Automatic Data Filtering for In Situ Workflows

Clément Mommessin, Matthieu Dreher, Tom Peterka, Bruno Raffin

Automatic Data Filtering for In Situ Workflows

Clément Mommessin
Argonne National Laboratory
9700 S. Cass Ave.
Lemont, IL 60439, USA
cmommessin@anl.gov

Matthieu Dreher
Argonne National Laboratory
9700 S. Cass Ave.
Lemont, IL 60439, USA
mdreher@anl.gov

Tom Peterka
Argonne National Laboratory
9700 S. Cass Ave.
Lemont, IL 60439, USA
tpeterka@mcs.anl.gov

Bruno Raffin
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG,
F-38000 Grenoble, France
bruno.raffin@inria.fr

***Abstract*—In situ workflows contain tasks that exchange messages composed of several data fields. However, a consumer task may not necessarily need all the data fields from its producer. For example, a molecular dynamics simulation can produce atom positions, velocities, and forces; but some analyses require only atom positions. The user should decide whether to specialize the output of a producer task for a particular consumer and get better performance or to send more data than required by the consumer. The first option limits task portability, while the second wastes resources. In this paper, we introduce *contracts* for in situ tasks. A contract specifies for a producer each data field available for output and for a consumer the data fields needed as input. Comparing a producer and consumer contract allows automatic selection of the data fields a producer has to send for that consumer. We integrated our contracts mechanism within Decaf, a middleware for building and executing in situ workflows. Contracts enable to automatically extract at the producer the data the consumer needs. We evaluate the cost and performance of message extraction at runtime with both synthetic examples and a real scientific workflow coupling a molecular dynamics simulation with three different data analytics codes. Our contract-based automatic data extraction removes the need to specialize producers while entailing small overheads.**

I. Introduction

As supercomputers grow in size and computational power each year [1], scientists using these platforms are generating and processing an ever-increasing amount of data. Traditionally, simulation and analysis applications communicate through files. As we approach exascale, however, the gap between computing rate and the I/O bandwidth of high-performance computing (HPC) platforms has become critical, and post processing data analysis is no longer practical. Moreover, the same amount of computing power may be necessary to analyze this data. These trends motivate the emergence of *in situ processing*, where data analysis is performed as close as possible to where and when the data is produced.

Many frameworks have been developed to design and execute in situ workflows ([2], [3], [4]), coupling scientific simulation applications with analytics and visualization tasks. In situ frameworks decouple as much as possible data management from the applications. A simulation application produces data, and the middleware takes care of routing these data through the communication channels to the tasks performing analytics.

Often the extraction of the data required by the different analysis is hard-coded in the simulation. If the frequency or the span of data required by the analysis changes, the simulation code needs to be updated accordingly. One conservative approach consists of packing and sending a superset of the needed data, with the analysis filtering what they actually need. However, this approach overloads the memory and network with unnecessary data.

In this paper we propose to augment in situ frameworks with a *contracting mechanism* to automatically extract from the simulation the data the analysis requires. Our goals are: (1) reduce the amount of data sent over the network to the strict minimum required by a consumer task, and (2) limit the intrusion of analysis in the simulation code. Thus, we move away from the model inherited from classical file writing, where the simulation is explicitly in charge of the data to output, to an analytics driven model where the simulation focuses on its core job, namely updates its internal variables as the time progress, while data extraction is actually defined by the analysis. A contract between a producer and a consumer enables checking that the consumer needs are compatible with the producer outputs and building a message containing only the data required by the consumer. We implement our model of contracts within Decaf [5], a middleware for building and executing in situ workflows. We evaluate the overheads and the performance within Decaf both with two synthetic examples and with *Gromacs*, a molecular dynamics simulation application [6], coupled with different data analyses.

The outline of the paper is as follows. Related work is reviewed in Section II. We present our model of contracts in Section III. Section IV details the modifications done to Decaf to implement the management of contracts and message filtering. Section V shows experimental results. Section VI presents our conclusion and discusses future work.

II. Related Work

A. Data models

We define a *data model* to be an abstraction of a structure containing different data objects, or *data fields*.

Several ways of describing and managing data models for large-scale parallel applications have been proposed in the literature. These involve methods to easily access and manipulate each field in a data model, but they do not support contracting.

Conduit [7] proposes a JSON-based data model for describing hierarchical in-core scientific data with a dynamic API for the construction and consumption of hierarchical objects. The JSON schema enables a data model description external to the code. Having the consumer and producer using the same JSON schema ensures that the consumer serializes messages that the consumer can deserialize and interpret correctly.

The Bredala [8] library provides a simple API to construct parallel data models with enough information to safely split and merge sub datamodels while preserving the data semantic. Bredala also provides various data redistribution components with an $M \times N$ communication pattern.

HDF5 [9] is a data model and a file format for flexible and efficient I/O in scientific workflows. FFS [10] is a data model that resembles C structures where each field is described by its name, type, size, and an offset in the data structure. FFS is used by the EVPath [11] middleware to perform data processing of dynamic variable-sized models and structures that can be described recursively, such as trees and graphs.

