illustration purposes (called `small_schedule`) and a large one for performance measurement (called `large_schedule`). For the total run time parameter, we check if the run time of the scheduled workflow is less than the runtime of the unscheduled workflow.

- `small_schedule`

  The demonstrator has the graph as shown in figure 5. In the figure, the colors mark the nodes on which the task is executed. As we can see, all the leaf tasks are executed on different nodes. After the workflow is run through the scheduler optimizer, we get the workflow graph as shown in 6

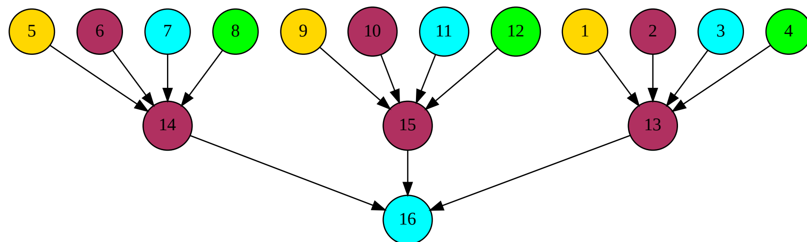

Figure 5: Unoptimized Workflow Graph

- `large_schedule` workflow

  In the large workflow case, we randomly generate a graph and then run the scheduler optimization over it with the maximum of 4 tasks per node. We also run the scheduler with 2 tasks per node. The number of nodes in the graph is 1365.

The test is parametrized on the ratio of intra-node transfers to the total number of transfers. We call this ratio, transfer locality ratio (TLR for short). In the small scheduler test case, unoptimised workflow, the number of intranode transfers is 3 while

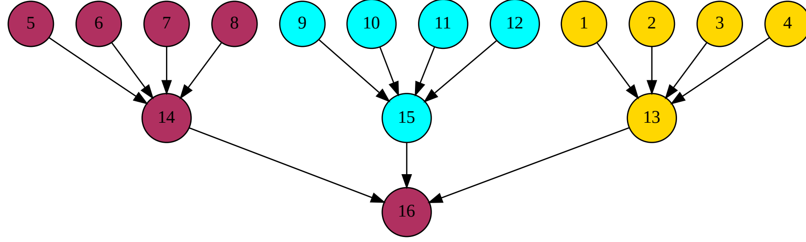

Figure 6: Optimized Workflow Graph

the total number of transfers is 15, therefore TLR 0.2. In the optimised version, we set the maximum number of tasks on a node to 4. We see that the TLR in that case is $13/15 \approx 0.86$. We can also run the optimizer such that the maximum number of tasks on a node is 2. In this case, the TLR is $7/13 \approx 0.46$.

4.2.5 Results

- `small_schedule`

  The demonstrator has the graph as shown in figure 5. In the figure, the colors mark the nodes on which the task is executed. As we can see, all the leaf tasks are executed on different nodes. After the workflow is run through the scheduler optimizer, we get the workflow graph as shown in 6

Maximum Jobs Per Node	1	2	4
TLR	Unoptimized (0.2)	0.46	0.86
Run Times	168s	104s	51s
Average Wait Times	13.6s	6.02s	3.12

Table 2: Performance Comparison for Small Workflow

- `large_schedule` workflow

  In the large workflow case, we randomly generate a graph and then run the scheduler optimization over it with the maximum of 4 tasks per node. We also run the scheduler with 2 tasks per node. The number of nodes in the graph is 1365.

Maximum Jobs Per Node	1	2	4
TLR	Unoptimized (0.06)	0.37	0.75
Average Wait Times	168s	104s	51s
Run Times	5341s	3889s	1187s

Table 3: Performance Comparison for Large Workflow

4.3 Large workflow benchmarking with Mocktage

4.3.1 Principle

As has been previously discussed, we may apply the workflow translator to arbitrary graphs that have been generated (in this case using the Pegasus API) and convert them to their equivalent CDO based versions. The graph of figure 3 used for BeeGFS/Lustre benchmarking for example, can be generated for CDO operation to produce figure 7. This graph differs from the simple example of figure 2 by omitting the cache (or staging) processes. Since we are using our own customized Mocktage processes that simulate Montage IO, we can skip the need for cache objects. In general, when converting a process from File to CDO based IO, a decision must be made on how to handle the lifetime of the generated data; it may be cached until the process(es) consuming it is/are ready to take it, or one can simply keep the original task generating the data alive until all consumers have made copies. For our benchmarking, it is a trivial process to move the output reference counting logic from the cache objects to the data producers themselves and produce the slightly simpler graph shown.

