
# D6.5
# Applications demonstration

| **Work package** | **WP6 Middleware Validation and Systems Software Demonstrators** | |
|---|---|---|
| Author | Domokos Sármány | ECMWF |
| Author | Miloš Lompar | ECMWF |
| Author | Simon Smart | ECMWF |
| Author | Jayish Badwaik | JUELICH |
| Author | Marc Pérache | CEA |
| Author | Christopher Bignamini | ETHZ |
| Reviewer #1 | Sai Narasimhamurthy | Seagate |
| Reviewer #2 | Dirk Pleiter | JUELICH |
| Dissemination Level | Public | |
| Nature | Report | |

| **Date** | **Author** | **Comments** | **Version** | **Status** |
|---|---|---|---|---|
| 16.11.2021 | Domokos Sármány | Initial version | v0.1 | Draft |
| 28.11.2021 | Domokos Sármány | First-review revision | v0.2 | Draft |
| 29.11.2021 | Domokos Sármány | First revision for EC | v1.0 | Draft |
| 30.11.2021 | Domokos Sármány | Second revision for EC | v1.1 | 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

# Contents

# Glossary

| Abbreviation | Expansion |
|---|---|
| **CDO** | Core Data Object, fundamental data unit understood by the Maestro middleware and communicated between the Maestro middleware and its applications |
| **CLM** | Community Land Model |
| **COSMO** | A non-hydrostatic limited-area atmospheric prediction model |
| **CSCS** | Swiss National Supercomputing Centre |
| **DFT** | Density Functional Theory, a computational method for quantum-mechanical simulation |
| **ECMWF** | European Centre for Medium-Range Weather Forecasts |
| **FDB** | The Fields Database |
| **GPU** | Graphics Processing Unit, a processor optimised for display functionality |
| **HPC** | High-performance computing |
| **IFS** | Integrated Forecast System |
| **MARS** | Meteorological Archival and Retrieval System, ECMWF's perpetual archive for meteorological data |
| **MCT** | Model Coupling Toolkit |
| **MIO** | Maestro I/O interface |
| **MPI** | Message Passing Interface |
| **MPMD** | Multiple Programs Multiple Data Streams |
| **NWP** | Numerical Weather Prediction |
| **OASIS3-MCT** | A coupler between numerical codes representing different components components of the climate system |
| **ParFlow** | An open-source, modular, parallel watershed flow model |
| **MP** | Pool Manager |
| **SCRIP** | Spherical Coordinate Remapping and Interpolation Package |
| **TSMP** | Terrestrial Systems Modelling Platform, TerrSysMP |
| **WP** | Work Package |

# 1 Introduction

This document describes the final evaluation of the Maestro middleware from the applications' perspective. For each application, deliverable D2.3 describes in detail at least one usage scenario, a small number of use cases derived from each usage scenario and a larger number of requirements that are defined based on the analysis of the use cases. The application partners have also created model workflows, which are described in detail in D2.4 [1]. These are simplified versions of existing workflows in actual use, designed to capture the most salient characteristics of data movement across workflow components and thus to be able to demonstrate the benefits of Maestro without the need for full re-engineering.

These model workflows form the basis of the current evaluation and represent four application areas:

- numerical weather prediction (ECMWF),
- computational fluid dynamics plus in-situ analysis (CEA),
- earth modelling system (JUELICH), and
- electronic structure calculation library (ETHZ).

For each of these applications, we provide a summary of both the actual usage scenario(s) and the original model workflow, with much more detailed discussions available in deliverables D2.3 and D2.4.

In order to be able to make use of some of the Maestro middleware's features, many of the model workflows' components, and sometimes the workflows themselves, have had to be modified. These Maestro-enabled workflows are used to validate those of the Maestro middleware's features that are most relevant for each of the applications. More specifically, a list of demonstrators have been created, each with a capability of validating a set of requirements.

In this final document, we describe the steps and modification we have had to apply to make each workflow Maestro-enabled. Then we provide a list of demonstrators we have built based on the Maestro-enabled workflows. We also outline how most of the requirements – derived from analysing the applications' workflows in D2.1 and D2.3 – have been validated and how many of those requirements have been shown to be satisfied by the Maestro middleware. When necessary, we also provide comparison with a similar demonstrator based on the original workflow.

Apart from validating specific requirements, the demonstrators are also used to validate the Maestro-enabled workflows against the performance metrics defined in D2.4.

# 2 IFS numerical weather prediction system

## 2.1 Introduction

ECMWF runs operational weather forecasts four times a day and each of those runs executes the full operational workflow. The workflow consists of three distinct parts: forecast model, post-processing, and dissemination. Crucially, the entire workflow needs to complete within a time-critical window of one hour. The forecast system is responsible for creating an ensemble of weather forecasts that predict the weather up to 15 days (or more) ahead. It also contains dedicated I/O servers that carry out the necessary operations to be able to produce global, two-dimensional fields, each representing the full horizontal global domain at a given level of the atmosphere. Each field is stored and indexed in a domain-specific database of meteorological data, the fields database (FDB).

Post-processing is triggered when all the necessary fields for a given product have been made available. It applies user-defined requirements to build pipelines of transformations, such as conversion between spherical-harmonic and grid-point representations, interpolation, rotation, and cropping. The products then are re-encoded and written to the filesystem. Finally, the dissemination system reads the products from filesystem and distributes them to ECMWF's member states and commercial customers. Figure 1 shows a sketch of the workflow.

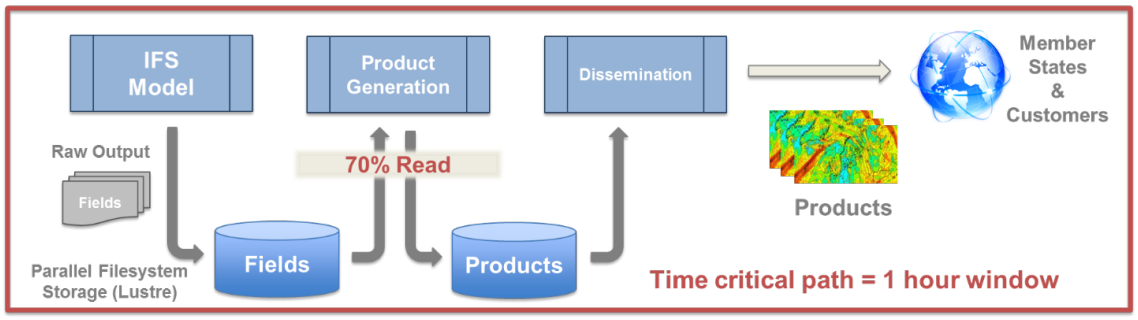



Figure 1: Full operational pipeline at ECMWF

Congestion in data movement typically occurs between the forecast model and post-processing because of the large number of simultaneous read and write operations. There are multiple I/O servers per ensemble member and the write pattern is per I/O server, while the read pattern is across the I/O servers. The congestion often comes from the metadata handling while managing this.

To capture this without running the entire operational pipeline, we have created a model workflow that contains most of the actual workflow components relevant for the Maestro project, but where the forecast model is replaced by a tool called `multio-hammer`. It mocks the output production from the I/O servers and it is configurable to reproduce output patterns of current operational (and research) workflows as well as to match expected patterns

from future forecasts (with higher resolution or more ensemble members, for example).

There are two versions of the model workflow: one with the original software components and one, called the Maestro-enabled workflow, where the some of these components have been adapted to use the Maestro middleware or replaced entirely by Maestro technologies.

## 2.2 Original model workflow

The original model workflow as well as the reasons and ways to derive it from the actual workflow are detailed in deliverable D2.4 [1]. It captures the essence of the I/O-related challenges being tackled within the Maestro project, whilst avoiding running the large number of tasks required to coordinate a full forecast. It simplifies the workflow to its three essential components.

- *Data generation* In an actual operational workflow, forecast data is routed through a number of dedicated I/O nodes, each running a predefined number of producer I/O tasks. These output data via the domain-specific I/O library, called `multio`. In the model workflow, we create a tool on top of the `multio` library to generate synthetic data.
- *Metadata handling* A key element of the actual workflow is that it is driven by metadata. The consumer needs to refer to the data it reads via its metadata. So it is important that the generated data have metadata that is identical to the actual data, even if the data itself is scientifically meaningless.
- *Data movement* The movement of data between I/O-node producers and the post-processing consumers is the bottleneck in the current operation workflow, so they must also be well represented in the model workflow. This is achieved by using the same consumer task in the model workflow as in the actual workflow.

Figure 1 shows both the operational and the model workflows, as well as the steps taken to derive the model workflow form the operational one. The model workflow is executed by `kronos`, a workload manager that is also used for benchmarking at ECMWF, and consists of the following main software components.

- `The Kronos executor` Amongst other things, `kronos` is an event-driven workload manager. Given a (configurable) schedule, it acts as a meta-scheduler, submitting tasks to the system scheduler. These tasks can notify `kronos` of events (startup, completion, error and the completion of intermediate steps) that will in turn trigger further submission of jobs. This component acts as a substitute for ECMWF's large-scale forecast workflow manager, `ecflow` [2].

- `multio-hammer` The real forecast model is large, uses significant computational resources, is complex to configure and needs to be carefully tuned to specific HPC systems. For the purposes of the Maestro project, the primary area of interest is the output from the model via its I/O servers. They can sink large number of fields of arbitrary size to a filesystem, object store or any other middleware. It can also send notifications to `kronos` to trigger the post-processing.
- `pgen` The product-generation engine at ECMWF, which transforms forecast model output into the required products according to customer-specified requirement definitions. One `pgen` task is launched per completed forecast step, and it reads output data from across all ensemble members. The pgen tasks included in this workflow perform tasks very close to those of the real post-processing, although on synthetic data output from `multio-hammer`.
- `fdb` The Fields Database (FDB) [3] is the library and data service used to handle, transfer, store and index meteorological objects in the forecast pipeline. It provides the I/O layer used by other components. This is the middleware through which data transfer between the I/O-node producers and the `pgen` consumers takes place.

## 2.3 Maestro-enabled model workflow

We aim to make use of the Maestro middleware to mitigate the congestion between the forecast and product generation. In the model workflow, it fully replaces `fdb`, which currently persists time-critical forecast data onto a globally-visible parallel file system. Figure 2 shows a high-level view of how the Maestro middleware is used to direct the time-critical part of the forecast output to the product-generation task, bypassing the global parallel file system.

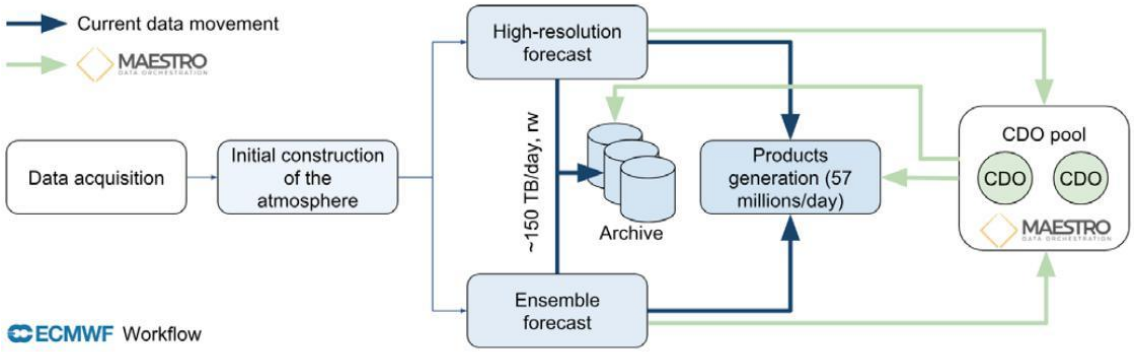



Figure 2: Potential use of the Maestro middleware in operational workflow

We have used the model workflow to highlight the parts of the software stack that need to be modified if we are to make efficient use of the Maestro

middleware. It has also provided a much simplified test environment to drive the development of these modifications.

We have identified and implemented the following changes to the model workflow.

