Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option Parallel, Distributed and Embedded Systems

A Programming and Data Model for In-Situ frameworks*

Jad DARROUS

22 June 2016

Research project performed at INRIA-LIG

Under the supervision of:
Bruno Raffin, INRIA

Defended before a jury composed of:
Noël De Palma, Professor
Martin Heusse, Professor
Olivier Gruber, Professor
Thomas Ropars, Assistant Professor
Arnaud Legrand, Chargé de recherche
Pierre Genevès, Chargé de recherche

June 2016

**This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir*

Abstract

Large-scale scientific simulations generate an enormous amount of data. Going to the Exascale era, this amount is expected to increase exponentially. Analyzing and extracting meaningful knowledge from these data become more and more a challenging task. Thus, relying on the classic ways for the analysis tends to be infeasible: Storing the simulation output to disk and reading it later for analysis can take more time than the simulation itself. The file system throughput does not increase with the same pace as computational power, making the file system the bottleneck when handling massive data.

The **in-situ** approach emerged as a prominent solution. In-situ frameworks allow the analysis codes to consume the simulation output, in place, as soon as it is generated. The data is processed while it resides in the main memory, consequently, bypassing the file system. The in-situ paradigm is to decouple the computational codes (simulation and analysis) from their I/O channels by employing a unified API to publish and consume the data through the in-situ framework. Following this paradigm, the simulation and analysis are no longer concerned about the source of the consumed data or the destination of the generated data. As a consequence, the framework is now responsible for optimizing the data movement through the scientific workflow.

Although it is a recent paradigm, many works have been done to design in-situ frameworks, but still there is no de-facto standard framework. Existing frameworks often rely on a data model exposing a very limited semantics about the transferred data. In this work, we propose to extend a dataflow oriented *Programming Model* with an advanced *Data Model*. We propose to annotate the I/O data with a *key* and describe it with a *schema*. The schema holds the semantic information about the data and serves as *Data Interface*. The key and schema bring more flexibility and allow higher level control for the scientist over the scientific workflow. The framework can automatically optimize data movement by transferring the required data declared by the components. Furthermore, the knowledge about the data movement patterns paves the way to sophisticated optimizations such as task placements and resource allocation.

Contents

Abstract **i**

1 Introduction **1**
1.1 What is In-situ Analytics? 1
1.2 In-situ Approaches 2
1.3 Our Scientific Contribution 3

2 State of the Art **5**
2.1 HPC Domain 5
2.2 Big Data Domain 8
2.3 Map-Reduce in Scientific Domain 10

3 Architecture Design **13**
3.1 Programming Model 13
3.2 Augmenting the Workflow with Metadata 14
3.3 Data Model 15
3.4 Data Distribution 17

4 Prototype **19**
4.1 The Conduit Library 19
4.2 Data Schema 20
4.3 Programming API 20
4.4 Configuration Script 21
4.5 Development Status 21
4.6 Example 22
4.7 Discussion 25

5 Conclusion **27**

Bibliography **29**

— 1 —
Introduction

Large-scale scientific simulation codes are often highly parallel optimized codes that run on a big cluster of Symmetric Multi-Processing (SMP) nodes. The simulation output, that can reach hundreds of terabytes or even petabytes, is stored to disk for analysis. The scientist connects to the disk and reads the data into his machine to do the analysis and visualization. This method is called post-mortem or post-processing. While the computational power of supercomputers increases at a steady pace, the gap between the computation and the memory bandwidth increases. As a result, the file system becomes more and more the bottleneck in this process.

