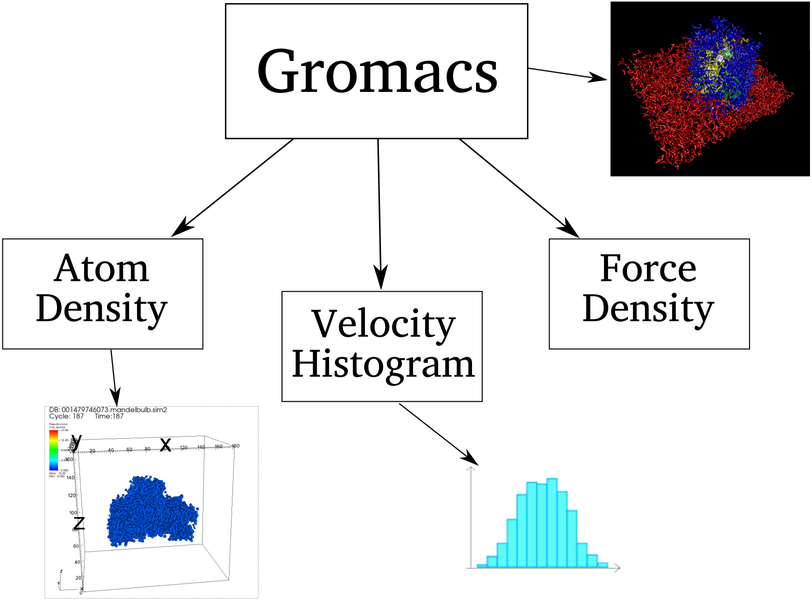
Automatic Data Filtering for In Situ Workflows

Clément Mommessin (ANL), Matthieu Dreher (ANL),
Tom Peterka (ANL), Bruno Raffin (INRIA)

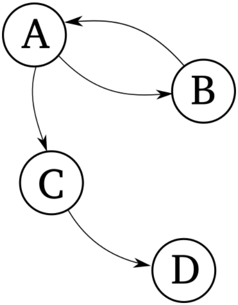
June, 16th, 2017

Scientific Workflow Example: Molecular Dynamics

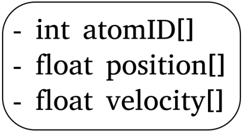


Definitions

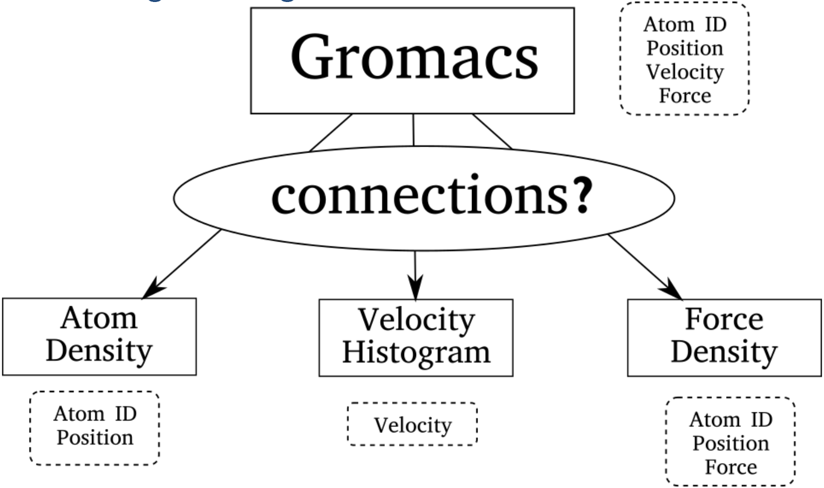
- An **in situ workflow** is a directed graph.
- **Nodes** are parallel tasks, sending and receiving data from other nodes.
- An **arc (or dataflow)** is a communication channel between a **producer** node and a **consumer** node.