B. In situ middleware

In situ middleware comes from various communities. VisIt [12] and ParaView [13] are widely used tools to analyze and visualize scientific data. Both tools have an in situ library, respectively Libsim [14] and Catalyst [13], to couple simulations to their visualization servers. VisIt uses a specific model of *contracts* [15] to describe the data modifications of each filter of the VisIt pipeline. These contracts are used to optimize the size of data objects retrieved from the simulation. Our contract mechanism applies to any type of data, and includes the temporality of the data. Matthes et al. present ISAAC [16], an open source library for in situ visualization and steering. Different visualization renderings acting directly on the simulation data are proposed, with possible data transformation before visualization and message passing to the simulation for steering.

Other in situ middleware originates from the I/O community, such as ADIOS [17], that uses helper cores and staging nodes to perform parallel I/O operations. ADIOS utilizes a data model defined in an external XML file to encode the data hierarchy, data type specifications, and process grouping and describes how to process the data. It enables changing the I/O behavior without code changes.

Damaris [18] uses dedicated helper cores and shared memory to perform asynchronous data processing and I/O in order to reduce variability in I/O performance. An in situ visualization framework, Damaris/Viz [19], built on Damaris, has proposed to directly couple simulations with VistIt requiring less code modification than using the Libsim library directly. Damaris also comes with a XML-based data model used as a tier to share information between the simulation and the helper cores.

Tiwari et al. [20] propose an approach to perform in situ data analysis on the solid-state disks (SSDs) present in HPC platforms, using the storage controllers to compute data already residing in the SSDs.

The FlexIO middleware [21] supports different placement strategies for analytics in a workflow and provides data management and communication between tasks. Moreover, FlexIO includes dynamics codelets, called *data conditioning (DC) plug-ins*, to perform selection and transformation of data transiting between two workflow tasks.

Champsaur et al. propose SmartBlock [22], an approach to design generic and reusable data manipulation and analysis components for in situ workflows, relying on the ADIOS data model. Specifically, the *Select* component permits the selection of a subpart of a multidimensional array by indicating the list of dimensions to keep for output.

General frameworks have been designed for connecting and managing different parallel applications in a workflow, such as FlowVR [23]. Originally designed for virtual reality at large scale, FlowVR has been adapted to support in situ applications with a runtime system to deploy the workflow tasks and manage communications between them. FlowVR messages are raw buffers that do not rely on a particular data model.

Decaf [5] is an in situ middleware for building and executing workflows, providing a simple put/get API for data communication between two parallel nodes. Decaf provides a Python API to describe an in situ workflow and relies on the data model and redistribution components of Bredala [8] to handle communication. Bredala provides mechanism to create and manipulate a data model field by field. Manala [**?**] is used on top of Decaf to filter messages based on their iteration number. Our contract mechanism extends that functionality to filter individual fields within a data model.

III. Contracts

A *contract* specifies for a producer each data object available per output and for a consumer the data objects needed per input. Knowing beforehand the list of data output by each producer and the list of data used by each consumer permits sending through the communication channels only the required data and thus reduces the bandwidth usage and message packing/unpacking costs.

A. Preliminaries

We define an *in situ workflow* as a directed graph, possibly containing cycles, where nodes are parallel tasks, usually simulation or analytics, sending and/or receiving data from other nodes through communication channels. A communication

channel, also denoted by $dataflow$, is an edge between a *producer* task and a *consumer* task in the workflow graph.

We denote by *field* the triplet $(name,\ type,\ periodicity)$ describing a data field with the name, type, and periodicity of the field. The periodicity is the frequency at which the field is available. For example, we can define the data model of a particle produced at each iteration to be the composition of the index, position of that particle, and velocity in the three spatial dimensions. Thus, the three fields describing the data model of a particle are as follows:

- (index, integer, 1)
- (position, array of floats, 1)
- (velocity, array of floats, 1)

B. Contract model

We denote by *output contract* the set of fields that a producer makes available for output and by *input contract* the set of fields that a consumer requires for input. A producer task can output several *data models*. In that case, an *output contract* is defined for each output. Similarly, a consumer task can receive several *data models*. An *input contract* is then defined for each input.

For a producer, a periodicity of k means that the data field will be sent every k iterations. For a consumer the requested periodicity is defined relative to the producer output periodicity. A consumer requesting a field at periodicity k will get the field every k iterations the producer makes available.

Knowing for each dataflow the list of data fields that are sent by the producer and required by the consumer, we can compute the list of data fields and their period at which they have to be transmitted to the consumer. We denote by *matching list* the list of *contract-fields* describing the data that the producer needs to send. The matching list of a dataflow is computed from the output contract of the producer and the input contract of the consumer.

Algorithm 1 presents the pseudo code of the function computing a matching list. A field of the input contract is in the matching list if and only if there is a field in the output contract with the same name and the same type. If a field in the consumer contract is not present in the producer contract, an error is raised (line 7 Alg. 1). A *contract-field* also contains the absolute periodicity extraction of that field at the producer. This periodicity is the product of the field periodicity in the producer and consumer contracts. For example, consider a data field with a periodicity of 3 for the producer and a periodicity of 2 for the consumer. The sequence of iterations where the data field is available at the producer is $S_1 = (0,\ 3,\ 6,\ 9,\ 12,\ 15,\ 18, \dots)$, and the consumer requires the data field only every two iterations. Thus, it is sufficient to send this data field at iterations $S_2 = (0,\ 6,\ 12,\ 18, \dots)$, namely, at an absolute periodicity of $3 \times 2 = 6$.