1.1 What is In-situ Analytics?

In-situ analytics is a new approach for analyzing the ever-growing amount of data that is generated from scientific simulations. Indeed, it is seen as a deep paradigm shift [8] because it radically changes data management and analysis. It has the potential to impact several HPC environment aspects such as resource allocation strategies, analysis algorithms, and data storage. In High-Performance Computing (HPC) domain, in-situ refers to process the data as it is generated, while the simulation is running. The data is analyzed when it is still in the main memory without any involvement of the file system. Only the final results, which usually are significantly smaller than the raw results, are stored in persistent storage. In addition, in-situ allows monitoring the simulation progress and stopping it when an error or a deviation of the expected behavior occurs, avoiding to waste computing resources. Also, in-situ allows the scientist to do a live steering of the simulation which is impossible by post-mortem analysis.

Scientific codes are maintained and optimized over years or even decades. They are complex codes and difficult to modify. The biggest challenge of in-situ analytics is to decouple the codes from their I/O channels and define a working environment that orchestrates the data movement between the simulation and decoupled analysis. The simulation is no longer responsible for the destination of the output data whether to write it to disk, send it over the network, or even discard it. Similarly, the analysis behaves independently of the source of data. Successful decoupling is a key point for in-situ frameworks.

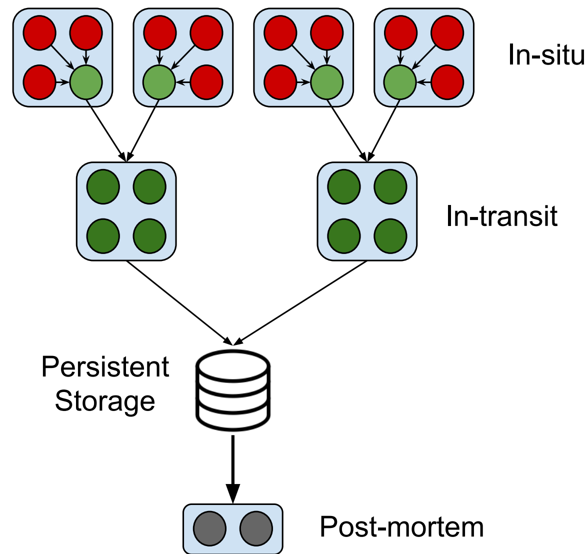
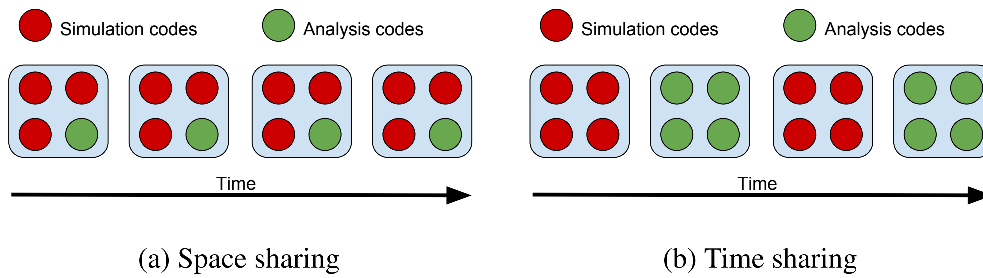


Figure 1.1: InSitu approaches

1.2 In-situ Approaches

In-situ approaches can be categorized according to *where* and *when* the analysis takes place. To optimize memory use, the simulation and analysis codes should avoid data replication as much as possible. Embedding the analysis codes directly into the simulation code allows sharing the same data structures without any data replication. This approach is called **in-simulation**. Also, it allows the analysis to take advantage of the already parallelized simulation code to perform the analysis in parallel. But the downside of this approach becomes clear when modifying or adding new analysis, which could break the integrity of the code base and make the maintenance more complicated. The **In-situ** approach runs the simulation and the analysis on the same node as two separate programs and takes the form of *time-sharing* or *space-sharing*, as depicted in Figure 1.2. In the time-sharing mode, the simulation and analysis run alternately on the same computation units. It requires performing context switches that have an effect on both performance and cache memory. In addition, it perturbs the synchronization between MPI processes that can lead to significant performance drops. This mode cannot be applied on some supercomputers that do not allow binding more than one thread to each core, like IBM Blue Gene for instance. While in the space-sharing mode, some cores of the simulation nodes, also called *helper cores*, are dedicated to run the analysis. The analysis runs asynchronously with the simulation and shares its data structure using a shared memory, managed by the framework. This mode allows sharing the data without any replication and provides the required flexibility to develop a modular analysis that can be integrated with different simulations. The same can be applied when using dedicated nodes instead of dedicated cores, called *staging nodes*. The data is moved over the network from the simulation nodes to the staging nodes. Staging nodes allow performing computation intensive analysis without disturbing the running simulation. This approach is called **in-transit** because the data is analyzed on its way to the final storage. The in-situ, in-transit and post-mortem can be combined as shown in Figure 1.1.