- A **data model** is a structure containing **data fields**
- A message is a serialization of a **data model**

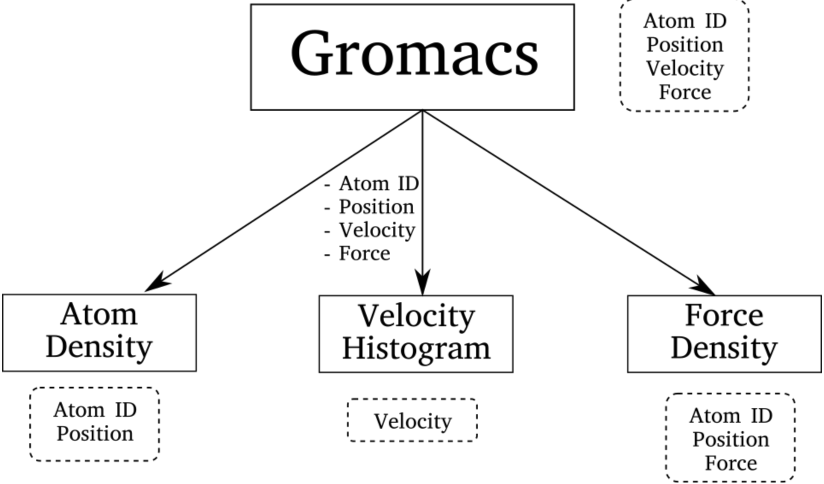


Data Exchange Management



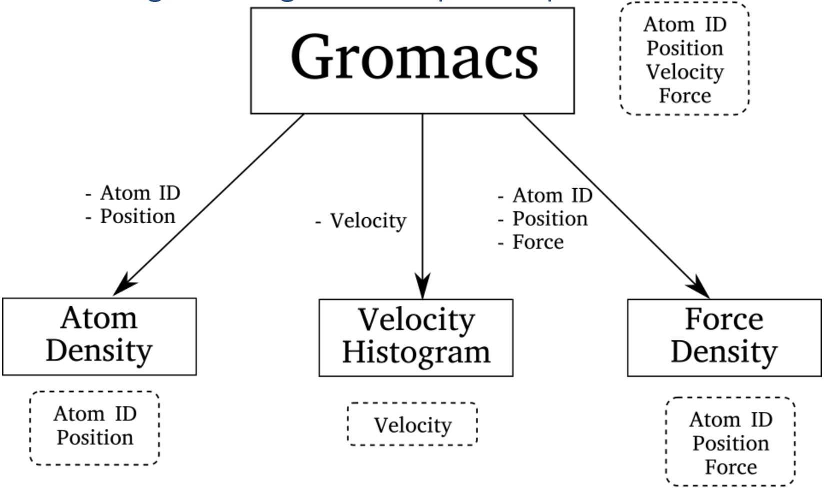
⇒ Typically two solutions

Data Exchange Management: Broadcast Everything



- ✓ No code modification
- X Extra cost to create and send the data
- X Unnecessary data on the network

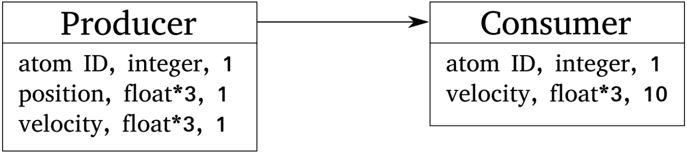
Data Exchange Management: Specific per Consumer



- ✓ Send only what is necessary
- X Code modification for every workflow

Solution Proposed: Automatic Data Filtering

Contract mechanism: a description of the inputs and outputs of each node for automatic message checking and filtering at runtime.



