D6.4
System software demonstrators

Work package	**WP6 Middleware Validation and Systems Software Demonstrators**	
Author	Marc Pérache	CEA
Author	Javier Novo Rodríguez	Appentra
Author	Manuel Arenaz	Appentra
Author	Jayish Badwaik	JUELICH
Author	Sai Narasimhamurthy	Seagate
Author	Christopher Haine	HPE
Author	Maxime Martinasso	ETHZ
Author	Alejandro Dabin	ETHZ
Author	Christopher Bignamini	ETHZ
Reviewer #1	Maxime Martinasso	ETHZ
Reviewer #2	Dirk Pleiter	JUELICH
Dissemination Level	Public	
Nature	Report	

Date	**Author**	**Comments**	**Version**	**Status**
21.11.2021	Marc Pérache	Initial version	v0.1	Draft
29.11.2021	Marc Pérache	Final version	v1.0	Finale

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
CEA	The French Alternative Energies and Atomic Energy Commission (CEA)
CDO	Core Data Object, fundamental data unit understood by the Maestro middleware and communicated between the Maestro middleware and its applications
CSCS	Swiss National Supercomputing Centre
DSRP	Dynamic Storage Resource Provisioning
ECMWF	European Centre for Medium-Range Weather Forecasts
HPC	High-performance computing
HSM	Hierarchy Storage Management
IFS	Integrated Forecast System
Lustre	Lustre is a type of parallel distributed file system, generally used for large-scale cluster computing.
MIO	Maestro I/O interface
MPC	Multi-Processor Computing runtime (MPC) is a unified runtime targeting the MPI, OpenMP and PThread standards. One of its main specificity is the ability to run MPI Processes inside threads, being a thread-based MPI
MPI	Message Passing Interface
MPMD	Multiple Programs Multiple Data Streams
NUMA	Non-uniform memory access
NVME	Non-Volatile Memory Express: logical-device interface specification for accessing a computer's non-volatile storage media usually attached via PCI Express (PCIe) bus.
WP	Work Package

1 Introduction

This document describes the current status for each of the use-cases detailed in deliverable D6.1. The use-cases presented here are maintained by four Maestro partners:

- HPE: Maestro framework
- CEA: Data aware runtime
- JSC: Workflow execution and optimisation demonstrator
- APP: Intelligent workload management and data preloading/staging
- ETHZ: Dynamic Provisioning
- Seagate: Guided I/O in pre/post-processing

2 Maestro framework

2.1 Objective

The aim of this demonstrator is to show the basic features of Maestro Core are operational, in particular multi-threaded and multi-application aspects.

2.2 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 framework.

2.3 Demonstrator

All Maestro Core demonstrators are publicly available on the project website [1] and are all built and run using `make check`. In the following, `$(MAESTRO_PATH)` denotes the Maestro Core root directory.

2.3.1 Multi-threaded setup

As per D3.1 [1], Maestro Core works in a multi-threaded setup as demonstrated by the following example: `$(MAESTRO_PATH)/tests/run_demo.sh` The principle is a multi-threaded application where threads for producer, consumer, and archiver use Maestro inside one address space and passing data

[1] https://gitlab.jsc.fz-juelich.de/maestro/maestro-core

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, i.e. each thread can be either a producer, or a consumer, or an archiver.

2.3.2 Coupling multiple applications

Application coupling in Maestro is achieved with a Pool Manager component. Typically, a Maestro-enabled workflow consists in a simple Pool Manager application, and a set of producer/consumer/archiver applications. For Maestro Core level demonstration purposes, we use a minimal script doing what the workflow coordinator (see WP4) would do to manage the workflow. A typical example would be `$(MAESTRO_PATH)/tests/check_pm_interlock.sh`, where two applications exchange a CDO. A number of other examples are implementing a multi-application setup, another notable one would be the shell script `$(MAESTRO_PATH)/tests/check_subscribe.sh`, which features an injector component which offers CDOs, and an archiver component that acts as a sink. This especially is also a demonstration of how events monitoring works, which is one of the key features of Maestro Core.

2.3.3 Adaptive Transport

Although most use-cases typically rely on RDMA transport, Maestro features a set of other transport backends as detailed in D5.3 [2]. Another role of the `$(MAESTRO_PATH)/check_pm_interlock.sh` test is that it demonstrates different adaptive transport backends as well, including parallel filesystem (PFS) and object storage (MIO).

2.3.4 CDO Transformations

Performing layout transformations based on CDO attribute is also a duty of Maestro Core, which can happen either outside of the pool, we talk about *explicit* transformations, or inside of the pool, we then talk about *implicit* or automatic transformations. The latter happens when Maestro Core notices non-matching CDO layout attributes on offer and demand side. This would typically be the case for an application written in C transferring a CDO to/from an application written in Fortran, where data layouts differ. `$(MAESTRO_PATH)/tests/check_layout` demonstrator showcases both implicit and explicit layout transformations, between row-major and column-major layouts.

3 Data aware runtime

3.1 Objective

The demonstrator will showcase the 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 Description

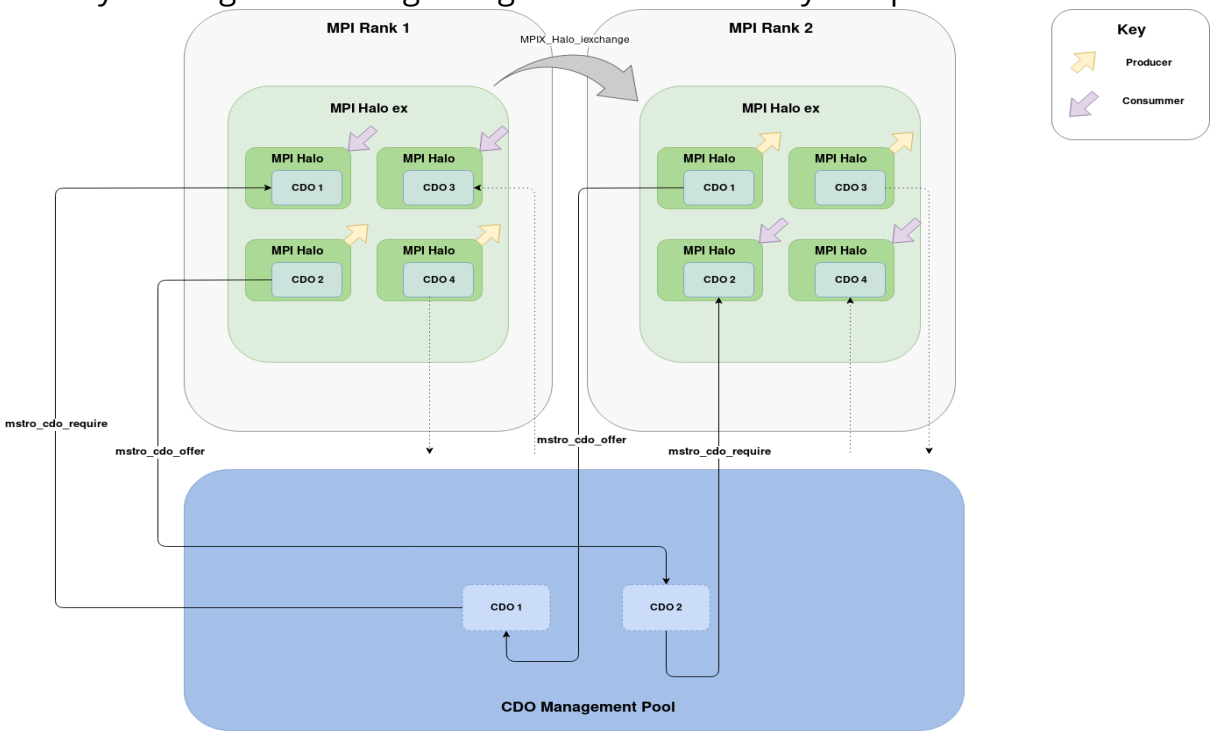
Usual simulation codes rely on domain decomposition to span the complete computation on multiple processes and multiple nodes. These domains must communicate information to have correct computations. These communications usually involve data at the boundaries of the domains. MPC [3] is an implementation of the MPI API, focusing on interactions with shared-memory models, and data locality. To alleviate shared-memory benefits for the data boundaries exchange, MPC proposes an MPI extension called MPI_Halo [4]. MPI_Halo is a feature in MPC allowing to encompass information about halo data-transfer in domain decomposition applications. MPI_Halo allows the exchange 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 lead to NUMA effects. According to the data access pattern, it might be more efficient to move data in a closer NUMA domain, avoiding NUMA effects. As the Maestro Core 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 thanks to 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 $\rightarrow$ mstro_core_init
- MPI_Finalize $\rightarrow$ mstro_core_finalize
- MPIX_Halo_cell_init $\rightarrow$ mstro_cdo_declare: initializes a halo-cell container with a given data-layout.
- MPIX_Halo_cell_set $\rightarrow$ mstro_cdo_attribute_set: sets the buffer pointer inside the MPI_Halo object (only possible on local MPI_Halo).
- MPIX_Halo_cell_get $\rightarrow$ 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.

