A big data analytics framework for scientific data management

Sandro Fiore[1,2*], Cosimo Palazzo[1,2], Alessandro D'Anca[1], Ian Foster[3], Dean N. Williams[4], Giovanni Aloisio[1,2]
[1]Centro Euro-Mediterraneo sui Cambiamenti Climatici, Italy
[2]Università del Salento, Lecce, Italy
[3]Computation Institute, Argonne National Lab and University of Chicago, Chicago, IL 60637, USA
[4]Lawrence Livermore National Laboratory, Livermore, CA, USA
*sandro.fiore@unisalento.it

Abstract—**The Ophidia project is a research effort addressing big data analytics requirements, issues, and challenges for eScience. We present here the Ophidia analytics framework, which is responsible for atomically processing, transforming and manipulating array-based data. This framework provides a common way to run on large clusters analytics tasks applied to big datasets. The paper highlights the design principles, algorithm, and most relevant implementation aspects of the Ophidia analytics framework. Some experimental results, related to a couple of data analytics operators in a real cluster environment, are also presented**.

Keywords—big data; data analytics; parallel I/O; eScience

I. Introduction

In several eScience domains [1,4] the analysis and mining of large volumes of data is increasingly becoming central to scientific discovery. The multidimensionality, volume, complexity, and variety of scientific data need specific solutions able to (i) support the management and analysis of large datasets [5,6], (ii) provide array-based functionalities, and (iii) support the scientific analysis process through parallel solutions able to deliver results in (near) real-time [7].

Unfortunately, the tools, libraries and frameworks used in many scientific domains are mostly sequential today. This barrier strongly limits scientific productivity, slowing down or entirely preventing data analysis activities on large datasets.

In the climate science context few tools exploit parallel paradigms for analysis and visualization [8,11], making urgent the need for new (big) data intensive parallel frameworks. Relevant projects in this area like ParCAT [12], ParNCL, and ParGAS [13], share this scientific goal, but provide a toolkit rather than a framework.

Scientific data needs data warehouse-like [14,15] platforms to perform data analytics. However, current general-purpose (enterprise-based) *On Line Analytical Processing* (OLAP) systems are not suitable for eScience due to their limited scalability, a lack of support for parallel solutions and for the analysis of large volumes of data, and their inadequate support for numerical and mathematical libraries. Additionally, specific array-based support (which is key for n-dimensional scientific data) is not sufficient both in terms of data types and data analysis primitives to address scientist needs and use cases.

The Ophidia project is addressing most of these challenges, in the context of a research effort addressing big data analytics issues for eScience. The key test case for Ophidia concerns the analysis of global climate simulations produced at the Euro-Mediterranean Centre on Climate Change, in the context of the international Coupled Model Intercomparison Project Phase 5 (CMIP5) [16,17]. On a daily basis, climate scientists need to run on large datasets data transformation, analysis, and processing tasks to reduce data, extract time series, compute ensemble means, run scenarios, perform model inter-comparisons, infer statistical indicators, and perform re-gridding. Most of these tasks are today performed via batch and sequential command line interfaces and tools that cannot provide needed answers efficiently or in real-time. In contrast, the Ophidia platform aims to provide an OLAP-like data management solution providing (through parallel "data kernels" running on HPC machines) real-time answers to scientists questions. An important building block of the Ophidia project is the *analytics framework*, which is also the key topic of this paper.

The remainder of this work is organized as follows. Section II describes the Ophidia architecture. Section III presents the Ophidia analytics framework, discussing in detail the functional and non-functional requirements that have driven its design, the analytics framework algorithm and implementation, the deployment diagram, a comprehensive set of operators, the internals of three operators, and some experimental results on a 12-node IBM iDataplex cluster. Finally, Section IV draws conclusions and highlights future work.

II. The Ophidia Architecture

As depicted in Fig. 1, the Ophidia architecture consists of several layers: the storage system, the I/O nodes, the OphidiaDB, the compute nodes, and the Ophidia server.

The *storage system* represents the hardware resource managing the data store of the Ophidia architecture. It consists of a set of disks storing *datacubes*.

The storage system is accessed via the *I/O nodes*, which host a set of I/O servers responsible for the parallel I/O with the underlying storage system. As described elsewhere [18], the current implementation of a single I/O server relies on the MySQL relational DBMS, which has been extended to support array-based data type and primitives. The *datacubes* in the MySQL databases are organized in a hierarchical structure and are partitioned in several tables (called *fragments*) distributed across multiple databases and MySQL servers. The data in the fragments are multidimensional arrays stored according to the Ophidia internal storage model, which exploits a key-value approach. A preliminary set of Ophidia array-based primitives, the storage model, and the hierarchical data structure have been described elsewhere [18] with some examples and use cases.

Metadata is stored in the *OphidiaDB*, a relational database running on top of a MySQL server.

The *compute nodes* are machines used by the Ophidia infrastructure to run the data analytics framework, which is based on a set of (parallel and sequential) operators providing the most relevant data and metadata datacube primitives. Each operator running on the compute nodes triggers a set of array-based primitives on the I/O nodes to analyze, transform and process a datacube as a whole. In more detail, the array based primitives and datacube operators work at two different levels: while the former focus on n-dimensional arrays stored in a single relational table (a single *fragment*), the latter focus on an entire datacube involving the complete set of associated fragments. Another important difference is that array-based primitives run on the I/O nodes (exploiting a software-based *active storage* approach), while datacube operators run on the compute nodes. The currently available array-based primitives are sequential, while most datacube operators have a parallel implementation.