Our objectives:

- Send only **necessary data** over the network
- Select data to compose a message **automatically**
- Improve **reusability** of the user code
- Enable **type checking** of data

Contract Model

A **contract** is a list of data fields present in a data model and can describe:

- The data **output by a producer**
- The data **needed by a consumer**

Each data field is represented by a triplet:

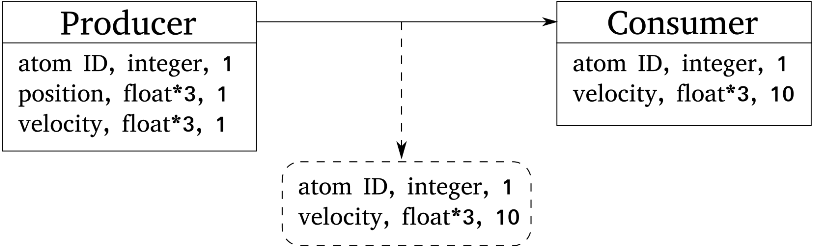
- **Name:** The name of the field
- **Type:** The type of the field
- **Periodicity:** The frequency at which the field appears in the data model

```
- atom ID, integer, 1
- velocity, float*3, 10
```

Contracts Checking

A **matching list** describes the minimal list of data fields a producer has to send to a consumer and is computed as follows:

1. Check that all fields required by the consumer are in the producer contract, with the same name and type
2. Add these fields to the matching list with the correct periodicities

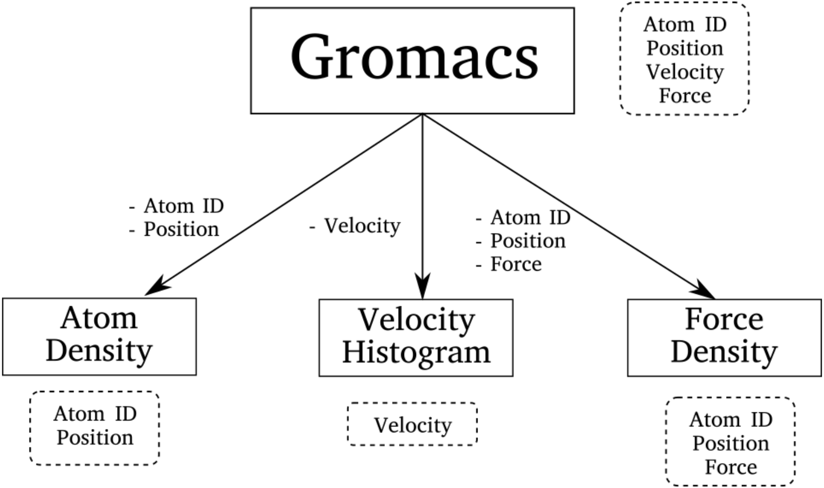


Message Filtering at Runtime

Matching lists are used for **automatic type checking and selection of data** by a middleware at runtime:

1. Take the full data model output by the producer
2. Compare the actual data types with the matching list
3. Form a data model containing only the required data using the matching list
4. Send the new data model to the consumer

Data Exchange Management: Use Contracts



- ✓ Send only what is necessary
- ✓ No code modification

Integration of Contracts Within Decaf

Decaf is a middleware for building and executing in situ workflows where:

- Nodes are **parallel tasks**
- Edges are **parallel communications** between nodes

Launching a workflow is done in 2 steps:

- Declaration of nodes and edges to construct the workflow graph with a **Python API**
- Task execution and management with a **runtime system**, providing **put/get** methods to exchange data between nodes

Modifications of Decaf

Modifications of the Python API:

- Creation of contracts for input and output when declaring nodes
- Checking that producer and consumer contracts are matching:
  - Type checking
  - Computation of matching lists

Modifications of the runtime, at each call to put:

- Type checking of the data using the matching list
- Automatic data selection of fields in the matching list

$\Rightarrow$ Transparent to the user

Performance Evaluation

Evaluation of the cost and performance of message filtering with 2 experiments:

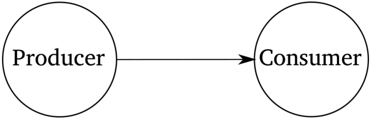
- **Overhead** of message filtering when contracts are not needed
- **Performance impact** on a real scientific workflow

Experiments conducted on the Froggy cluster (https://ciment.ujf-grenoble.fr), 190 nodes, 16 cores per node, FDR InfiniBand network of 60 Gbit/s.