A matching list is associated with a communication channel between a producer output and a consumer input. If a producer is connected to multiple consumers, a separate matching list is computed between the producer output and each consumer. If several producers are connected to a single consumer, a separate matching list is computed between each producer and the consumer.

In situ middleware can use the computed matching list to send only the required data and thus reduce the size of the message sent through the communication channel.

Algorithm 1: Pseudo code to compute the matching list of contract-fields between two contracts.

Input : A producer and a consumer contracts (*prod-contract* and *cons-contract*).
Output: The matching list of contract-fields (*matching*).

```
1  matching = ∅
2  forall (name, type, cons-period) ∈ cons-contract do
3      if ∃ (name, type, prod-period) ∈ prod-contract then
4          periodicity = cons-period × prod-period
5          matching = matching ∪ {(name, type, periodicity)}
6      else
7          print "ERROR: data field mismatch"
8      end
9  end
10 return matching
```

C. Intermediate contract

Several in situ packages run intermediate code to perform data manipulations on a message while in a communication channel. These intermediate *data operators* can be either user or system defined (e.g., FlowVR filters [23], FlexIO Data Conditioning plug-ins [21], Decaf links [5]). These operators act only on a subset of the data fields composing the message. They are used, for instance, to convert data field units. The nonmodified fields are automatically forwarded by the middleware without explicit calls from the operator. That automatic forwarding guarantees the genericity of the operator. As a result we cannot use the same type of contract as with for producers and consumers because these contracts expect a description of all the data fields transiting through the operator.

We present here an extension of the contract model, called *middle-contract*, to support such intermediate operators. The goal is to deduct which data field must be forwarded by the operator given the fields modified by the operator, the producer, and consumer contracts.

When a data operator is present in a dataflow, a message is sent from the producer to the consumer in two steps. First, the message is sent from the producer to the operator. Second, the resulting message is sent from the operator to the consumer. Thus, this intermediate data operator plays the role of a consumer in the first step and of a producer in the second step. We can view this dataflow as a succession of two sub-dataflows, one between the producer and the data operator and one between the data operator and the consumer.

A middle-contract, attached to a data operator, is composed of an input contract that is used during the first step of the

communication and an output contract that is used during the second step. Thus, the matching list of the dataflow is split in two lists: *list_prod* and *list_middle*. The *list_prod* describes the data model of the messages transiting in the first sub-dataflow. It corresponds to the matching list between the producer output contract and the data operator input contract. The *list_middle* describes the data model of the messages transiting in the second sub-dataflow. It corresponds to the matching list between the data operator output contract and the consumer input contract.

Actually these intermediate operators may modify only a subset of the fields. We thus attach a Boolean value *forward-fields* to a middle-contract to include in the matching list the fields that are sent by the producer and required by the consumer without explicitly adding their contract-fields in the middle-contract. This way, a middle-contract describes only the modifications performed by the data operator and does not have to explicitly declare the other fields exchanged between the producer and the consumer. For example, consider a producer sending 10 fields to a consumer, with a data operator modifying only one field. Setting *forward-fields* to *True* allows one to put only the modified field in the middle-contract while still including the other 9 fields in the computed matching lists.

If *forward-fields* is set to *True*, the two computed matching lists are updated with contract-fields of the consumer contract if and only if there is a contract-field with the same name and the same type in the producer contract but not in the middle-contract. As for regular contracts, we update the periodicity of each added contract-field to be the product of periodicities of this field in the producer contract and in the consumer contract.

Algorithm 2 presents the pseudo code to update the two matching lists when *forward-fields* is set to *True*. If set to *False*, no update of the matching lists is required, and the function is not called. We can summarize the computation of the two matching lists when a data operator is present in a dataflow by the three following steps:

1) Compute *list_prod* with the producer output contract and the data operator input contract with Algorithm 1.
2) Compute *list_middle* with the data operator output contract and the consumer input contract with Algorithm 1.
3) If *forward-fields* is set to true, complete *list_prod* and *list_middle* with Algorithm 2.

Figure 1 summarizes our model with a simple two-nodes example. The contracts of the producer and the consumer as well as the middle-contract are in the solid-line rectangles while the two corresponding matching lists are in the dashed-line rectangles. The producer outputs two fields, an integer and a floating-point value, every iteration. The consumer requires two fields: an integer *dataA* every two iterations and an integer *dataB* every iteration. The data operator between these two nodes converts the field *dataB* from a floating-point value to an integer every iteration and forwards the other fields present in the data model. Thus, the matching lists of the dataflow between the two nodes contain the field *dataA* with a periodicity of 2 and the field *dataB* whose type correspond to the transformation from a floating-point value to integer performed by the data operator, with a periodicity of 1.

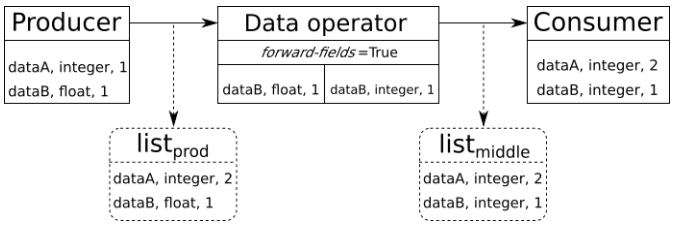

Fig. 1: Schema representing contracts and middle-contract (solid-line rectangles) for a two-tasks example and their corresponding matching lists (dashed-line rectangles).