- Extend the functionality of the `multio` library to be able to support arbitrary, configurable processing chains driven by metadata and thus to act as an I/O-server for the ocean as well as the atmospheric components of IFS (as opposed to being a purely I/O-library of the fortran-based I/O-server inside the atmospheric model). It has been re-designed to accept a configurable, extendable and composable list of actions. For the demonstrations, we use just one action – the output (or 'sink') action – that is responsible for passing meteorological data to a storage system, database or, indeed, a data-management system. A backend to `multio` to support output to the Maestro core middleware has been developed.

- Create a software tool, based also on the `multio` and `pgen` libraries, that is able to request data from the Maestro middleware using metadata queries. It uses the select and subscribe mechanisms from the Maestro middleware's event-handling architecture, which have been included in the design for this purpose. This component is also responsible for passing the data onto `pgen` for post-processing and distributing such work in a scalable manner.

- Adapt `pgen` to be multithreaded within a node. It is primarily designed as a multi-process application in the original workflow. In the Maestro-enabled version, there is a single 'broker' thread on every node that posts metadata queries based on the post-processing requirements and subscribes to events that signal when a data object matching the query is available for consumption. It then places the requirement and the corresponding data handle into multithreaded data structures. Multiple worker threads may thus simultaneously process the requirements, one at a time, using the data handle to facilitate the transfer of data on which actual post-processing (interpolation cropping, etc.) is carried out. Figure 3 provides a visual description of the Maestro-enabled workflow. It shows how the metadata-handling is decoupled from the actual data transfer and how each consumer has been re-designed to make the best possible use of the Maestro functionalities.

- Launch the consumers immediately at the beginning of the workflow. The Maestro middleware's support for event subscriptions have made it possible to do this, as opposed to than waiting for data from each step from each producer to be persisted on the parallel filesystem. Indeed, the dependency in the Maestro-enabled workflow is reversed as we need to ensure that the consumers join the workflow before any of the producers, otherwise data may be lost. We have implemented two different ways to support this: one through dependencies in the `kronos`

workload manager, and one via subscriptions to the Maestro pool manager's 'join' events.

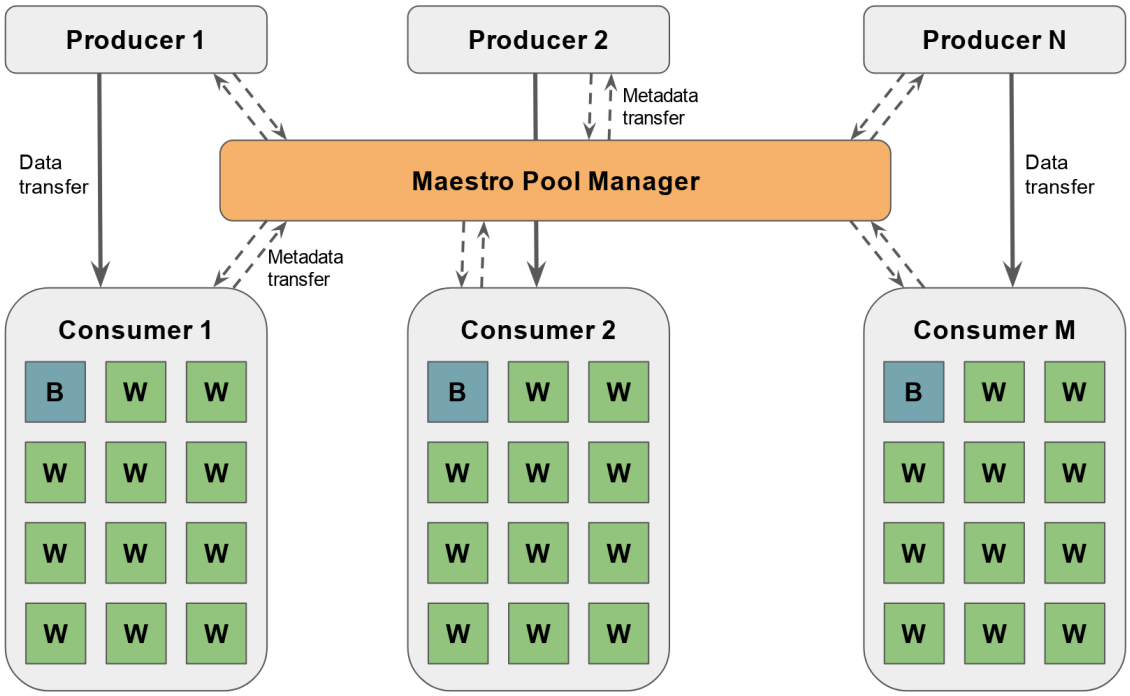



Figure 3: High-level overview of the Maestro-enabled workflow, emphasising changes to `pgen`, the post-processing component that acts as the consumer. A multi-threaded design with one broker (B) and multiple workers (W) enables the use of the Maestro architecture and the decoupling of metadata handling from data transfer. The dashed lines indicate network communication necessary for metadata handling, while the solid lines indicate data transfer.

An important corollary of the last point is that data transfer through the entire Maestro-enabled model workflow is now completed once the last producer leaves the workflow. In the original workflow, there is always a delay because each consumer job can only launch once the corresponding data is known to have been persisted to the filesystem. Figure 4 shows an overview of how the model workflow has been re-designed as part of making it Maestro-enabled. This development, together with the design changes to the consumer shown in Figure 3, has been critical for make maximum use of the Maestro technologies.

We note that the multio library only uses the Maestro core middleware directly (i.e. it only links against `libmaestro.so` directly). What technologies to use from the low-level middleware may be configurable at compile/run time, but the low-level APIs will not be exposed to `multio`. Equally, the multio library provides an I/O abstraction in the ECMWF software stack so that the scientific/computational layers do not need to know about what underlying technology (multio backend) is being used.

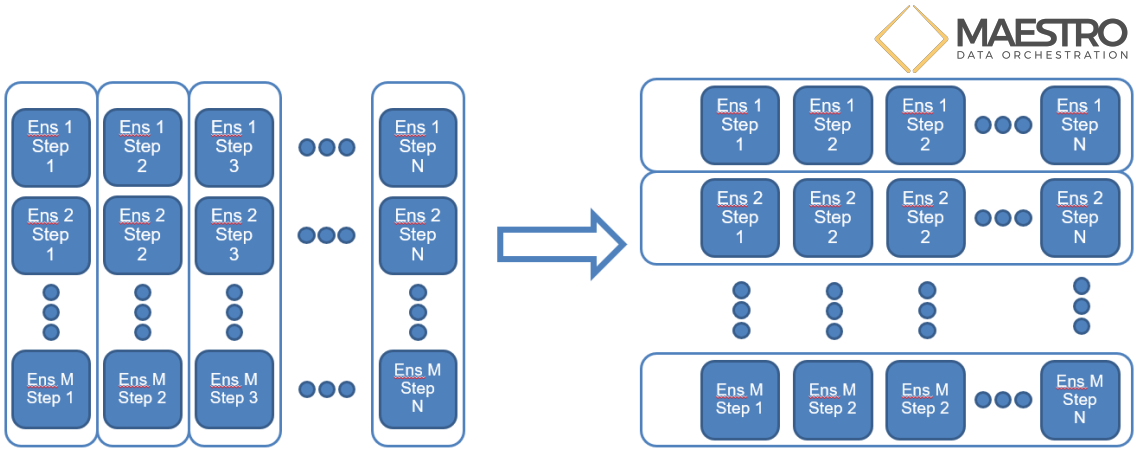



Figure 4: Overview of the design changes between the fdb-based original workflow and the Maestro-enabled workflow. The ensemble-based post-processing makes it possible to start the consumers immediately at the beginning of the workflow. They will be able to process fields as they become available, rather than having to wait until every producer completes a given step and persists the corresponding data on the filesystem.

## 2.4 Demonstrators description

Deliverables D2.1 [4] and D2.3 [5] define the list of requirements the applications impose on the Maestro middleware. Being able to create a Maestro-enabled version of the IFS model workflow, which executes the same work as the original workflow, already suggests that the Maestro middleware satisfies the majority of the most important requirements. Nevertheless, it is instructive to validate each requirement using a small number of dedicated demonstrators.

The demonstrators focus on a few broad categories of features: requirements to cover basic functionalities, metadata manipulations, stress conditions, lifetime management and failure resilience. As a result, the demonstrators aim to be able to validate whether the following high-level features have been correctly implemented.

- *Intelligent data movement.* The general concept to consider information about where the data are likely to be needed in the decisions about where to store and how to transfer them.
- *Data encapsulation.* The concept that data can be encapsulated into a Maestro-specific abstraction and annotated with scientifically meaningful attributes. The Maestro middleware supports this through the abstraction of the Core Data Object (CDO).
- *Metadata handling.* The ability of applications to access semantically related data sets in the workflow. Data objects are referenced via their scientific meaning, rather than, for example, by their unique ids or names. The Maestro middleware supports these operations through its support for CDO attributes and through an event-detection mechanism:

users can subscribe to certain events based on attributes defined by an application-defined attribute schema.

- *Stress conditions.* The ability of the workflow to produce and consume meteorological objects at a high rate. Operational weather forecasting is time-critical and one of the main motivations for the Maestro middleware is to be able to accommodate such use. This requires a management solution that supports multiple ways to transfer data. The Maestro middleware achieves this by creating a dedicated pool manager and by decoupling the data transfer from the pool operations.
- *Avoiding the globally-visible parallel file system.* The concept that, for time-critical runs, it is possible to pass data from the forecast system to the post-processing system without the requirement for disk-based indexing and read/write operations.

**Demonstrator components** Based on the concepts above, we implement a set of demonstrators for the Maestro-enabled workflow. These are used to evaluate the extent to which the requirements outlined in D2.3 [5] are satisfied. Each consists of one or more of the following components, depending on level of complexity.

- *A pool-manager application.* This is the component that manages data request, access and transfer between other components of the workflow. A single pool manager is used, although we demonstrate the point at which multiple pool managers may be required.
- *One or more* `multio-hammer` *instances, acting as producers.* It is often enough to run a small number of these to validate parts of the Maestro middleware's features. However, a large number of these will be used to demonstrate some functionality at scale.
- *One or more consumer applications.* Similarly, a small number of these are enough to validate many of the requirements. But a larger number of these are needed to demonstrate the Maestro middleware at scale.
- *The* `kronos` *workload manager.* This is unnecessary, and thus omitted, for smaller demonstrators that consist of only a few number of producers and consumers. It is used, however, to demonstrate the Maestro-enabled version of larger workflows.

**List of demonstrators** In this final report, the evaluation efforts focus both on the semantic requirements – many of which are prerequisites to building a full Maestro-enabled model workflow – and on measuring its performance against the metrics and baselines defined in D2.4 [1]. We list below the demonstrators that have been developed to validate one or more requirements from D2.3 [5].

- *Demonstrator 1* One producer and one pool manager. This is a minimal configuration, used to validate that communication between a client application and the pool manager can be established and that the data can be wrapped in a Maestro-defined data structure – the Maestro CDO. It can also verify whether there is support for setting user-defined attributes and whether the producer-side semantics of creating, passing, reclaiming and destroying CDOs works according to the specifications.

- *Demonstrator 2* One producer, one consumer and one pool manager without data transfer. In addition to the validation from the previous demonstrator, this is used to validate whether the event detection mechanism in the Maestro middleware behaves as expected within the context of the IFS model workflow. The consumer creates a metadata query and subscribes to an event that indicates if a CDO with metadata matching that query is available for consumption. It then also validates whether the event contains sufficient information to execute the producer-side interface calls of creating, requesting and destroying the corresponding CDO and whether the behaviour matches the Maestro specifications. This demonstrator can also be used as a minimal benchmark to measure the overhead of metadata communications only.

- *Demonstrator 3* One producer, one consumer and one pool manager with data transfer. In addition to the validation from the previous demonstrators, this is used to validate that the data transfer does indeed take place, that it corresponds to the correct description and that it is identical to the data that has been produced. It can also be used as a minimal benchmark to measure data transfer rates and their relative costs to both metadata manipulations and to the costs of product generation.

- *Demonstrator 4* Multiple producers, multiple consumers, one pool manager and executing inside `kronos`. This is used to evaluate the same requirements as the previous demonstrators but at scale. It is designed for the execution of the full Maestro-enabled model workflow via `kronos` and is highly configurable. It can thus be used to validate the stress conditions and performance requirements from D2.4 [1].