Message Filtering Overhead

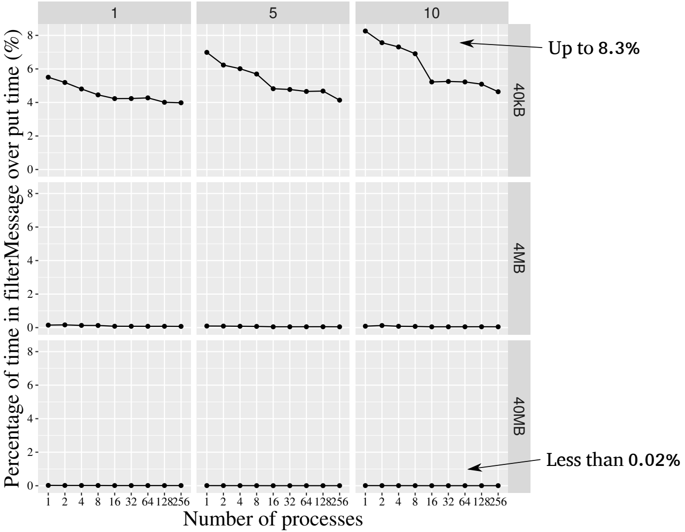
What is the cost of filtering messages when not needed?

- Hand-made example of one producer and one consumer
- Identical producer and consumer contracts
- Variable number and size of fields sent, variable number of processes per node



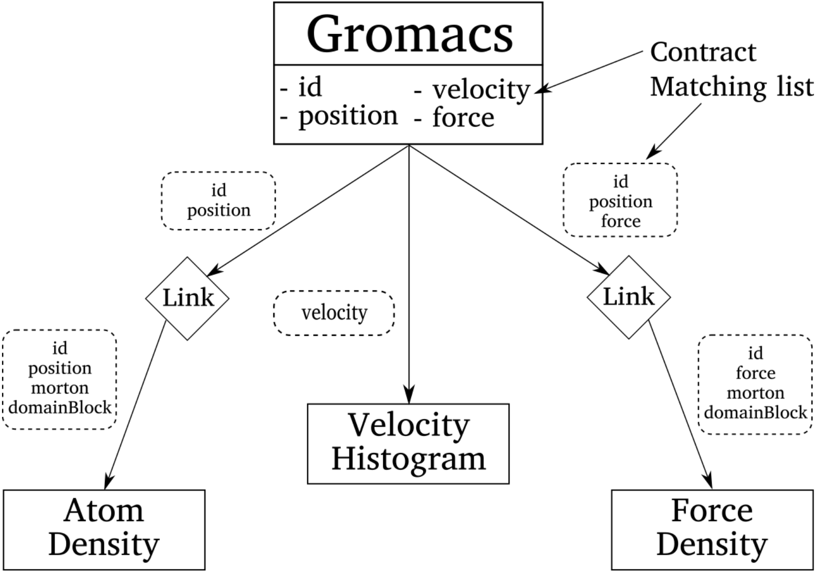
- Measurement of time spent in put and in the filtering function for 1000 messages sent

Message Filtering Overhead: Results



⇒ Virtually no cost when size of a message is a few MB

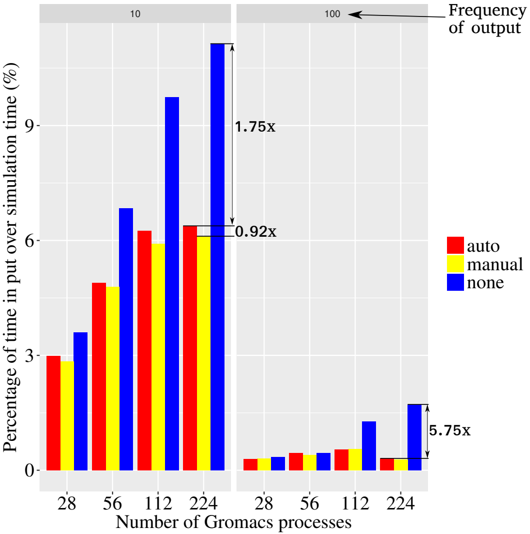
Performance on Real Workflow: Setting



Real Scientific Workflow: Setting (cont.)