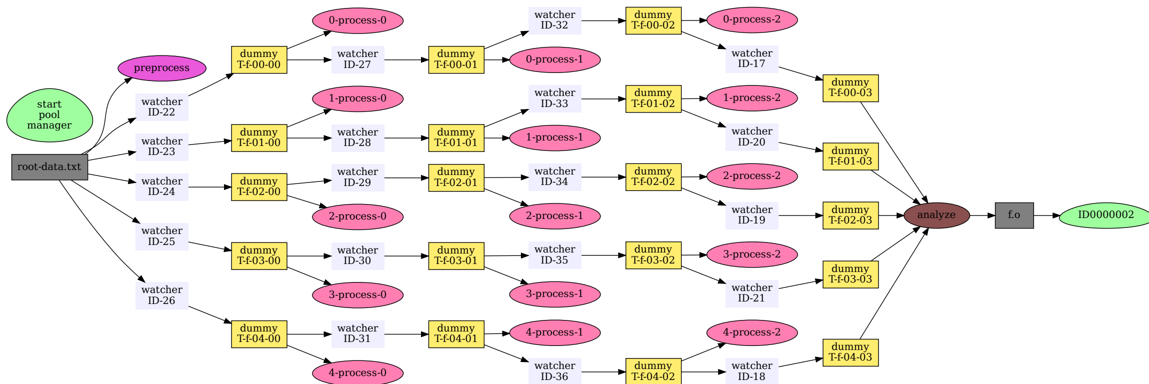

Figure 7: Comparison for workflows running over Lustre, BeeGFS and memory CDOs.

We can now run the same set of 48 graphs with different numbers of forks and branches to make a direct comparison of the same workflow going via disk, via dynamic provisioning or via CDOs.

4.3.2 Results

The results of our CDO enabled Mocktage benchmarking are shown in figure 4 alongside BeeGFS and Lustre. The performance improvement overall when using CDOs is clear. On graphs with smaller data transfer sizes, where the total runtime is somewhat dependent on the slurm start/stop times of each graph node, the throughput is on a par with the filesystem, and slightly behind BeeGFS in some cases. The CDO based graph uses watchers to signal when data is ready and therefore requires an extra task to be run for each data generating process – this extra task comes with a cost and so the smaller graphs do not benefit.

As the number of branches increases - more parallelism of reads/writes becomes possible for all 3 of the scenarios (BeeGFS/Lustre/CDO) and so we expect throughput to

increase accordingly – however the CDO version should finish slightly faster because each branch can begin execution independently as the watcher processes signal to each when data is available. To see if this is the case, we plot (a subset of) the data from figure 4 using the number of forks along the X axis, as in figure 8. Whilst the trend is positive for increasing numbers of branches (more parallelism), the synchronization introduced by the final *analyze* process in the graph reduces the impact and we do not see a clear signal to show the improvement. Other graph topologies could be considered as candidates to better illustrate the CDO gain.

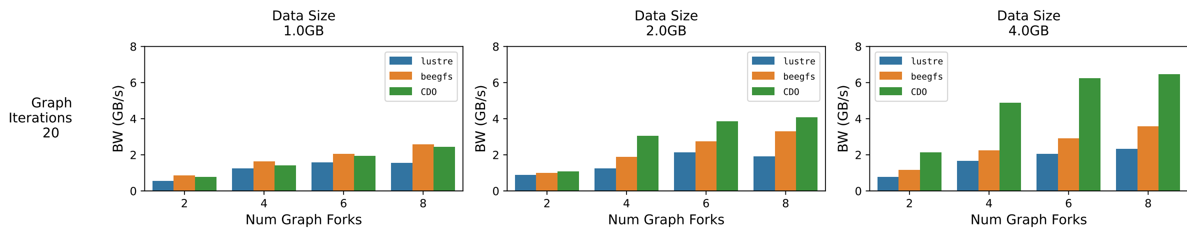

Figure 8: Comparison of workflow results (from fig 4) as a function of the number of graph branches.