Algorithm 2: Pseudo code to update the matching lists if *forward-fields* is set to *True*. The variables x at lines 2 and 8 denote any value of periodicity.

Input : The producer and consumer contracts (*prod-contract* and *cons-contract*) and the two matching lists ($list_{prod}$ and $list_{middle}$) of a dataflow.
Output: The updated matching lists.

1 **forall** *(name, type, cons-period)* $\in$ *cons-contract* **do**
2 **if** *(name, type, x)* $\notin list_{prod}$ **then**
3 **if** $\exists$ *(name, type, prod-period)* $\in$ *prod-contract* **then**
4 $periodicity = cons\text{-}period \times prod\text{-}period$
5 $list_{prod} = list_{prod} \cup \{(name, type, periodicity)\}$
6 **end**
7 **end**
8 **if** *(name, type, x)* $\notin list_{middle}$ **then**
9 **if** $\exists$ *(name, type, prod-period)* $\in$ *prod-contract* **then**
10 $periodicity = cons\text{-}period \times prod\text{-}period$
11 $list_{middle} = list_{middle} \cup \{(name, type, periodicity)\}$
12 **else**
13 **print** "ERROR: data field mismatch."
14 **end**
15 **end**
16 **end**
17 **return** $(list_{prod},\ list_{middle})$

IV. Integration within Decaf

We implemented our model within the Decaf middleware. We present in this section the modifications made to include contracts within Decaf. We first describe the modifications done at the Python level to compute the matching lists. Second, we describe the changes in the code of the runtime to filter the data that are sent with respect to these matching lists.

```
#Node declaration
producer = Node('node0', start_proc=0, nprocs=4,
                func='producer', cmdline='program')
producer.addOutput('dataA', 'int', 1)
producer.addOutput('dataB', 'float', 1)

consumer = Node('node1', start_proc=6, nprocs=2,
                func='consumer', cmdline='program')
consumer.addInput('dataA', 'int', 2)
consumer.addInput('dataB', 'int', 1)

#Edge declaration
edge = Edge('node0', 'node1', start_proc=4,
            nprocs=2, func='link', path=link_path,
            prod_dflow_redist='count',
            dflow_con_redist='count', cmdline='program')
edge.addInput('dataB', 'float', 1)
edge.addOutput('dataB', 'int', 1)
edge.setForwardFields(True)

#Populating the graph
graph = DiGraph()
graph.addNode(producer)
graph.addNode(consumer)
graph.addEdge(edge)

#Generation of the JSON and SH files
processGraph(graph, 'program')
```

Listing 1: Python example for the declaration of a two-nodes workflow with Contracts.

A. Python API modifications

To handle the utilization of contracts, we updated the Python API of Decaf to be able to declare contracts and middle-contracts before attaching them to workflow nodes and links. We implemented the two functions presented in Algorithms 1 and 2, Section III, to check that the producer and consumer contracts of a dataflow are matching and to compute their matching lists. The Python script generates an intermediate *JSON* file used to configure the runtime of Decaf. We integrated the matching lists of each dataflow into the *JSON* file for the filtering of data at runtime.

Listing 1 presents the Python script describing the two-nodes example of Section III, including the description of each contract.

B. Runtime modifications

The management of contracts at runtime is done at the Decaf level and is completely transparent to the user. Thus, no modification of the user code is required.

The Decaf API exposes to the user simple methods to receive (`get()`) and send (`put()`) data from and to other tasks. The Decaf `put()` is called by producer tasks or by *data operators* included in Decaf called *links*. The *put* augmented with our contracts is composed of three steps: automatic data filtering, serialization, and data sending (`MPI_ISend()`). Automatic data filtering implements a function $filterMessage$ presented in Algorithm 3. The function verifies whether the producer or the link respects its output contract by checking, for each contract-field in the matching list, whether the field is present in the data with the correct type. If so, the field is added to a new message $filtered\_data$ if the current iteration is a multiple of the field periodicity. This way, every other field present in the original data that is not in the matching list, or present at a different iteration, is not sent through the communication channel. We also modify the Decaf `get()` function to optionally check that the data received respect the *input contract* of the consumer or the link.

Decaf relies on Bredala [8] for its data model. A Bredala data model is a map between field names and pointers to user data. The function $filterMessage$ manipulates only the pointers to the data to create $filtered\_data$. After filtering, Decaf serializes the data into a message and sends it to its destination.

Algorithm 3: Pseudo code of the function *filterMessage* to check the matching lists and perform data field selection at runtime.

Input : The original *data*, the matching list *list* and the current *iteration*
Output: The extracted message $filtered\_data$

```
1  filtered_data = Empty message
2  forall (name, type_contract, periodicity) in list do
3      if iteration % periodicity == 0 then
4          if name ∉ data then
5              ERROR: "field not in data"
6          end
7          field ← getData(data,name)
8          type_field ← getType(field)
9          if type_contract ≠ type_field then
10             ERROR: "types do not match"
11         else
12             Add field in filtered_data
13         end
14     end
15 end
16 return filtered_data
```

V. Performance Evaluation

We present in this section the analysis of the cost and performance of message filtering using contracts within Decaf. We define two synthetic tests and one real scientific workflow coupling Gromacs, a molecular dynamics simulation application, with three different data analytics codes.