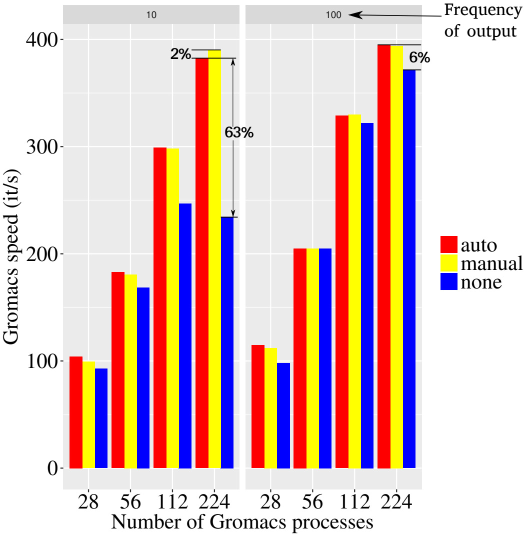
- Performance impact with 3 filtering methods:
  - Automatic filtering with contracts in Decaf (auto)
  - Manual filtering at the producer level (manual)
  - No filtering of message (none)
- Simulation data output every 10 or 100 iterations
- Measurement of time spent in simulation and in put for 200,000 iterations
- Molecular model of the FepA protein (about 70,000 atoms)
- Up to 224 cores for the simulation (maximum scalability), 4 cores for each analysis

Performance on Real Workflow: Results



Less data to serialize and send ⇒ less time spent in put

Performance on Real Workflow: Results (cont.)



⇒ Frequency gain more significant because less contention in network

Conclusion

- Design of a contract model to describe producer data outputs and consumer data requirements
- Automatic type checking and data filtering of message by a middleware
- Integration into the Decaf middleware

- Removes the I/O management from the user code
- Improves reusability of user code in different workflows
- No unnecessary data in communication channels

Future Work

- Integrate contracts mechanism in FlowVR and EVPath.
- Declaration of contracts at runtime

Work submitted to 2017 IEEE Cluster

Source code available at: https://bitbucket.org/tpeterka1/decaf

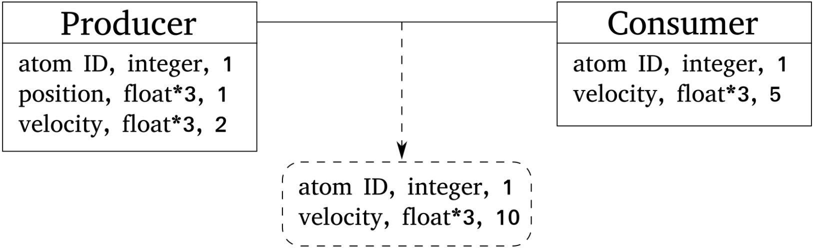
References

- *Decaf: Decoupled dataflows for in situ high-performance workflows*, Submitted to 2017 IEEE Cluster, Sept. 2017.
- *Bredala: Semantic data redistribution for in situ applications*, 2016 IEEE Cluster, Sept. 2016.

Thank you for your attention!

Any question?