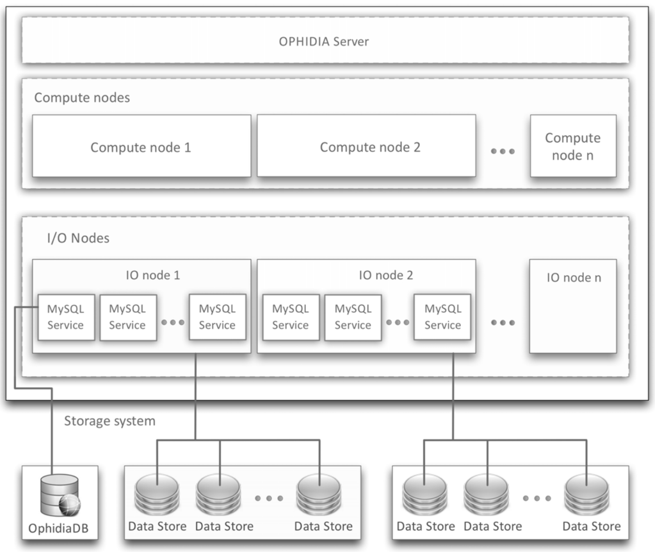

Fig. 1. The Ophidia architecture

Finally, the *Ophidia server* represents the front-end of the system. It is a web service implemented in gSOAP and exposing a standard WS-I interface. The server manages both the user authentication and authorization. It also provides complete session management support to (i) track existing user sessions, (ii) limit the maximum number of active concurrent sessions per user, and (iii) manage session lifetime, caching and logging. The Ophidia server is strongly tied to the Ophidia analytics framework. Indeed, it is responsible for submitting the analytics operators (through the analytics framework) on the compute nodes infrastructure, thus dispatching jobs and managing queues and priorities.

In the sections that follow, we describe the Ophidia analytics framework and its operators. We will review its design, present a deployment diagram, classify and list operators, describe implementation details for some operators, and present some benchmark results.

III. The Ophidia Analytics Framework

As stated before, the analytics framework in the Ophidia system is responsible for *atomically processing and manipulating datacubes, by providing a common way to run distributive tasks on large set of fragments*. The following subsections present and discuss in detail several aspects of the Ophidia design and the implementation.

A. Functional and non-functional requirements

The following major functional requirements drove the design of the Ophidia analytics framework:

- *plugin-based infrastructure*: the analytics framework has to provide a plugin-based infrastructure to make straightforward the implementation, integration, monitoring, and execution of new operators. To this end, we design a well-defined and general API with which each operator must comply, and implement a runtime analytics framework execution environment;
- *data and metadata operators*: we must provide a wide set of core primitives to manipulate both the data contained in datacubes (e.g., sub-setting, duplication, transformation, export/import, array-based analysis, data reduction) and the *metadata* associated with datacubes (e.g., list of datacubes, datacube size, datacube number of elements, datacube provenance information);
- *domain-oriented and domain-agnostic operators*: the framework has to support both domain-oriented and domain-agnostic operators. Import and export of a datacube into/from the Ophidia data store can represent two cases of domain-oriented operators (e.g., import/export of NetCDF [17] files to/from the Ophidia data store in the climate change domain);
- *logging and bookkeeping support*: the analytics framework should provide logging and bookkeeping for every operator. Thus, we include these capabilities in the runtime analytics framework execution environment rather than in each operator module.

In addition, we also took into account the following major non-functional requirements in the Ophidia analytics framework design:

- *dynamicity*: adding new operators to the system must not imply rebuilding the entire framework (dynamic approach);
- *flexibility*: the framework API must be sufficiently general and flexible to allow the implementation of whatever task running on the Ophidia infrastructure in the form of a framework operator;
- *robustness*: the framework has to be able to properly react to and cope with errors and exceptions during the execution of any operator;
- *extensibility*: the system must be easily extensible to keep rapidly evolving in terms of set of functionalities through the implementation of new operators. Besides the core/system operators available in the Ophidia middleware, the framework API has to be easy and clear enough to facilitate the implementation of operators by the end-users. This will help making this effort community based and incorporating new functionalities in the form of new operators;

- *paradigm-agnostic*: the framework must not be tied to a specific execution paradigm. It has to allow the implementation of sequential, parallel, and distributed operators. Thus, paradigm-specific instructions must be incorporated in operator implementations rather than in the framework itself;
- *efficiency*: the runtime analytics framework execution environment has to be lightweight and efficient without representing a bottleneck for the entire system.

B. Analytics framework design

Starting from the aforementioned requirements, the Ophidia analytics framework has been designed and implemented as an algorithm based on seven functions, plus a wide set of operators implemented as dynamic libraries. To address flexibility and deal with both complex and simple operators, three functions have been defined as mandatory and four ones as non-mandatory. Each operator must implement the three mandatory functions and may choose to implement the other four ones as well.

To dynamically bind an operator at runtime, the analytics framework exploits the GNU Libtool *libltdl* library, which *hides the complexity of using shared libraries behind a consistent, portable interface* [19]. Such an approach makes the framework lightweight, simple, and manageable. The set of interfaces has been designed to guarantee strong code modularity and separation of concerns.

The framework manages status and logging information; thus, operator implementations need not be concerned with these issues. Conversely, the framework does not include any MPI, OpenMP, or distribute calls/APIs since the adopted paradigm (sequential, parallel, or distributed) is operator-specific. Even though the Ophidia software is being tested today on a single cluster configuration and most current operators are MPI-based, in the future the same software could run on a distributed set of clusters, by exploiting new operators based on distributed libraries and middleware.

The seven interfaces reflect the following general tasks: environment *setup/clean-up*, operator *init/destroy*, task *distribution/reduction* and operator *execution*.

