HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Maestro
801101

D6.1
Systems Software Test Description

WP 6: Middleware Validation and Systems Software Demonstrators

Christopher Haine
Utz-Uwe Haus
Adrian Tate

Date of preparation (latest version): 2019-12-30

DOCUMENT INFORMATION

Deliverable Number	D6.1
Deliverable Name	Systems Software Test Description
Due Date	2019-12-31 (M16)
Deliverable lead	SEA
Authors	Christopher Haine Utz-Uwe Haus Adrian Tate
Responsible Author	Christopher Haine (CRAY) e-mail: chaine@cray.com
Keywords	[Memory Systems, Data Middleware, HPC]
WP/Task	WP 6/Task(s) 6.1
Nature	R
Dissemination Level	PU
Planned Date	December 2019
Final Version Date	2019-12-30
Reviewed by	Dirk Pleiter Simon Smart

DOCUMENT HISTORY

Partner	Date	Comment	Version
CRAY	2019-11-29	Initial draft	0.1
CRAY	2019-12-06	Internal review version	1.0
CRAY	2019-12-30	Version after review	1.1

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 6 (WP6) of the Maestro project develops the Systems Software Demonstrators and validates the middleware.

This deliverable defines, for each individual functionality developed in the project, a set of "minimal viable products" (MVPs) and measurable quantities that permit evaluating the correctness and performance of the feature. This deliverable deliberately focuses on describing tests for the individual features; see Deliverable D6.2 for a set of tests and validation criteria based on demonstrators employing Maestro as a whole.

Contents

1 Terminology/Glossary

Term	**Meaning**
Core Middleware / Maestro Core	The internal software layer that constitutes the Maestro framework as developed in WP3
Core Data Object (CDO)	Fundamental data unit understood by Maestro and communicated between Maestro and Application
MPI	Message Passing Interface
WLM	Workload Manager
Workflow	A set of computational tasks that require scheduling due to task or data dependencies
Producer	An application or process creating a CDO and making it available to other Maestro-enabled applications
Consumer	An application or process requiring a CDO provided by a Producer
Archiver	An application or process listening to a set of CDOs transactions and writing said CDOs to disk.
MVP	Minimal Viable Product – a software artifact that showcases a particular previously defined functionality with just enough features to satisfy early customers and provide feedback for future product development

2 Introduction

This deliverable defines, for each individual functionality developed in the project, a set of "minimal viable products" (MVPs) and measurable quantities that permit evaluating the correctness and, where appropriate, performance of the feature. This deliverable deliberately focuses on the functionality of the individual features; see Deliverable D6.2 for a set of tests and validation criteria based on demonstrators employing Maestro as a whole. Performance measurements, while targeted for some aspects like 3.1, are not the main goal of this deliverable.

3 System Software Demonstrators

Each of the six sections below covers a certain aspect – from the System Software layer through Guided I/O – of the Maestro middleware. The MVPs described will provide examples of how to exercise that particular aspect, as much as possible in a self-sufficient manner. As such, they can serve as illustrative examples to new users, and for the implementers of the actual demonstrators of D6.2. At the same time they will provide suitable test for the continuous integration process during late phases of the development.

The MVPs will be judged by criteria set forth in the respective sections. Each such criterion will be labelled by the type of requirement it covers: exercising a Maestro subsystem for which telemetry parameters will show statistics (*Telemetry*), providing explicit benchmark data in relation to requirements defined in D2.1 [2] or to be defined later (*Benchmark*), or providing qualitative functionality or correctness tests (*Qualitative*). The first and last will be deemed successful if the parameters can be obtained, or the functionality is demonstrated.

Since D2.1 sets forth only a very limited amount of quantitative requirements, most test criteria are of a qualititive nature, or provide quantitative data without specifying benchmark goals. Defining those and evaluating them will be part of a feedback process towards completion of D6.3, the application porting report, as well as D6.4 (the implementation of this deliverable), and D6.5 (implementation of D6.2).

Each demonstrator has a *Lead* in charge of its design and implementation.

3.1 Maestro framework use (Lead: CRAY)

3.1.1 Objective

This section defines four tests of the Maestro Core functionality, as well as a number of parameters that will be more generally useful as telemetry values to observe Maestro Core behavior (see WP5 for details on telemetry in Maestro).