In addition to performance which is one of the main goals in HPC, in-situ frameworks should be easily used and deployed by physicists, biologists, chemists and other computational



(a) Space sharing (b) Time sharing

Figure 1.2: In-situ modes: In space-sharing mode (a) one core of the four is dedicated to the analysis and the other for the simulation. In time-sharing mode (b) the same cores alternate running the simulation and the analysis.

scientists that know about programming and coding but are neither their job nor their interest to work with complex libraries and frameworks or write long and complex configuration files. Tolerating some performance degradation in favor of easier programming and fast development is often accepted by the scientists who use the framework.

Large-scale data problems beyond the scientific and HPC domain are often referred as **Big Data**. Big data frameworks focus on the ease of development to increase developers' productivity. Also, these frameworks have built-in fault tolerance mechanisms as they are usually deployed in the Cloud. On the other hand, the computations are often not intensive but the amount of data to be processed is huge. In this work, we try to use some features provided by big data frameworks and apply them in the scientific domain.

1.3 Our Scientific Contribution

In-situ frameworks involve defining a programming model, a data model, and a workflow management that allows independently developed simulations and analysis to work together. The power of an in-situ framework comes from how efficiently it decouples the simulation and analysis from their I/O routines. In the classical way, the simulation decides what to do with the generated data, whether to store it to disk, or send it over the network to another remote node, or even discard it. In opposite we propose to make the framework responsible to decide how to handle the generated data. For a proper and efficient handling of data, a strong programming model and data model should be employed. Using the right programming model has a significant effect on the whole framework. It controls the extensibility, scalability, and flexibility in addition to the performance and ease of use of the framework. We propose a data model where the exchanged data between the computation components is annotated as a *Key/-Value* pair. The Key is a tuple that is exposed to the framework as filtering criteria. Whereas, the Value part describes the schema of the data with semantic information such as names and types. The Key/Value pair gives the scientist more control over the workflow using the key and allows the framework to optimize data movement by transferring just the needed data as described by the value. Each component declares the list of input and output Key/Values pairs. These declarations can be seen as a *Data Interface*. This interface allows the framework to do a static type checking and discover data mismatch at compile time preventing many hard to debug runtime type errors. In addition, the interface makes the component self-described and simplifies the integration of independently developed components in the same scientific work-

flow. Furthermore, it gives the framework more information about the data movement patterns than can be used for sophisticated optimizations as task placements. We show that combining the dataflow model as a programming model and the proposed data model can deliver a strong base for in-situ frameworks.

The rest of this report is organized as follows. Chapter 2 presents the state of the art of in-situ frameworks in the HPC domain and later presents an overview of the Big Data frameworks. Chapter 3 presents our main contribution with regard to the programming model and the data model. In chapter 4, we present the developed prototype. And chapter 5 concludes the report.

— 2 —
State of the Art

2.1 HPC Domain

Though the in-situ paradigm is considered a recent research domain, many works already attempted to solve the problem of big data movement. Optimizations can be applied to the whole software stack, from the underlying hardware to the application level. We briefly present the works done to improve the lower level layers and then go more into details to study the proposed in-situ frameworks and middlewares and discuss their main characteristics.