The experiments were conducted on Froggy, a cluster of 190 computing nodes of the Ciment infrastructure. Each node has two 8-core processors Intel Sandy Bridge EP E5-2670 at 2.6 GHz and 64 GB of memory and nodes are interconnected with an FDR InfiniBand network. We used OpenMPI version 1.10.2 to compile Decaf and Python version 2.7.5 for the scripts.

For the test with the molecular dynamics application, we used the version 4.5.5 of Gromacs simulating a FepA protein, composed of about 70,000 atoms.

A. Filtering cost

In the first test, we analyze the overhead of the filtering function when the utilization of contracts is not necessary, that is when the contracts of the producer and the consumer of a dataflow are identical. We describe our workflow as a single dataflow between two nodes with no link. The producer and the consumer have the same number *nprocs* of processes,

ranging from 1 to 512, and a message is sent every iteration during 1,000 iterations. The message contains a number of fields *nfields* ranging from 1 to 10 with the size of each field ranging from 40 kB to 40 MB. We ran each experiment with two configurations: *filter* with contract-based filtering active, and *no_filter* with contracts disabled (all data available at the producer are sent to the consumer). In both configurations, the periodicity of each field is set to 1, and the data checking is disabled during the call to Decaf `get()`.

Figure 2 presents the average processing time of a *put* for a total of 1,000 iterations as a function of the number of processors *nprocs* (x-axis), the number of exchanged fields *nfields* (columns) and the size of each field *size* (rows). We observe that the two versions present similar results in terms of time spent during the *put* function, with an average difference of 3.5%.

To explain this result, we measured (Figure 3) the ratio of the time spent in *filterMessage* over the time spent in *put* for *filter* as a function of the three parameters *nprocs*, *nfields*, and *size*. We see that the proportion of time spent in the filtering function increases as the number of fields increases but decreases as the array size increases. With a field size of 40 kB, the time spent in the *filterMessage* function is 5.3% on average and decreases to less than 0.01% with a field size of 40 MB.

Decaf *put* augmented with our contracts is composed of three steps: automatic data filtering, serialization, and sending data (`MPI_ISend()`). The automatic data filtering complexity is linear in the number of fields, whereas the serialization and communication are linear in function of the message size. Consequently, the ratio of automatic filtering decreases as the message size increases up to the point where, if messages are large enough (40MB per field), the automatic data filtering comes virtually at no cost.

B. Automatic filtering evaluation

For the second test, we analyze the performance of message filtering using contracts compared with a manual filtering and no filtering of messages (all data are sent to all consumers). In both configurations, the periodicity of each field is set to 1 and the data checking is disabled during the call to Decaf `get()`.

The workflow of this second test consists of one producer sending several fields and three consumers requiring only one field each. Each consumer has *nprocs* processes, ranging from 1 to 128, and the producer has $3 \times nprocs$ processes. The data sent by the producer contains 1, 5, or 10 fields with a size ranging from 40 kB to 40 MB. We ran each configuration with three different filtering methods. The first methods uses automatic filtering (*auto*), the second applies a manual filtering (*manual*), and the third does not filter data (*none*). Automatic filtering filters messages at the Decaf level during the call to Decaf *put*. Manual filtering required a modification of the simulation code to explicitly build the expected messages at the producer side before calls to *put*. Instead of creating a single message containing all fields, three different messages

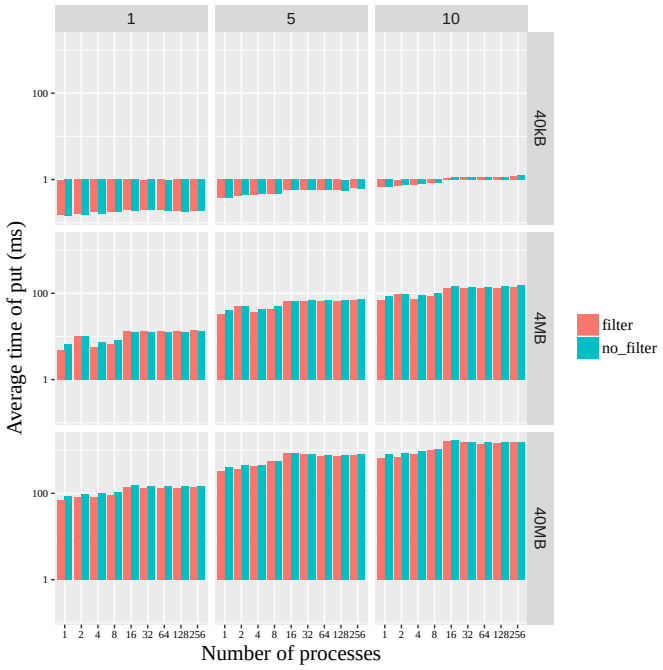

Fig. 2: Average time spent in *put* (log-scale) as a function of *nprocs* (x-axis), *nfields* (columns), and *size* (rows) for the first test.