- *Demonstrator 5* Multiple producers, multiple consumers, multiple pool managers and executing inside `kronos`. This is the same as the previous demonstrator except for using multiple pool managers to alleviate contention in a single pool manager in larger workflows.

- *Demonstrator 6* Multiple producers, multiple consumers, pool manager, a telemetry listener and using kronos. It is a modified version of the previous demonstrator, when a listener application is added to the workflow to provide telemetry information about the pool manager.

## 2.5 Requirements evaluation

The list of tables below provides a summary of the requirements that the demonstrators have attempted to evaluate as well as the final evaluation status for those requirements. Note that there may be requirements defined in D2.3 [5] that have not been considered for demonstrator evaluation.

| **Reference in D2.3** | **R1.1** |
|---|---|
| **Description** | Objects handled within Maestro can be described and handled according to user-defined domain-specific metadata. |
| **Validation** | Demonstrator 1 describes data similar to the MARS[a] language before handing it over to Maestro. |
| **Evaluation status** | *Validated.* There exists a mechanism to create a user-defined attribute schema for domain-specific metadata. We have been able to use it to set attributes and to subscribe to events associated with data whose metadata matches certain attributes. |

[a] The Meteorological Archival and Retrieval System, operational at ECMWF since 1990

| **Reference in D2.3** | **R1.2, R1.27** |
|---|---|
| **Description** | User-defined metadata should take the form of a dictionary of key-value pairs, or equivalently as a set of arbitrarily named attributes. |
| **Validation** | Demonstrator 1 and the following demonstrators defines metadata as a set of named attributes, where the attributes match a user-provided schema. Calls to the Maestro core API are expected to behave according to the Maestro core specifications. |
| **Evaluation status** | *Validated.* The Maestro middleware's support for named attributes is adequate and its use has been successfully demonstrated. |

| **Reference in D2.3** | **R1.3a-d** |
|---|---|
| **Description** | Object retrieval should support metadata queries that describe lists of values. |
| **Validation** | Demonstrator 2 makes queries using lists of values. These queries are expected to succeed. |

| | |
|---|---|
| **Evaluation status** | *Validated.* The existing Maestro interface for handling metadata queries supports lists of values and those queries are successful. Furthermore, polling based on values of arbitrary metadata is successful and we have been able to initiate data transfer after polling when the metadata query is successful. |

| **Reference in D2.3** | **R1.7a-c** |
|---|---|
| **Description** | Once data objects are handed over to Maestro, managing these must no longer be the application's concern. |
| **Validation** | Demonstrators 2 and 3 make the Maestro API calls that are required by the ECMWF pipelines. They will not make any attempt to access or transport objects once they are handed over to Maestro. |
| **Evaluation status** | *Validated.* All ECMWF demonstrators have successfully completed without concern to transport and access once ownership has passed to Maestro. Also, data objects have remained uniquely identifiable and there is support for locating and accessing objects through their metadata. |

| **Reference in D2.3** | **R1.8** |
|---|---|
| **Description** | Objects may have a minimum lifetime (possibly zero). |
| **Validation** | Demonstrator 2 implements a consumer that needs access to the data some time after the data was handed over to Maestro. The access is expected to be granted during a user-defined period. |
| **Evaluation status** | *Validated.* The applications can control the data objects' lifetimes through the four-step interface [6] of the maestro-core library. |

| **Reference in D2.3** | **R1.10, R1.11, R.12** |
|---|---|
| **Description** | Requirements on Maestro's error-handling functionality. |
| **Validation** | During the demonstrators' development, configuration, and execution, errors have, as expected, occurred and thus these requirements could be validated. |
| **Evaluation status** | *Validated.* There exists an extensive error-handling mechanism, most of which is also useful for users. Both the hard-failure requirements of R1.10 and R1.12, and the soft-failure support of R.11 have been met. |

| **Reference in D2.3** | **R1.14** |
|---|---|
| **Description** | Maestro to record and communicate usage statistics – type of storage used and their load, write (production) rate, read (consumption) rate, etc. – to human operators or system monitoring/logging tools. |
| **Validation** | Demonstrator 6 has a workflow component that is capable of collecting timestamp-based statistics about the Maestro pool-manager. The usage statistics are expected to be available and ready to be interpreted according to the specifications. |
| **Evaluation status** | *Partially validated.* Timestamped statistics about any of the pool-manager's events are available. With simple post-processing it is possible to infer where computational time is spent in the pool manager and where the bottlenecks may occur. |

| **Reference in D2.3** | **R1.15a-d** |
|---|---|
| **Description** | The Maestro middleware must be able to cope with at least 20,000 object creations per second per workflow. This is sustained for most of the workflow duration. |
| **Validation** | Demonstrators 4 and 5 can be configured to create an arbitrarily large amount of mock data, which is scientifically meaningless but otherwise has the same characteristics as actual data. These are fed into the Maestro-enabled pipeline. |
| **Evaluation status** | *Partially validated.* Synthetic meteorological data has been generated and passed to Maestro to validate its metadata-handling capabilities, which has been successful. On workflows that include data transfer for each CDO, we have managed to generate around 17,500 objects of 804KiB each when running 50 producers, 50 consumers and 5 pool managers. This suggests the target can be achieved when using more resources. The division of work to be able to use multiple independent pool managers has been made possible as a result of Maestro developments. However, the use of multiple pool managers is not a truly general solution; work on the multi-threaded and/or distributed pool manager is ongoing. |

| **Reference in D2.3** | **R1.16a-b** |
|---|---|

| | |
|---|---|
| **Description** | The Maestro middleware must be able to support multiple non time-critical workflows, creating at least a combined 150TiB data in an hour. |
| **Validation** | Demonstrators 4 and 5 can be configured to create a large volume (many hundreds of TiB) of mock data per hour to demonstrate Maestro's 'limits'. |
| **Evaluation status** | *Partially validated.* Demonstration on a workflow with 50 producers, 50 consumers and 5 pool manager can process around 40TiB of data in an hour. This suggests it may be possible to achieve this requirement on a system with multiple pool managers on a few hundred nodes. However, the use of multiple pool managers is not a truly general solution; work on the multi-threaded and/or distributed pool manager is ongoing. |

| **Reference in D2.3** | **R1.17** |
|---|---|
| **Description** | Object visibility within Maestro is handled transactionally. No partial state must be accessible. |
| **Validation** | All of the demonstrators carry out sanity checks that objects returned from Maestro are not in a partial state. Objects in partial state will trigger a hard failure. These sanity checks, however, will not be able to prove that partial states cannot exist. |
| **Evaluation status** | *Validated.* During the demonstrator's development, invalid requests have resulted in errors indicating that partial states cannot be accessed. |

| **Reference in D2.3** | **R1.19** |
|---|---|
| **Description** | Objects within Maestro are handled consistently. No risk of undefined behaviour or data corruption. |
| **Validation** | All demonstrators carry out sanity checks for consistency. |
| **Evaluation status** | *Validated.* No data corruption or signs of undefined behaviour have been observed in the final version of the Maestro core middleware[a]. |

[a] Version 0.2

| **Reference in D2.3** | **R1.21** |
|---|---|
| **Description** | A single consumer must be able to iterate over a set of objects, even when this is too large to fit in memory at once. |

| | |
|---|---|
| **Validation** | Demonstrator 2 makes an attempt to iterating over large data sets that typically do not fit into memory. The operations are expected to succeed. |
| **Evaluation status** | *Validated.* There is support for this in the Maestro interface and we have not experienced out-of-memory errors related to the number and sizes of data objects we are iterating over. For large data sets, we observe a decrease in the rate at which the data is transferred. |

| **Reference in D2.3** | **R1.24** |
|---|---|
| **Description** | Once data objects are handed over to Maestro, they must be immutable. |
| **Validation** | The demonstrator attempts to create data with semantic meaning (metadata) identical to data that already exists within Maestro. That attempt is expected not to mutate the existing data. It is expected either to fail or to create new data. |
| **Evaluation status** | *Partially validated.* There has been no systematic validation, but this is strongly supported in the Maestro middleware design and we have found no evidence to the contrary. |

| **Reference in D2.3** | **R1.26** |
|---|---|
| **Description** | Consumer applications should not be required to know the size of the data prior to requesting it. |
| **Validation** | Demonstrators 2, 3, 4 and 5 all have consumer applications that request data from Maestro without specifying the size of the data. |
| **Evaluation status** | *Validated.* Requesting data objects via their metadata only – without specifying its name, unique identifier, or size – has been used extensively and the behaviour is as expected. |

## 2.6 Performance evaluation

We evaluate the performance of the Maestro-enabled workflow, both for its scalability and for its comparison with the original one. Deliverable D2.4 defines the main performance baselines and evaluation metrics for the original IFS model workflow. We use demonstrators 4, 5 and 6 from Section 2.4 because those are the most complex and the most configurable. As a special case, with one producer and one consumer, we can also reduce to demonstrator 3. We carry out four sets of experiments.

- *Experiment 1 – scalability with a single pool manager* Run a full Maestro-enabled workflow with producers and consumers. We maintain the same workload per producer-consumer pair, running an equal number of 50 steps per simulation, and vary only the number of producers (equivalent to forecast ensemble members) and consumers (post-processing tasks). This is a fair configuration for testing the scalability of the Maestro middleware – in particular the pool manager, which is the only shared resource.

- *Experiment 2 – scalability with multiple pool managers* Run one pool manager for every 10 producer-consumer pairs and keep everything else identical to experiment 1. This allows testing the scaling of the Maestro data communication patterns separately from the scalability of the Maestro metadata handling in the pool manager.

- *Experiment 3 – direct comparison* Run both a full Maestro-enabled workflow and full original, fdb-based one. We vary the number of producer-consumer pairs as in the previous experiments, but we now also vary the number of steps of the mock forecast to keep them equal to the number of producers. This is to maintain parity between the simulations – since the number of consumers is equal to the number of steps in the original workflow, while they equal to the number of ensemble members (producers) in the Maestro-enabled one. It is the fairest way we can directly compare the two workflows.

- *Experiment 4 – scalability with increasing steps and data-object sizes* Run a Maestro-enabled workflow with 10 producer-consumer pairs and one pool manager. We then vary only the number of steps and the sizes of the data objects. This ensures that the number of objects the PM needs to manage simultaneously remains flat, while the total amount of data processed by the workflow is increased.

In essence, we can increase the workload in three different ways: by increasing the number of ensemble members, by increasing the number of steps and by increasing the size of each data object. In experiments 1 and 2, we increase only the ensemble numbers, in experiment 3, we increase both ensemble members and steps for the sake of fair comparison with `fdb`, and in experiment 4, we increase that data volume while keeping the rate of metadata communications constant.

Figure 5 illustrates the difference in the ways the workloads increase in experiments 1 and 2 compared with experiment 3. Under ideal scaling, the run times of experiments 1 and 2 would be expected to remain flat, while that of experiment 3 would be expected to increase linearly.

All experiments are executed on Piz Daint[1] at the Swiss National Supercomputing Centre.

[1] `https://www.top500.org/system/177824/`

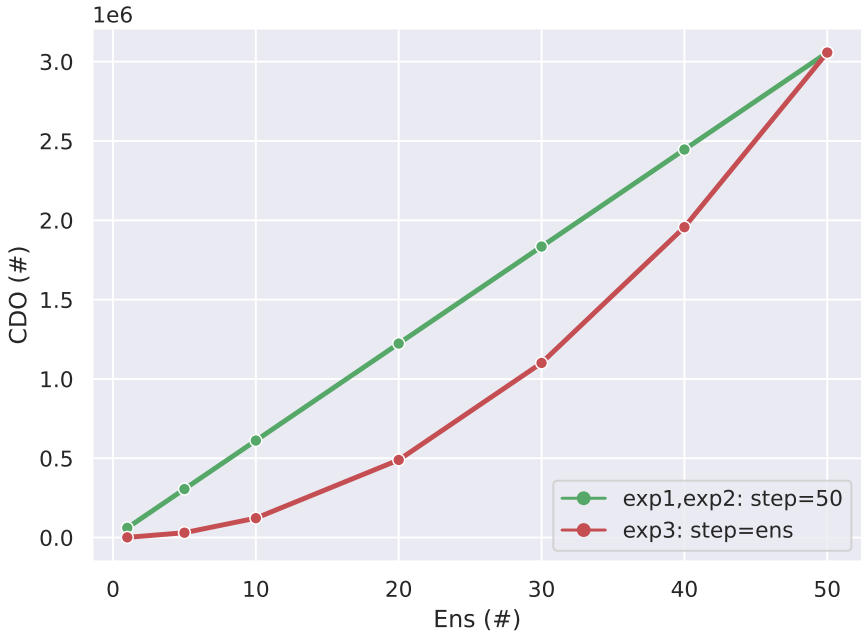



Figure 5: *Experiments 1,2 and 3* Comparison of the total numbers of CDOs processed in the workflows with increasing number of ensemble members (producers). In experiments 1 and 2 (green), the number of steps always equals to the number of producers; in experiment 3 (red), the number of steps is always 50.