3.1.2 Description

The demonstrators will showcase the coordinated startup of multiple Maestro-enabled applications, comprising producer(s) and consumer(s) of CDOs, with artificial workloads. Each will exercise a certain aspect of the Maestro core functionality and can serve as example code for that aspect of the Maestro system.

3.1.3 MVPs

M1.1 Multithreaded single process Maestro usage. A first version was created in D3.2, and was already reviewed in this context. It consists of a multithreaded application where threads for producer, consumer, and archiver use Maestro inside one address space to pass data objects around. The number of threads for producers, consumers and archivers respectively is configurable via a configuration file. Each of the threads have a single role, ie each thread can be either a producer, or a consumer, or an archiver.

M1.2 Upgrading M1.1 to multi-application (but workflow-coordinator-free) operation, with a script doing what the workflow coordinator would do to manage the applications. The CDO content transfer itself might not be performed, however relevant transfer information will be printed.

M1.3 Transport benchmark for Task 5.1. Scalable 1-1, 1-N, N-1, N-M, single node and multi-node distributed CDO transport scaling test run through some parametric workflow definition.

M1.4 Transformation operations on CDOs: non-pool transformations using explicit API, automatic transformations when DEMANDing from pool.

3.1.4 Success criteria and performance metrics

Number	Description	Type	Requirement in D2.1
T1.1	Number of CDOs created per second (per Maestro app, per node, on pool manager...)	Telemetry, Benchmark	R1.15 (20000/s creation rate)
T1.2	Histogram of CDO sizes over Maestro apps	Telemetry	
T1.3	Histogram of transfer bandwidth over CDO sizes for various memory layers	Telemetry	
T1.4	Wait times on DEMAND and WITHDRAW operations by object size and other CDO attributes	Telemetry	
T1.5	CDO search times when creating/accessing CDO GROUP by attribute matching	Telemetry	
T1.6	Type of storage used and their load, write (production) rate, read (consumption) rate, etc	Telemetry	R1.14 (statistics collection)
T1.7	Allocation statistics from Mamba memory management layer	Telemetry, Benchmark	R.16 (150TB/h created)

3.2 Data aware runtime (Lead: CEA)

3.2.1 Objective

The demonstrator will showcase Maestro Core Data Model to encompass relevant data-locality information and to integrate them in a runtime system to improve data-related performance.

3.2.2 Description

Usual simulation codes rely on domain decomposition to span the complete computation on multiple processes and multiple nodes. These domains must communicate information to have correct computations. These communications usually involve data at the boundaries of the domains. MPC [1] is an implementation of the MPI API, focusing on interactions with shared-memory models, and data locality. To alleviate shared-memory benefit for the data boundaries exchange, MPC proposes an extension called MPI_Halo [1]. MPI_Halo is a feature in MPC allowing to encompass information about halo data-transfer in domain decomposition applications. MPI_Halo allows the exchange

[1] `http://www-hpc.cea.fr/en/red/docs/MPC-en.pdf`

of pointers in shared memory regions instead of all the data range as defined in the regular MPI standard.

As the Maestro API is designed to express data-locality and help realize efficient data migration, it is possible to use the Maestro API to provide another implementation of the MPI Halo shared-memory part. As of now, in shared-memory MPI Halos only exchange pointers, which can leads to NUMA effect. According to the data access pattern, it might be more efficient to move data in a closer memory, avoiding these NUMA effects. As Maestro orchestrator deals with data migrations, combining it with MPI Halos can lead to a data aware runtime moving either data or pointer depending on the data usage. The information exchanged via MPI_Halo is stored in a specific read-only data structure gathering all data concerned by the Halo exchange, which would be a CDO in Maestro semantics. The concept of MPI_Halo and Maestro CDOs are close enough that MPI_Halo routines may serve as wrapper for Maestro Core Data Model routines, as presented below:

- `MPI_Init` -> `mstro_core_init`
- `MPI_Finalize` -> `mstro_core_finalize`
- `MPIX_Halo_cell_init` -> `mstro_cdo_declare`: initializes a halo-cell container with a given data-layout.
- `MPIX_Halo_cell_set` -> `mstro_cdo_attribute_set`: sets the buffer pointer inside the MPI_Halo object (only possible on local MPI_Halo).
- `MPIX_Halo_cell_get` -> `mstro_cdo_attribute_get`: retrieves the buffer pointer from the MPI_Halo object (only possible on remote MPI_Halo).
- `MPIX_Halo_cell_bind_local` -> `set producer flag for Maestro`: registers a halo buffer as local (available for read in remote processes).
- `MPIX_Halo_cell_bind_remote` -> `set consumer flag for Maestro`: registers a halo buffer as remote (retrieved from a remote process).
- `MPIX_Halo_iexchange` -> `mstro_cdo_offer` (producer) or `mstro_cdo_require` (consumer): triggers MPI_Halo exchanges in a non-blocking fashion.
- `MPIX_Halo_iexchange_wait` -> `mstro_cdo_withdraw` (producer) or `mstro_cdo_demand` (consumer): waits for the end of communications associated with this exchange.

3.2.3 MVPs

M2.1 Implement shared-memory feature of MPI_Halo with CDOs and Maestro Core Data Model. When in shared memory, MPI_Halo API will serve as a wrapper to Maestro Core Data Model. A comparison between existing shared memory implementation in MPI Halo and new implementation based on CDOs will be realized. The goal of this MVP is to demonstrate the possibility to use Maestro model to improve an already existing runtime regarding data-related locality and performance.

M2.2 Leverage data-migration features of Maestro Core Data Model to improve performance.

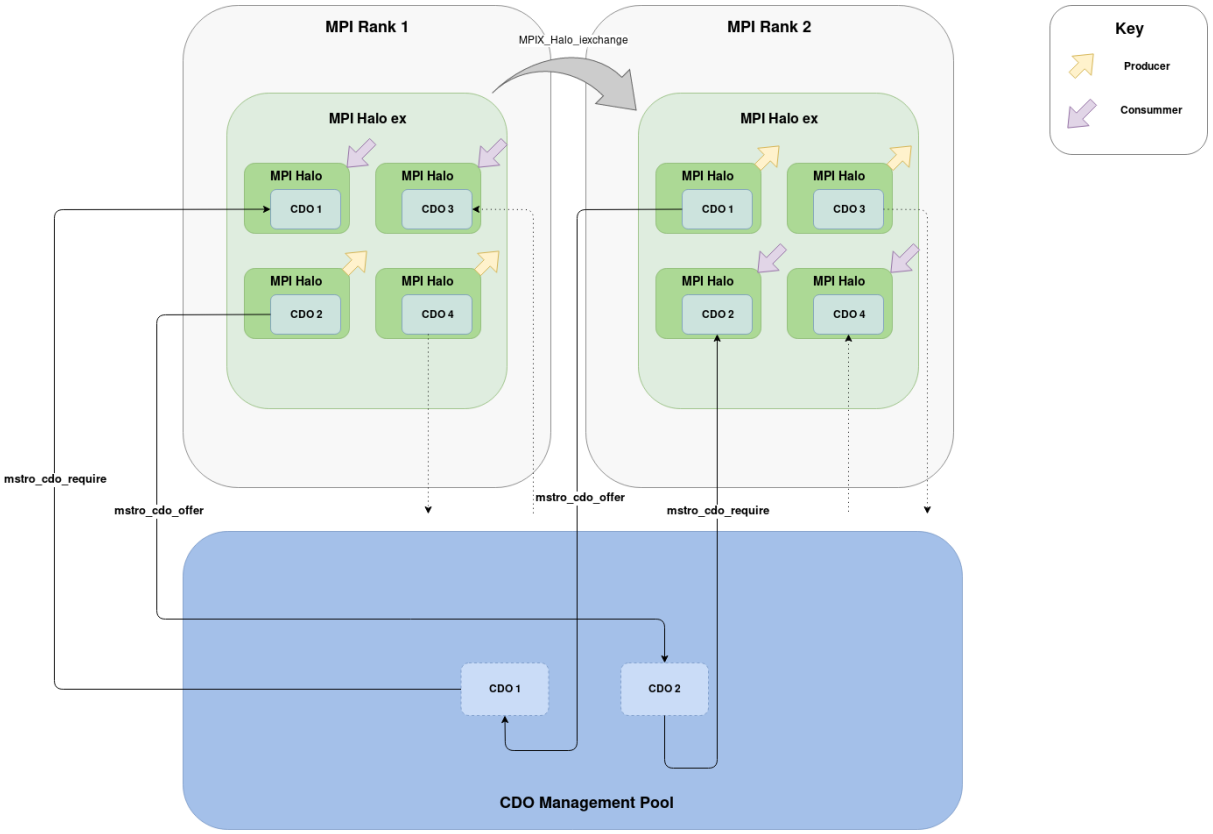

Figure 1: MPI_Halo handling with Maestro Core Data Model

3.2.4 Success criteria and performance metrics

Number	Description	Type
T2.1	Number of CDOs created	Telemetry
T2.2	Number of CDOs per communication	Telemetry
T2.3	Number of task migrations	Telemetry
T2.4	Impact on latency (global and per communication) over current MPC implementation	Benchmark