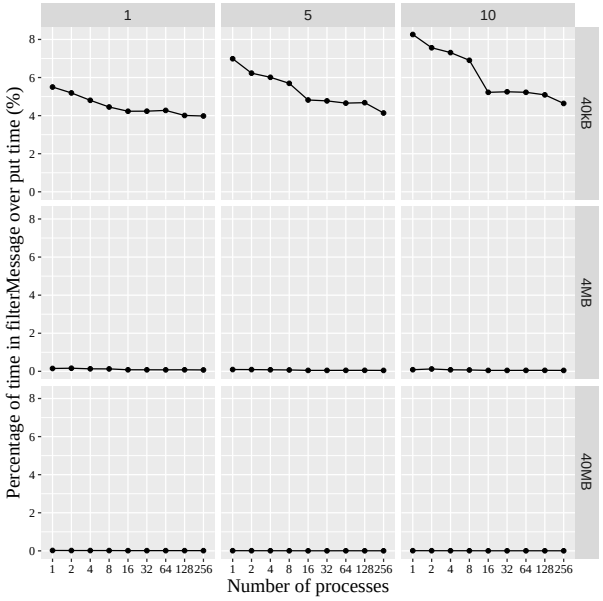

Fig. 3: Percentage of time spent in *filterMessage* over time spent in *put* as a function of *nprocs* (x-axis), *nfields* (columns) and *size* (rows) for *filter*.

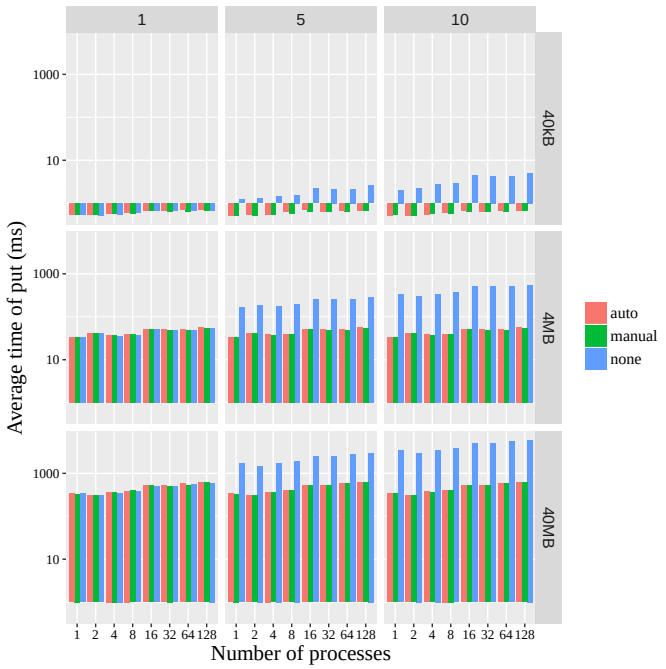

Fig. 4: Average time spent in *put* (log-scale) as a function of *nprocs* (x-axis), $nfields$ (columns), and *size* (rows) for the second test.

containing the fields needed by the three consumers are created and sent separately to the corresponding consumer. We built three different messages because each consumer requires a different subset of the original data model.

Figure 4 presents the average processing time of a *put* for a total of 1,000 iterations as a function of the number of processors *nprocs* (x-axis), the number of exchanged fields $nfields$ (columns), and the size of each field *size* (rows) for the second test.

We observe that *none* outperforms *auto* only for $nfields = 1$. For this configuration, no filtering is needed; therefore filtering comes as an extra unnecessary cost. When more fields are in the messages than are required by the consumer, the time spent in *put* without filtering increases because of the time spent to serialize and send these unnecessary data. For $nfields = 5$, *auto* is on average 4.2 times faster than *none* and 8.2 times faster for $nfields = 10$.

The performance of *manual* and *auto* are close for all configurations tested. On average, we observe a 2% slowdown of *auto* compared with *manual*. This slowdown is due to the extra cost of the filtering function during the *put* for automatic filtering. This result shows that our automatic data filtering can achieve performance similar to that of manual filtering without the need for the developer to write custom code. Alternatively, the user can define contracts for only a subset of tasks requiring data filtering.

C. Molecular dynamics

We now test the performance of message filtering with Gromacs [6], a molecular dynamics parallel simulation

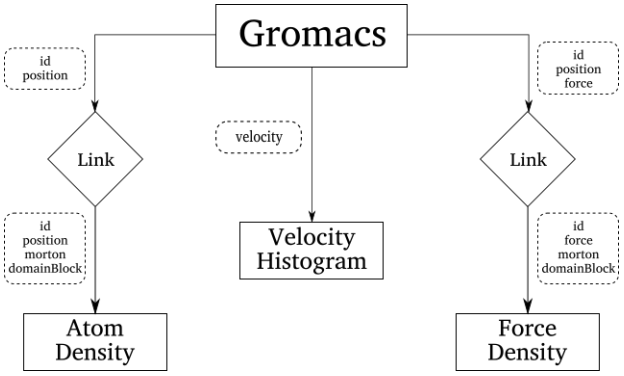

Fig. 5: Workflow graph of the experiments with Gromacs. The dashed-line rectangles correspond to the lists of fields transiting in each dataflow.

application, coupled to three in situ analysis tasks requiring different data generated by the simulation. We measured the impact on the performance of the simulation in terms of its slowdown due to the coupling of the simulation with in situ analyses. We tested three different modes to send the data, which are the same as for the second test: automatic message filtering performed by Decaf, *auto*; manual filtering directly in the Gromacs code, *manual*; and no message filtering, *none*.

We define our workflow with one producer task, Gromacs, and three consumer tasks. Figure 5 summarizes the workflow and shows the matching lists computed for each dataflow. The producer sends every 100 simulation iterations a message containing four different fields: *id*, *position*, *velocity* and $force$. These fields corresponds respectively to arrays of atom indexes, positions, velocities, and forces.