3.3 Demonstrator

When in shared memory, MPI_Halo API will serve as a wrapper to Maestro. The demonstrator allows us to compare between existing shared memory implementation in MPI_Halo and new implementation based on CDOs. The goal is to demonstrate the possibility to use the Maestro model to improve an already existing runtime regarding data-related locality and performance.



MPI Halo exchange process

3.4 Performances results

We did a performance evaluation of the Hydro benchmark with MPI_Halo on top of Maestro. We realized experimentations on the Inti 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. Table 1 documents.

Nodes	1	1	1	1	2	5
Cores	4	8	16	32	64	160
Time (sec)	187.85	99.02	50.31	26.31	14.45	5.87
Speed (MCells/s)	4.37	8.29	16.31	31.23	59.26	151.06

Table 1: Execution time of HydroC with MPI_Halo on top of the Maestro framework

Compare to the original version MPI_Halo without Maestro, the performances are in the the range of the noise of the measurement with less than 5% of differences in the execution time.

4 Workflow Execution and Optimisation Demonstrator

4.1 Objective

The objective of this demonstrator is to show that the basic features of the workflow execution framework are in operation, in particular in complex workflows.

4.2 Description

The demonstrators will showcase the execution of complex workflows and show that the workflow execution framework can correctly execute the workflows given and perform the correct optimizations. The complete description of the workflow execution framework is done in the deliverable D4.5.

4.3 Demonstrator

We take a small demonstrator and show that the workflow execution framework can take an input of a simple workflow graph and automatically add the correct tasks to the graph to make the graph work correctly with the Maestro-enabled applications. We also show how the monitor application makes sure that the CDOs are not lost when the producer process for a CDO ends before the consumer process can request the CDO.

The demonstrator has the workflow graph in Figure 1. There, you can see different process and the workflow execution dependency between them. Once the graph is passed through the workflow execution engine, we get graph in Figure 2. In this graph, you can see the watchers have been added

to track the CDO lifetime thereby ensuring the correctness of the workflow. The cache workers jobs have also been added to make sure that the CDOs are cached on the correct nodes.

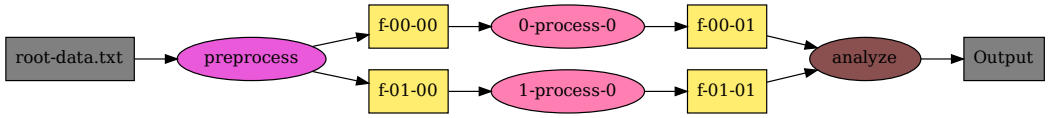


Figure 1: A simple workflow with a fork and join of tasks. The execution of each branch can only proceed when the *preprocess* step has completed and generated data for both branches of the pipeline.

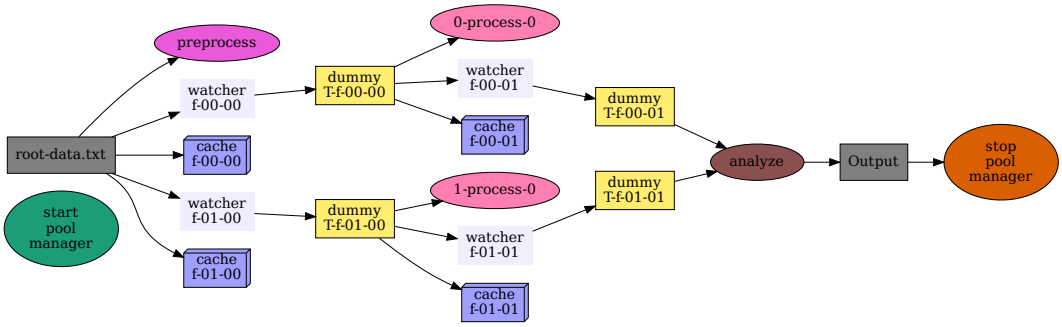


Figure 2: A simple workflow with maestro watcher and cache objects added. When watchers for each CDO signal that data is available for each branch, the execution can proceed independently on each branch. (Note that the 'dummy' file objects do not need to be generated, they exist solely as placeholders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

5 Intelligent workload management and data preloading/staging

5.1 Objective

The objective of this demonstrator is to showcase the *pwmaestro* tool for the analysis of the data access semantics of an application code. The demonstrator will exercise the static code analysis capabilities to extract information from the source code and then leverage such an information to facilitate the development of Maestro-enabled applications that make intelligent decisions about data movement and data placement.

5.2 Description

The demonstrators will showcase the *pwmaestro* command-line tool in the following two scenarios:

- **Identify non-consecutive array accesses in Montage using *pwmaestro***: Analyze the Montage project written in C/C++ to find memory-related issues that impact of the performance of the code. The *pwmaestro* tool will be used to identify non-consecutive array accesses that exhibit poor data locality at run-time. The demonstrator shows the summary report provided by *pwmaestro*, which enables the characterization of Montage from the point of view of an efficient usage of the memory.
- **Development of Maestro-enabled applications assisted by *pwmaestro***: Create a Maestro-enabled version of a sequential C/C++ code that manipulates a two-dimensional array using a column-major memory access pattern. The *pwmaestro* tool will be used to identify this type of non-consecutive array access and suggest using a row-major array access instead. It will also take advantage of the source code rewriting capabilities of *pwmaestro* to create a Maestro-enabled version by inserting the proper calls to the Maestro data transformation in order to change the data layout of two-dimensional arrays. The demonstrator also evaluates the performance of the Maestro transpose operation with respect to data movement code explicitly by the programmer.

