D3.4
Transformations and Map optimisation release

Work package	WP3 Core Memory and Data Aware Middleware	
Author(s)	Salem El Sayed Jayesh Badwaik Utz-Uwe Haus Christopher Haine Javier Novo Rodriguez Manuel Arenaz Ganesan Umanesan Sai Narasimhamurthy Sining Wu	JUELICH JUELICH HPE HPE APPENTRA APPENTRA SEAGATE SEAGATE SEAGATE
Reviewer #1	Dirk Pleiter	JUELICH
Reviewer #2	Maxime Martinasso	ETHZ
Dissemination Level	Public	
Nature	Other	

Date	Author	Comments	Version	Status
12.11.2020	Salem El Sayed	Initial content list	V0.1	Draft
30.11.2020	Salem El Sayed	Changes to content list	V0.2	Draft
23.11.2020	Javier Novo	Added 'Data access semantics'	V0.3	Draft
30.11.2020	Salem El Sayed	Major changes to content organisation	V0.4	Draft
07.12.2020	Utz-Uwe Haus	Added mapping table wrt. mamba	V0.5	Draft
01.02.2021	Javier Novo	Added a list of possible transformations	V0.6	Draft

05.03.2021	Sining Wu	Added MIO section	V0.7	Draft
08.03.2021	Javier Novo	Reduced the transformations list	V0.8	Draft
08.03.2021	Christopher Haine	Added introduction to transformations	V0.9	Draft
09.03.2021	Christopher Haine	Added Maestro software information	V0.10	Draft
10.03.2021	Sai Narasimhamurthy	Added introduction and other sections	V0.11	Draft
16.03.2021	Christopher Haine	Clarified mappings and feedback questions	V0.12	Draft
18.03.2021	Christopher Haine	Some text on qualitative performance	V0.12	Draft
31.03.2021	Jayesh Badwaik	Resource description content	V0.13	Draft
14.04.2021	Jayesh Badwaik	Added architecture section	V0.14	Draft
27.04.2021	Maxime Martinasso	General comments and text improvements	V0.15	Draft
28.04.2021	Jayesh Badwaik	Rewritten mapping section	V0.16	Draft
28.04.2021	Jayesh Badwaik	Resource description for workflows	V0.17	Draft
28.04.2021	Jayesh Badwaik	Added constraints on workflows	V0.18	Draft
29.04.2021	Javier Novo	Multiple changes to the transformation section	V0.19	Draft
30.04.2021	Salem El Sayed	Full document edit, ready for internal review	v1.0	Draft
20.05.2021	Jayesh Badwaik	Addressed Most Comments	v1.1	Draft
27.05.2021	Sai Narasimhamurthy	Addressed MIO comments	v1.2	Draft
31.05.2021	Jayesh Badwaik	Addressed remaining Comments	v1.2	Draft
31.05.2021	Salem El Sayed	Final version	v2.0	Final

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

Executive Summary

This report documents the software release targeting transformations and map optimisation. The released software builds on top of the Maestro framework developed as part of WP3 and introduced in D3.1, D3.2 and D3.3 [1, 2, 3]. It also refers to some of the work done in WP4 on data semantics and WP5 on MIO, which were introduced in D4.1 [4] and D5.1 [5] respectively. Any missing requirements of the released software as part of this deliverable will be completed within the work done by WP4 and WP6.

Contents

Glossary

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

1 Introduction

The deliverable deals with how to reference data in the “physical” resources from the Maestro stack and the workflows deployed on it. The references to the physical resources are provided from the different components of the Maestro stack, namely, Mamba, Maestro Core, Maestro IO (MIO) components and the Maestro workflow.

The physical resources are layouts across different layers of Memory and persistent storage - that eventually map to Maestro Objects. The memory resources could be DRAM, HBM, NVDIMM and GPU memory. The persistent storage resources could be NVRAM used as persistent storage, SSDs and Disks.

Note that this deliverable also deals with transformations of data layouts in physical memory, enabled by Maestro and benefitted by having readily available use cases. The main goal of data mapping and data transformations is to optimise execution. Tools to help Maestro to do this better are also discussed in this deliverable, including the Parallelware static code analysis tool specializing in performance to help developers write Maestro-enabled applications.

As part of this report we introduce resource description tools developed to enable these mapping and transformation operations.

The remaining document is organised as follows, Section 2 introduces the data mapping in Maestro. Section 3 introduces a set of transformations and Section 4 describes how some of them are being handled in Maestro. Section 5 describes a Maestro-developed library to describe resources. Finally, Section 6 provides information on the released software.

1.1 Relation to other parts of the project

This deliverable follows the initial core Middleware API specification work (reported in D3.1 [1]) and the full implementation of the core data model and memory system model of the Maestro middleware described in D3.3 [3]. The smart usage of this middleware is presented in this deliverable by providing the ability to reference appropriate physical resources and enabling appropriate transformations of data within these physical resources for application and workflow optimization. This work is also closely linked to the work on MIO in WP5 (T5.2 and T5.3) [5] and the data access semantics work in WP4 (T4.1) [4]. The final results of the developed tools discussed in this report will be demonstrated as part of the work done in WP6.