The Ophidia analytics framework algorithm goes through all of these steps, as depicted in the following pseudo-code. In this pseudo-code, the three mandatory calls are shown in boldface.

```
int oph_analytics_framework_algorithm (char* input_args) {
  oph_setup_env (char* input_args,
                 oph_operator_struct * handle);
  oph_init (oph_operator_struct * handle);
  oph_distribution (oph_operator_struct * handle);
  oph_execute_operator (oph_operator_struct * handle);
  oph_reduction (oph_operator_struct * handle);
  oph_destroy (oph_operator_struct * handle);
  oph_cleanup_env (oph_operator_struct * handle);
  return 0;
}
```

To address readability, this pseudo-code does not include any logging, bookkeeping or error management sections/instructions. For a more complete understanding of the analytics framework algorithm, it is important to look at the *oph_operator_struct* data structure (used by all of the framework functions) definition:

```
typedef struct {
  char *operator_type;
  void *operator_handle;
  int proc_number;
  int proc_rank;
  char *lib;
  void *dlh;
} oph_operator_struct;
```

In addition to a few general and intuitive attributes regarding all operators, the *oph_operator_struct* always includes a *void* operator_handle* pointer to an internal handle containing any additional attributes needed by the target operator. This feature makes the analytics framework interfaces highly flexible, allowing for each operator the definition and management of an operator-specific set of attributes.

Another important property of the analytics framework algorithm lies in its symmetry (*oph_init/oph_destroy*, *oph_setup_env/oph_clean-up_env*, *distribution/reduction*).

Moreover, note that in a generic operator implementation, only *oph_setup_env*, *oph_cleanup_env* and *oph_execute_operator* are mandatory. Thus, the implementation of simple operators can be lightweight.

Due to the framework generality, each interface/task objective has been properly defined to better guide the user through the correct operator implementation process. In particular:

- *oph_setup_env* is a mandatory task responsible for setting up the operator runtime environment (e.g., allocating the operator-specific structure pointed by the *operator_handle* pointer). The *oph_cleanup_env* (mandatory task) does the opposite of the *oph_setup_env*, releasing the memory or deactivating modules initialised in the *oph_setup_env*;
- *oph_init* is a non-mandatory task responsible for activating a set of preliminary actions to be executed before the distribution task (e.g., the creation of a new datacube entry in the OphidiaDB); conversely the *oph_destroy* task finalises/closes those actions;
- *oph_distribution* task is a non-mandatory task in charge of distributing the load among the available *workers* (e.g., processes, threads, agents). Scheduling strategies and policies are implemented as part of this task; on the other hand, the *oph_reduction* is the non-mandatory and complementary task in charge of gathering/reducing data and results from all the workers. Reduction strategies and policies are implemented as part of this task;
- *oph_execute_operator* is a mandatory task responsible for running the *core* operator function. It is the central and more intuitive task of the framework. It exploits all

the memory structures, connections and information initialised and distributed in the previous tasks. It is also responsible for providing the main output of the operator.

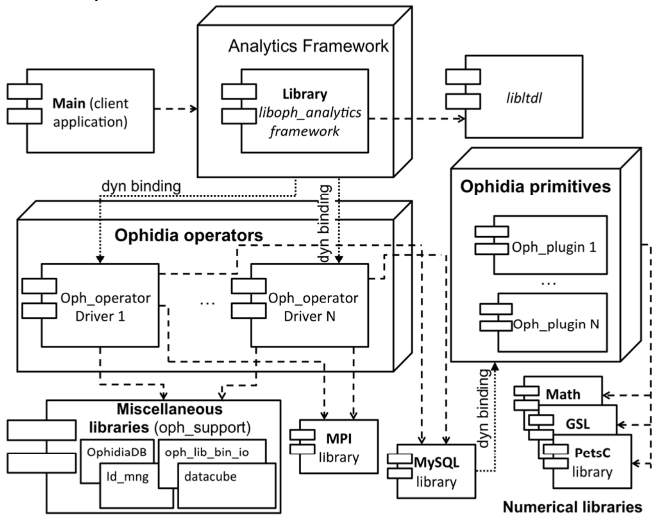

Fig. 2. The deployment diagram of the Ophidia analytics framework

C. *Analytics framework implementation*

We next review some important aspects of the implementation. The analytics framework is based on the pseudo-code presented in the previous section and is implemented in C language. In more detail, the seven functions of the Ophidia API represent the static part of the *analytics framework* library. They basically deal with the *libltdl*, to properly manage the dynamic libraries binding associated to each operator. It is important to remark that they are not directly available to developers. Indeed, the analytics framework library supplies the developers with just the following single algorithm function:

```
int oph_analytics_framework_algorithm (char* input_args)
```

which is the only one called by each analytics framework client application built on top of the Ophidia software. This interface has only one input parameter (*input_args*) consisting of a semicolon-separated list of *argument=value* strings. It is mandatory to have an *operator* argument in the *input_args* string so that the framework can properly identify the target operator and the associated dynamic library.

The deployment diagram reported in Fig. 2 shows that each operator exploits several libraries. Two of the most relevant ones in the current release of the Ophidia analytics framework (v1.0) are the Message Passing Interface (MPI) and the MySQL client. The former has been used to implement a set of parallel *data* operators, whereas the latter has been exploited in the *data* and *metadata* operators code to establish the connection with the I/O servers (MySQL DBMSs) and trigger array-based primitives (SQL statements). Moreover, the diagram highlights the adoption of several numerical libraries in the Ophidia framework (e.g., GSL [20] and PETSc [21]) to build array-based primitives.