On the lowest software level, parallel file systems play an essential role to exploit the maximum throughput of the disk. These systems spread data across multiple storage nodes to achieve high performance and scalability. The Luster file system [3] is designed specifically for HPC computing environments. Google File System [22] is a distributed file system designed for large distributed data-intensive applications and it is optimized for huge files of multi-GB. GPFS [24] is IBM's parallel, shared-disk file system for cluster computers. GPFS provides, as closely as possible, the behavior of a general-purpose POSIX file system running on a single machine. Despite the constant development in file systems, the disk throughput remains a limiting factor that cannot be surpassed and a new approach to handle huge data becomes a must.

For files formats, many formats are developed that focus on performance, expressiveness and preservation. NetCDF [6] (Network Common Data Form) is a set of software libraries and self-describing, machine-independent data formats that support array-oriented scientific data. HDF5 [7] (Hierarchical Data Format) is a data model, library, and file format for storing and managing data. It is designed for flexible and efficient I/O and for high volume and complex data. NetCDF and HDF5, and their parallel versions, are mostly used to store scientific data instead of simple binary files or text files. These formats keep the semantic information about the stored data, which facilitate browsing it and accessing it later, but it is applied only to the data stored in files.

In contrast to files, Conduit [2] provides a schema to describe in-memory data structures. Conduit is a *Scientific Data Exchange Library for HPC Simulations*. It can describe, by giving names and types, a raw array of bytes with a schema. The schema is a JSON string that contains the name of the variables with their types and their position in the array. The data represented by Conduit can be serialized and deserialized into an array of bytes. In addition to primitive types, Conduit supports hierarchical data to further increase its expressiveness by supporting more complex data structures.

On the parallel libraries level, MPI-IO is an MPI extension to handle file I/Os. It allows collaborative reading and writing to the same file by the parallel processes. ROMIO is a high-performance, portable MPI-IO implementation. MPI data types are concerned about data size and data structure in memory rather than data semantic. Parallel I/O libraries can increase the I/O performance in parallel programs, but still do not employ any model for the data that shows the semantic of the data.

ADIOS [23] is a middleware that can greatly improve parallel I/O performance, simplify parallel I/O coding, and improve code portability. Integrating ADIOS into simulation codes is done by replacing I/O API calls by ADIOS API that is very similar to the POSIX API. The exact data transport method is specified in an external XML configuration file. The data types of the variables are also specified in the XML file. ADIOS supports many transport methods such as POSIX, MPI, HDF5, MPI_LUSTRE and NULL for testing. ADIOS improves the I/O performance by buffering the data before invoking the actual transport method. Changing the I/O method is done without recompilation, which allows fast debugging for the application to select the best one. ADIOS uses a custom BP (Binary Packed) file format, and provides tools to convert it to other standard formats as NetCDF and HDF5. This step becomes critical as the data increases in size and the conversion time become significant. ADIOS describes the data structure in an external XML file, but this description is used to optimize the I/O operations, and not used as a data model.

To enable live analysis and visualization, VisIt [4] and ParaView [11] developed their own pipeline for in-situ coupling, using the in-simulation approach. These libraries are the leading visualization libraries and use the Visualization Tool Kit (VTK) [5] as the data processing and rendering engine. VisIt provides the Libsim library [31] for in-situ visualization and analysis. The simulation code needs to be instrumented so VisIt can connect to it and access its data directly. For ParaView, the Catalyst library [15] uses an adapter to mediate between the simulation and ParaView. The visualization requests the required data from the adapter. The Adapter plays the role of mapping the simulation data structure to the visualization format. Although these methods provide some modularity for the development, they impose a rigid coupling with the visualization software; Changing the visualization library or integrating another analysis requires many code changes.