The first consumer is an analysis task computing a 3D density grid of the atoms based on their position, which is used for visualization. The task requires four fields: the *index*, *position*, a *morton code* [24] of each atom, and a data field representing the *domain block* that contains these atoms. To include the fields *morton code* and *domain block* in the message, we create a link between Gromacs and this consumer that will compute and add these two fields in the message. The second consumer is similar except that it computes a 3D grid in which each cell contain the sum of the forces applied to each of the atoms belonging to the cell. The third consumer is a task computing a histogram of atom velocities and thus requires only the field *velocity*.

The integration of Decaf within the source code of Gromacs to send the four fields took about 150 lines of code, with no modification required for the contracts. The manual filtering of the fields with the creation of one specific message for each consumer required about 20 extra lines of code.

We measured the execution time of Gromacs iterations and calls to *put* of Decaf during 200,000 simulation iterations, which corresponds to at least 5 minutes of computation time for each instance. Scientists usually save the data produced by

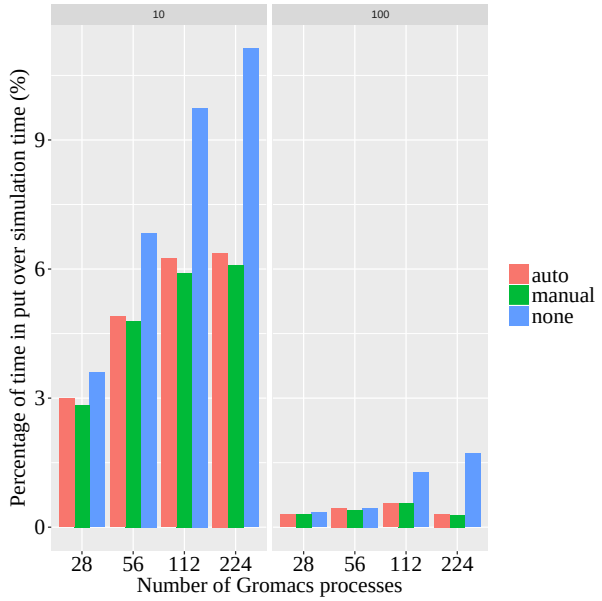

Fig. 6: Percentage of time spent in *put* over simulation time of Gromacs, as a function of the number of processes (x-axis) and the number of iterations between two outputs (columns), for the three instances.

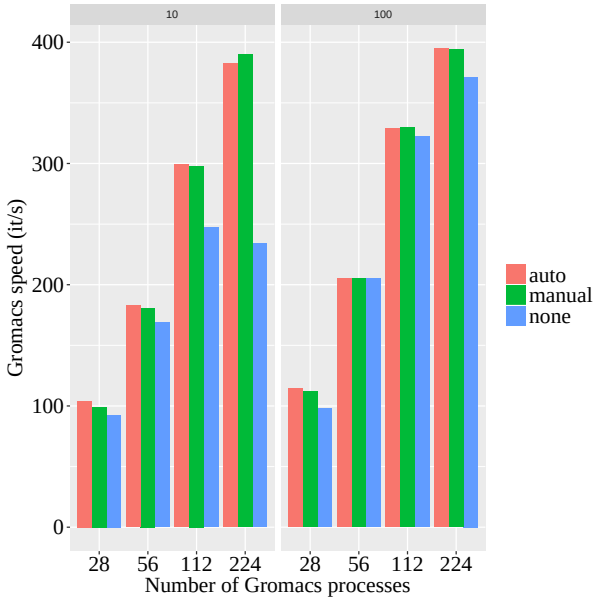

Fig. 7: Gromacs speed (number of iterations per second) as a function of the number of processes (x-axis) and the number of iterations between two outputs (columns), for the three instances.

the simulation every 5,000 iterations, or even less, because of the costly synchronous I/O of Gromacs. Because Decaf communications are asynchronous, we can afford to output data every 100 simulation iterations without slowing the simulation noticeably [5]. We also ran the experiments with an output of the data every 10 simulation iterations.

We ran the experiments with up to 224 cores (16 nodes, 14 cores out of 16 per node) dedicated to Gromacs. Experimentally we found that increasing the number of cores further did not improve the simulation performance (no I/O) for this molecular model. The two remaining cores per simulation node were dedicated to the links that transform the data for the analytics computing 3D grids. We used 3 dedicated nodes for the analytics: one node per consumer task and 4 cores per node.

Figure 6 shows the ratio of time spent in the *put* function over the running time of the simulation. We see that *none* is outperformed by the two other instances because of the cost of serializing and sending the unnecessary data fields. The time spent in *put* by *none* is 5.75 times longer than for *auto* for an output of data every 100 simulation iterations at 224 cores, and 1.75 times longer for an output every 10 iterations.

Filtering data decreases the amount of data sent over the network, which can reduce the impact made on applications. Figure 7 presents the frequency of Gromacs as a function of the number of processes for Gromacs. We observe that the two methods filtering messages outperform the method without filtering: *auto* improves Gromacs frequency by 6% on 224 cores for an output every 100 iterations compared with *none* and by 63% for an output every 10 iterations.

The gain in performance for Gromacs is more significant than the time saved during the Decaf `put()`. The reason is that Gromacs is bounded by communications at scale. Reducing the size of the messages sent over the network reduces the contention over the network card, leaving more room for Gromacs communications.