The improvements of CDO transfer compared to either filesystem is clear for larger datasets and the CDO results outperform even BeeGFS by almost a factor of 3. Since CDO transfers are happening among a group of nodes sending data to each other, rather than funneling through the filesystem, this is exactly the improvement we expect to see.

5 Summary and Concluding Remarks

This report has documented the workflow execution framework in Maestro. By implementing the design from D4.3, we have shown that it is possible to improve the performance of the workflow by adding resource awareness to the workflows. In particular, we have shown that dynamic provisioning and locality-aware scheduling improve the performance of the workflows significantly. In addition, we have shown that the data awareness of a scheduler itself is also instrumental in providing some optimizations for workflows in the future.

In the course of implementing some optimizations, we discovered aspects of workflow managers and schedulers that can be changed to provide for a better performance of the workflows. In general, the recommendation is to make the workflow managers more resource aware and to make the schedulers more workflow aware. Specifically, we recommend the following directions of investigation:

1. The workflow managers, that we investigated, are task oriented. In that, they would respond to triggers which were activated when tasks would either start up or shut down.

   In addition to this, we feel, the workflow managers would benefit from being data oriented as well. In particular, currently, the workflow execution framework can only be triggered when at task either finishes or ends. We recommend that the

workflow managers should also provide a data based trigger mechanism. And then this information can be used as a trigger to launch new tasks.

2. The schedulers can be more workflow aware. Specifically, they can be aware of the relation between a task and a helper task and allow both the applications to be scheduled on a single node concurrently.

3. The scheduler and the workflow manager can be combined into a single application, where the users can submit workflows directly to a workflow manager daemon. The workflow manager daemon would then be able to compute the optimal scheduling of task and corresponding resource allocation.

A Intermediate Workflow Description Language

In order to optimize the workflows, it is necessary to express the workflows in a manner which captures all the relevant information. In particular, for Maestro based resource aware workflows, it is necessary to express resource requirements of tasks, and the data dependencies between tasks. In order to do so, we have designed an intermediate workflow description language (IWDL) which is implemented in terms of JSON.

In D4.3, we described resource aware workflows, where we modeled a resource aware workflow in terms of tasks to be executed, and data to be produced/consumed by the tasks. Continuing the theme, the IWDL contains two lists, one of the tasks which are to be executed, and another of the data dependencies which are to be satisfied.

A.1 Specification

We now describe the specification of tasks and data. The whole workflow can be described as JSON document consistent of two arrays, one of tasks and another of data. An empty workflow is described an empty JSON document and is a valid IWDL workflow.

```
1 "task_array" : {
2     {"task01" : "details"},
3     {"task02" : "details"}
4 },
5 "data_array" : {
6     {"data01" : "data"},
7     {"data02" : "data"}
8 }
```

A.1.1 Tasks

```
1 "taskname" : {
2     "id" : 1,
3     "command" : "ls -1",
4     "resources" :  {
5         "cores" : 1,
6     }
7     "dependency_info" :  {
8         "task" : [1],
9         "filesystem" : {},
10        "cdo" : {}
```

```
11      },
12      "provider_info" : {
13      },
14  }
```

Tasks in IWDL are uniquely identified by their ID in a workflow. The "id" and the "command" are the only mandatory fields in IWDL, all the other fields are optional. Their descriptions are given below

1. "id" : A unique id in the workflow.
2. "command" : The command to run to execute the task.
3. "resources" : The resources that should be allocated to the task. For example, in case one is using SLURM as a backend to execute the tasks, then the resources will be the account allocation for the task, the number of nodes, the resources required on the node and the time for which the resources might be needed.
4. "dependency_info" : All entities which should be satisfied for the current task to execute. This includes, for example, all the files that must be present, all the CDOs that must be available in the pool manager and all the tasks that must have completed beforehand for the current task to execute.
5. "provider_info" : All the entities which need the current task to be successfully finished for the entity to be available or ready to execute. It follows the same structure as "dependency_info".