Damaris [20] is a middleware that exploits the benefits of helper cores to run asynchronous I/O operations. On each node, some cores are dedicated to the in-situ analytics. Damaris consist of a set of MPI processes, one process on each helper core. The simulation outputs of other cores are aggregated on the helper cores and transfered asynchronously to the visualization software. To get the benefit of the computation power of the helper cores, Damaris is enriched with a plug-in system to perform simple local operations as filtering, indexing and compression. Sharing data between the cores of the same node is managed by the middleware, and done using shared memory. Damaris uses an XML file to specify the name, type and size for the output variables of the simulation. This description of the data allows automatic conversion between the simulation data structure and the visualization or analytics software data structure.

DataStager [10] uses staging nodes as a cache to hold data on it way to the file system, and as a service that delegates some metadata management operations that can be a bottleneck for the file system. DataStager addresses the jitter problem of parallel applications by moving data asynchronously to staging nodes by leveraging RDMA technology. PreDatA middleware [34], the successor of DataStager, allows pluggable analysis to be performed on either computation node or staging nodes. PreDatA exploits the additional computational and memory resources

provided by the staging area. PreDatA is introduced to the compute nodes through ADIOS, thus, no changes to the application codes is required and the used data model is what ADIOS provides.

DataSpaces [19] takes a different approach to manage the staging area. It implements a virtual shared space abstraction that can be associatively accessed by all the components of the workflow. DataSpaces relies on Distributed Hash Table (DHT) to store simulation data. The simulation output is sent to the staging nodes where it is indexed and stored in the main memory. This indexed data can be later accessed by the DataSpaces query engine. DataSpaces was integrated with ADIOS as an I/O transport method.

GLEAN [28] is a flexible and extensible framework that takes application, analysis and system characteristics into account to facilitate simulation-time data analysis and I/O acceleration. GLEAN is integrated into two scientific codes, FLASH and PHASTA. GLEAN uses in-situ to perform some custom analysis and in-transit for visualization. GLEAN can scale up to 160K cores and achieve up to 48 GiBps for data movement. Analysis in GLEAN leverages the data models and structures of the studied scientific simulations (FLASH and PHASTA), but the framework does not offer a generic way for data modeling.

To select analytics placement, FlexIO [36] is designed mainly for flexible placement of in-situ data analytics. It is implemented as an extension of ADIOS. FlexIO offers the flexibility in where analytics codes are placed. It could be on compute nodes (either inline or on helper cores), on staging nodes, on both, or offline. Also, FlexIO offers the ability to alter such placements without requiring application codes to be changed or updated.

GoldRush [35] aims to improve performance and resource usage by exploiting idle CPU cycles. GoldRush runs in-situ analysis codes on idle cores when executing sequential periods of an OpenMP parallel program. The aggregate duration of these periods can be up to 65% of total execution time on the real-world codes tested.

Most of the previous works do not have a model for their data and treat them as raw byte streams. Others, as ADIOS, use simple data model but for I/O optimization that does not describe the semantic of the data. Damaris data model is mainly to map the simulation data structure to the visualization format, and it just describes the output of the simulation. MPI data types are concerned about data size and data structure in memory rather than data semantic. To the best of our knowledge, this work is the first to study a comprehensive data model for in-situ scientific analytics.

2.1.1 FlowVR

FlowVR is a framework that was initially developed for Virtual Reality applications [14]. FlowVR facilitates the coupling of heterogeneous parallel codes to build large-scale applications. FlowVR uses a dataflow model and it is based on a component model. In the dataflow model, computation programs are represented as *Modules*, also called components. These modules are linked through unidirectional communication channels. Modules are developed independently and each represents a standalone program. FlowVR defines the simulation and visualization as modules that communicate through implicit links managed by the framework. Modules define input and output ports to receive and send data respectively. The links between input/output pair of ports are declared in a configuration file. The configuration file can be changed to assemble a new workflow without any recompilation of the modules. The FlowVR API for developing a module relies on 3 main primitives: *wait*, *get* and *put*. The main loop

of the simulation should be instrumented with these primitives to integrate with FlowVR. The main loop structure follows this schema:

```
// holds while there is no message
while wait():
  // read the message
  message = get(input_port)
  // do the simulation
  new_message = process(message)
  // send the new message
  put(output_port, new_message)
```

The *put* or *get* does specify explicitly a source or destination. The source and destination of a message are defined by the link set between the producer and consumer modules through the module assembly. Then, the FlowVR runtime that knows where the modules are running and how they are linked takes care of transporting the message. If the modules run on the same node, it simply consists of a pointer exchange. If they run on different nodes, FLowVR takes care of building the message to deliver it to the destination using TCP/IP of MPI as a transport layer.