5.3 Demonstrator

5.3.1 Identify non-consecutive array accesses in Montage using *pwmaestro*

The Montage code has been selected during the execution of the Project [5] as a use case for demonstrator purposes. In the scope of this demonstrator of *pwmaestro*, the *cfitsio* subsystem of Montage that provides simple high-level routines for reading and writing FITS (Flexible Image Transport System) data format files[2] was found to be of particular interest. Thus, all the directories and source files of *cfitsio* were scanned by *pwmaestro* in order to look for non-consecutive array accesses that exhibit poor data locality, and thus are candidates to take advantage of Maestro data transformations. The invocation of the tool and the corresponding output summary metrics are shown below:

```
$ pwmaestro --summary --config compile_commands.json  lib/src/cfitsio-3.25/*.c
pwmaestro: error: failed to analyze 'lib/src/cfitsio-3.25/vmsieee.c'

lib/src/cfitsio-3.25/vmsieee.c:1:10: fatal error: 'cvtdef.h' file not found
#include <cvtdef.h>
         ^~~~~~~~~~
1 error generated.
error: file was not processed correctly: 'lib/src/cfitsio-3.25/vmsieee.c'
```

[2] https://heasarc.gsfc.nasa.gov/fitsio/

```
pwmaestro: error: failed to analyze 'lib/src/cfitsio-3.25/windumpexts.c'

lib/src/cfitsio-3.25/windumpexts.c:24:10: fatal error: 'windows.h' file not found
#include <windows.h>
         ^~~~~~~~~~
1 error generated.
error: file was not processed correctly: 'lib/src/cfitsio-3.25/windumpexts.c'


KPI SUMMARY:

  Total array references:                                          35931
  Total number of memory-related issues:                           113
    MSTR001: use row-major instead of column-major array access      1
    MSTR002: avoid strided array access                             50
    MSTR003: avoid non-consecutive array access                     57
    MSTR004: avoid indirect array access                             5
```

A *compile_commands.json* file is passed to *pwmaestro* through the *–config* argument. As described in Section "3.3.5 Advanced analysis setup" of Deliverable "D4.4 Final Specification and Release of Software Tool"[6]), this file contains the compile commands to build the project, including the compiler flags that required to build the source code and to enable its analysis by *pwmaestro*. In order to generate *compile_commands.json* file, the *bear* tool was invoked from on the Montage folder root as follows: *bear make*. For more details see the advanced set up of *pwmaestro* in its user manual.

Two files could not be analyzed due to missing header files (e.g. *windows.h* which is not available in the Linux platform where *pwmaestro* was run). Files such as this, can be excluded from the analysis explicitly through the *–exclude* argument. Also note that the *pwmaestro –summary* flag is used to count the total number of memory-related issues in the *cfitsio* subsystem, which accounts for 113 out of 35931 array references found in the code. The breakdown shown by the tool reports 1 column-major access that could be rewritten as a row-major access using *pwmaestro*. Moreover, other 112 issues are reported corresponding to different types of non-consecutive array accesses, including 50 strided memory accesses and 5 indirect memory accesses.

The next step would be to invoke *pwmaestro –list* to get a detailed listing of the exact location of the memory-related issues along with a suggestion to use the *pwmaestro –rewrite* command to actually annotate the code with Maestro data transformation API calls. For the sake of brevity, let's focus on the issue *MSTR001* reporting the existence of a column-major array access. It is located in the *checksum.c* source file of *cfitsio*. The corresponding output of *pwmaestro –list* is as follows:

```
$ pwmaestro --list checksum.c
...
checksum.c:103:5: MSTR001: 'asc' multi-dimensional array not accessed in row-major
    order
  Use --rewrite to create a new version using Maestro transformations to enable row
      -major access:
    /home/javi/dev/maestro/pwmaestro/pwmaestro checksum.c:103:5 --rewrite -i
...
```

The use case found by *pwmaestro* consists of a column-major array access to *asc[4*jj+ii]* to a two-dimensional *4 x 4* array in the scope of two nested loops

for_ii and *for_jj*. The code uses a one-dimensional array *asc* of 16 elements.

The second demonstrator described in the following section shows a use case of *MSTR001* as well. Typically, HPC codes process two-dimensional arrays of size *M x N*, where the array size is very large and it grows with the problem size in order to provide more accurate solutions to the scientific and engineering problem. A synthetic code of this *M x N* use case will be used in the second demonstrator presented in the next section.

5.3.2 Development of Maestro-enabled applications assisted by *pwmaestro*

The input source code of *pwmaestro* used in this demonstrator is shown in the following listing, which computes the sum of all the elements of a bi-dimensional array in C/C++ language. The computation loop has a column-major array access that is not optimal since the data is stored in memory following a row-major layout.

Listing 1: Original *matrixSumJ.c* file with a column-major array access.

```
/****
MATRIX summatory Column-major access.
Dinamic arrays.
****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#ifndef M
    #define M 6
#endif
#ifndef N
    #define N 8
#endif


void print_matrix_info(const char *desc, int *array, int rows, int cols)
{
    printf("\n␣␣%s\n===============\n", desc);

    printf("Contiguous␣memory:␣");
    for (int i = 0; i < rows * cols; i++)
        printf("%d␣", array[i]);
    printf("\n");

    printf("\n%dx%d␣matrix:\n", rows, cols);
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
            printf("%d␣", array[i * cols + j]);
        printf("\n");
    }

    printf("\n");
}


double mysecond()
{
/* struct timeval { long        tv_sec;
            long        tv_usec;        };

struct timezone { int   tz_minuteswest;
```

```
            int         tz_dsttime;       };      */

        struct timeval tp;
        struct timezone tzp;
        int i;

        i = gettimeofday(&tp,&tzp);
        if (i == 0)
            return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
        else
        {
            fprintf(stderr, "Get time of day error\n");
            exit(1);
        }
}

int main()
{
    double t1, t2;
    int i, j;
    double sum; // Matrix summatory data output

    // Data declaration NxM
    int *A = (int *)malloc(M * N * sizeof(int)); // Source matrix

    // Fill the matrix with incremental values
    for (i = 0; i < M; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            A[(i*N)+ j] = (i * N) + j;
        }
    }

    sum = 0;

    // Data computation : Matrix summatory
    t1 = mysecond();
    for (j = 0; j < N; j++)
    {
        for (i = 0; i < M; i++)
        {
            sum += A[(i * N) + j];
        }
    }

    t2 = mysecond();

    #ifdef DEBUG
    print_matrix_info("Original", A, M, N);
    #endif

    free(A);

    printf("Matrix Summatory %18.2f\n", sum);

    printf( "\nBaseline Matrix dinamic summatory by columns time\n");
    printf( "Walltime, s:       %18.8lf\n", (t2 - t1)  );

    printf("\n");
}
```

Running *pwmaestro –list* for the previous code yields the following list of non-consecutive memory access patterns:

```
$ pwmaestro --list matrixSumJ.c
matrixSumJ.c:82:5: MSTR001: 'A' multi-dimensional array not accessed in row-major
    order
  Use --rewrite to create a new version using Maestro transformations to enable row
```

```
    -major access:
  pwmaestro matrixSumJ.c:82:5 --rewrite -i
```

The output shows that the tool reported a "MSTR001" issue highlighting the aforementioned inefficient column-major access. It also reports a suggestion on how to invoke the tool to generate a new version of the code that uses the Maestro API to transpose the matrix so that the matrix is accessed in row-major order. There are other solutions for this problem such as loop interchange but here we intend to demonstrate how to enable row-major access through a matrix transposition, with and without the Maestro framework in order to evaluate its performance compared to an explicit data transposition coded by the programmer.