### 2.6.1 Experiment 1 – scalability with a single pool manager

We aim to take a closer look at how the Maestro-enabled workflow itself performs with increasing workload and see when the pool manager may hit a contention point. For this, we define the following setting.

- Each of the producers, consumers and the Maestro pool manager runs on a separate node.
- Each producer creates 1223 fields of size 804KiB per step, amounting to a total of around 960MiB data per step.
- Each consumer is running one broker and eleven workers, making use of all the cores available on the node (hyperthreading is turned off). The broker is responsible for requesting the data and every worker is responsible for picking up a request and execute the corresponding data transfer and post-processing.
- The number of steps is set to 50 across all runs to guarantee that each producer and each consumer processes the same amount of data.

We measure three sets of runs for a range of producer-consumer numbers – 1, 5, 10, 20, 30, 40, 50 – and use three different consumer configurations.

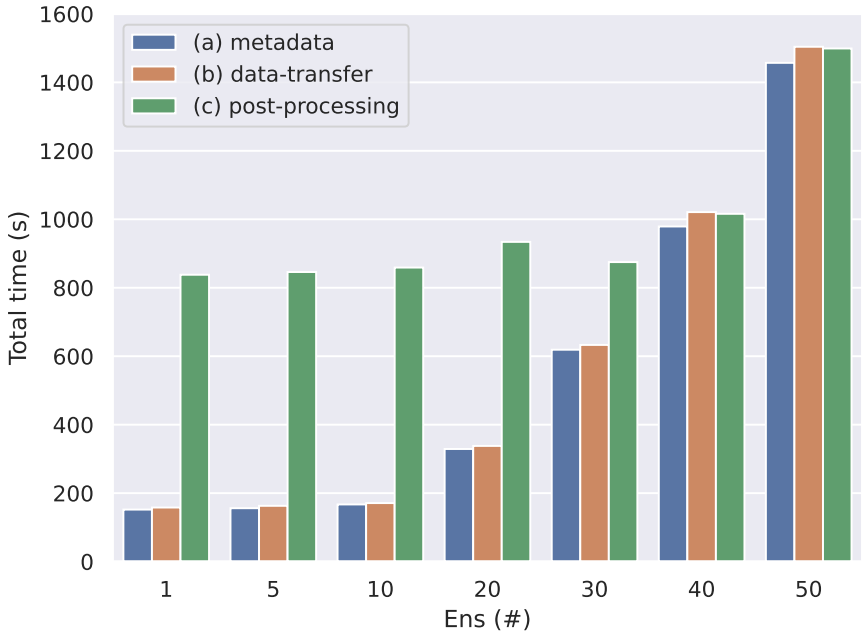



Figure 6: *Experiment 1* Total execution time for the Maestro-enabled workflow for three different configurations: *(a)* metadata communications only; *(b)* metadata communications and data transfer; *(c)* metadata communications, data transfer and post-processing.

1. Consumer retracts the request for the data – no data transfer takes place but there is metadata communication.
2. Consumer demands the data but does not post-process it – metadata communication and data transfer are measured. This is in effect an I/O benchmark.
3. Consumer demands the data and post-processes it – actual scientific work is carried out instead of only executing the I/O benchmark.

Since every producer and every consumer processes the same amount of data across the entire experiment, perfect scaling would indicate constant execution times (bars with similar heights). Figure 6 shows the results for different producer-consumer numbers.

The first observation to make is that for relatively small numbers of nodes, the majority of the work is spent on post-processing computations. This is an ideal position to be in because most of the post processing task is 'embarrassingly paralleliseable' and we can simply increase the number of workers[2].

The second point to note is that there is practically no difference between the case when the data transfer is initiated and when the data transfer is skipped, suggesting that data transfer is very fast compared with the

[2]A limit to that is the number of available cores on the node as the current Maestro-enabled workflow is thread-based.

metadata communications. The current experiment has a large number of relatively small fields, each with a size of 804KiB, and that may contribute to the fact that metadata operations dominate.

A third observation worth making is that the performance measurements for larger number of nodes (between 30 and 50) indicate that the post-processing work and the work associated with obtaining the required data are overlapped successfully. In cases where the metadata operations dominate, there is about 10% overhead for carrying out the post-processing tasks in addition. This is because most of the communication with the pool-manager is handled by the broker thread.

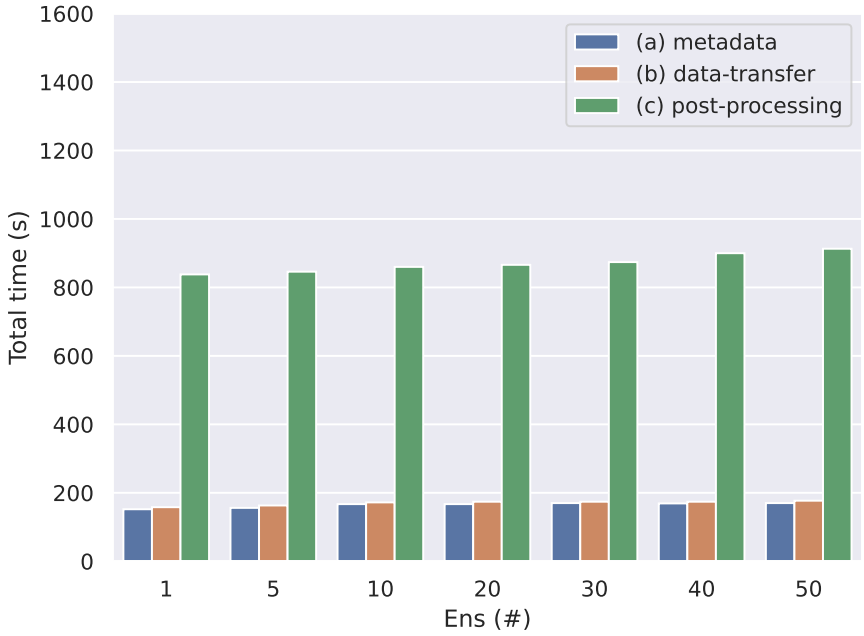



Figure 7: *Experiment 2* Total execution time for the Maestro-enabled workflow, using one pool manager per ten producer-consumer pairs, and for three different configurations: *(a)* metadata communications only; *(b)* metadata communications and data transfer; *(c)* metadata communications, data transfer and post-processing.

### 2.6.2 Experiment 2 – scalability with multiple pool managers

To work around the limitations of the single-threaded pool manager, we have run the same scaling experiment as shown in Figure 6 but configured the workflow so that there is one pool manager for every 10 producers and 10 consumers. This is possible with the Maestro-enabled workflow as we have modified the logic so that data is consumed per producer, rather than per step and across all producers. Nevertheless, it introduces a reduction in the generality of the Maestro-pool manager because the assumption that it can initiate data transfer between any two components of the workflow is lost.

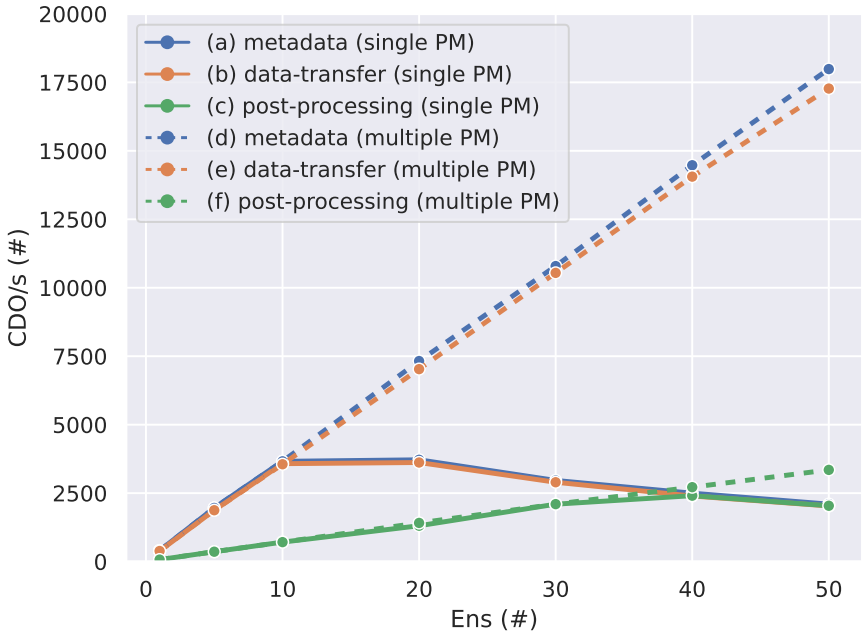



Figure 8: *Experiment 1 and 2* The total number of CDOs processed per second for the Maestro-enabled workflow for six different configurations: *(a)* single PM metadata communications only; *(b)* single PM metadata communications and data transfer; *(c)* single PM metadata communications, data transfer and post-processing. *(d)* multiple PM metadata communications only; *(e)* multiple PM metadata communications and data transfer; *(f)* multiple PM metadata communications, data transfer and post-processing.

Consequently, it is primarily meant to demonstrate the potential of a multi-threaded and/or distributed pool manager, which is the Maestro design.

Figure 7 shows that perfect scaling is achieved for metadata communications and data transfer, while the scaling is slightly suboptimal when the post-processing is also carried out. Figure 8 depicts the rate at which the number of CDOs are processed across the entire workflow for both the single PM case (experiment 1) and the multiple PM case (experiment 2). The single-threaded PM scales perfectly until about ten producers and ten consumers, and there is a clear deterioration of performance beyond that point. The workflow with multiple PMs maintains the near-perfect scaling up to 50 producer-consumer pairs, the largest workflow studied.

### 2.6.3 Experiment 3 – direct comparison

We aim to compare the Maestro-enabled workflow and the original, fdb-based workflow from the perspective of the I/O workload and, in particular, from the perspective of how quickly can data pass through the workflow. Both workflows are thus executed in dry-run mode where each pgen job's work is restricted to retrieving and taking ownership of the data while

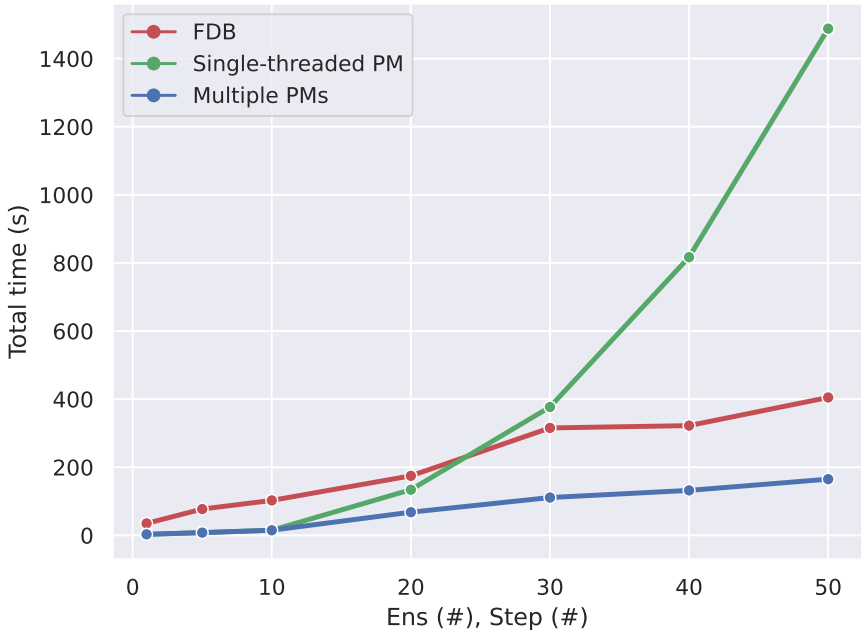



Figure 9: *Experiment 3* Total execution time for workflows based on FDB and the Maestro middleware. The Maestro-enabled workflow is run with either a single pool-manager or with multiple pool managers, where there is one pool manager for every 10 producer-consumer pairs.

no post-processing work takes place. This configuration is commonly used in I/O benchmarks. Furthermore, for the sake of fair comparison, we run the workflows with the following setup.