2 Data Mapping in Maestro

In this section we introduce the connection from physical resources to Maestro core, to Mamba, to persistent storage including MIO and the workflow optimisations. To that end, we develop the resource description tool `radle`, which is further introduced later in the document.

2.1 Architecture

There are multiple abstractions of resources in a Maestro enabled workflow, with different libraries having their own abstractions. Each library which is involved in making decisions about resources in the workflow is labeled as a resource-aware component or just component for short. One of the main objectives of the deliverable is to create a map of resources across the different abstractions so that multiple resource-aware components can talk to each other.

While there is no strict hierarchical structure between the resource abstractions, the abstractions can be still be classified into relatively higher level and lower level abstractions. Highest level abstractions are those which the user of Maestro will be interacting with directly, while at the lowest level, we will have abstractions that interact directly with libraries such as `hwloc`.

At the highest level, it is desirable to describe the resources in a very functional manner. For example, a memory might be marked as set aside for the function of "carrying out transpose transformations". At the lowest level, the description would have more details about the hardware. For example, at a hardware level, the same memory will be described at as being of type DRAM on the NUMA domain `0` of the host `node23` with a capacity of `8192` MiB. Apart from a declarative resource description, the components also have their own way of discovering and describing the resources, and hence, there is the need of a method to map both resources description between themselves.

In the general case, the problem of mapping a description of resource from one arbitrary component to another is unsolvable. Moreover, the effort of creating a map between every pair of components in a combinatorial fashion is an unreasonable amount of work. Instead we have decided on a convention to describe the resources of provided or discovered devices using a base description from where one can map all the descriptions. For this purpose, in Maestro, we have developed a resource description framework `radle` as the base language for describing resources. `radle` uses `hwloc` to discover the resources on a node, making the naming convention of `hwloc` the default convention for naming the resources in the workflow. All the resource in other components are then mapped to `radle`.

There are two different scenarios of looking at the mapping of resource description in a component. The first scenario is where there is description of the resource at the lowest-level of description, and there is a need to find the corresponding description of the resource at a higher level. This is useful, for example, when a component will need to match the resources available to

different tasks, and some resources are reserved for certain kinds of uses. For example, a certain node might be reserved for doing transform computations. This process is fairly explicit in implementation. The strategy for matching the resources and providing a corresponding resource description can all be done inside the resource description framework (`radle`). Furthermore, this strategy can be achieved at the time of instantiation of the workflow tasks on the target machine, for which, an actual resource description is available.

The second view is the developer view, where the developer can specify some constraints on the resources that they need for their programs in terms of a component, and then it is the responsibility of the component to find the optimal resource which matches the listed constraints. In this scenario, the mapping is not straightforward. Often, a developer will want to run his program on a diverse set of systems and with a diverse set of parameters. Therefore, it is not really possible for the developer to anticipate, both the target system and the requirements of the program at development time. In such a scenario, the developer is expected to request the resources in terms of the resource description syntax of different component And then the libraries can map the resources to the `radle`-provided resources.

In this section, we describe the relevant resources exposed by each of the Maestro components, like Mamba. At the end of each component description, we can create a map between those resources and the resources exposed by `radle`. However, the types of resources is quite large. Therefore, in order to simplify the problem space, we classify all resources into five categories namely:

1. Compute
   This category contains resources like the compute capabilities of the CPUs and accelerators such as GPUs.
2. Memory
   This category contains resources like DRAM, GDDR, HBM, NV RAM and similar memory technologies.
3. Storage
   Storage contains resources like global parallel file system and object storage.
4. Network
   Network contains specific interconnects and fabrics like InfiniBand or Ethernet.
5. Meta
   Meta resources are resources that describe a group of resources. They are important to identify the locality of the resources. Examples includes nodes, packages, NUMA domains and so on. The meta resources can contain multiple type of resources, for example, a NUMA domain might contain its own CPU, DRAM and PCIe based devices. So, allocating a meta resource is equivalent to allocating all the resources contained in the meta resource.

2.2 Resource description tool: `radle`

`radle` provides a mechanism to describe and query resources available to the various components of the Maestro framework. This allows tools using `radle` to make informed decisions on allocating resources to the various tasks, for example in a workflow execution or CDO placement.

Multiple types of resources might be accessed by the same API, or the same hardware resource might be accessed by different APIs based on the function. In order to allow for this separation, `radle` characterizes the resources by drivers, and resources are then different instances of the driver. For example, all DRAM would be grouped under a `dram` driver of the `memory` category. A classification of resources is given in Table 2.