Since both the operators and the primitives are dynamic libraries, new ones can be added to the system without doing any change to (or rebuild) the existing code. This makes the entire software infrastructure (both in terms of datacube operators and array-based functionalities) dynamic and easily extensible by external users, thus addressing both the extensibility and the dynamicity non-functional requirements.

Even though the primary goal of the framework is to support data operators for analysing and processing datacubes, the framework is also extensively used to run *metadata* operators—usually single process tasks accessing to the OphidiaDB to get metadata information and implementing only the three mandatory framework interfaces.

TABLE I. THE OPHIDIA DATA AND METADATA OPERATORS

OPERATOR NAME	OPERATOR DESCRIPTION
Operators "Data processing" – Domain-agnostic	
OPH_APPLY(*datacube_in*, *datacube_out*, *array_based_primitive*)	Creates the *datacube_out* by applying the *array-based primitive* to the *datacube_in*
OPH_DUPLICATE(*datacube_in*, *datacube_out*)	Creates a copy of the *datacube_in* in the *datacube_out*
OPH_SUBSET(*datacube_in*, *subset_string, datacube_out*)	Creates the *datacube_out* by doing a sub-setting of the *datacube_in* by applying the *subset_string*
OPH_MERGE(*datacube_in*, *merge_param*, *datacube_out*)	Creates the *datacube_out* by merging groups of *merge_param* fragments from *datacube_in*
OPH_SPLIT(*datacube_in*, *split_param*, *datacube_out*)	Creates the *datacube_out* by splitting into groups of *split_param* fragments each fragment of the *datacube_in*
OPH_INTERCOMPARISON (*datacube_in1, datacube_in2, datacube_out*)	Creates the *datacube_out* which is the element-wise difference between *datacube_in1* and *datacube_in2*
OPH_DELETE(*datacube_in*)	Removes the *datacube_in*
Operators "Data processing" – Domain-oriented	
OPH_EXPORT_NC (*datacube_in*, *file_out*)	Exports the *datacube_in* data into the *file_out* NetCDF file.
OPH_IMPORT_NC (*file_in*, *datacube_out*)	Imports the data stored into the *file_in* NetCDF file into the new *datacube_in* datacube
Operators "Data access"	
OPH_INSPECT_FRAG (*datacube_in, fragment_in)*	Inspects the data stored in the *fragment_in* from the *datacube_in*
OPH_PUBLISH(*datacube_in*)	Publishes the *datacube_in* fragments into HTML pages
Operators "Metadata"	
OPH_CUBE_ELEMENTS (*datacube_in*)	Provides the total number of the elements in the *datacube_in*
OPH_CUBE_SIZE (*datacube_in*)	Provides the disk space occupied by the *datacube_in*
OPH_LIST(void)	Provides the list of available datacubes.
OPH_CUBEIO(*datacube_in*)	Provides the provenance information related to the *datacube_in*
OPH_FIND(*search_param*)	Provides the list of datacubes matching the *search_param* criteria

As a general rule, all interactions with the Ophidia system come in the form of operators. Yet, some operators run *system* (parallel and/or sequential) tasks that do not interact with the Ophidia data/metadata infrastructure at all. Some examples are operators that clean up directories on cluster nodes (I/O or compute), that perform RPMS update/installation, or that check system logs. This flexibility illustrates the generality of the designed framework, which is able to go even beyond the data/metadata analytics infrastructure needs.

The current version of the Ophidia framework provides about forty operators (parallel and sequential). The parallel ones are all concerned with *data* (both domain-oriented and domain-agnostic) and the sequential ones are all concerned with either *metadata* or *system*. Table I lists, describes, and classifies 16 of these operators.

D. The internals of three relevant operators

This section presents in detail the OPH_APPLY (Fig. 3), OPH_PUBLISH (Fig. 5) and OPH_CUBE_ELEMENTS (Fig. 4) operators from an implementation point of view. In each case, we discuss how the seven steps of the analytics framework are mapped onto the seven functions described before. We also review the main differences among the operators. OPH_APPLY (*data processing*), OPH_PUBLISH (*data access*), and OPH_CUBE_ELEMENTS (*metadata management*) are all parallel operators implemented using the Message Passing Interface (MPI) library.

OPH_APPLY applies an array-based primitive to all fragments of an input datacube. The input string for this operator is quite simple and involves few arguments like in the following example:

```
operator=oph_apply;datacube_input=cube_in;datacube_output=
cube_out;query=oph_reduce(oph_subarray(measure,1,120),'OPH_A
VG',30)
```

The array-based primitive in this example is the following nested query:

```
oph_reduce(oph_subarray(measure,1,120),'OPH_AVG',30)
```

which extracts the array elements in the interval [1,120], and then computes the average on the sub-intervals [1,30],[31,60],[61-90],[91-120]. Assuming that the arrays contain daily temperature values (360-value based arrays according to a 360-day calendar, 12 months of 30 days each), this example could represent a monthly-based data reduction carried out on a subset of data (four months, from January to April).

OPH_PUBLISH publishes on the HTTP server of the Ophidia system a HTML based version of the input datacube, which can be remotely accessed through a web browser. The operator creates one HTML page for every single fragment and builds an index page linking all of the other pages. In this case, no output datacube is created in the Ophidia data store. The input string for this operator is even simpler than the OPH_APPLY one:

```
operator=oph_publish;datacube_input=cube_in
```

Finally, OPH_CUBE_ELEMENTS computes in parallel the total number of elements stored in a datacube without counting *NotANumber* values, which are common in sparse datacubes. The input string is similar to the OPH_PUBLISH:

```
operator=oph_cube_elements;datacube_input=cube_in
```

Note that the inputs strings reported before contain the mandatory arguments only. In some cases additional non-mandatory arguments are also available (e.g., the scheduling policy for the distribution task).