In order to generate the Maestro-enabled version, we followed the previous suggestion, although using *-o maestroSumMaestro.c* to generate a new file:

```
$ pwmaestro matrixSumJ.c:82:5 --rewrite -o matrixSumMaestro.c
Rewriting matrixSumJ.c:82:5 with row-major access using Maestro to transpose the
    array 'A'
Successfully created matrixSumMaestro.c
```

The resulting *maestroSumMaestro.c* file is omitted for brevity. The following output of the *diff* command is provided instead to highlight the changes performed by *pwmaestro*:

Listing 2: Difference between the original and Maestro-enabled versions.

```
0a1,3
> #include "maestro.h"
> #include "transformation/transformation.h"
>
67c70,72
<     int *A = (int *)malloc(M * N * sizeof(int)); // Source matrix
---
>     int *A = (int *)malloc(M * N * sizeof(int));
>     int *At = (int *)malloc(6 * 8 * sizeof(int));
>      // Source matrix
81a87,132
>     mstro_init("small-matrix", "transpose", 0);
>
>     // Information common to both matrices
>     size_t A_elem_size = sizeof(int);
>     size_t A_size = 6 * 8 * A_elem_size;
>     size_t A_num_dims = 2;
>
>     mstro_cdo A_original_cdo = NULL;
>     mstro_cdo_declare("A_original", MSTRO_ATTR_DEFAULT, &A_original_cdo);
>
>     // Common attributes
>     mstro_cdo_attribute_set(A_original_cdo, MSTRO_ATTR_CORE_CDO_RAW_PTR, A, false
    );
>     mstro_cdo_attribute_set(A_original_cdo, MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE,
     &A_size, false);
>     mstro_cdo_attribute_set(A_original_cdo,
    MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &A_elem_size, false);
>     mstro_cdo_attribute_set(A_original_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &
    A_num_dims, false);
>
>     // Matrix A specific attributes
>     size_t A_dims_original[] = {6, 8};
>
```

```
>     mstro_cdo_attribute_set(A_original_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE,
     &A_dims_original, false);
>     size_t A_layout_original = MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER_COLMAJOR;
>     mstro_cdo_attribute_set(A_original_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER, &
    A_layout_original, false);
>     mstro_cdo_declaration_seal(A_original_cdo); // Mark CDO declaration as
    complete
>
>     /* Create an unpooled CDO to specify how the data must be transformed and
    stored into the 'transposed' matrix */
>     mstro_cdo A_transposed_cdo = NULL;
>     mstro_cdo_declare("unpooled", MSTRO_ATTR_DEFAULT, &A_transposed_cdo);
>
>     // Common attributes
>     mstro_cdo_attribute_set(A_transposed_cdo, MSTRO_ATTR_CORE_CDO_RAW_PTR, At,
    false);
>     mstro_cdo_attribute_set(A_transposed_cdo,
    MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE, &A_size, false);
>     mstro_cdo_attribute_set(A_transposed_cdo,
    MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &A_elem_size, false);
>     mstro_cdo_attribute_set(A_transposed_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &
    A_num_dims, false);
>
>     // Transposed matrix specific attributes
>     size_t A_dims_transposed[] = {A_dims_original[1], A_dims_original[0]};
>     mstro_cdo_attribute_set(A_transposed_cdo,
    MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, &A_dims_transposed, false);
>     size_t A_layout_transposed = MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER_ROWMAJOR;
>     mstro_cdo_attribute_set(A_transposed_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER, &
    A_layout_transposed, false);
>
>     mstro_cdo_declaration_seal(A_transposed_cdo); // Mark CDO declaration as
    complete
>
>     // Perform the transformation: implicitly, it is a transposition due to the
    different layout orders
>     mstro_transform_layout(A_original_cdo, A_transposed_cdo,
>                            A_original_cdo->attributes, A_transposed_cdo->
    attributes);
>
86c137
<             sum += A[(i * N) + j];
---
>             sum += At[(j * 6) + i];
88c139,149
<     }
---
>     }
>
> // Transpose back into original
>     mstro_transform_layout(A_transposed_cdo, A_original_cdo,
>                            A_transposed_cdo->attributes, A_original_cdo->
    attributes);
>
>     mstro_cdo_dispose(A_original_cdo);
>     mstro_cdo_dispose(A_transposed_cdo);
>
>     mstro_finalize();
>     free(At);
```

As it can be seen, the new code contains the declaration of a new *At* array that will store the transposed version of the original *A* array and two CDOs, one for each array. In addition, the Maestro transpose operation is invoked and the array accesses have been rewritten as needed to use *At* and access it properly.

In order to compare the performance of the Maestro-enabled version,

we also developed the version of the code that performs the transposition without Maestro. Changes are limited to the *main*, thus we omit the rest of the file for brevity:

Listing 3: Manual (non-Maestro) transposition version *main* function.

```
int main()
{
    double t1, t2, t3, t4;
    int i, j;
    double sum; // Matrix summatory data output

    // Data declaration NxM
    int *A = (int *)malloc(M * N * sizeof(int)); // Source matrix
    int *At = (int *)malloc(N * M * sizeof(int));

    // Fill the matrix with incremental values
    for (i = 0; i < M; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            A[(i*N)+ j] = (i * N) + j;
        }
    }


        // Data movement : Manual Transpose
    t1 = mysecond();
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            At[j*M+i] = A[i*N+j];

    sum = 0;

    // Data computation : Matrix summatory
    t2 = mysecond();
    for (j = 0; j < N; j++)
    {
        for (i = 0; i < M; i++)
        {
            sum += At[(j * M) + i];
        }
    }

    t3 = mysecond();

        // Data movement back : Manual Transpose
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            A[j*M+i] = At[i*N+j];
    t4 = mysecond();

    #ifdef DEBUG
    print_matrix_info("Original", A, M, N);
    #endif

    free(A);
    free(At);

    printf("Matrix␣Summatory␣%18.2f\n", sum);

    printf( "\nBaseline␣Matrix␣manual␣trnasposition␣dinamic␣summatory␣by␣columns␣
        time\n");
    printf( "Data␣Movement␣time,␣s:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%18.8lf\n", (t2 - t1)  );
    printf( "Data␣Movement␣time␣(reverse␣Tr),␣s:␣%18.8lf\n", (t4 - t3)  );
    printf( "Computation␣time,␣s:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%18.8lf\n", (t3 - t2)  );
    printf( "Walltime,␣s:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%18.8lf\n", (t4 - t1)  );

    printf("\n");
```

```
}
```

The last goal of this demonstrator is to compare the performance of the Maestro-enabled data transposition versus the explicit data transposition coded manually by the developer. The three source code versions evaluated are the original code (*Original*), the Maestro-enabled code (*Maestro*) and the non-Maestro-enabled code (*Non-maestro*). The benchmarking platform is the SAGE2 system [7] using different matrix sizes. More specifically, node *client-23* was used, equipped with an Intel Xeon E5630 CPU and 2 GBs of memory, while the compiler used was *clang 11.0.0*. The resulting run-times are presented in the following table:

Matrix size	Original	Maestro	Non-maestro
900 x 1150	0.01	0.06	0.02
30000 x 31000	14.16	52.68	59.42
45600 x 47000	44.39	149.24	177.07
465000 x 200	1.09	3.62	2.45
1071600 x 2000	40.24	139.74	133.90
2000 x 1071600	38.92	150.92	443.07

The following two observations are in order here. First, for illustrative purposes this demonstrator uses a simple synthetic code that exhibits an inefficient memory access pattern that can be handled through the Maestro transpose operation. As a consequence, the run-time of the *Maestro* and *Non-maestro* versions is slower than the *Original* version. It should be noted that HPC code typically perform compute-intensive calculations where the data transposition helps to reduce the run-time of the application as a whole.

The second observation is that, in general, the Maestro framework provides a more efficient implementation of the transpose operation. In general, the run-time of *Maestro* is comparable to that of *Non-maestro* (900 x 1150, 30000 x 31000, 45600 x 47000, 465000 x 200 and 1071600 x 2000). There are also some cases like matrix size *2000 x 1071600* where it is significantly faster that an explicit naive implementation by the programmer. In the Maestro framework the management and implementation of multi-dimensional arrays relies on Mamba, which provides an abstraction layer where data transformation operations can be efficiently optimized for the target hardware system.