Compute	Memory	Storage	Network	Meta
CPU	DRAM	Disk	Ethernet	Node
GPU	GDDR	Motr	Infiniband	NUMA
Stencil Accelerators	HBM			Package
	NV Memory			Network Locality

Table 2: `radle`'s classification of Drivers

2.3 Mamba

Mamba brings the concept of *memory layers* to Maestro as a native concept. The current mamba version (0.1.6) and roadmap specify the memory layers listed in Table 3.

Layer Name	Description	Access	auto-discovered	user-targetable	sublocality
`MMB_DRAM`	Main memory	Posix, SICM, jemalloc, umpire	yes	yes	yes: NUMA
`MMB_GDRAM`	GPU memory	CUDA, OpenCL, HIP	yes	yes	yes: multiple devices
`MMB_HBM`	High-bandwidth memory	libmemkind	yes	yes	yes: NUMA domains and devices
`MMB_NVDIMM`	Non-volatile memory	libpmem	yes	yes	yes: NUMA
`MMB_FS`	File system	C FILE*	no	yes	many (FS roots)

Table 3: Mamba 0.1.6 memory layers. "auto-discovered": indicates whether mamba will identify instances using `hwloc` automatically; "user-targetable": mamba API permits individual selection of instances; "sublocality": Support for partitioning of a given layer instance.

Users of the Maestro CDO API may want to use layer names and sublocality information to determine (or inquire about) CDOs location. Maestro configuration data may need to include mapping of user-defined names to layer

names and sublocality information, and may also need to configure layers that do not support auto-discovery.

Resource descriptions serve 3 purposes:

- Descriptive configuration – which involves system inventory, admin decision, workflow-owner decision, and workflow manager decision on which components to make visible to a Maestro workflow
- Prescriptive configuration – which involves resource limits (maximum usage, forced used of certain resources) as well as resource requirements of (part of a) Maestro workflow as per admin/workflow owner/-workflow manager decision
- Mapping of internal names and system-specific names to portable names: To make workflows portable across sites and composable they should only refer to Maestro-defined internal names ('well-known resource names'), like those coming from Mamba or MIO, or names defined in a workflow (component). A site-specific mapping to physical resource names needs to be performed, with one part of this mapping defined at site level, another part per-workflow, and possibly a part per application.

Examples:

- Configuration of usability/unusability of certain memory regions
  - Descriptive configuration of per-node memory layers (e.g., 'Node host42, has 21G MMB_GDRAM on device 0, 21G MMB_GDRAM on device 7')
  - Prescriptive configuration of per-node-memory layers ('use only GPU device 7')
- Configuration of one or more file system locations for out-of-core mamba storage
  - Mapping of a symbolic name to a file-system storage resource specifier block

2.4 MIO

The Maestro IO Interface (MIO) is served between Maestro middleware and object stores such as Mero. MIO not only defines a generic object-based interface to allow Maestro middleware to store and load data (such as CDOs) to and from backend object store, but also provides with guided-IO (hints) interface to help Maestro make better use of object store. Resource discovery and description is a critical part of MIO.

2.4.1 Resource description and MIO configuration

MIO is designed as an abstract layer on top of multiple object stores. The purpose of MIO is to allow Maestro applications to access different types of object store systems in a uniform way. But different object stores have different configuration settings. Manually maintaining these configurations is non-trivial and error prone. Automatic collection of object store configurations and creation of MIO configuration inputs can mitigate Maestro's burden.

2.4.2 Resource description and MIO guided IO

MIO organises objects into pools. An object store could serve multiple pools and each of these pools are configured with storage devices of a certain type-/technology. A heterogeneous object store is assumed to be configured with pools of NVRAM, SSD and HDD devices. Different pools have different characteristics in term of capacity and performance. Knowing these characteristics (from resource descriptions) is key to MIO's guided IO.

A storage pool is configured with certain data layout, such as replication or de-clustered parity group. To achieve better IO performance from a pool, MIO and its applications (Maestro middleware) typically need to know a pool's layout parameter in order to match their IO workload patterns with the data layout.

Examples of MIO guided IO are discussed below to show how the resource description of pools can be used to help Maestro to achieve better performance. The main goal of the MIO's guided IO interface is to enable Maestro applications to pass storage related hints to the underlying object storage systems so that various IO optimizations can be performed at the storage level. The IO performance of any storage is dependent on many factors such as configuration, access pattern, randomness etc., however, in general, SSDs are several times faster than HDDs, and NVM storage devices are at least twice as fast as SSDs. Maestro applications can pass <Slow>, <Medium> or <Fast> as an IO throughput hint during object creation. The object will be created in a suitable storage pool, for example the object will be placed entirely on an SSD storage pool if the hint is <Medium>. The resource description of which pool contains what kind of storage devices help MIO select a suitable pool.