The three operators have a similar implementation of the *oph_setup_env* and *oph_cleanup_env* interfaces. In the first case (*oph_setup_env*), the master reads and checks the input string arguments against the information related to the input datacube in the OphidiaDB. It then sends the input arguments to all slave processes. In the second case (*oph_cleanup_env*), all processes de-allocate the memory structures previously allocated.

Similarly, in the three operators, *oph_destroy* is an empty function and *oph_distribution* distributes the fragments among the parallel processes, applying a simple and intuitive distribution algorithm based on the process identifier.

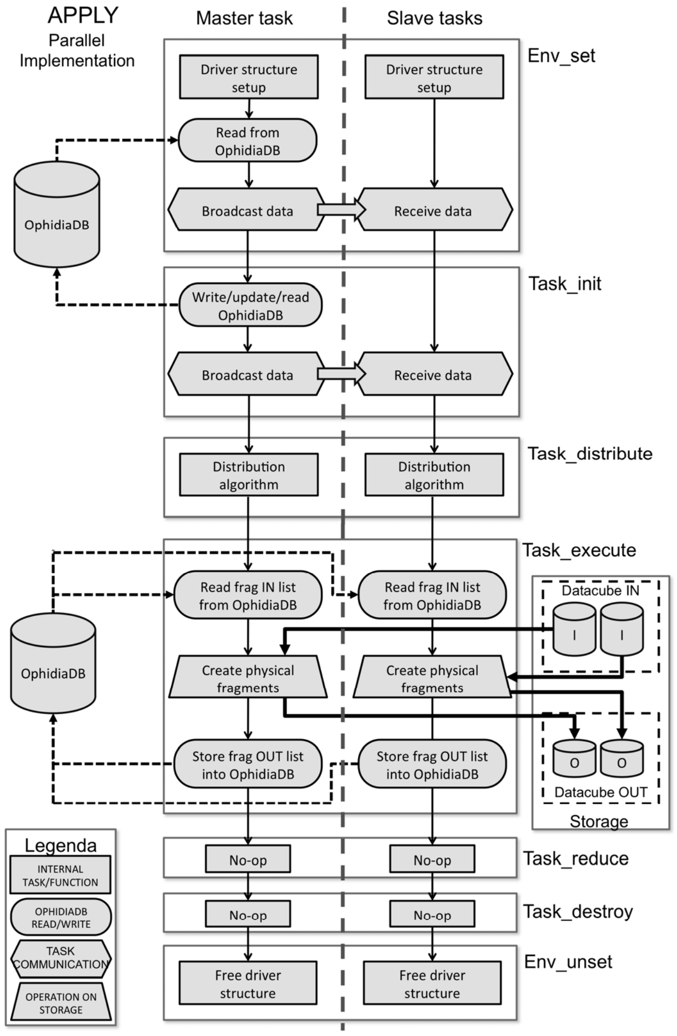

Fig. 3. The OPH_APPLY internal diagram. The schema shows how the 7 steps of the analytics framework are mapped onto the implementation of the OPH_APPLY operator.

The principal differences among the three operators are:

- OPH_CUBE_ELEMENTS implements the *oph_reduce* interface to perform a reduction task and sum the total number of elements from the partial results computed by all the processes. The reduction task is not needed by the other two operators;

- OPH_APPLY and OPH_PUBLISH implement the *oph_init* interface to create, respectively, the *datacube_out* entry in the OphidiaDB and the web directory containing the HTML pages. The OPH_CUBE_ELEMENT does not need the implementation of this interface since the result is managed in memory and delivered to the end user as a metadata output information;

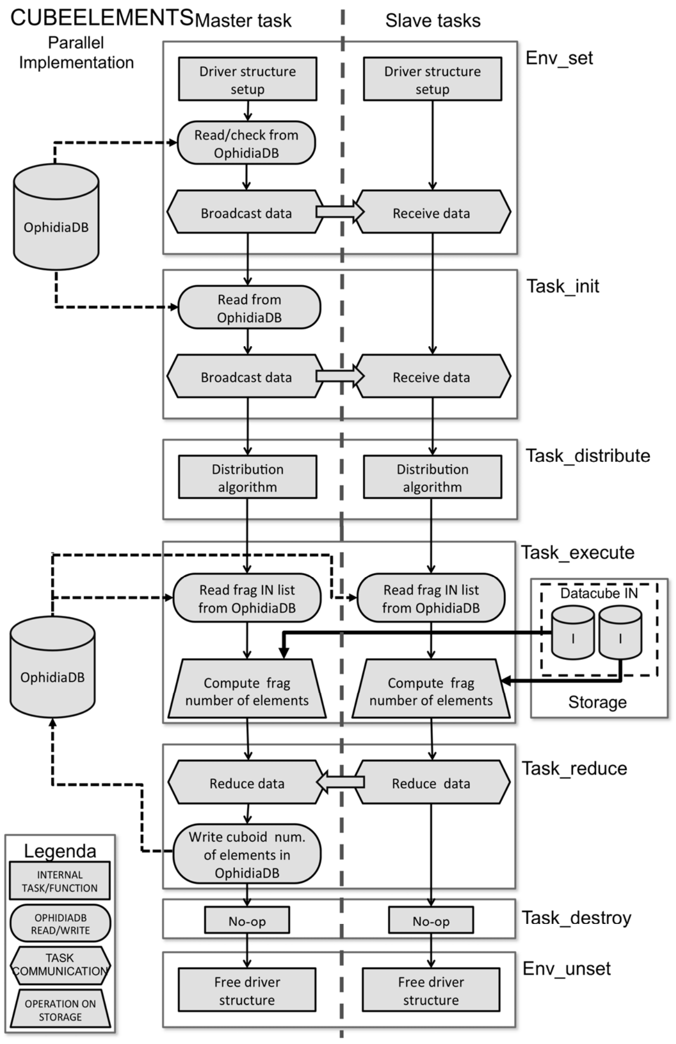

Fig. 4. OPH_CUBE_ELEMENTS internal diagram. The schema shows how the seven steps of the analytics framework are mapped onto the implementation of the OPH_CUBE_ELEMENTS operator.