The last step of this demonstrator is to evaluate the performance of several parts of the *Maestro*-enabled version. The following Figure 3 shows a breakdown in the following parts: first, Maestro initialization and matrix transposition (*Maestro transpose*); second, the matrix summary computation (*Matrix sum computation*); and third Maestro matrix transpose back to the original data placement (*Maestro transpose back*). The bars display the breakdown of the run-time of the *Maestro* version. The figure also shows a line that represents the run-time of the *Original* version. As expected, most of the run-time accounts for the two required matrix transpositions, one before the computation to place the data so that locality is exploited and another one after the

computation to place data in the original locations so that the rest of the program can run normally. Moreover, the computation time for the transposed version is faster than that of the original version. Moreover, note that the *Matrix sum computation* (blue portion of the bars) is faster than the *Original* code (line), showing the potential benefit of the Maestro transpose operation for real HPC codes.

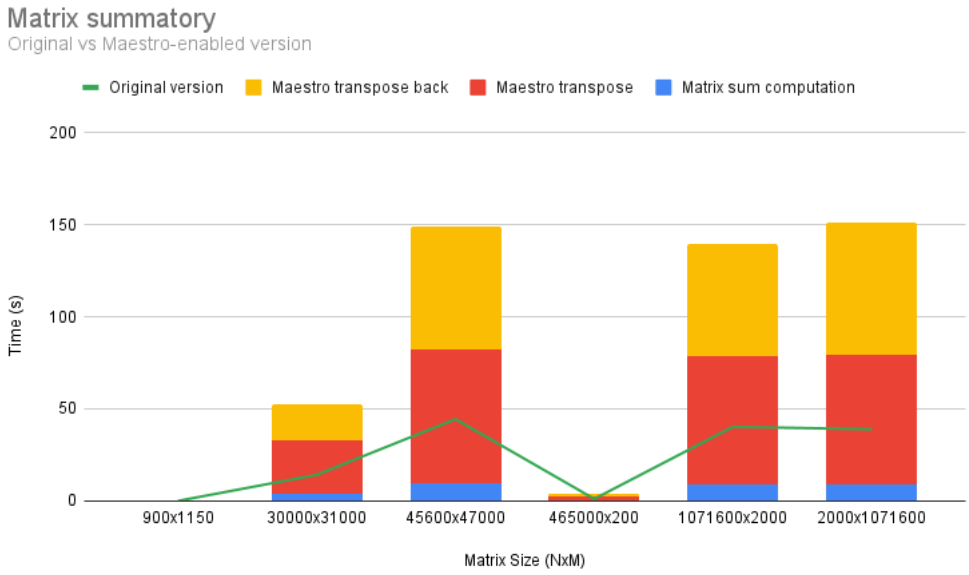


Figure 3: Maestro-enabled version breakdown

6 Dynamic Provisioning

6.1 Objective

The objective of this demonstrator is to showcase Dynamic Storage Resource Provisioning (DSRP), in particular data manager deployment and performance for filesystem data manager.

6.2 Description

HPC systems usually provide storage resources shared among all users, which can be suboptimal for I/O intensive workloads. In the context of Maestro, a dynamic provisioning mechanism as part of the middleware allows allocation of dedicated storage resources to a workflow and deploy an appropriate data manager on top of these resources. 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 minimum requisites for running DSRP on a system are Python 3, Ansible[3] 2.x and Sarus[4]. While Python 3 is a standard language available on most system and Ansible, a popular automation and configuration management tool, can be easily installed by a standard user; Sarus is an OCI-compatible container engine specialized for HPC usage which requires administrative permissions to be installed.

Container images can be fetched from public and private repositories, enabling easy use of artifacts provided by third parties as well as specific system-tuned or custom artifacts. However, please note that the BeeGFS[5] filesystem requires a local kernel module which does not belong to the container image and needs to be installed separately.

This demonstrator shows DSRP capabilities and basic performance metrics.

6.3 Demonstrator

All tests were performed on the Piz Daint[6] system at CSCS. There are 8 near-compute storage nodes that have 3 NVME devices of 6TB each one (aggregate raw capacity of 144 TB). In the case of the filesystem data manager (BeeGFS), test times are compared against a shared cluster-wide 8.8 PB Lustre filesystem, so results can have variations related to the shared usage of the global Lustre filesystem.

- **M5.1** Currently DSRP can deploy three different data managers which are suitable for three different data models: a POSIX parallel filesystem (BeeGFS), an object storage (MinIO) and a NoSQL database (Apache Cassandra).

  DSRP provides a simple command line syntax to deploy these dynamic data managers to simplify integration with other components.

  The syntax to start a data manager has the following format:

  ```
  dsrp_deploy.py start <beegfs|minio|cassandra> -t<cluster_name> \
  -s<storage_nodelist>
  ```

  which requires specifying the data manager type, cluster name which contains the description of all storage nodes and devices and storage nodes names allocated for that data manager instance.

[3] https://www.ansible.com
[4] https://sarus.readthedocs.io
[5] https://beegfs.io/
[6] https://www.cscs.ch/computers/piz-daint/

To get access of the filesystem data manager (BeeGFS) from the compute nodes or nodes executing the application reading/writing data, an extra step to mount the filesystem on these compute nodes is required:

```
dsrp_deploy.py start beegfs -t<cluster_name>\
-c<nodelist> [-m<client_mount_path>]
```

To facilitate deployment, both storage nodes and compute nodes can be setup in one single command.

Following the same syntax, individual clients or servers can be stopped as required:

```
dsrp_deploy.py stop <beegfs|minio|cassandra> -t<cluster_name> \
[-c<nodelist>] [-s<storage_nodelist>]
```

By default, DSRP deletes data on the storage devices when stopped. However, a usage policy of an HPC system may allow users to persist data on storage nodes after a job finishes, thus, we introduced the option `-keep` to skip the deletion step.

Tests were run to compare performance between a dedicated dynamically provided BeeGFS filesystem and a cluster-wide shared Lustre filesystem. IOR [7] was the selected benchmark as it is a standard parallel I/O storage benchmark. BeeGFS servers were deployed with default options on 8 storage nodes and additional 4 compute nodes mounted the filesystem. Tests were run on compute nodes to measure performance for independent file access, which could be a common case for Maestro-enabled workflows. On each node, 4 processes read and wrote files with sizes of 500, 1000 and 1500 MB.

File size [MB]	Lustre read	BeeGFS read	Lustre Write	BeeGFS write
500	43	14	10	13
1000	5.8	12.6	11	13.1
1500	6.1	12	11.8	13

Table 2: I/O bandwidth [GiB/s], one file per process, 16 processes across 4 nodes

For most cases, a dedicated BeeGFS filesystem provides a clear improvement over the shared filesystem. There is an important cache effect for the read benchmarks on Lustre with files up to 512MB. Additional results were previously published in the article *Dynamic Provisioning of Storage Resources: A Case Study with Burst Buffers*[8].

For the object storage MinIO, the 'S3-benchmark' was used to measure performance of GET, PUT and DELETE operations. The client used 10 threads on a compute node with different size objects.

[7] https://github.com/hpc/ior

Object size [MB]	GET [GiB/s]	PUT [GiB/s]	DELETE [ops/s]
1	2.6	1.0	2615
10	4.9	2.1	2755
100	5.4	2.8	2820
1000	5.4	2.9	2826

Table 3: Operations performance, one client with 10 threads

Results show an expected performance increase for bigger objects. However, this tests use a default configuration which could be optimized for particular requirements.

Finally, simple tests were carried for Apache Cassandra using 'tlp-stress' to verify functionality. The 'BasicTimeSeries' was used with a mixed rate of 20% reads and 80% writes. It sent 10k and 20k queries per seconds, obtaining a throughput of 9.9k and 19.8k operations per seconds respectively, showing the database is working correctly.

These results display the capabilities to dynamically deploy different storage resources and to obtain expected performance.