3.3 Workflow execution and optimisation demonstrator (Lead: JSC)

3.3.1 Objective

This section describes how to test the functionality and performance of the workflow execution and optimization framework of Maestro.

3.3.2 Description

The goal of this demonstrator is to show how Maestro's execution framework can manage workflows. From the workflow manager to the job scheduler, a set of components is in charge of translating a workflow description enhanced with Maestro annotations to a graph of tasks to be scheduled on the HPC system. Those tasks can include both workflow applications as well as Maestro tasks such as a CDO pool manager or a dynamically provisioned file system. The figure below depicts the software stack that will be in place to manage the scheduling of Maestro-enabled workflows. We will focus here on validating the production of a Maestro-enabled set of tasks. That is to say, we will give as input a workflow description that will be transmitted from one component to another until a new graph of dependencies with co-existing applications and Maestro tasks is produced. Those workflow descriptions will be based either upon use-cases derived from applications if applicable or will come from another existing compatible workflow. Deliverable D6.2, submitted in parallel to this one, details use-cases and could be a relevant portfolio to feed the current demonstrator.

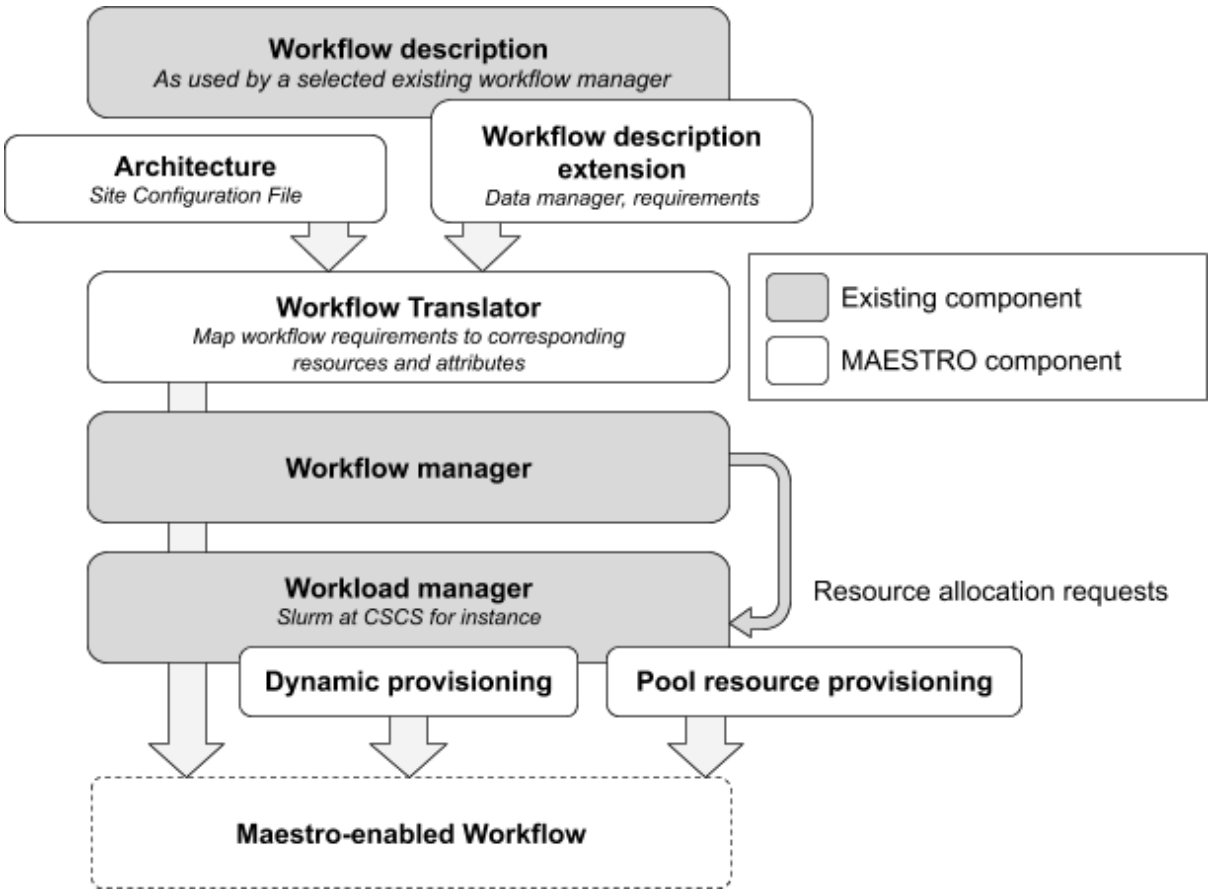

Figure 2: Components of the Maestro workflow execution framework

3.3.3 Success criteria and performance metrics

M3.1 The workflow manager has to be able to deal with basic dependencies scenarios such as diamond or cyclic workflows. This MVP will ensure that the selected

manager can take such a scenario as input and properly schedule it on the target system.

M3.2 We will show how Maestro annotations at the workflow description level can be either passed to the Maestro core for decision making or evaluated at the Workflow Translator stage to modify or improve the graph of tasks.

M3.3 We will demonstrate the ability to update the graph of tasks, that is to add typical Maestro components tasks such as the pool manager or dynamically provisioned storage resources. M3.3 and the M3.2 could be merged if it appears to be more relevant.

3.3.4 Success criteria and performance metrics

Number	Description	Type
T3.1	Validity of the graph of dependencies produced by the connected components	Qualitative
T3.2	Benchmark performance changes due to taking attributes into account at various levels of workflow handling and translation.	Benchmark
T3.3	Profile memory and compute resource usage over the duration of the workflow when taking or not taking attributes into account, and with/without dynamic provisioning.	Qualitative

3.4 Intelligent workload management and data preloading/staging (Lead: APP)