- each operator provides a different implementation of *oph_execute_operator*. OPH_APPLY creates a new set of fragments by applying the array-based primitive on the input datacube. OPH_PUBLISH creates a set of HTML pages containing all the data reads from the input datacube (no filtering or data transformations are applied by this operator on the input data). OPH_CUBE_ELEMENTS counts the number of elements from each fragment by running the following SQL statement:

```
SELECT sum(oph_count(array_attr)) FROM fragment
```

which invokes an array-based primitive (*oph_count*) to get the number of elements from each tuple (*array_attr* attribute) and sums these results to infer the total number of elements related to the table *fragment*.

Finally, we note that more complex use cases can be defined by simply nesting several operators like in the following example:

```
OPH_PUBLISH(OPH_APPLY(OPH_APPLY(OPH_IMPORT_NC(input_
file), transformation1), transformation2), HTTP_folder)
```

which imports the NetCDF *input_file* file into the Ophidia data store, applies two data transformations (*transformation1* and *transformation2*), and then publishes the results on the HTTP server (*HTTP_folder*) available in the Ophidia infrastructure.

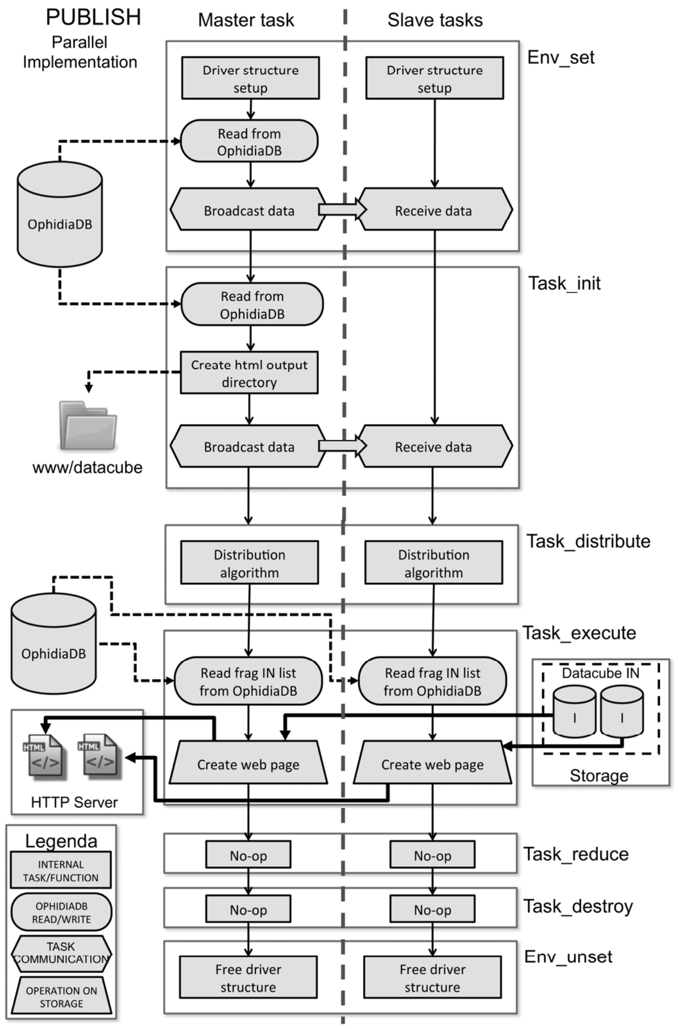

Fig. 5. The OPH_PUBLISH internal diagram. The schema shows how the seven steps of the analytics framework are mapped onto the implementation of the OPH_PUBLISH operator.

E. Operators benchmark: preliminary insights

A comprehensive evaluation of the Ophidia analytics framework is out of the scope of this paper and will be presented in future work. However, a preliminary performance evaluation provides some interesting insights about both the analytics framework and the current implementation of some key operators. The benchmark environment consists of a cluster equipped with 12 dual processor nodes IBM iDataplex (2 Intel Sandy Bridge processors, 2x8 cores, 64GB RAM, 500GB local disk, 20MB chipset cache). Four nodes are configured as *compute* and are used to run the parallel operators. The remaining eight nodes host 4 MySQL servers each, which are used for I/O purposes. The MySQL servers are configured with default settings.

The benchmark regards two different operators and orthogonal use cases:

- use case A "*compute intensive*": OPH_APPLY applies a *reduce_all_max* array-based primitive on a 500GB datacube. The datacube is partitioned in 64 fragments with 10^4 tuples (each tuple stores a 10^5-element array). Each tuple is reduced to a single element (the maximum element of the array) through the *reduce_all_max* primitive. The operator generates an output datacube of about 14MB in size. The runs involve from 1 to 64 parallel processes.
- use case B "*I/O intensive*": OPH_PUBLISH runs on a 2.5GB datacube. The datacube is partitioned into 64 fragments consisting of 500 tuples each one. Each tuple stores a 10^4-element array. The full datacube is published on a 5GB set of HTML pages (the HTML tags and the different way the data is stored – ASCII characters in HTML files instead of binary data in a database table - double the datacube size). The runs involve from 1 to 64 parallel processes.