- Each of the producers, consumers and the Maestro pool manager runs on separate node.
- Each producer creates 1223 fields of size 804KiB per step, amounting to a total of around 960MiB data per step.
- The number of steps and number of ensemble members (producers) is identical for any given experiment in both the fdb-based and the Maestro-enabled workflow. This will guarantee that the number of consumers is the same between the two workflows, even though one consumes data per step while the other consumes data per ensemble member. This configuration also ensures that the same amount of data will be produced by one producer as consumed by one consumer.
- The number of workers per consumer job is fixed. The fdb-based workflow is multi-processed while the Maestro-enabled workflow is multi-threaded, and we need do ensure the same number of CPU cores are dedicated to workers in both workflows. Workers do the majority of the processing, including retrieving data from the middleware.

Figure 9 shows the total workflow execution time of the two workflows and for a range of different number of ensemble members (producers), varying between one and 50. Up to 20 produce-consumer pairs, the Maestro middleware shows clear signs of performance benefits even with just one pool manager. Beyond that point, however, the single-threaded pool-manager seems saturated with the large number of requests, which in turn slows down the entire Maestro-enabled workflow below that of the respective fdb-based one.

On the other hand, when multiple pool-managers are used, the performance benefits of the Maestro-enabled workflow are maintained all the way to 50 ensemble members. There is a significant gain over the fdb-based workflow and the workflow also scales linearly, which is expected given that the number of steps are increased at the same rate as the number of producers.

### 2.6.4 Experiment 4 – scalability with increasing steps and data-object sizes

Experiments 1 and 2 show that there is no performance difference between the cases when we are only carrying out the metadata manipulations necessary for the data transfer and when we are also executing the data transfer. That is because our current model workflow (based on our current operational ensemble forecast) consists of a large number of relatively small (804KiB) data objects. However, we are also interested in much larger data objects as we anticipate increases in the resolution of our ensemble system in the coming years. Figure 10 shows the Maestro-enabled workflow times of a set of ten-producer, ten-consumer experiments, and for three different data-object sizes: 804KiB, 4.4MiB and 15MiB. These sizes represent a typical field from the current operational ensemble forecast and two from possible future ensemble forecasts with increased resolution. We are, again, executing dry-runs, where the post-processing step is omitted, once the data is delivered to the consumer.

Crucially, we observe linear scaling with step sizes and no significant differences with object sizes. This is a significant result that suggests that the Maestro-enabled workflow is capable of transferring data between the workflow components at a much higher rate than the fdb-based workflow. So much so that the cost of the data transfer is still negligible compared with the metadata communications in all three cases.

The achieved transfer rate seen from the application averages at around 700MiB/s measured by a single worker; it peaks at 1.12GiB/s per worker for 15MiB-sized objects. The transfer rate per node may be higher depending on the extent to which data transfer is serialised at the network interface. With 15MiB data objects the total transfer rate is around 4.47GiB/s per node. For a workflow of 50 producers, 50 consumers and 5 PMs that amounts to an aggregate total of 220GiB/s.

Another relevant observation is that at a relatively small workflow of 10 producer-consumer pairs can handle at least 3500 objects per second at a sustained rate. This is highly encouraging for workflows that can make use of

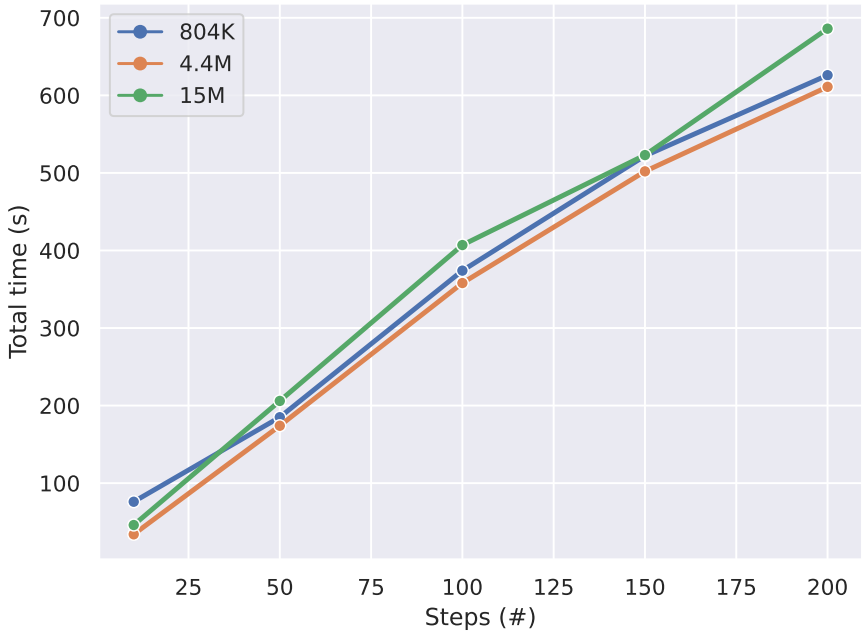



Figure 10: *Experiment 4* Total execution time with increasing number of steps and for data-object sizes 804KiB, 4.4MiB and 15MiB. 10 producers, 10 consumes and 1 single-threaded PM are used. The number of CDOs per step is constant across all experiments.

multiple pool managers, and for the Maestro development of multi-threaded and distributed pool managers.

# 3 Computational Fluid Dynamics plus in-situ analysis

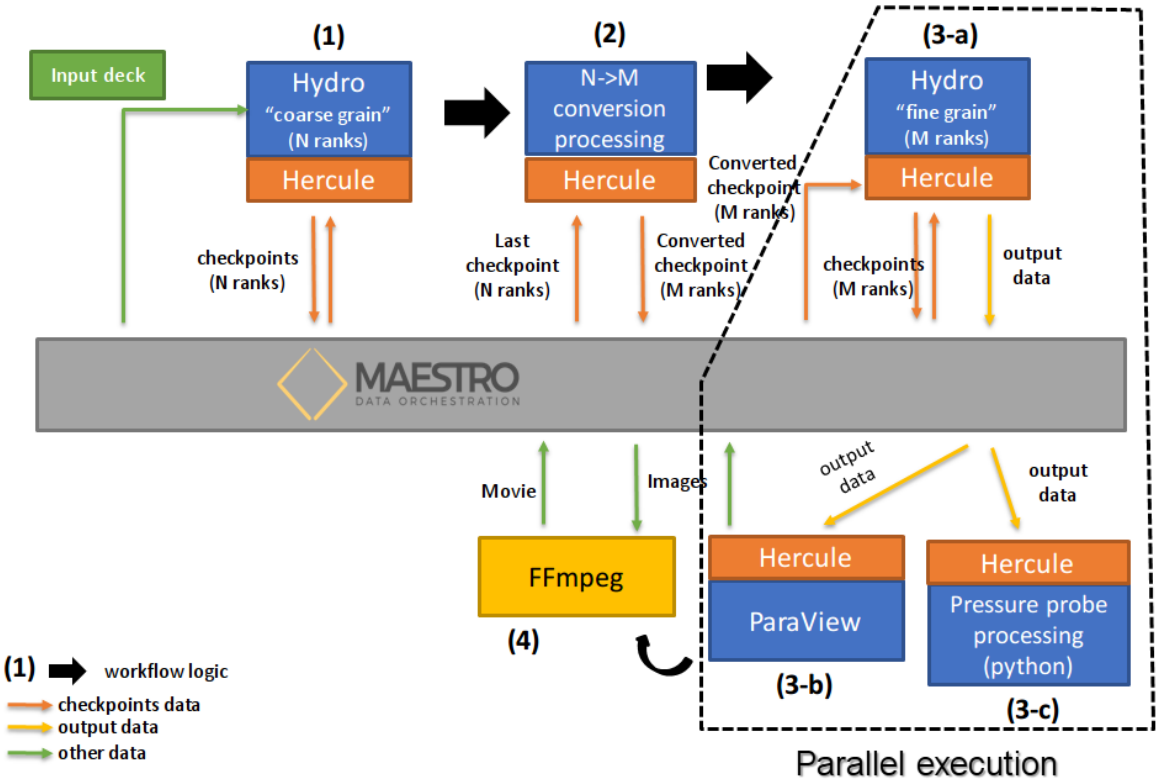
## 3.1 Introduction

The CEA workflow provided for Maestro is representative of a typical chaining scenario of two simulation codes. Chaining of simulation codes is often used to model different physics and/or scales, with the output of the first code being used as the input of the second with an intermediate data transformation/preparation step and a final post-processing step.

Due to the nature of CEA's activities, our actual simulation codes cannot be publicly disseminated. As a substitute we will use the open-source proxy application HydroC, a 2D hydrodynamic simulation code, to model our two simulation codes. The chaining mechanism will be represented by a restart of the simulation with a different domain decomposition, the checkpointing data serving as data exchange between the "two" codes. Two post-processing pipelines are then run consuming HydroC output data: 1/ a custom analytical processing written in python, 2/ a graphical processing with ParaView and

FFmpeg to produce a movie of the simulation. HydroC uses the CEA Hercule I/O library in order to produce post-processing outputs and checkpoint/restart.

The Maestro-enabled version will consist of modifying the Hercule I/O library to allow using Maestro CDOs to produce and/or consume simulation data with the primary objective of executing the processing steps, no longer *post-*, but *in situ* in a loosely coupled fashion (also called *in transit*). In other words, it consists of executing in parallel steps 3-a), 3-b) and 3-c), with simulation output data "streamed" from the simulation to the two processing. This is shown in the figure below:




## 3.2 Maestro features

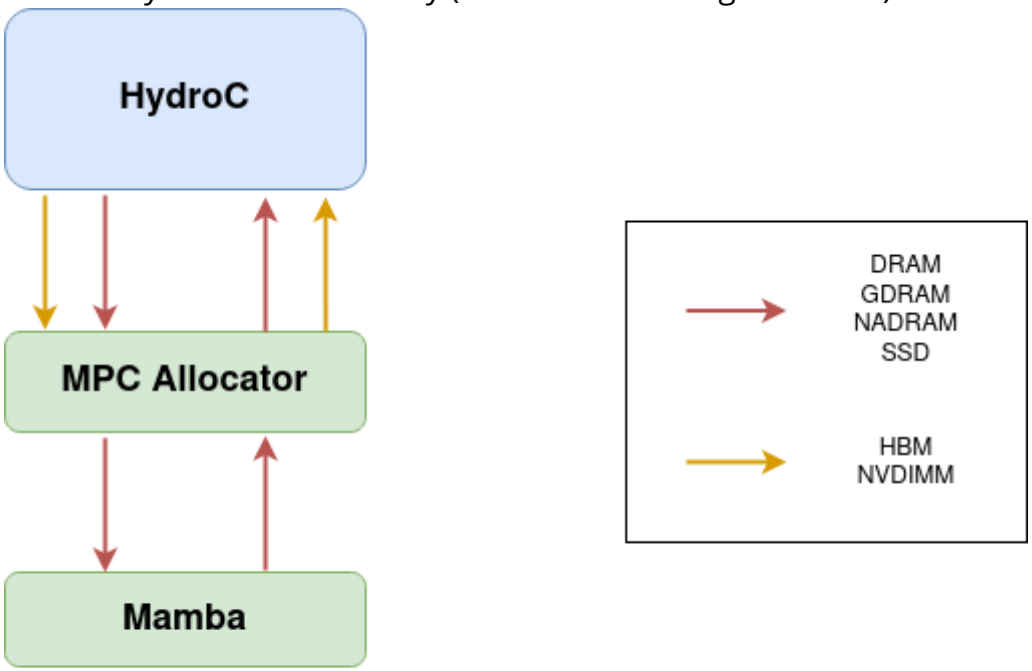
### 3.2.1 Attribute-based memory choice

The Mamba[3] library is providing an abstract memory model which allows the programmer to transparently allocate, access, and transport data across heterogeneous memory tiers. In the case of the Data-Aware Runtime Demonstrator, the hydrodynamic application HydroC is parallelized with MPC-Halos. MPC-Halo is implemented on the top of the Maestro-core so that the application processes are communicating with Maestro CDOs.