As the number of objects stored in a storage system grows, sometimes we need to migrate objects into a different tier by HSM (Hierarchical storage management). How to choose which objects to be moved is difficult. Object hotness provides with an index showing how active an object is during a specified period, by using this object hotness, a user can enumerate the top-N least or most active candidate objects. The definition of hotness takes into account access history of an object, including how the numbers of READ/WRITE change over time and the time of last access etc. HSM moves an object in term of its hotness, and pool resource description is used in the following places:

- Make use of ordered pool performance in pool selection for hot/cold objects, for example, moving the hottest object into the best performance

pool and moving the coldest objects to least performance pool.

- Moving objects across pools using optimised IO parameters, for example, the optimised buffer block size when running MIO with Mero. The parameter is calculated using pool's data layout information.

2.5 Workflow Management

At the level of workflow management, we generate workflows with annotations. The annotations are then checked by workflow translator together with a system configuration of programmable hooks, and create tasks in the workflows or add constraints on the tasks.

3 Transformations in Maestro-core

This section introduces the basic data transformation primitives. The usage of these primitives is described in section 4.

There is a wide variety of data transformations that are frequently used in scientific and engineering applications. For example, the transposition of a matrix, the compression of a large amount of data in sparse storage formats to reduce the memory requirements, the flattening of multidimensional arrays into plain arrays to enable more efficient communications, or extracting a subset of a larger data set like a row of a matrix. The Maestro applications TerrSysMP (Juelich), SIRIUS (ETH Zurich/CSCS) and Hydro (CEA) have been analyzed from this perspective, selecting one data transformation for each application.

The rest of this section introduces the three data transformations selected, namely, *Transpose* (representative of TerrSysMP from Juelich), *Pack* (Hydro from CEA) and *Split* (SIRIUS from ETH Zurich/CSCS).

3.1 Transpose

Description:

- Change the data layout so that the order given by the C/C++/Fortran programming language matches the order given by the memory access pattern used in the application code.

Motivation examples:

- *[Transpose.1]* In C/C++ (or Fortran) sequential applications, there are two code snippets (typically two loops) where one loop produces a multi-dimensional array in row-major order and the other loop consumes the multi-dimensional array in a column-major order. Alternatively, the data can be produced in column-major order and consumed in row-major order.

- *[Transpose.2]* In sequential C/C++ and Fortran applications, there is one C loop that produces a multi-dimensional array in row-major order and one Fortan loop that consumes the multi-dimensional array in column-major order. Alternatively, the data can be produced in column-major order and consumed in row-major order.

- *[Transpose.3]* In distributed parallel applications (e.g. MPI), there is a multi-dimensional array distributed across several processes, each process holding an array slice. There is a data exchange between two processes, where one process produces the data in row-major order and the other consumes that data in the column-major order. Alternatively, the data can be produced in column-major order and consumed in row-major order. Transposing the data when it is distributed from producers to consumers, shall ease the work to be performed by the application and increase data access locality.

The TerrSysMP application from Juelich fits into the *Transpose.3* motivation example described above.

3.2 Pack

Description:

- Copy non-consecutive data into a consecutive data layout using a data mapping function (e.g. non-unit stride access, indirect access).

Motivation examples:

- *[Pack.1]* In C/C++ (or Fortran) sequential applications, there is a loop that reads non-consecutive array elements that are separated by a non-unit stride (e.g. stride-3 access, indirect access). This may a negative impact on performance because costly gather SIMD instructions may be required. Correspondingly, the same stands for write operations on non-consecutive array elements that require costly scatter SIMD instructions.
- *[Pack.2]* In the scope of MPI communications, it is necessary to send non-consecutive array elements from one MPI process to another. To handle this use case, the MPI provides an interface that requires the base address of the first element, the number of elements to be sent and the stride that separates two array elements. It also provides interfaces to declare the data structure so its transmission can be optimized by packing and unpacking it. However, oftentimes users find them too complex to use and either perform these operations manually or skip them altogether.
- *[Pack.3]* In the scope of porting codes to GPUs, efficient data transfers requires data allocated in consecutive memory locations. In real applications that use complex data types based on structs and pointers, it is common practice to pack the target data in temporary data structures to avoid the redesign of the data layout, which usually impacts on the application as a whole. In addition, GPU programming APIs like OpenMP and OpenACC are designed to handle consecutive data efficiently.
- *[Pack.4]* A similar situation arises in applications that use external libraries (e.g. scientific linear algebra routines), which usually require consecutive data as input.

The Hydro application from CEA, particularly the Halo exchange, fits into the *Pack.2* motivation example described above.

3.3 Split

Description:

- Split a data set into non-overlapping subsets.