The two use cases share the same fragmentation and distribution settings at the node level. More specifically:

§ = 4 (number of DBMSInstances/IOnode)
ß = 2 (number of databases/DBMSInstance),
µ = 1 (number of fragments/database)

We define the *absolute fragmentation index Fi* = §*ß*µ as the total number of fragments per I/O node, which is equal to 8 in both use cases. We plan in future work to study how changing *Fi* impacts performance: this topic is out of the scope of this paper for page limit issues.

Table II shows the execution time and the efficiency for the first use case. Efficiency is always more than 95%. As an additional test, doubling the datacube input (1TB) and the number of nodes in the infrastructure (24 in total), the efficiency is about 94% on 128 cores.

Table III presents the results for the second use case. In this case the efficiency is always more than 92%, even though the generated output is quite different in size (366 times bigger than in the previous use case).

Fig. 6 summarizes the results from Tables II and III. Even though the two use cases are quite different in terms of output size and processing needs, they behave in a similar manner and scale linearly. This is due to several factors:

- the distributive nature of the two use cases, which helps in having a large number of sub-tasks working in parallel without strong communication needs or issues;
- the low number of fragments does not introduce a high overhead both in the distribution and execution tasks.
- the analytics framework design which guides the operators implementation through seven well defined interfaces (see Section III.B) and fits perfectly distributive tasks like the ones in the two use cases.

Table IV shows the different behaviour of the two operators in terms of I/O. We see that the aggregate throughput of the I/O nodes is small in the OPH_APPLY use case (less than 1MB/sec) due to the compute intensive nature of the use case and the small output generated. Conversely, aggregate I/O is much higher in the OPH_PUBLISH use case (due to the massive I/O to generate the HTML output) and it scales linearly with the number of cores (up to 100MB/sec with 64 parallel processes, see Fig. 7).

TABLE II. USE CASE A (OHP_APPLY) EXECUTION TIME AND EFFICIENCY RESULTS

#cores	Use Case A time (sec)	Efficiency
1	1647.87	1.00
2	824.01	1.00
4	412.40	1.00
8	206.21	1.00
16	103.44	1.00
32	52.43	0.98
64	27.18	0.95

TABLE III. USE CASE B (OPH_PUBLISH) EXECUTION TIME AND EFFICIENCY RESULTS

#cores	Use Case B time (sec)	Efficiency
1	3157.41	1.00
2	1578.33	1.00
4	790.76	1.00
8	398.71	0.99
16	199.22	0.99
32	101.74	0.97
64	53.89	0.92

TABLE IV. THROUGHPUT RESULTS RELATED TO THE TWO USE CASES

#cores	Use Case A (MB/sec)	Use Case B (MB/sec)
1	0.01	1.59
2	0.02	3.18
4	0.04	6.35
8	0.07	12.58
16	0.14	25.19
32	0.28	49.32
64	0.54	93.11

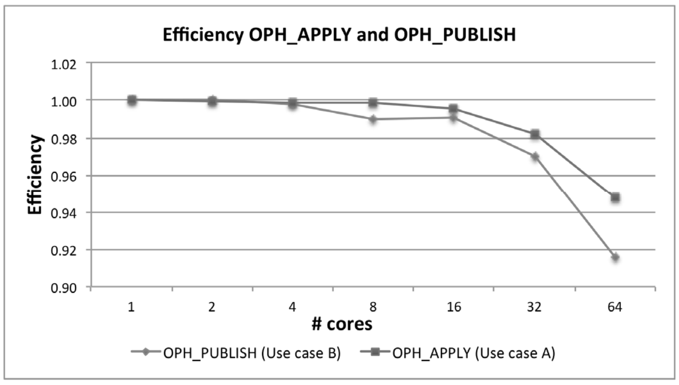

Fig. 6. OPH_APPLY and OPH_PUBLISH efficiency in the two use cases. The chart is not meant to compare the efficiency of the two operators, but rather to show they have a similar behavior in terms of scalability.

We note that the current tests and cluster configuration use the compute nodes to run the parallel operators only. I/O servers were just triggered by processes running on the compute nodes. In future work, we will also evaluate performance when using a different cluster configuration named *super-node*, in which all cluster nodes participate in both I/O and compute tasks.

Thus, each node will be used to run the operators as well as to host the I/O servers. Preliminary results on a four-node cluster configuration are really promising and show that the compute and I/O node difference is basically functional rather than physical (this means the I/O and compute functionalities can coexist on the same node).

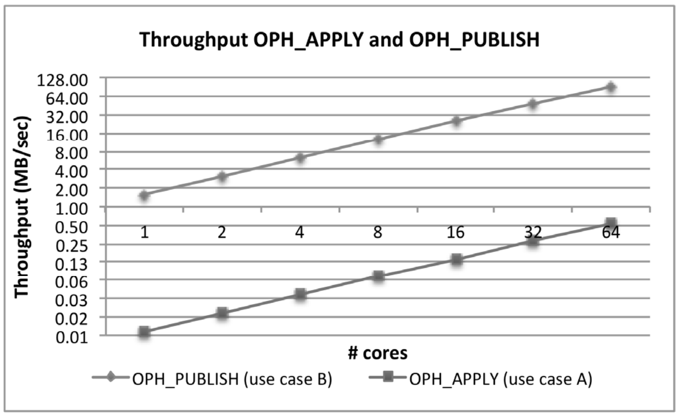

Fig. 7. The OPH_APPLY and OPH_PUBLISH throughput in the two use cases.