3.4.1 Objective

This section describes how to demonstrate Parallelware's static code analysis capabilities to extract information from the source code of an application and inject such information in the Maestro middleware to enable intelligent decisions about data movement and data placement: Information about data object lifetime, access patterns, read- vs. write access in certain code regions can be turned into CDO attributes; Parallelware could also be used to generate all Maestro API calls in a non-Maestro-enabled code to guide porting efforts, in a similar way as it can already be used for adding OpenMP pragmas.

3.4.2 Description

The Maestro middleware addresses the fundamental problem of efficient data movement and data placement in complex memory hierarchies. In general a solution to this data-movement problem requires making well-informed decisions using a description of the memory semantics of the application (e.g. memory access patterns, data layout) as well as a description of the memory hierarchy of the hardware (e.g. tiers, capacity, bandwidth). On the application side, a precise description of the memory semantics requires a combination of static and dynamic code analyses. This demonstrator focuses on static

code analysis using Parallelware tools to extract the source code information that is relevant for Maestro middleware to enable intelligent decision-making about data movement and data placement in complex memory hierarchies. Our demonstrator will consist of command-line tools run on miniapps (representative subset) of Maestro applications, one for each application, checking that the output of the tools will be either source codes annotated with Maestro APIs and/or configuration files for Maestro middleware.

3.4.3 MVPs

M4.1 Inject application's source code information into Maestro middleware. The Maestro middleware will do intelligent data movement and data placement using information extracted from the source code of an application. Such information can be injected into Maestro middleware in several ways, for example, by creating a configuration file to setup Maestro middleware or by developing a Maestro middleware module that uses Parallelware's APIs. This demonstrator will provide an API and a command-line tool to showcase the interaction between Parallelware and Maestro.

M4.2 Assisted development of Maestro-enabled applications. An alternative way of injecting information into Maestro middleware is by developing Maestro-enabled applications that use the Maestro API directly. In general, the refactorization of the source code of an application using the Maestro API (e.g. the low-level Mamba API, the high-level CDO API) is time-consuming and error-prone. This demonstrator will provide a command-line tool to help the programmer in building Maestro-enabled applications.