The MPC component responsible for the allocation of the simulation data is called MPC Allocator. The idea is to benefit from the Mamba library to the extent of the handled memory layers: HBM and NVDIMM allocations will

[3]Mamba has been developed in the partner project Epirgam-HS

be performed by the MPC Allocator and DRAM, GDRAM, NVRAM, and SSD allocations by the Mamba library (as shown in the figure below).



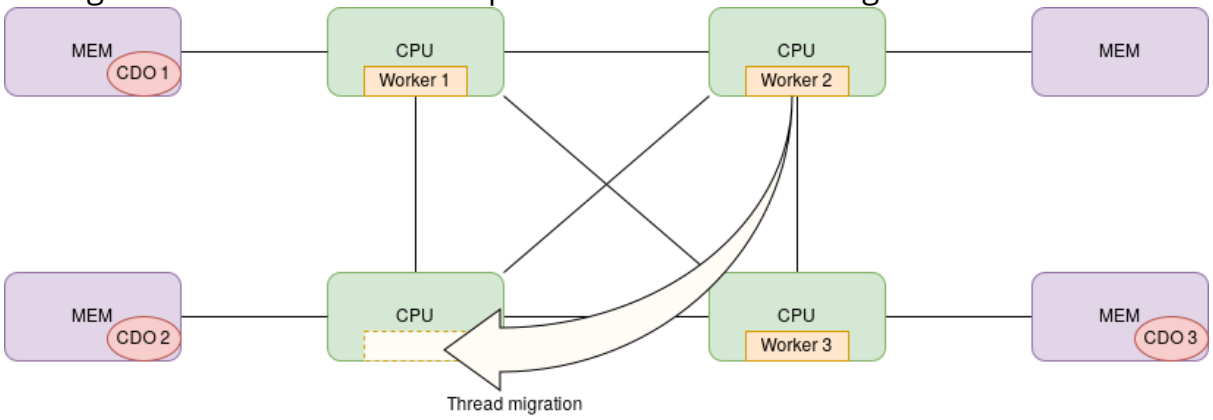

**Memory allocation process**

### 3.2.2 Data-aware task migration

This feature aims to migrate threads closer to the memory. To do so, 2 steps are performed:

1. Get the NUMA domain for each CDO
2. Thread worker should be pinned to cores within the same NUMA domain as the CDO

The figure below shows an example of data-aware task migration.



### 3.2.3 Guided I/O

There are 2 kinds of I/O in the demonstrator: the checkpoint I/O and output I/O. On the one hand, the priority is very low for checkpoint I/O since it is only used to restart the demonstrator after a system crash. Yet, on the other hand, output I/O is used at every simulation step for the data post-processing.

Guided I/O allows us to prioritize outputs over checkpoints and optimize the global performances of the whole demonstrator.

### 3.2.4 CDO data wrapping

The demonstrator will use our I/O library, Hercule, to pass simulation data to/from Maestro. This library attaches a series of attributes to data arrays including, dimensionality, domain-specific semantic information, and relationships between arrays.

User-defined attributes are used to generate custom CDOs that contain the metadata needed by Hercule.

## 3.3 Middleware requirements and current status

The table 26 shows the baseline performances at the beginning of the project. No major algorithmic changes that affect performances have been performed since the D2.4 deliverable. Nevertheless, the Inti system has changed and now performances are slightly better.

| **System** | **Inti[4]** | | | **Sage[5]** | | |
|---|---|---|---|---|---|---|
| **Test case** | 2000x2000 ; time (end) = 5 | | | | | |
| **Nodes** | 1 | 5 | 10 | 1 | 3 | 5 |
| **Total cores** | 32 | 180 | 320 | 8 | 24 | 40 |
| **Time (sec)** | 150.031 | 59.654 | 49.11 | 163.24 | 83.17 | 73.14 |
| **Cycles (MCells/s)** | 31.41 | 104.64 | 159.18 | 6.07 | 16.10 | 20.36 |

Table 15: Execution time of Computational Fluid Dynamics plus in-situ analysis Baseline

This is the list of the requirements provided in the deliverable “D2.3 Middleware Requirements and API Design”.

| **Reference in D2.3** | **R1.21** |
|---|---|
| **Description** | Maestro must allow a conceptual data organization based on the following data container concepts: |
| **Validation** | Most simulation codes and tools at CEA use the in-house Hercule library to perform IO-related tasks. To minimize the effort to use Maestro, we intend to port Hercule on top of Maestro (using Maestro as a backend store). This is a fundamental organization of data within Hercule. |

| | |
|---|---|
| **Evaluation status** | *Partially validated.* Records and collections are effectively supported in the Maestro middleware with the use of groups. However, the ability to handle multi-dimensional arrays is still to be validated. |

| **Reference in D2.3** | **R2.2** |
|---|---|
| **Description** | Maestro must provide the ability to add (and query) named attributes (metadata) to data arrays, records, and collections, where these terms are defined in R2.1. |
| **Validation** | As the Hercule I/O library (which will use Maestro) manipulates data objects but also groups of data objects, the ability to use named attributes on data objects and groups of data objects will facilitate it. |
| **Evaluation status** | *Not validated.* |

| **Reference in D2.3** | **R2.3** |
|---|---|
| **Description** | Maestro should provide a way to hierarchically group related data arrays within a record (and query this group structure), similar to HDF5 groups. |
| **Validation** | This will assist in porting the Hercule I/O library on top of Maestro (though this hierarchical organization could be implemented in the data naming, having a native system for hierarchical grouping would be more efficient). |
| **Evaluation status** | *Not validated.* |

| **Reference in D2.3** | **R2.4** |
|---|---|
| **Description** | A data collection must accommodate records produced by multiple independent producers (see R2.1 for the definition of 'collection' and 'records'). Hence, the labelling of data records is at least a 2-tuple made of a sequence integer (the sequence number of a record with the list of records) and a producer ID integer. Maestro must provide an indexing or attribute system to be able to support this. |

| | |
|---|---|
| **Validation** | When one of the simulations writes data with the Hercule I/O library, each rank produces a record that is self-describing and contains all data and metadata produced by that rank. Unlike in other I/O libraries, these records are not combined to produce a global view of the domain (or a different domain decomposition). They are written as-is (keeping the same domain decomposition). The recombination into a different domain decomposition is deferred until reading if needed. This is the default behaviour and is part of optimisation for writing.<br>This principle implies that records are indexed by their issuing sequence (here the time step number) and by the producer id (here the MPI rank). Therefore, to allow porting Hercule to leverage Maestro, it must provide an indexing or attribute system compatible with this approach. |
| **Evaluation status** | *Partially validated.* Records seem to be quite well accommodated within groups. Yet, validation is still to be performed. |

| **Reference in D2.3** | **R2.5** |
|---|---|
| **Description** | In 'streaming' mode, a data collection should accommodate multiple consumers organized into groups, similar to Kafka groups (or Redis Stream consumer groups). Records are broadcasted to all consumer groups. Consumers within a consumer group retrieve records from a queue. |
| **Validation** | This allows mixing two types of pattern: |
| **Evaluation status** | *Partially validated.* Records are broadcast to all consumers. A queue is set when multiple consumers are retrieving the same record simultaneously. However, groups of consumers are not yet validated. |

| **Reference in D2.3** | **R2.6** |
|---|---|
| **Description** | Maestro should provide a configuration system to describe properties of the workflow/pipelines for which it will manage the data exchange, where these properties are known in advance. |

| | |
|---|---|
| **Validation** | An example property that could be leveraged by Maestro at configuration time would be the identifiers of data that are known to be consumed by a consumer application. Let's say a simulation code dumps many physical quantities, but for a particular workflow, the consumer application only requires a few of them. This is a piece of information that Maestro could leverage to avoid transporting data from the producer application that will not be consumed at all. |
| **Evaluation status** | *Validated.* The Maestro middleware's support for workflows is appropriate. For each workflow, a pool manager may be started to handle records separately. |

| **Reference in D2.3** | **R2.7** |
|---|---|
| **Description** | In 'streaming' mode, a consumer should have the opportunity to declare in advance (in configuration) the data arrays it will consume. |
| **Validation** | In CEA's simulation pipelines, upstream simulation codes produce many arrays that are not necessarily consumed by downstream consuming applications. The arrays consumed is information that is often available ahead of time (e.g. the user knows that its consuming application only reads pressure fields). Maestro could leverage this knowledge to optimise data flow (filtering) between producer and consumer applications. |
| **Evaluation status** | *Validated.* Consumers have the opportunity to require records and post a demand request once the data is effectively needed. |

| **Reference in D2.3** | **R2.8** |
|---|---|
| **Description** | Maestro must provide the ability to persist indefinitely a data object provided that one of the storage tiers managed by Maestro is non-volatile. |
| **Validation** | Required if Maestro is to replace an existing I/O stack. |
| **Evaluation status** | *Not validated.* Persistency is not yet supported in the current Maestro development version. |

| **Reference in D2.3** | **R2.9** |
|---|---|

| | |
|---|---|
| **Description** | Relates to R2.8<br>When used for persisting data produced by an application, Maestro should provide the ability to configure which group of data objects is persisted. |
| **Validation** | An application may produce different data collections serving different purposes. The user should be able to instruct Maestro which collection needs to be persisted. |
| **Evaluation status** | *Not validated.* Relates to R2.8 |

| **Reference in D2.3** | **R2.10** |
|---|---|
| **Description** | Relates to R2.8.<br>When used for persisting data produced by an application, Maestro should provide the ability to specify a subsampling rate of persistence or filter. |
| **Validation** | This allows different branches of a pipeline to work on different data flow rates. In particular, the main simulation branch executed 'in transit' can work on a higher data flow rate to have refined data processing. |
| **Evaluation status** | *Not validated.* Relates to R2.8 |

## 3.4 Performance results

The table 26 shows the performances of the workflow without Maestro. We realized experimentations on Inti: a platform composed of dual-socket Intel(R) Xeon(R) CPU E5-2698 v3 (Haswell) nodes, each with 16 cores at 2.3 GHz, equipped with Mellanox MT27600 (Connect-IB) InfiniBand boards. Has the Inti system has changed during the project, we decided to re-run all the test cases to have a fair performance comparison.

| Test | A | B | C |
|---|---|---|---|
| Nodes | 1 | 3 | 5 |
| Total cores | 32 | 96 | 160 |
| Time (sec) | 47.863 | 30.096 | 28.302 |
| Cycles (MCells/sec) | 44.89 | 107.46 | 165.89 |

Table 26: Execution time of Computational Fluid Dynamics plus in-situ analysis without Maestro

The same tests have been performed with the Maestro enabled version (last version at the end of the project). The results are summarized in table 27

As we can see in the tables, the overall execution time with the Maestro enabled version is faster than the execution time without Maestro. Never-

| Test | A | B | C |
|---|---|---|---|
| Nodes | 1 | 3 | 5 |
| Total cores | 32 | 96 | 160 |
| Time (sec) | 30.661 | 10.368 | 6.245 |
| Cycles (MCells/sec) | 27.56 | 84.08 | 143.91 |

Table 27: Execution time of Computational Fluid Dynamics plus in-situ analysis with Maestro

theless, we can also see that the throughput (cells per time) is lower with the Maestro enabled version. The throughput is calculated using the execution of the overall HydrocC application. Thus, the throughput is impacted by the performances of the Hercule library. Due to a lack of time, the Hercule library is less efficient with Maestro than the original one without Maestro. We still have to optimize the critical path for the Hercule + Maestro version.