- **M5.2**

  A simple workflow description that requires an isolated filesystem was made, and then translated into tasks. Then, the Splinter Maestro-enabled workflow manager (described on D4.5[5], section 3) was used to execute the tasks, which included starting and stopping a deployment of an isolated BeeGFS filesystem and running simple commands through Slurm. As described previously, integration was easy as DSRP only requires a command execution to start and stop a data manager.

  A complete Maestro-enabled workflow with Maestro-core features is presented on **M5.3**.

- **M5.3**

  For testing purposes, the Mocktage application (presented on D4.5[5], section 2) was chosen because it provides a Maestro-enabled workflow, and Splinter is used as a Maestro-enabled workflow manager.

  The test workflows consist of a set of graphs with several iterations and forks that produce and consume objects. No additional tasks that require computational resources were added, as the objective is to benchmark I/O intensive operations. These workflows are run with different object sizes over a shared filesystem and a dedicated dynamically provisioned one. Finally, total amount of transferred data is summed and timed to give the overall bandwidth, which allows to compare and measure the potential benefits if this optimization. Detailed information on how to incorporate DSRP to a Maestro workflow is provided on D4.5[5] section 4.1.

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

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

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

With these tests, it is shown how a dynamically provided filesystem data manager can reduce execution time on several cases compared with a shared cluster-wide one. As this component is easy to integrate, it can be incorporated into existing workflows with minimal modifications.

6.4 Performance results

- **T5.1** Deployment time depends on Ansible tasks parallelism, SSH connection time to nodes and services startup time. The code starts data managers services in parallel so elapsed time is almost constant for any quantity of nodes. For this test, each data manager was deployed 10 times. For each data managers we present the average elapsed times, standard deviations are less than 5 seconds between the runs, which is negligible for large scale systems.

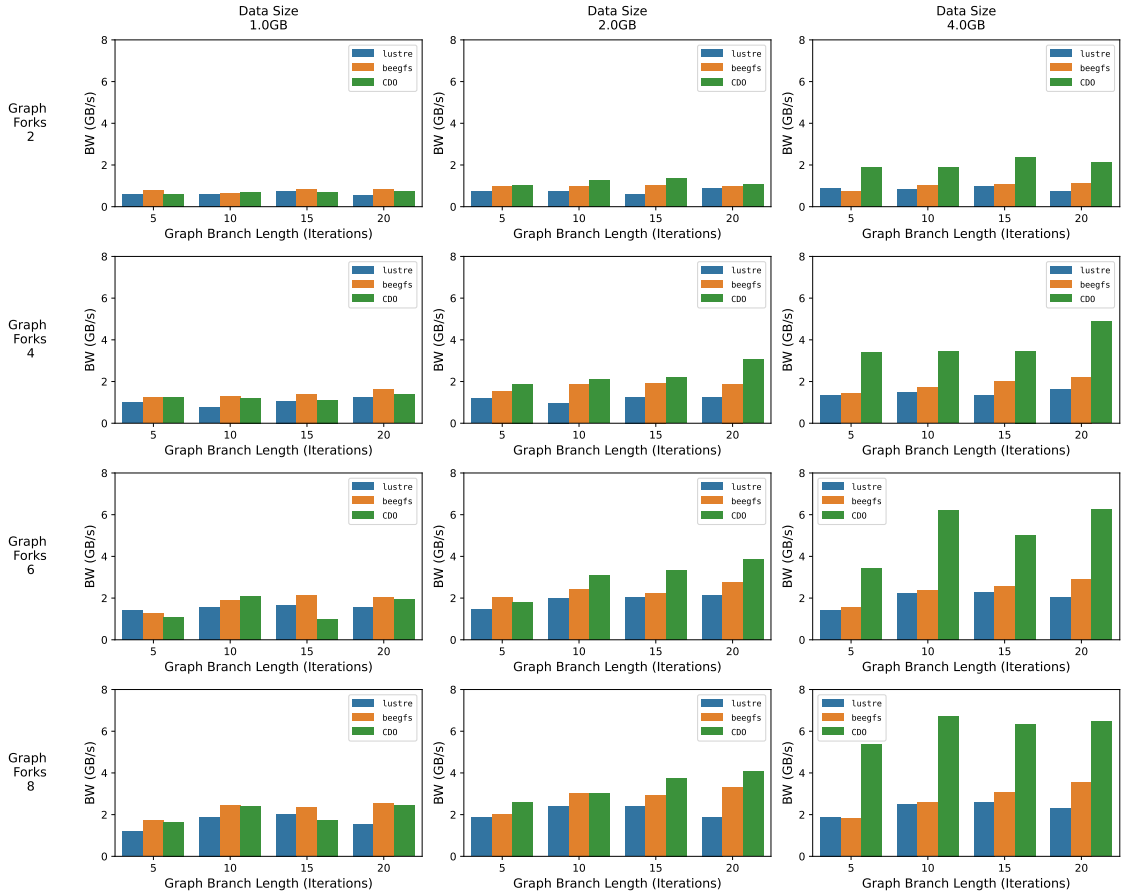


Figure 4: Comparison for workflows running over Lustre, BeeGFS and memory CDOs.

The BeeGFS filesystem has the larger quantity of tasks to perform: starting a management service on the first storage node, and two services for each device on storage nodes. All these services have to register to the BeeGFS Management service, which could take up to 5 seconds. To deploy the first storage node it took 20 seconds, and 12 seconds for additional ones. For compute nodes, the deployment step includes launching a helper service and mounting the filesystem, which took 10 seconds over all nodes.

For both MinIO and Cassandra, a server service is started in parallel on each selected nodes, respectively requiring 6 and 25 seconds to deploy.

- **T5.2**

  DSRP provides a stage-in/stage-out capability for the filesystem data manager (BeeGFS) as a simple way to copy files to/from the dynamic provisioned storage and a permanent filesystem. The current code only uses one process so parallel performance from the Lustre filesystem can not be achieved. However, it can be expanded to a parallel mechanism using Ansible tasks or a MPI-based approach.

In this test, a 'tar' file from the Lustre filesystem is decompress to the provisioned filesystem for stage-in, and a group of individual files is archived into a 'tar' file on Lustre for stage-out. Then, times are compared between these steps and standard copies of individual files between the filesystems. For the later copies, option '-a' is used to preserve metadata (permissions, modification timestamp, etc) to have the same behaviour as creating/decompressing 'tar' files. A thousand files where used for each file size, and results are expressed in seconds.

File size [MB]	Copy from Lustre	Stage-in	Copy to Lustre	Stage-out
0.1	15.5	2.5	10.3	1.6
1	43.0	9.6	12.5	4.6
10	40.0	99	21.2	30.5

Table 4: Comparison between stage-in/stage-out feature and standard copy

Results show this feature can be useful when a large quantity of small files is involved, as large parallel filesystems are usually not optimized for this case.

7 Guided I/O in pre/post-processing

7.1 Objective

The objective of this demonstrator is to show Guided I/O usage, wherein Maestro MIO can be used with hints from the user and/or the system to take specific actions - for example to optimize data movements to improve application I/O performance.

7.2 Description

MPI-IO's hints enable MPI applications to select IO preferences and parameters on data access layouts, collective IO optimizations and file system specific configurations, but these hints are generally coarse-grained and only portray a static view on how an application uses IO. When processing hints, MPI-IO performs optimisations according to a set of pre-defined rules without considering the state of the underlying file system and the application's workload characteristics, not being able to achieve the best performance.

The missing information of the state of the underlying file system can be gained from telemetry data provided by the I/O software stack or the underlying storage system. A new component, the telemetry processor, has been added in the guided IO framework of Maestro IO (MIO) as illustrated in Figure 5.