Comparing *auto* and *manual*, we observe that the performance of the two methods is similar (less than 1% difference). We notice also that the ratio of time spent in *put* compared with the total simulation time is on average 5% smaller for *manual*, which corresponds to the overhead of the *filterMessage* function since the simulation times are similar for the two instances. These results are consistent with the results of Section V-B and show that our automatic message filtering is competitive with manual filtering in a real scientific scenario, avoiding the need for the user to modify the simulation code for each analysis workflow. Improving the I/O performance by filtering messages also enables scientists to output data more frequently for comparable simulation impact and improve the time resolution of their analysis. In other words, the contract mechanism allows Decaf to obtain better performance by automatically filtering the content of the messages without requiring any modifications of the application from the users.

VI. Conclusion

We introduced contracts for in situ task communications. A contract specifies for a producer each data field available for output and for a consumer the data fields needed for input. We proposed a contracting mechanism to automatically select

which data fields to send from a producer to a consumer. We extended this mechanism to incorporate data operators found on the I/O path of several in situ infrastructures able to transform data between a producer and a consumer.

We integrated our mechanism of contracts within Decaf, a middleware for building and executing in situ workflows. We showed that our automatic filtering comes at little cost while providing performance comparable to that of manual filtering. We demonstrated that our contract mechanism removes the need for developers to specialize their tasks for a particular workflow.

We plan to introduce the declaration of contracts at runtime to allow the creation of new connections of tasks. We also plan to implement our contract mechanism in two other in situ middlewares: FlowVR and EVPath.

ACKNOWLEDGMENT

This material was based upon work supported by the U.S. Department of Energy, the Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, program manager Lucy Nowell.

Some of the computations presented in this paper were performed on the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA) and the Equip@Meso project (reference ANR-10-EQPX-29-01) of the programme Investissements d'Avenir supervised by the ANR.

REFERENCES

[1] "Top500 supercomputer." [Online]. Available: http://www.top500.org

[2] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin, "Lessons learned from building in situ coupling frameworks," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ACM, 2015, pp. 19–24.

[3] S. G. Parker and C. R. Johnson, "Scirun: A scientific programming environment for computational steering," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[4] M. Dreher, J. Prevoteau-Jonquet, M. Trellet, M. Piuzzi, M. Baaden, B. Raffin, N. Férey, S. Robert, and S. Limet, "ExaViz: A Flexible Framework to Analyse, Steer and Interact with Molecular Dynamics Simulations," *Faraday Discussions of the Chemical Society*, vol. 169, pp. 119–142, May 2014.

[5] M. Dreher and T. Peterka, "Decaf: Decoupled dataflows for in situ high-performance workflows," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017 (Submitted).

[6] S. Pronk, S. Pall, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, "Gromacs 4.5: A high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013.

[7] Lawrence Livermore National Laboratory, "Conduit: A scientific data exchange library for hpc simulations." [Online]. Available: http://software.llnl.gov/conduit/index.html

[8] M. Dreher and T. Peterka, "Bredala: Semantic data redistribution for in situ applications," in *CLUSTER - IEEE International Conference on Cluster Computing*. IEEE, Sep. 2016.

[9] "The HDF group: Hierarchical data format version 5." [Online]. Available: https://support.hdfgroup.org/HDF5/

[10] G. Eisenhauer, M. Wolf, H. Abbasi, S. Klasky, and K. Schwan, "A type system for high performance communication and computation," in *2011 IEEE Seventh International Conference on e-Science Workshops*, Dec 2011, pp. 183–190.

[11] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "Event-based systems: Opportunities and challenges at exascale," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: ACM, 2009, pp. 2:1–2:10.

[12] S. Ahern, E. Brugger, B. Whitlock, J. S. Meredith, K. Biagas, M. C. Miller, and H. Childs, "VisIt: Experiences with sustainable software," *arXiv preprint arXiv:1309.1796*, 2013.

[13] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen, "The ParaView coprocessing library: A scalable, general purpose in situ visualization library," in *2011 IEEE Symposium on Large Data Analysis and Visualization*, Oct 2011, pp. 89–96.

[14] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '11. Aire-la-Ville, Switzerland: Eurographics Association, 2011, pp. 101–109.

[15] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max, "A contract-based system for large data visualization," in *Proceedings of IEEE Visualization 2005*, 2005, p. 190–198.

[16] A. Matthes, A. Huebl, R. Widera, S. Grottel, S. Gumhold, and M. Bussmann, "In situ, steerable, hardware-independent and data-structure agnostic visualization with ISAAC," *CoRR*, vol. abs/1611.09048, 2016. [Online]. Available: http://arxiv.org/abs/1611.09048

[17] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[18] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (ToPC)*, 2016.

[19] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/Viz: A nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, Oct 2013, pp. 67–75.

[20] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, pp. 119–132.

[21] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O middleware for location-flexible scientific data analytics," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 320–331.

[22] A. Champsaur, J. Lofstead, J. Dayal, M. Wolf, G. Eisenhauer, P. Widener, and A. Gavrilovska, "SmartBlock: An approach to standardizing in situ workflow components," in *Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, 2017.

[23] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, IL, May 2014. [Online]. Available: https://hal.inria.fr/hal-00941413

[24] Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Tech. Rep. Ottawa, Ontario, Canada, 1966.