## 3.5 Conclusion

During the Maestro project, we were able to optimize the overall execution of the Computational Fluid Dynamics plus in-situ analysis benchmark. This benchmark is representative of real workflows at CEA. The gain is significant on a small number of nodes (36% with 1 node) and even larger when the number of nodes increases (78% with 5 nodes). The gains are obtained in the data management between codes. Regarding the execution of the simulation code, there are still some optimizations to do in the Hercule library to improve Maestro core usage.

# 4 Global Earth Modelling system TerrSysMP

## 4.1 Introduction

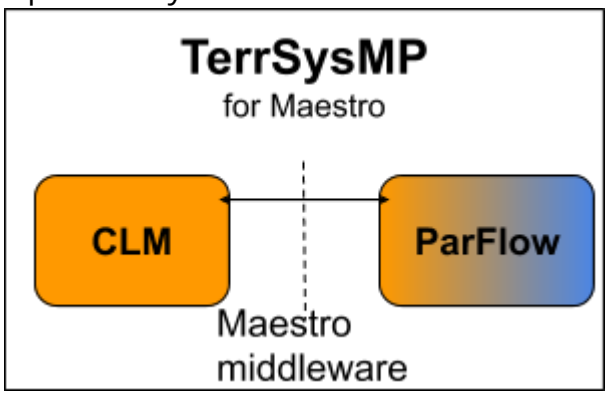
As described in D2.1, the Terrestrial Systems Modelling Platform (TerrSysMP or TSMP) targets the simulation of interactions between lateral flow processes in river basins and lower atmospheric layers. This is achieved by combining three model components COSMO, CLM, and Parflow. The coupling is done using OASIS3-MCT which is an external coupler. The COSMO-model is a non-hydrostatic limited-area atmospheric model developed by the COnsortium for Small-scale MOdelling. The Community Land Model (CLM) is used to study the effect of terrestrial ecosystems on climate. Finally, Parflow (PARallel FLOW) is an integrated hydrology model used to simulate surface and subsurface flow. To couple all three models OASIS3-MCT is used, which is a fully parallel implementation for coupling field re-gridding and exchange.

## 4.2 Maestro features

Given the highly optimized and fine-tuned nature of the OASIS-MCT3 coupler, we do not expect any performance benefit from porting the application to Maestro. The objective instead is to show the suitability of Maestro's general approach as a coupler for the applications which currently used a specialized coupler. As such, the objective is to at least achieve comparable performance with the OASIS-MCT3 coupler. Apart from that, the use of Maestro removes the requirement to write an application specific coupler for the use case of the application. This is described in more detail in the Results section.

### 4.2.1 Transparent Transformation of 2D Fields

As part of the effort done in WP2, we have identified two possible opportunities for leveraging Maestro in TerrSysMP. First, we consider the opportunity of exchanging 2D fields between climate models. For demonstration purposes, we will focus only on the interaction of two models, specifically CLM and ParFlow. The target is to replace OASIS3-MCT as a coupler in the process of exchanging the 2D fields. The fields need to be transferred, remapped, and interpolated by the Maestro middleware.




## 4.3 Covered Middleware Requirement and Current Status

| **Reference in D2.3** | **R4.1** |
|---|---|
| **Description** | Redistribution of Parallel Fields |
| **Validation** | Given a function which takes in data from multiple MPI Ranks, and outputs data to multiple MPI Ranks, a redistribution schema allows for abstracting away the redsitribution logic. |

| | |
|---|---|
| **Evaluation status** | Partially Validated (The data members which are transferred through OASIS coupler have been ported to use Maestro CDOs. In that case, in terms of programming design, it was a better design to do a point to point communication, instead of a gather and scatter paradigm. For a more complete validation, there needs to be support for Distributed CDOs in Maestro. Distributed CDOs support in Maestro will allow sections of CDOs to be marked with the destination node of the CDOs. This will reduce ) |

| **Reference in D2.3** | **R4.2** |
|---|---|
| **Description** | Row and Column-Major Layout Transformation of the Array |
| **Validation** | Required for functional correctness. |
| **Evaluation status** | Validated (Maestro CDO Attributes make the transformation automatic without any special attention needed from the programmer.) |

## 4.4 Results

There are two modes along with the evaluation of the Maestro port of TerrSysMP is expected to be done:

1. Runtime Performance
2. Programming Effort

### 4.4.1 Runtime Performance

In terms of performance, we can see that the OASIS and the Maestro version perform similarly.

1. OASIS version : 30 seconds mean (min: 28, max 45)
2. Maestro version: 34 seconds mean (min: 30, max 40)

### 4.4.2 Programming Effort

We have to make some assumptions in order to compare the programming effort of using TerrSysMP with Oasis Coupler as against using TerrSysMP with Maestro coupler. In particular, we make the following assumptions:

1. For the baseline, we assume that the codebases of Parflow, CLM, Oasis and Maestro are given to us, and that no programming effort is involved in obtaining that.

2. We qualitatively evaluate the changes that we have to make in the code in order to make a functioning port of TerrSysMP with both Maestro and Oasis Coupler.

Given these assumptions, we have the following advantages of Maestro in comparison with the OASIS-MCT3 coupler

1. Maintainability

   As described earlier, the use of Maestro removes the requirement to write an application specific coupler for the use case of the application. This is described in more detail in the Results section. For example, the TerrSysMP software is currently implemented by patching the component application with the patching code. A separate set of patches has to be maintained for each separate use case. This can often result in bugs, and different use cases falling out of sync with each other. Currently, there are at least three different use cases that have to be kept in sync manually by manually reimplementing patches from one use case into another.

   With Maestro, only the component applications have to be ported to use Maestro, and the coupler part is not needed to be implemented. Furthermore, the Maestro-enabled port of the application can be used in a standalone manner as well, without any loss of either functionality or performance. This allows the coupling code to be incorporated into the upstream, and therefore reduce the maintenance burden of the coupling code.

2. Debuggability

   On Oasis MCT3 coupler based TerrSysMP, the data transfer between Parflow and CLM is assymetric. This is reflected in the code where majority of data send and receive code is mostly written in Parflow. This also affects the execution flow of TerrSysMP with CLM's execution flow being dependent on Parflow's execution flow.

   In contrast, the data sending/receiving code for Maestro based implementations are symmetric. This also separates the execution flow of the two applications. During development, a major advantage of this method is that the applications can now be developed and tested in isolation from each other by using pre-generated data from the other applications and feeding it into the Maestro pool manager.

3. Amount of Glue Code

   The loosely coupled execution flow allows means that the implementation requires less glue code. Due to the intrusive nature of the OASIS-MCT3 coupler, the amount of glue code needed to be maintained in patches increases significantly. More specifically, the glue code contains not just the code required to transfer the data but also coupler specific code which cannot be up streamed. Specifically, the glue code needs to

define special buffers to send and receive data and convert them into the form which can then be consumed by the component application.In Maestro, this code can be up streamed into the component application, and the need for special buffers go away.

# 5 Electronic structure calculation library SIRIUS

## 5.1 Introduction

The DFT Loop mini-app is part of the SIRIUS package, a domain-specific library for electronic structure codes developed at ETHZ/CSCS. This mini-app is used to benchmark all components of the DFT self-consistency cycle – diagonalization of the Kohn-Sham Hamiltonian, charge density construction, mixing and generation of the effective potential – using a pseudopotential or full-potential method. The diagonalization-based methods used in those core algorithms rely heavily on linear algebra operations. They are usually compute-intensive and thus are the perfect candidates for GPU acceleration. However, the matrices involved in the calculation are usually large and do not always fit into the memory of a device. This implies significant data movements between CPU and GPU memories.

## 5.2 Maestro features

The Maestro-enabled version of our workflow will impact the low-level layer of the SIRIUS library. Maestro will be a substitute for memory management and the memory_pool classes. Chunks of data will be encapsulated into a CDO (Core Data Object) and given to Maestro. The GPU component will ask ownership of the data either using Maestro pool operations or Maestro's transformation API and Mamba, process it, and give it back to Maestro and SIRIUS.

### 5.2.1 Move data from CPU to GPU memory

When the data cannot fit into GPU memory, allocation/deallocation is currently manually done by blocks, meaning that the memory management is fully controlled by the developer. This memory management is implemented in a dedicated library within SIRIUS. The data movements are synchronous. A GPU kernel cannot start until all the data is available on the device. It is also important to note that the source of the data is not in use during the transfer. In addition, the host memory always stores the correct and valid data arrays.

This use case goal is to replace the smartness of moving data created by a developer of SIRIUS with Maestro's capabilities to handle CPU-GPU data movements using a generic abstraction. In that way, it simplifies the code of SIRIUS or any other applications using Maestro and enables such applications

to delegate the smartness and portability among different GPU hardware of data movements to Maestro.

In terms of workflow, SIRIUS creates the initial dataset in the host memory and gives the dataset ownership to Maestro. Later on, SIRIUS requests such data to Maestro in the GPU memory. Maestro is responsible to manage the transfer of the data from the CPU to the GPU to fulfil the request. Once the data is on the GPU, the ownership is transferred back to SIRIUS.

### 5.2.2 Management of memory resources

Copying data to the GPU memory for computation implies repeatedly allocating and deallocating GPU memory. The same behaviour is also observed on host memory. As the frequency of allocation and deallocation is correlated to the number of steps required for computation like described in UC3.1, a performance overhead is observed when the number of memory operations performed on both memories is high. A memory pool where allocated memory spaces are re-used is necessary to mitigate this overhead. Maestro should provide a similar capability.

## 5.3 Middleware Requirement and Current Status

| **Reference in D2.1** | **R3.1** |
|---|---|
| **Description** | Grant ownership of data to Maestro |
| **Validation** | Data allocated through the mdarray library will be "given" to Maestro that will get ownership on it. That way, Maestro will be able to take data movement decisions from host to GPU memory or the opposite way. |
| **Status/feedback** | *Validated*, ownership is managed by means of wrapper pointer types provided by Mamba. |
| **Evaluation metric** | *Semi-automatic data partitioning.* |

| **Reference in D2.1** | **R3.2** |
|---|---|
| **Description** | Indicate Maestro to move or allocate data to a particular tier of memory. |
| **Validation** | While giving ownership of data to Maestro, the calling process (host) implicitly or explicitly requests an allocation on the GPU to move data. |
| **Status/feedback** | *Validated*, data movement and allocation on a particular memory tier is handled by means of specific Mamba functions. |
| **Evaluation metric** | *Semi-automatic data partitioning.* |

| Reference in D2.1 | R3.3 |
|---|---|
| Description | Require data owned by Maestro. |
| Validation | Both the CPU and the GPU will require data owned by Maestro. The GPU will ask for ownership to apply linear algebra algorithms. The CPU will retrieve ownership afterward. |
| Status/feedback | *Validated*, CPU and GPU data management functions are provided by Mamba to obtain data access on both memory layers. |
| Evaluation metric | *Semi-automatic data partitioning.* |

| Reference in D2.1 | R3.4 |
|---|---|
| Description | Maestro can manage memory resources (including allocation and deallocation) across multiple layers of the memory hierarchy. |
| Validation | Allocation and deallocation will be carried out by Maestro on both sides, CPU and GPU memories. |
| Status/feedback | *Validated*, memory allocation and deallocation is fully managed by Mamba on both memory layers. |
| Evaluation metric | *Automatic memory pool management* |

| Reference in D2.1 | R3.6 |
|---|---|
| Description | Maestro can move data from one memory space to another |
| Validation | Maestro will have to move data from the CPU memory to the GPU high-bandwidth memory for computation. |
| Status/feedback | *Validated*, Mamba provides a data tile duplicate function for CPU to GPU data transfer management. |
| Evaluation metric | *Semi-automatic data partitioning.* |

| Reference in D2.1 | R3.7 |
|---|---|
| Description | Maestro can manage different memory spaces within a node |
| Validation | In our demonstrator, Maestro will manage two types of memory: the CPU memory and the high-bandwidth memory available on the accelerator. |
| Status/feedback | *Validated*, data is managed on both memory spaces by means of Mamba functions. |
| Evaluation metric | *Automatic memory pool management* |

In order to test the discussed capabilities of the Maestro-enabled workflow, a DFT Loop mini-app simulating the data movement involved in Sirius computations has been implemented. In the mini-app, all the basic memory management and computational operations performed in a SIRIUS calcula-

tion relevant for this project are performed:

- CPU matrix and vector memory allocation
- Blocking of these allocations to make them fit in GPU memory
- Loop over the blocks and transfer to GPU memory
- CUDA kernel execution (matrix-vector internal product)
- Transfer of the result to CPU

All the memory management operations of the above list are manually performed in the mini-app. Particularly, this means that the developer is responsible for computing the correct data index ranges for matrix and vector blocks, taking into account data strides and incomplete blocks, and for the calls to CUDA API for data transfer. By means of the Mamba library, an updated version of the mini-app has been developed: thanks to the APIs exposed by the library, all the manual memory management operations have been moved to the library itself. This include the block construction (tiling, in Mamba language), loop over blocks and data transfers. Also the data access at CUDA kernel level is performed by making use of Mamba functions. While the resulting source code is larger, in terms of lines of code number, the porting effort is completely justified by simplification in developing some error prone code blocks. Moreover, by only changing a Mamba keyword identifying the so called GPU execution context, the mini-app has been successfully ported to an AMD accelerated system. The only other required code change, not related to Mamba, were of course the CUDA function calls and keywords that has been translated to HIP. As an example of the code resulting from the porting activity, the loops over the matrix and vector tiles is reported:

Original version

```
for ( block_id_row = 0; block_id_row < num_blocks_height; block_id_row++ )} {
    for ( block_id_col = 0; block_id_col < num_blocks_width; block_id_col++ ) {

        offset_mat = block_id_row * gpu_block_height * matrix_width +
                     block_id_col * gpu_block_width;
        offset_vector = block_id_col * gpu_block_width;

        if ( (block_id_row + 1) * gpu_block_height > matrix_height )
            current_block_height = matrix_height % gpu_block_height;
        else
            current_block_height = gpu_block_height;

        if ( (block_id_col + 1) * gpu_block_width > matrix_width )
            current_block_width = matrix_width % gpu_block_width;
        else
            current_block_width = gpu_block_width;

        sirius_generate_valence_gpu_kernel(matrix, matrix_width, offset_mat,
                                           current_block_height, current_block_width,
                                           vector, offset_vector);
    }
}
```

Mamba version

```
for (size_t ti = 0; ti < matrix_tiling_dims->d[0]; ++ti) {
    for (size_t tj = 0; tj < matrix_tiling_dims->d[1]; ++tj) {

        stat = mmb_index_set(vector_block_idx,tj);
        stat = mmb_tile_at(mba_vector, vector_block_idx, &vector_block);
        stat = mmb_index_set(matrix_block_idx, ti, tj);
        stat = mmb_tile_at(mba_matrix, matrix_block_idx, &matrix_block);

        mmbArrayTile *vector_gpu_block;
        stat = mmb_tile_duplicate(vector_block, gpu_interface, MMB_READ_WRITE,
                                  layout_vector_host, &vector_gpu_block);

        mmbArrayTile *matrix_gpu_block;
        stat = mmb_tile_duplicate(matrix_block, gpu_interface, MMB_READ_WRITE,
                                  layout_matrix_host, &matrix_gpu_block);

        sirius_generate_valence_cpu_kernel_mamba(matrix_gpu_block, vector_gpu_block);

        stat = mmb_tile_merge(matrix_gpu_block, MMB_OVERWRITE);
    }
}
```

## 5.4 Performance results

Performances are evaluated in terms of the following indicators:

1. Code implementation complexity
2. Portability
3. Number of memory allocations/transfers

Results concerning the first two parameters have been already discussed in the previous section, we will therefore focus on the memory management aspects. Focus has been given to the number and size of CPU/GPU allocations and to the number and size of CPU to GPU (and back) memory transfers. The result reported below have been measured using some specifically developed functions to count the number and size of the above operations, by means of Mamba statistics measurement functions and by means of the NVIDIA NSIGHT performance analysis tool. Measurements were done with the following input data:

- Matrix sizes: 100000x1000 (data type: double)
- Vector size: 1000
- Block sizes: 32x32
- Cuda block size: 32x32 threads
- Cuda grid size: 3125x32 blocks

| | Number (size) CPU allocations | Number (size) GPU allocations | Number (size) GPU to CPU transfers | Number (size) CPU to GPU transfers |
|---|---|---|---|---|
| Original RM | 100003 (2400 MB) | 200000 (825 MB) | 100000 (800 MB) | 200000 (825 MB) |
| Original CM | 100003 (2400 MB) | 100032 (800 MB) | 100000 (800 MB) | 100032 (800 MB) |
| Mamba RM | 200003 (1681.6 MB) | 400000 (906.6 MB) | 100000 (800 MB) | 400000 (906.6 MB) |
| Mamba CM | 100032 (1643.2 MB) | 200064 (843.2 MB) | 100000 (800 MB) | 200064 (843.2 MB) |

Table 30: SIRIUS mini-app memory profiling

The particular shape of the matrix, with a much larger number of rows with respect to the number of columns, has been selected to reflect the typical shape of the "tall and skinny" matrices characterizing the matrix-vector inner product calculations needed by SIRIUS. Results are reported for two different variants of the Mamba mini-app version: the loop over matrix blocks is composed by two nested loop, since matrix blocks are identified by a row and a column index pair. The two loops can be ordered in two different ways, row major and column major: the original mini-app implements a row-major ordering that while the Mamba version has been implemented in both variants. Loop reordering has been introduced in the original mini-app as well, for a fair comparison. The obtained results are reported in table 30, where RM and CM in the mini-app version rows identify the Row Major and Column Major variants respectively.

As can be seen from the above Table, the number of GPU allocations and CPU to GPU transfer are twice the number in a comparison between the original mini-app and the one ported to Mamba, if the same loop ordering is considered (i.e., Original RM compared to Mamba RM): the additional memory operations represents the overhead caused by the usage of the Mamba library. A similar increment in memory operations is present by comparing the RM and CM versions of the same mini-app implementations, in this case caused by the missed data reuse in the RM version for the vector elements, whose block set is transferred during each new row iteration to the GPU. GPU to CPU number and size of transfers is instead exactly the same in all cases since the only operation involving data transfer in that direction is the update of the output matrix and the same strategy is adopted in all mini-app version.

## 5.5 Conclusions

Porting of the Sirius mini-app, from the original manual memory management version to the one where this task is fully based on Mamba library usage, has shown some valuable advantages mostly related to code readability and maintainability: the total number of lines of code increased but, as shown in the previous section, the maintainability of the updated code has increased. Thanks to the functionalities provided by the Mamba library, all the data tiling operations are automatically managed and tile indexing calculations, which could be error prone, are not needed anymore. Code abstraction and memory access safety has increased as well, since developers can use iterators to access the tiles instead of dealing with index ranges. It must be observed that the adoption of the Mamba library does not affect code portability: the Mamba based mini-app has been ported to AMD GPU with just a small change in the code, corresponding to the request of an execution context for GPU memory registration. All the above advantages come at the cost of a small overhead in terms of performances that does not invalidate the advantages of the described approach.

# 6 Concluding Remarks

This document presents the final evaluation of the Maestro middleware from the applications' perspectives. Figure 11 shows the summary of the final requirement evaluation effort, as a breakdown per application partner. From the requirements and performance evaluation, we can draw the following final conclusions.

- The Maestro middleware is capable of handling complex metadata-driven workflows, primarily thanks to its dedicated pool manager, support for metadata attached to data object, and its subscription-based event-handling architecture.
- With an essentially single-threaded pool manager, the Maestro middleware has been able to outperform the existing, filesystem-based version of the IFS model workflow up to a 61-node (30 producers, 30 consumers, 1 pool manager) configuration.
- Using multiple pool managers, the Maestro middleware has outperformed the existing IFS model workflow up to a 105-node (50 producers, 50 consumers, 5 pool managers) configuration by a significant (around 2.5 times) margin. This was the largest workflow we have executed as part of application demonstration. However, the workflow indicates scaling that we would also expect to achieve with much more nodes and bigger workflows, as we move towards exascale computing.
- Using multiple pool managers, it is also possible to achieve 'perfect' scaling, in particular when one pool manager is used for every 20 client ap-

plications (10 producers, 10 consumers). However, this reduces the generality inherent in the Maestro design, so it used only to demonstrate the strong promise of a multithreaded or distributed implementation of the pool manager.

- The Maestro-enabled version of the CFD plus in-situ analysis benchmark has shown significant performance gains, reaching 78% with 160 cores on five nodes, over the original benchmark. The gains are mainly thanks to improved data management between code components.
- The use of the Maestro middleware removes the requirement to write an application-specific coupler. This means that the Maestro-enabled version reduces the amount of glue code needed for coupling applications and improves the ability to debug applications during development.
- The Maestro middleware has also been shown to improve code maintainability in a heterogeneous GPU/CPU environment as it takes over managing data tiling operations and increases code abstraction and safety of memory access.

# References

[1] D. Sármány, J. Capul, F. Tessier, S. El Sayed, J. Badwaik, and D. Pleiter, "*Maestro D2.4* Final Model Workflows," 2020.

[2] A. Bahra and S. Kertész, "ecFlow." `https://github.com/ecmwf/ecflow`.

[3] S. Smart, T. Quintino, and B. Raoult, "A high-performance distributed object-store for exascale numerical weather prediction and climate," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, pp. 1–11, 2019.

[4] D. Sármány, S. Smart, T. Nikolov, J. Capul, S. Morais, and F. Tessier, "*Maestro D2.1* Workload characterisation and Middleware Requirements," 2019.

[5] D. Sármány, S. Smart, J. Capul, F. Tessier, S. El Sayed, and D. Pleiter, "*Maestro D2.3* Middleware Requirements and API Design," 2020.

[6] C. Haine, U.-U. Haus, and A. Tate, "*Maestro D3.1* Initial Core Middleware Specification, API Document," 2019.

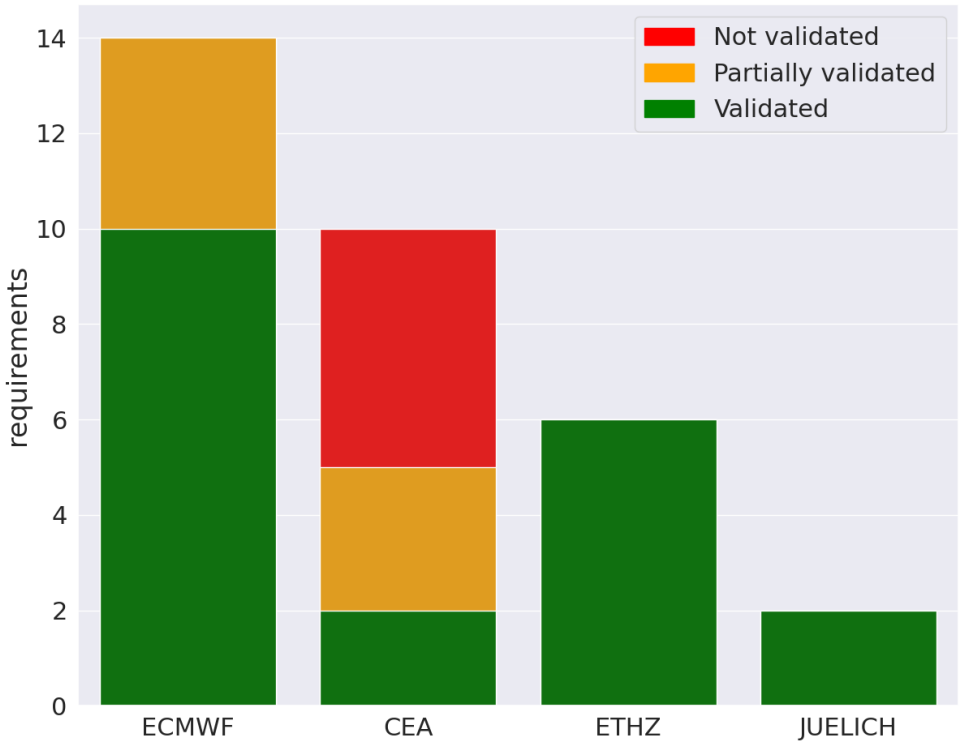



Figure 11: Number of validated, partially validated and unvalidated requirements per applications partners. The basis for this is the set of requirements defined by each application partner during the Maestro co-design effort, although not every requirement was planned to be demonstrated.