IV. Conclusions and Future Work

The Ophidia analytics framework, a core part of the Ophidia research project, has been presented. As discussed in this work, the analytics framework is responsible for atomically processing, transforming and manipulating array-based data, by providing a common way to run on large clusters analytics tasks applied to big datasets. The paper has highlighted the design principles, the algorithm and the most relevant implementation aspects associated to the Ophidia analytics framework. Besides a comprehensive list of the most relevant operators, three of them have been also presented in detail (OPH_APPLY and OPH_PUBLISH for the data part, OPH_CUBE_ELEMENTS for the metadata part). We have also presented some promising experimental results involving two operators (OPH_APPLY and OPH_PUBLISH) executing in a real cluster environment.

We plan in future work to develop an extended set of parallel operators to support new scientific use cases. Array-based primitives extensions, a data analytics query language and an optimized query planner will be considered to support more complex operators and dataflow driven requests. A comprehensive analytics benchmark will be also defined and implemented to further evaluate the performance of the system.

Acknowledgments

This work was supported in part by the Italian Ministry of Education, Universities and Research (MIUR) under contract GEMINA and in part by the U.S. Department of Energy, Office of Science through grant DE-AC02-06CH11357.

References

[1] S. Fiore and G. Aloisio, “Special section: Data management for eScience”. *Future Generation Computer System* 27(3): 290-291 (2011).
[2] Julio Saez-Rodriguez, Arthur Goldsipe, Jeremy Muhlich, Leonidas G. Alexopoulos, Bjorn Millard, Douglas A. Lauffenburger, and Peter K. Sorger, “Flexible informatics for linking experimental data to mathematical models via DataRail”. *Bioinformatics* 24, 6 (March 2008), pp. 840-847, 2008, doi-10.1093/bioinformatics/btn018 http://dx.doi.org/10.1093/bioinformatics/btn018.
[3] William Hendrix, Isaac Tetteh, Ankit Agrawal, Fredrick Semazzi, Wei-keng Liao, and Alok Choudhary, “Community dynamics and analysis of decadal trends in climate data”, (ICDM Climate 2011).
[4] Rob Latham, Chris Daley, Wei-keng Liao, Kui Gao, Rob Ross, Anshu Dubey and Alok Choudhary, “A case study for scientific I/O: improving the FLASH astrophysics code”. Comput. Sci. Disc. 5 (2012) 015001.
[5] J. Dongarra, P. Beckman, et al., “The International Exascale Software Project roadmap”. International *J. High Performance Computing Apps.* 25, no. 1, pp. 3-60 (2011), ISSN 1094-3420 doi: 10.1177/1094342010391989.
[6] G. Aloisio and S. Fiore, “Towards exascale distributed data management”, International *J. of High Performance Computing Apps.* 23, no. 4, pp. 398-400 (2009) doi: 10.1177/1094342009347702.
[7] J. Taylor, Defining eScience http://www.nesc.ac.uk/nesc/define.html.
[8] Climate Data Operators (CDO) - https://code.zmaw.de/projects/cdo.
[9] C. S. Zender, “Analysis of self-describing gridded geoscience data with netCDF Operators (NCO)”, Environmental Modelling & Software, 23, no. 10–11, pp. 1338–1342, 2008.
[10] P. Tsai and B.E. Doty, “A prototype Java interface for the Grid Analysis and Display System (GrADS)”, in Fourteenth International Conference on Interactive Information and Processing Systems, Phoenix, Arizona, 1998.
[11] The NCAR Command Language (Version 6.0.0) [Software]. (2012). Boulder, Colorado: UCAR/NCAR/CISL/VETS. http://dx.doi.org/ 10.5065/D6WD3XH5.
[12] B. Smith, D. M. Ricciuto, P. E. Thornton, G. M. Shipman, C. A. Steed, D. N. Williams, M. Wehner, “ParCAT: Parallel Climate Analysis Toolkit”, ICCS2013, pp. 2367-2375
[13] R. L. Jacob, J. Krishna, X. Xu, T. Tautges, I. Grindeanu, R. Latham, K. Peterson, P. B. Bochev, M. Haley, D. Brown, R. Brownrigg, D. G. Shea, W. Huang, D. Middleton, “ParNCL and ParGAL: Data-parallel Tools for Postprocessing of Large-scale Earth Science Data”, ICCS2013, pp. 1245-1254.
[14] J. Han and M. Kamber, “Data mining: Concepts and Techniques”, Morgan Kaufmann Publishers, 2005.
[15] M. Golfarelli. “The DFM: A conceptual model for data warehouse”, in Encyclopedia of Data Warehousing and Mining (2nd Edition), John Wang (Ed.), IGI Global, pp. 638-645, 2008.
[16] K.E. Taylor, R. J. Stouffer, and G. A. Meehl, “An overview of CMIP5 and the experiment design”, Bulletin of the American Meteorological Society 93, no. 4, pp. 485-498 (2012), doi:10.1175/BAMS-D-11-00094.1.
[17] R. K. Rew and G. P. Davis, “The Unidata netCDF: Software for scientific data access”, in 6th Int. Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology, *American Meteorology Society*, pp. 33-40, February 1990.
[18] S. Fiore, A. D'Anca, C. Palazzo, I. Foster, D. Williams, G. Aloisio: “Ophidia: toward big data analytics for eScience”, ICCS2013, pp. 2376-2385.
[19] GNU Libtool - The GNU Portable Library Tool. http://www.gnu.org/ software/libtool/
[20] The GNU Scientific Library (GSL) http://www.gnu.org/software/gsl/.
[21] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, et al., PETSc web page http://www.mcs.anl.gov/petsc, 2012.