Motivation examples:

- *[Split.1]* In a CPU-GPU memory environment, split a large array that does not fit in the GPU memory in sub-arrays and offload to the GPU one sub-array at a time. This is typically known as “tiling” in the scope of loop transformations for software performance optimization.
- *[Split.2]* In a CPU memory environment, split a large array that does not fit in the cache memory in sub-arrays to exploit data locality in the CPU memory (e.g. cache-awareness). It is typically know as “tiling” or “blocking”.
- *[Split.3]* In a complex memory environment, distribute disjoint chunks of an array over separate memory spaces managed by different tasks (either processes or threads). A well-known example in the scope of MPI applications is to scatter an array across multiple MPI processes using a block data distribution (also applies to cyclic and block-cyclic).

The SIRIUS application from ETH Zurich/CSCS fits into the *Split.1* motivation example described above.

4 Handling Transformations in Maestro

Two major execution modes are presented by Maestro, as reported in D3.1 [1]:

- a Management mode: A 4-step protocol ('Give-Take' semantics) for the giving of data objects to Maestro and the taking of data objects back from Maestro
- a Transformation mode that is concerned with layout-sensitive transformations
- a Hybrid API, for those users with a clear interest in both layout-insensitive management, and layout-sensitive transformation

This in turn translates into two main ways of handling data transformations in Maestro:

- Explicit (unpooled): Corresponds to Maestro's transformation execution mode. The user triggers the transformation via the API call `mstro_transform_layout`, which necessarily operates on a non-pooled CDO that is users' responsibility.
- Implicit (pooled): Corresponds to Maestro's management mode. In a producer/consumer paradigm, the transformation is triggered by Maestro between the excution of an OFFER call and the execution of a DEMAND call. Maestro will trigger a transformation if there is a (rightful wrt. Maestro semantics) mismatch between the OFFERed layout and the DEMANDed layout, both of which need to be explicited to Maestro through CDO attributes.

In both cases the execution of the data transformations is fully managed by the Maestro middleware.

The Maestro middleware has been designed to easily enable implementing data transformations. The current release of the Maestro middleware supports *Transpose* (*Pack* and *Split* are not implemented in this release). The following subsections present a working example of a transpose transformation, a potential implementation of a pack transformation and how Maestro also provides tooling like Parallelware to help developers to write Maestro-enabled applications.

4.1 Explicit Transpose transformation example

For illustrative purposes consider the Listing 1, which shows a Maestro-enabled source code example written in C to transform a 10×8 matrix into its transposed 8×10 matrix.

Two matrices are declared at line 31. For each one of them, a CDO is created at lines 51 and 69. Both CDOs have the same properties except for the

pointer to the matrix (*MSTRO_ATTR_CORE_CDO_RAW_PTR* is set to *A* and *transposed* at lines 54 and 73, respectively), the transposed dimensions of the two matrices (*MSTRO_ATTR_CORE_CDO_LAYOUT_DIM_SIZE* at lines 61 and 79) and the data layout ordering attribute *MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER* which states that the first matrix has a row-major layout (**_ROWMAJOR* at line 62) while the second requires a column-major layout (**_COLMAJOR* at line 62). These different layout order attributes configures the *Transpose* data transformation executed by *mstro_transform_layout* at line 85, which fills the destination *transposed* matrix with the transposed version of matrix *A*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <maestro.h>
4  #include <transformation.h>
5
6  const int M = 10;
7  const int N = 8;
8
9  void print_matrix_info(const char *desc, int *array, int rows, int
       cols)
10 {
11     printf("\n  %s\n===============\n", desc);
12
13     printf("Contiguous memory: ");
14     for (int i = 0; i < rows * cols; i++)
15         printf("%d ", array[i]);
16     printf("\n");
17
18     printf("\n%dx%d matrix:\n", rows, cols);
19     for (int i = 0; i < rows; ++i)
20     {
21         for (int j = 0; j < cols; ++j)
22             printf("%d ", array[i * cols + j]);
23         printf("\n");
24     }
25
26     printf("\n");
27 }
28
29 int main()
30 {
31     int A[M][N],           // Source matrix
32         transposed[N][M]; // Destination matrix
33
34     // Fill the matrix with incremental values
35     for (int i = 0; i < M; ++i)
36         for (int j = 0; j < N; ++j)
37             A[i][j] = (i * N) + j;
38
39     mstro_init("small-matrix", "transpose", 0);
40
41     // Information common to both matrices
42     size_t elem_size = sizeof(int);
43     size_t size = M * N * elem_size;
44     size_t num_dims = 2;
```