3.4.4 Success criteria and performance metrics

Number	Description	Type
T4.1	Source code coverage in terms of the number of source variables turned into CDOs by that can be managed by the Maestro middleware.	Qualitative
T4.2	Level of automated insertion of Maestro API calls and number of new statements added to the source code to modify an existing application in order to make it Maestro-enabled.	Qualitative
T4.3	Benchmark of Maestro Pool scheduling behaviour with/without taking attributes generated by static analysis into account.	Benchmark

3.5 Dynamic Provisioning (Lead: ETHZ)

3.5.1 Objective

This section describes how to demonstrate and measure functionality of the dynamic provisioning component of Maestro.

3.5.2 Description

HPC systems provide dynamic access to compute nodes through a batch scheduler. However, little has been done for dynamically provisioned storage resources. Such resources are traditionally shared among all users. In the context of Maestro, a dynamic provisioning mechanism as part of the middleware allows allocation of dedicated resources to a workflow and deploy an appropriate data manager on top. This set of resources can be explicitly requested through the workflow description or granted based on quantitative and qualitative requirements expressed by the workflow (bandwidth, metadata intensive, and so on). The dynamic resource provisioning mechanism is an independent component that fits in the software stack in charge of the workflow execution (workflow manager, translator, workload manager). More precisely, the component is meant to be scheduled as one or more tasks by the job scheduler, itself steered by the workflow manager. The new tasks will integrate the graph of Maestro-enabled task dependencies (applications). The Figure below presents the general functioning of this Maestro component. Expressing the workflow requirements as a graph of tasks is one of the topics of Section 3.3. This will come as an input of the job scheduler. As we cannot cover all the available data managers, we will implement a subset of commonly used ones including a parallel file system, a database or an object store.

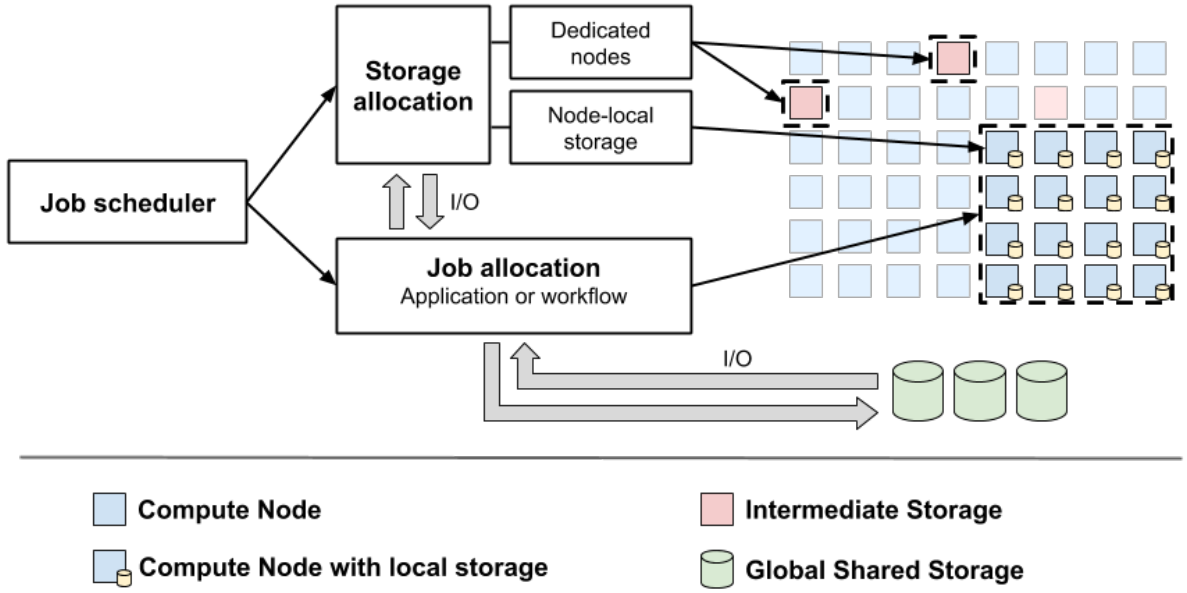

Figure 3: Overview of our dynamic resource provisioning mechanism

3.5.3 Demonstrators

The demonstrators will show how storage resources can be dynamically provisioned by means of a workflow description or a workload manager. In particular, we will show:

M5.1 A set of storage resources dynamically provisioned for an I/O intensive benchmark. This demonstrator will be Maestro-independent, meaning that no Maestro component will be involved in the execution.