It accesses and analyses telemetry data to generate hints with richer representation and workload profiles. Unlike the application hints which are usu-

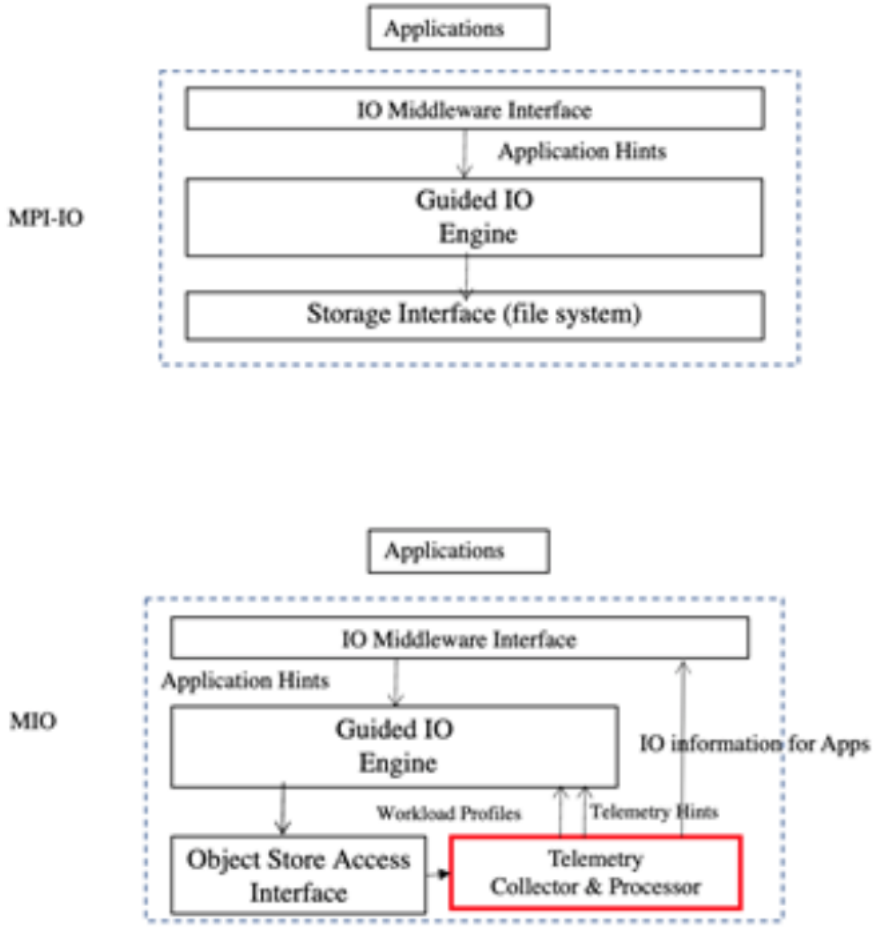


Figure 5: Guided I/O frameworks in MPI-IO and MIO

ally pre-set, the hints from the telemetry processor are distilled from continually generated telemetry data and will be constantly updated. An example of hint from telemetry data is object hotness which is captured and calculated during object IO operations. We demonstrate the application of object hotness hint using the HSM (Hierarchy Storage Management) system for pre-processing and post-processing.

The MIO Guided IO also provides the MIO applications with the ability to pass "Hints" to the object store backend on upcoming data usage attributes. Each hint inside MIO is expressed as in a key-value format, hint_key, hint_value, where hint_key is a predefined and unique integer for each hint in MIO. Unlike MPI-IO hints which are set for opened files only and the values of hints are lost when a file is closed, MIO differentiates 2 types of hints:

- Session hints which are set only for an object that is opened and are kept alive only during the time where the object is opened. Session hints will be destroyed when an object is closed, and the hints won't be valid for the next session. Example of session hints: object hotness and pool

selection hint.

- The persistent hints are valid for the lifetime of the object. The persistent hints keep track of an object's IO activities over its lifetime instead of one single object open-close session. This is particularly useful for post-processing data. For example, the object placement optimisation can be done based on long term observation instead of each individual IO session.

With the purpose of showing how the guided IO helps data pre-processing and post-processing, the demonstrator focuses on one hint and its optimisation, MIO_HINT_OBJ_HOT_INDEX, to demonstrate the benefit of the Guided IO framework and its usage. MIO_HINT_OBJ_HOT_INDEX passes information on how frequently the object will be accessed to MIO and MIO internally decides, which pool to store the object according to the hotness value. MIO also defines 2 system level hints, MIO_HINT_SYS_HOT_OBJ_THRESHOLD and MIO_HINT_SYS_COLD_OBJ_THRESHOLD to set the thresholds of object access frequency for hot and cold objects.

MIO also defines another hint, MIO_HINT_OBJ_WHERE, which allows users to influence the choice of a pool according to performance requirements, MIO_POOL_GOLD, MIO_POOL_SILVER or MIO_POOL_BRONZE. This hint can be used in data pre processing to choose which storage pool to store the objects according to known performance requirement or IO characteristics. For example, we can place the frequently accessed short-lived temporary objects in high performance storage pool to speed up the data processing. These hints are available for Maestro and its applications for use to place objects in different tiers according to its object "hotness".

7.3 Demonstrator

An HSM (Hierarchical storage management) application is used as demonstrator to show how the MIO guided IO work. As the number of objects stored in an object store system grows, it will at some point be necessary to migrate objects into a different tier. This is facilitated by the HSM depending on the hotness of an object. Hot objects are moved into a better performing pool and cold objects into a lower performing pool that provides higher capacity. How to identify hot and cold objects is difficult. Object access frequency provides with an index showing how active an object is during a specified period. The object access frequency is captured by the MIO telemetry collector/processor. The current version of MIO determines if an object is hot (or cold) by simply examining whether the access frequency is higher (or lower) than MIO_HINT_SYS_HOT_OBJ_THRESHOLD or
MIO_HINT_SYS_COLD_OBJ_THRESHOLD,
a system level hint that is set by the application.

The Mini-HSM application (included in the MIO source code[8]) makes use of the MIO and the MIO backend, the Motr object store's multi-pool feature to

[8] https://github.com/Seagate/cortx-mio

create and access objects in different pools. For a system with multiple storage tiers, such as NVME, SSD and HDD tiers in the Sage cluster[9] , Motr usually configures it with multiple pools, each is built with devices of the same tier. Mini-HSM is also shipped with a workload generator to simulate object access pattern. While the workload generator is running, object access frequency is collected and used as the object's hotness as explained above.

HSM often needs to move objects between pools of different storage tiers. Typical steps to move an object when combining MIO guided IO are: (1) Query hotness of an object by calling MIO guided IO API; (2) The hotness can be used as the hint when creating a new object to which the original object will be moved. MIO internally decides which pool to create this new object based on the object's hotness hint. Hot objects will be created in a better performance pool, while a cold object is created in a lower performance pool; (3) The old object is copied to the new object if a different pool is selected for the new object. Currently the thresholds for hot and cold objects are set as system level hints as explained above.