```
45
46      mstro_cdo original_cdo = NULL;
47      mstro_cdo_declare("original", MSTRO_ATTR_DEFAULT, &original_cdo
        );
48
49      // Common attributes
50      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_RAW_PTR, &A);
51      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE, &size);
52      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &elem_size);
53      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &num_dims);
54
55      // Matrix A specific attributes
56      size_t dims_original[] = {M, N};
57      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, &dims_original);
58      size_t layout_original =
        MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER_ROWMAJOR;
59      mstro_cdo_attribute_set(original_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER, &layout_original);
60
61      mstro_cdo_declaration_seal(original_cdo); // Mark CDO
        declaration as complete
62
63      /* Create an unpooled CDO to specify how the data must be
        transformed and stored into the 'transposed' matrix */
64      mstro_cdo transposed_cdo = NULL;
65      mstro_cdo_declare("unpooled", MSTRO_ATTR_DEFAULT, &
        transposed_cdo);
66
67      // Common attributes
68      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE, &size);
69      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_RAW_PTR, &transposed);
70      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &elem_size);
71      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &num_dims);
72
73      // transposed matrix specific attributes
74      size_t dims_transposed[] = {dims_original[1], dims_original
        [0]}; // NxM
75      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, &dims_transposed);
76      size_t layout_transposed =
        MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER_COLMAJOR;
77      mstro_cdo_attribute_set(transposed_cdo,
        MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER, &layout_transposed);
78      mstro_cdo_declaration_seal(transposed_cdo); // Mark CDO
        declaration as complete
79
80      // Perform the transformation: implicitly, it is a
```

```
    transposition due to the different layout orders
81   mstro_transform_layout(original_cdo, transposed_cdo,
82                          original_cdo->attributes, transposed_cdo
    ->attributes);
83
84   print_matrix_info("Original", A, dims_original[0],
    dims_original[1]);
85   print_matrix_info("Transposed", transposed, dims_transposed[0],
     dims_transposed[1]);
86
87   mstro_cdo_dispose(original_cdo);
88   mstro_cdo_dispose(transposed_cdo);
89
90   mstro_finalize();
91 }
```

Listing 1: Matrix transposition example

4.2 Extending Maestro with new transformations: explicit Pack transformation example

The Maestro API can easily be extended to support new transformations. Listing 2 shows a C-code example of how a pack transformation could be performed using the Maestro API. As exhibited, no actual new API calls should be required to implement the support for this transformation in Maestro as its design allows to rely on merely extending the attribute set. Note that this is not working code as this transformation is not part of the initial implementation.

Attributes names and values defined between lines 7-12 are the new attributes that would be added to Maestro to support the pack transformation. The example defines two matrices: *A* of size 1000×1000 and *A_packed* of size 200×200. The CDO for the first matrix is defined without any special attributes regarding the transformation (see line 30). All this information is also set for the second CDO which also includes the declaration of the pack transformation as well as a stride of 5 elements in each dimension (see lines 55 and 56). Invoking *mstro_transform_layout* for these CDOs at line 61, causes the data of *A* to be packed into the 200×200 *A_packed* variable.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <maestro.h>
4  #include <transformation.h>
5
6  // Potential new Maestro attributes to support the pack
       transformation
7  #define MSTRO_ATTR_CORE_CDO_LAYOUT ".maestro.core.cdo.layout"
8  #define MSTRO_ATTR_CORE_CDO_LAYOUT_PACK 1
9  #define MSTRO_ATTR_CORE_CDO_LAYOUT_PACK_STRIDES ".maestro.core.cdo.
       layout.strides"
10
11 const int M = 1000;
```

```
12 const int N = 1000;
13
14 int main()
15 {
16     int A[M][N];
17
18     mstro_init("examples", "pack", 0);
19
20     // Declaration of the original CDO that is linked with the an
   existing variable in the application code
21     size_t elem_size = sizeof(double);
22     size_t size = M * N * elem_size;
23     size_t num_dims = 2;
24     size_t dims[] = {M, N};
25
26     mstro_cdo original_cdo = NULL;
27     mstro_cdo_declare("original", MSTRO_ATTR_DEFAULT, &original_cdo
   );
28     mstro_cdo_attribute_set(original_cdo,
   MSTRO_ATTR_CORE_CDO_RAW_PTR, &A);
29     mstro_cdo_attribute_set(original_cdo,
   MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE, &size);
30     mstro_cdo_attribute_set(original_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &elem_size);
31     mstro_cdo_attribute_set(original_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &num_dims);
32     mstro_cdo_attribute_set(original_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, &dims);
33     mstro_cdo_declaration_seal(original_cdo); // Mark CDO
   declaration as complete
34
35     // Declaration of the destination CDO that is linked with the a
      new variable in the application code
36     double A_packed[M / 5][N / 5];
37     size_t packed_elem_size = sizeof(double);
38     size_t packed_size = (M / 5) * (N / 5) * elem_size;
39     size_t packed_num_dims = 2;
40     size_t packed_dims[] = {M / 5, N / 5};
41     size_t packed_layout = MSTRO_ATTR_CORE_CDO_LAYOUT_PACK;
42     size_t packed_strides[] = {5, 5};
43
44     mstro_cdo packed_cdo = NULL;
45     mstro_cdo_declare("unpooled", MSTRO_ATTR_DEFAULT, &packed_cdo);
46     mstro_cdo_attribute_set(packed_cdo, MSTRO_ATTR_CORE_CDO_RAW_PTR
   , &A_packed);
47     mstro_cdo_attribute_set(packed_cdo,
   MSTRO_ATTR_CORE_CDO_SCOPE_LOCAL_SIZE, &packed_size);
48     mstro_cdo_attribute_set(packed_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &packed_elem_size);
49     mstro_cdo_attribute_set(packed_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &packed_num_dims);
50     mstro_cdo_attribute_set(packed_cdo,
   MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, &packed_dims);
51     mstro_cdo_attribute_set(packed_cdo, MSTRO_ATTR_CORE_CDO_LAYOUT,
      &packed_layout);
52     mstro_cdo_attribute_set(packed_cdo,
```

```
    MSTRO_ATTR_CORE_CDO_LAYOUT_PACK_STRIDES, &packed_strides);
53   mstro_cdo_declaration_seal(packed_cdo); // Mark CDO declaration
     as complete
54
55   // EXPLICIT transformation based on the pack information on the
     destination CDO
56   mstro_transform_layout(original_cdo, packed_cdo,
57                          original_cdo->attributes, packed_cdo->
    attributes);
58
59   mstro_cdo_dispose(original_cdo);
60   mstro_cdo_dispose(packed_cdo);
61   mstro_finalize();
62
63   double sum = 0;
64
65   // Compute a scalar reduction using consecutive memory access
66   for (int i = 0; i < M / 5; i++)
67       for (int j = 0; j < N / 5; j++)
68           sum = sum + A_packed[i][j];
69 }
```