M5.2 A workflow description that depicts a simple I/O intensive benchmark and explicitly requires an isolated parallel file-system will be translated into two tasks to be carried out by the job scheduler. Here, a Maestro-enabled workflow manager (i.e. with the support of data-aware Maestro annotations), as described in Section 3.3 will be necessary. However, the benchmark does not need to be Maestro-enabled (i.e. use Maestro-core features)

M5.3 An Maestro-enabled workflow from the applications portfolio will be used to demonstrate the benefit of the dynamic provisioning mechanism. This workflow or this subset of a workflow will require a specific data manager and a set of resources expressed as capacity and/or bandwidth requirements

Work has already been done on this component and a first proof of concept has been published [2]. The current status of the dynamic resource provisioning has reached a certain level of maturity. While the proof of concept was implemented with Python scripts, the current version is now using a tool dedicated to this task, namely Ansible [3], and is on production on several large-scale systems.

3.5.4 Success criteria and performance metrics

Number	Description	Type
T5.1	Deployment time according to two parameters: the scalability (number of nodes, number of disks) and the data manager	Benchmark
T5.2	Stage-in / stage-out efficiency, in terms of exploiting available capabilities, for different types of datasets (lots of small files, large files, ...)	Benchmark

3.6 Guided I/O in pre/post-processing (Lead: SEA)

3.6.1 Objective

The objective of this demonstrator is to show that MIO is usable by the Maestro applications through Maestro middleware.

3.6.2 Description

The MIO interface acts as a generic object store I/O interface for the Maestro middleware providing a gateway to various persistent storage backends. Guided-I/O is one of the important aspects of the MIO interface. Maestro can specify optimization and data usage hints to the persistent storage backend through MIO, which is helpful to meaningfully organise data and optimize the data management of Maestro Objects. Guided I/O involves hints provided to the backend persistent storage which stores Maestro data,

[2] F. Tessier, M. Martinasso, M. Chesi, M. Klein, M. Gila, "Dynamically Provisioning Cray DataWarp Storage" in Cray User Group Conference 2019 (CUG 2019), Montréal, Canada (May 2019), http://www.francoistessier.info/documents/CUG2019.pdf

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

through which the storage system is able to pre and post process data appropriately. The demonstrator focuses on these two aspects, as Guided I/O by itself relies on the working of MIO.

3.6.3 MVPs

3.6.3.1 MIO The following are the MVPs planned to be demonstrated as part of the MIO interface:

M6.1.1 Working of the backend storage (Mero and the Clovis API) with MIO - which is able to accept basic read and write operations.

M6.1.2 MIO is able to accept scatter gather read/write operations from Maestro

M6.1.3 Maestro middleware is able to query the state of an ongoing operation

M6.1.4 Maestro can perform Key-Value operations on the backend object store, namely create, delete, query, insert and remove key-value pairs

M6.1.5 Maestro can create and delete composite objects that are spread across the different persistent storage tiers

3.6.3.2 Guided I/O within MIO The following are the MVPs planned to be demonstrated as part of the Guided I/O MIO Interface:

M6.2.1 Basic working on the guided I/O interface, ability to get and set hints through the MIO guided I/O interface

M6.2.2 Demonstration of hints on data lifetimes corresponding to Mero object data (which in turn maps to Maestro objects)

M6.2.3 Demonstration of hints to explicitly place data in specific persistent storage tiers

M6.2.4 Demonstration of hints to explicitly place data in certain tiers based on basic performance categories (Gold/Silver/Bronze)

3.6.4 Success criteria and performance metrics

Number	Description	Type
T6.1	Number of backend object reads and writes corresponding to a CDO read/write list of persistent object data lifetimes recorded for the CDOs, during the duration of the workflow, before which data was downmigrated/deleted	Telemetry
T6.2	Number of Object Reads/Writes per second observed corresponding to the different storage tiers when hints are used	Telemetry

References

[1] Jean-Baptiste Besnard, Allen Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. An mpi halo-cell implementation for zero-copy abstraction. In *Proceedings of the 22nd European MPI Users' Group Meeting*, EuroMPI '15, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Domokos Sarmany, Simon Smart, Teodor Nikolov, Julien Capul, Sebastien Morais, and Francois Tessier. *Maestro D2.1*, 2019.