The mini-HSM demonstrator is run on the Sage cluster at Jülich Supercomputing Centre (JSC). Figure 6 shows the Motr pool configuration and the HSM demonstrator used 2 pools: the first pool is constructed using NVME devices with a 128-bit ID 0x6f00000000000001:0x239, while the second pool with ID 0x6f00000000000001:0x253 is configured with HDD devices. The HDD pool is configured as the default pool for objects.

```
Data pools:
    # fid name
    0x6f00000000000001:0x239 'tier1-nvme-p1'
    0x6f00000000000001:0x246 'tier2-ssd-p2'
    0x6f00000000000001:0x253 'tier3-hdd-p3'
    0x6f00000000000001:0x25e 'tier2-ssd'
    0x6f00000000000001:0x27a 'tier3-hdd'
```

Figure 6: SAGE Cluster Pool Configurations

The Figure 7 shows the screenshot of one run of Mini-HSM. The Mini-HSM emulates common commands of an HSM system: creating objects, reading/writing objects and moving objects between tiers. To demonstrate how to move objects according to object hotness hint, we also add a few commands to display the object's pool, set and get hotness thresholds for hot and cold objects and generate emulated IO workload. In the simple example, we first create 5 objects and populate them with data. At the end of the IO workload command (workload), the hotness for each object is displayed. Using the command show, we can clearly see that all objects are stored in the HDD pool. We also run the command move on 2 objects: the hottest object (object 4) is moved to the pool with the best performance (NVME pool); the cold object (object 0) doesn't move as the object is created at the default HDD pool and its object hotness suggests that it is a cold object. The pools for each object can be verified running the command show. WRITE and READ (1GB data) are

[9]https://sagestorage.eu/

tested on the moved objects. It shows that moving the hot object to higher performance pool increases its IO bandwidth.

```
[wu5@client-21 demo]$ ../examples/mio_hsm -s 1m -c 8 -o 3:12345678 -n 5 -y ./mio_config_hsm.yaml
hsm> help
 actions:
    create
    write <index> <number of objects> <number of blocks>
    read <index> <number of objects> <number of blocks>
    move <index> <number of objects>
    show <index> <number of objects>
    set_hot_thld <hot object threshold>
    set_cold_thld <cold object threshold>
    get_thlds
    workload
hsm> create 0 5
[hsm_create_objs] create 5 objects ...
hsm> write 0 5 8
[HSM_WRITE] Time = 0.898439 secs, BW = 44.521665 MB/s
hsm> show 0 5
Object     POOL_ID
[Object 0] (6f00000000000001:253)
[Object 1] (6f00000000000001:253)
[Object 2] (6f00000000000001:253)
[Object 3] (6f00000000000001:253)
[Object 4] (6f00000000000001:253)
hsm> workload
OBJ0 Hotness: 8
OBJ1 Hotness: 232
OBJ2 Hotness: 112
OBJ3 Hotness: 264
OBJ4 Hotness: 432
hsm> set_hot_thld 400
hsm> set_cold_thld 100
hsm> move 0 1
Object stays in the same pool, not moving :)
hsm> move 4 1
hsm> show 0 5
Object     POOL_ID
[Object 0] (6f00000000000001:253)
[Object 1] (6f00000000000001:253)
[Object 2] (6f00000000000001:253)
[Object 3] (6f00000000000001:253)
[Object 4] (6f00000000000001:239)
hsm> write 0 1 1024
[HSM_WRITE] Time = 10.906418 secs, BW = 93.889671 MB/s
hsm> write 4 1 1024
[HSM_WRITE] Time = 7.681222 secs, BW = 133.312121 MB/s
hsm> read 0 1 1024
[HSM_READ] Time = 4.914210 secs, BW = 208.375303 MB/s
hsm> read 4 1 1024
[HSM_READ] Time = 4.429178 secs, BW = 231.194140 MB/s
hsm> quit
[wu5@client-21 demo]$
```

Figure 7: Mini-HSM Demonstrator Using MIO Guided IO

7.4 Performances results

The SAGE cluster [10] is used in the MIO guided IO performance tests. The cluster uses the Cortx Motr object storage system to provide applications with accesses to its storage. The Motr storage pools are configured the same as described in the last section. The NVME pool is used to store hot objects, while the HDD pool is configured as the default pool for objects. The pools are configured with the same de-clustered parity group settings, for example, data unit size and the numbers of data and parity units etc. 1MB data units are

[10] https://sagestorage.eu/

configurated in all tests. We compare the write and read bandwidths before and after moving the hot objects from the HDD pool to the NVME pool by mini-HSM using the hotness hints. Be aware that mini-HSM is implemented as a single thread application.

The Figure 8 shows the bandwidth improvement on different IO sizes. The IO size for object writes and reads varies from 64MB to 4096MB (4GB). The write and read bandwidths for 64MB IOs improve 2.12 times, and 4.52 times respectively. For larger IO of 256MB, 1024MB and 4096MB, the MIO guided IO increases the write bandwidths by 39%, 34% and 34% while the read bandwidth is improved by 26%, 29% and 26%

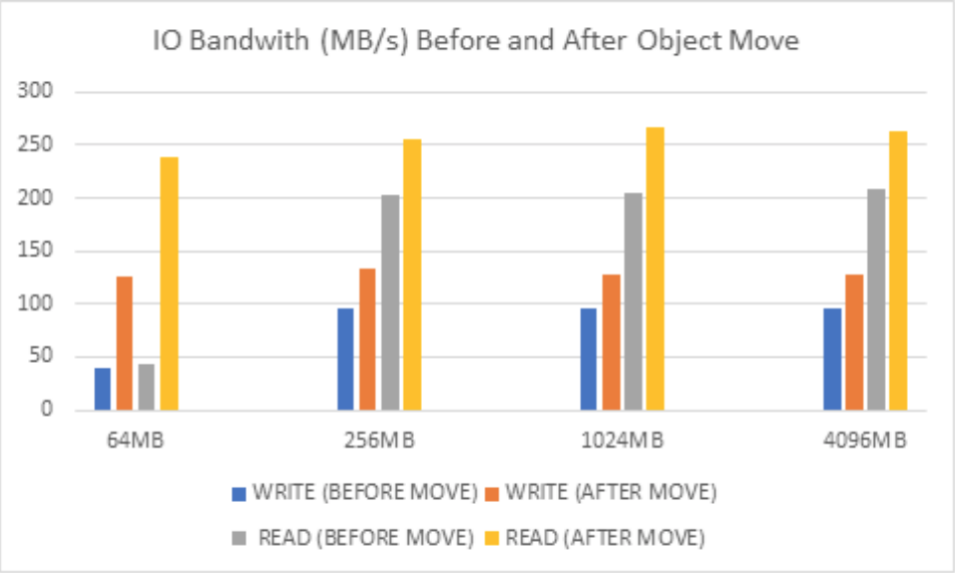


Figure 8: IO Bandwidth Comparison Before and After Object Move

Please note that the absolute performance of the SAGE system continues to be optimized and higher absolute numbers are expected for these tiers - though the ratio of their performances is not expected to change.

8 Concluding Remarks

In this deliverable, we presented a demonstration of the Maestro framework and illustrate the usage of Maestro for five system software demonstrators. An evaluation of the performance has been completed for each of these demonstrators. The demonstrators show the advanced capabilities of this framework that has been developed from scratch. For some of the demonstrator also significant performance benefits based on using Maestro could be shown.

References

[1] C. Haine, U.-U. Haus, and A. Tate, *Maestro D3.2*, 2019.

[2] C. Haine, U.-U. Haus, and A. Tate, *Maestro D5.3*, 2019.

[3] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *European Conference on Parallel Processing*, pp. 78–88, Springer Berlin Heidelberg, 2008.

[4] J.-B. Besnard, A. Malony, S. Shende, M. Pérache, P. Carribault, and J. Jaeger, "An MPI halo-cell implementation for zero-copy abstraction," in *Proceedings of the 22Nd European MPI Users' Group Meeting*, pp. 1–9, 2015.

[5] J. Badwaik, J. Biddiscombe, and A. Dabin, *Maestro D4.5*, 2021.

[6] J. Novo and M. Arenaz, *Maestro D4.4*, 2021.

[7] S. Consortium, *Sage Prototype System Documentation*, 2020.

[8] F. Tessier, M. Martinasso, M. Chesi, M. Klein, and M. Gila, "Dynamic provisioning of storage resources: A case study with burst buffers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Los Alamitos, CA, USA), pp. 1027–1035, IEEE Computer Society, may 2020.