Listing 2: Pack transformation example

4.3 Guided implementation of Maestro-enabled transformations

The Maestro stack provides software components to facilitate the development of Maestro-enabled applications. More specifically, the data access semantics work in WP4 (T4.1) provides a new software tool based on Parallelware's static code analysis capabilities to extract information about the data memory layout (e.g. statically allocated arrays, dynamically allocated arrays), how it is accessed in the code (particularly loops) through memory access patterns (e.g. sequential, strided, indirect, row-major) and the data flow of the program (e.g. temporary computations). In the Parallelware tools, this information is provided to the developer through an optimization report that contains human-readable actionable hints tailored for Maestro.

The Maestro-enabled codes described above in this section for the *Transpose* and *Pack* data transformations show that Parallelware tools will need to provide human-readable actionable hints for the following:

- Identifying CDO opportunities: point the developer to those source code variables that may benefit from delegating their management to the Maestro middleware.
- Identifying data transformation opportunities: point the developer to uses of variables that may benefit from data transformations (e.g. colum-major to row-major access inside loops).
- Identifying data transport opportunities: point the developer to uses of

variables that may benefit from data transport (e.g. producer/consumer dependencies between programs).

- Suggest insertions of Maestro API calls: point the developer to the point of the code where a CDO can be defined and list the specific Maestro calls needed for this purpose. In a similar manner, Maestro API calls to do explicit data transformations (*mstro_transform_layout*) or implicit data transformations (OFFER/DEMAND).

Note that this is work in progress following a co-design process while preparing demonstrators based on the data transformations supported in the Maestro middleware.

5 Resource Description

The main goal of data mapping and transformations is to enable optimized execution of workflows. One of the primary methods of achieving this is to opportunistically use available non-critical resources to perform the transformations or ensure that the objects are available to the applications on a quick path when required. To achieve this, it is important for us to be able to describe the resources of a system and their performance both topographically and temporally.

There are a variety of resource discovery and description resources available. The most popular of them are `hwloc` and `LIKWID` among others. Most of these tools are node level tools. The main objective of these tools is to describe resources as they exist in a system. A Maestro-enabled workflow requires to do resource management and allocation, through a workflow manager. For this purpose, we also need tools that can annotate resources with information about the intended use of resources in a workflow. We also need facilities to be able to manipulate a resource description to select a subset of resources and pass them to applications. To accommodate these requirements, we have designed a resource description language called `radle` and a library with the same name to handle the requirements of the description.

Resources on an HPC system are generally allocated by a job scheduler, and hence allocations are completely dynamic in nature. This necessitates to identify a resource description topology at job runtime. Such identification of resources is inexpensive, and therefore, does not put a significant overhead over the tasks/jobs of a workflow. However, deriving performance metrics is expensive to compute, furthermore, they depend on a host of factors including the workload and the state of system at any given time among other things. Due to the expensive nature of metrics and the shared nature of compute systems, it is not feasible to for the metric to deterministically predict the performance. Instead, we use them as approximate values that are mostly true over a long term.

The dependence on workload implies that different systems might need different metrics in their performance description. For this reason, we have separated the resource description and topology from the performance description of the resources. We have defined a resource description language (RDL) to enumerate all the resources in the system and describe their topology. For the performance metrics, we are planning to define a performance schema, which is based on top of resource description language.