A.1.2 Data

```
1  "taskname" : {
2      "id" : 1,
3      "type" : "maestro_cdo" | "file" | "cortx_object"
4      "backend_identifier" : "cdo_01"
5      "resources" :  {
6          "memory" : 1,
7      }
8      "dependency_info" :  {
9          "task" : [1],
10         "filesystem" : {},
11         "cdo" : {}
12     },
13     "provider_info" : {
14     },
15 }
```

Data in IWDL are uniquely identified by their ID in a workflow. The "id", "type" and "backend_identifier" fields are mandatory, all the rest of the fields are optional. Their descriptions are given below

1. "id" : A unique id in the workflow.
2. "type" : This field indicates the type of data, which can either be a maestro_cdo, or a file on a file system or a cortx object.
3. "backend_identifier" : This is the unique identifier which can be used to identify this data on the data storage backend. For Maestro, it would be CDO name for example, for file, it would be the path to the file.
4. "resources" : The resources that should be allocated for the data" by the task that is looking to consume it.
5. "dependency_info" : All entities which should be satisfied for the current task to execute. This includes, for example, all the files that must be present, all the CDOs that must be available in the pool manager and all the tasks that must have completed beforehand for the current task to execute.
6. "provider_info" : All the entities which need the current task to be successfully finished for the entity to be available or ready to execute. It follows the same structure as "dependency_info".

A.2 A Simple Example

A simple IWDL producer consumer example looks like follows:

```
1  {
2      "task_array" : [
3          "producer" : {
4              "id" : 1,
5              "command" : "./produce cdo_01",
6              "resources" :  {
7                  "node" : 1,
8              },
9              "provider_info" : {
10                 "data"  : [
11                     "cdo_01"
12         ]}},
13         "consumer" : {
14             "id" : 2,
15             "command" : "./consume cdo_01",
16             "resources" :  {
```

```
17                  "node" : 1,
18              }
19              "dependency_info" :  {
20                  "data" : [
21                      "cdo_01"
22              ]}},
23              "provider_info" : {},
24          }
25      ],
26      "data_array"  : [
27          "cdo_01" : {
28              "type" : "maestro_cdo",
29              "dependency_info" :
30              {
31                  "task" : ["producer"]
32              },
33              "provider_info" : {
34                  "task"  : ["consumer"]
35  }}]}
```

You can see here that the producer has an "id" 1, the command required to run the producer is given by "./produce cdo_01". The task takes 1 node, and provides a data named "cdo_01". If then we move to the "data_array" part of the description, there we see that "cdo_01" is a data of type "maestro_cdo" which depends on the task "producer" and provides for the task "consumer".

An important thing to remark here is that the "consumer" does not directly depend on "producer" and instead only depends on the data which is required. This allows the following different opportunities for the workflow execution framework which would not have been possible otherwise. For example, a workflow execution framework can detect that "consumer" only depends on the "cdo_01" and therefore can insert additional tasks which watch for the CDO to become available, and start the "consumer" even before the producer has finished.

References

[1] 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.

[2] "Montage performance profile." http://montage.ipac.caltech.edu/docs/grid.html.

[3] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[4] N. Dun, K. Taura, and A. Yonezawa, "Paratrac: A fine-grained profiler for data-intensive workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), p. 3748, Association for Computing Machinery, 2010.

[5] R. F. da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny, "Toward fine-grained online task characteristics estimation in scientific workflows," in *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, WORKS '13, (New York, NY, USA), p. 5867, Association for Computing Machinery, 2013.

[6] "Splinter." `https://gitlab.jsc.fz-juelich.de/maestro/splinter`.

[7] J. Badwaik, S. E. Sayed, J. Biddiscombe, and M. Martinasso, *Maestro D4.3*, 2021.