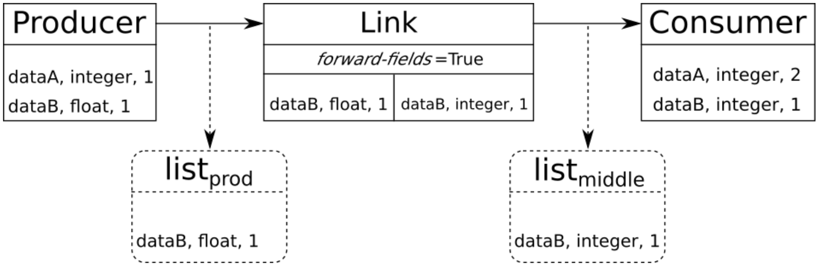
Periodicity Example



Middle Contract Example

Two matching lists are computed:

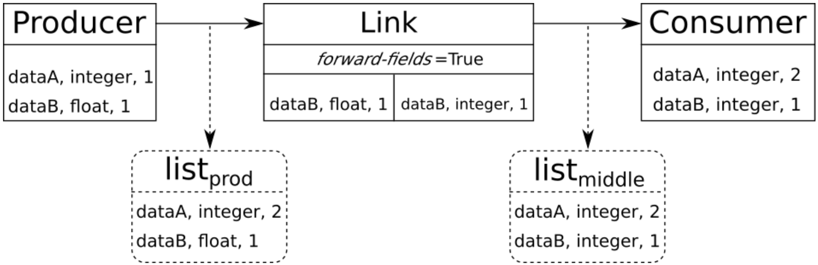
- $\textbf{list}_{\textbf{prod}}$ between the producer contract and the middle input contract
- $\textbf{list}_{\textbf{middle}}$ between the middle output contract and the consumer contract



Middle Contract Example

Two matching lists are computed:

- **$\text{list}_{\text{prod}}$** between the producer contract and the middle input contract
- **$\text{list}_{\text{middle}}$** between the middle output contract and the consumer contract



Python Example

```python
#Node declaration
producer = Node('producer', start=0, nprocs=4, cmdline='program')
producer.addOutputFromDict({'dataA':['int', 1],
                            'dataB':['float', 1]})

consumer = Node('consumer', start=6, nprocs=2, cmdline='program')
consumer.addInputFromDict({'dataA':['int', 2],
                           'dataB':['int', 1]})

#Edge declaration
edge = Edge('producer', 'consumer', start=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.setForwardField(True)

#Populating and processing the graph
graph = DiGraph()
graph.addNodes([producer, consumer])
graph.addEdge(edge)

processGraph(graph, 'program')
```

Algorithm 1: Computing a Matching List

Input: A producer and a consumer contracts (*prod-contract* and *cons-contract*).

$matching = \emptyset$
forall $(name,\ type,\ cons\text{-}period) \in cons\text{-}contract$ **do**
  if $\exists\ (name,\ type,\ prod\text{-}period) \in prod\text{-}contract$ **then**
    $periodicity = cons\text{-}period \times prod\text{-}period$
    $matching = matching \cup \{(name, type, periodicity)\}$
  else
    print "ERROR: data field mismatch"
  end
end
return *matching*

Algorithm 2: Data Filtering at Runtime

Input: The original *data*, the matching *list* and the current *iteration*

```
filtered_data = Empty message
forall (name, type_contract, periodicity) in list do
    if iteration % periodicity == 0 then
        if name ∉ data then
            ERROR: "field not in data"
        end
        field ← getData(data,name)
        type_field ← getType(field)
        if type_contract ≠ type_field then
            ERROR: "types do not match"
        else
            Add field in filtered_data
        end
    end
end
return filtered_data
```

Runtime Example

```
// retrieve message from input
pConstructData in_data;
while(decaf->get(in_data, "In"))
{
    // retrieve the value recieved
    int value = 0;
    SimpleFieldi field = in_data->getFieldData<SimpleFieldi>("var");
    value = field.getData();

    // create a field with the new value
    // and add it to a new data model
    SimpleFieldi new_field(value + 1);
    pConstructData out_data;
    out_data->appendData("new_var", new_field,
                         DECAF_NOFLAG, DECAF_PRIVATE,
                         DECAF_SPLIT_KEEP_VALUE,
                         DECAF_MERGE_ADD_VALUE);

    // send the data model containing the new value
    decaf->put(out_data, "Out");
}
```