5.1 Related Work

`hwloc` [6] is a node based portable abstraction which is able to provide a hierachial topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or

GPUs.

LIKWID [7] stands for “Like I Knew What I’m Doing.” It is an easy to use yet powerful command line performance tool suite for the GNU/Linux operating system. While the focus of LIKWID is on x86 processors, some of the tools are portable and not limited to any specific architecture. For the upcoming release, LIKWID has been ported to ARMv7/v8 and POWER8/9 architectures as well as for Nvidia GPU co-processors.

Resource description is also implicitly available in a multitude of tools. Workload managers such as SLURM require a list of resources to allocate the scheduled jobs. Additionally, workflow managers provide a representation of the available resources that enable workflows to assign lists of resources to a chosen task.

When working with Maestro-aware workflows, there’s a requirement to look at the resources available on a system to generate workflows which are optimized for the system. This requires a resource description system which has the following requirement:

1. Allow for discovery of resources on a complete systems, rather than a single node.
2. Allow for selecting a subset of resources and treat the resulting set as an independent description of resources in its own right.
3. Allow for splitting of a resource into multiple resources.
4. Allow for annotating resources with tags which can then be used by workflow translator to properly modify the workflow.
5. Allow for tools to store to file and load the resource description from files.
6. Allow for storing additional data like metrics based on performance schema for each resource.

Due to the nature of most resource description/management tools, we find that some of the above tasks, especially tasks like annotation and manipulation of resources are not readily available in other tools. This has been a major motivation to create a resource description language (radle) which has been described further in D4.3.

5.2 Resource Description Tools in Maestro

There are two tools, both currently in development, to deal with resource description in Maestro, namely `radle` and `mapi`. `radle` is a tool to read, write and manipulate the description of resource in a system and the topology of those resources in terms of the resource description language (RDL). `mapi` is planned to compute and store various performance metrics.

6 Software Release

This section provides some information on the released software as part of this deliverable.

6.1 Maestro-core

Source code	`https://gitlab.version.fz-juelich.de/maestro/maestro-core`
Tag	`d3.3`
Prerequisites	• C compiler capable of understanding C11 • POSIX threads • *refer to INSTALL.MD for more info*
License	BSD 3-Clause "New" or "Revised" License

A basic code excerpt and example of transformation is shown in Figure 1. This is a synthetic benchmark located in the Maestro repository `./tests/check_layout.c`. This unit test also shows example of explicit transformation usage.

```
/* Consumer side */
mstro_cdo_declare(name, MSTRO_ATTR_DEFAULT, &cdo_dst);
mstro_cdo_attribute_set(cdo_dst,
            MSTRO_ATTR_CORE_CDO_LAYOUT_ELEMENT_SIZE, &elsz);
mstro_cdo_attribute_set(cdo_dst,
            MSTRO_ATTR_CORE_CDO_LAYOUT_NDIMS, &ndims);
mstro_cdo_attribute_set(cdo_dst,
            MSTRO_ATTR_CORE_CDO_LAYOUT_DIMS_SIZE, dimsz);
mstro_cdo_attribute_set(cdo_dst,
            MSTRO_ATTR_CORE_CDO_LAYOUT_ORDER, &patt_dst);
mstro_cdo_declaration_seal(cdo_dst);
mstro_cdo_require(cdo_dst);
mstro_cdo_demand(cdo_dst);
 /* at this point automatic transformation was performed */
```

Figure 1: Example of implicit usage of Maestro transformations. Attributes fed to the CDO will allow Maestro to detect transformation opportunity, in the case where the produced CDO layout differs.

6.2 Resource Description

Source code	`https://gitlab.version.fz-juelich.de/maestro/radle`
Tag	D3.4
Prerequisites	• C compiler capable of understanding C11 • *refer to readme.md for more info*
License	Apache-2.0

7 Concluding Remarks

This document is a report accompanying a software release targeting transformations and map optimisations. To that end we have described the data mapping in Maestro, referencing the physical resources in different components in the Maestro framework. Additionally, we have introduced a set of transformations and the mechanisms by which Maestro potentially handles them. Finally, as a requirement of mapping and allocating operations such as transformations, we introduced a resource description library. Future work as part of WP4 will integrate some of the work introduced here as part of the workflow handling mechanisms. Additionally, WP6 will create demonstrations and quantify their impact.

References

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

[2] C. Haine, U. Haus, and A. Tate, “D3.2 low-level middleware reference implementation,” 2019.

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

[4] J. Novo and M. Arenaz, “D4.1 data access semantics, initial specification and release of software tools,” 2019.

[5] S. Wu, G. Umanesan, and S. Narasimhamurthy, “D5.2 storage data mapping and memory integration design,” 2019.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, “hwloc: A generic framework for managing hardware affinities in hpc applications,” in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 180–186, 2010.

[7] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, “Rrze-hpc/likwid: likwid-5.1.1,” Mar. 2021.