Later on, the framework was redesigned to run analysis in-situ [21]. It enables fine-grained control over module placement. Modules can be assigned to specific cores within specific nodes. The configuration file has been replaced with a configuration script written in Python. This script is used to represent the scientific workflow as a direct graph where modules are vertices and links are the edges; modules are defined and later linked. Modules definition includes, in addition to the placements, the command line argument to run the module's code. Changing the configuration script allows changing the workflow without any recompilation of the modules. This simplifies finding the best placement of the modules that brings the highest performance. For efficiency, FlowVR employs zero copy; Shared memory is used to send messages between modules running on the same node. RDMA (Remote DMA) technology is used to transfer data between modules running on different nodes. Notice that, FlowVR cannot exploit the full power of BlueGene operating systems, because these systems do not allow the execution of multiple processes on the same core, which forces binding the daemon process alone to a single core, thus wasting the computational power of one core per node to execute non intensive operations.

In FlowVR, data semantic is left to the module developer. Data are transferred as arrays of bytes, which mean an implicit convention should be employed between modules to treat correctly the data.

2.2 Big Data Domain

The Map-Reduce paradigm [18] is proven to be easy to write and can solve a lot of business problems. Map-Reduce operates in two phases. During the map phase each row of the input data is mapped to a Key/Value pair. Then, the pairs that have the same key are grouped and each group is processed by a reducer, which is the second phase. The Key/Value organization is the important element of the Map-Reduce data model. It is a simple yet efficient way to structure data. The Key accepts any type with defined comparator. Whereas, the Value part does not specify any model and it is treated as *byte stream* data.

Map-Reduce programming model is widely adopted for large-scale, data-intensive applications. Its power comes from the fact that sequential code can by executed massively in parallel.

Apache Hadoop [30] is the widely deployed open-source implementation of the Map-Reduce programming model. Map-Reduce is specifically designed for scalability and provides built-in fault-tolerance. Map-Reduce is employed to solve various large scale data problems, i.e. indexing the web. Hadoop runs on the resource manager YARN and uses Hadoop Distributed File System (HDFS) as the interface for the persistent storage. Hadoop can be deployed on any GNU/Linux cluster or personal computer running on GNU/Linux.

Hadoop uses *batch processing* to handle the data. Batch processing is very efficient in processing huge volume of data. Where data is collected, entered to the system, processed and then results are produced in batches. In batch processing, the data should be available upfront in a (distributed) file system before running the jobs. This model can achieve high throughput but suffers from the high latency problem. Thus, it's not suitable for iterative processing; the data should be written to disk and another job is launched and read the data again, from the disk.

Spark [32], in contrast with Hadoop, deals efficiently with iterative jobs by keeping intermediate results into memory. Spark output is managed by Resilient Distributed Datasets (RDD). RDDs are immutable, fault-tolerant, and distributed collections of objects that can be operated in parallel.

Another processing mode starts to emerge, mainly to meet business needs, which is *stream processing*. In contrast to batch processing, stream processing involves continuous input and outcome of data. The main goal is to process data with low latency. Stream processing frameworks as Storm [26] from twitter and MillWheel [12] from Google are prominent examples.

In order for Spark to deal with stream processing, it came with the idea of discretized streams [33] (Apache Spark Streaming); converting a stream of data to micro batches that can be processed in batch mode. This idea was criticized as it does not represent the right semantic of streaming. Later on, frameworks that are characterized as true stream processing were developed. Flink [1] is a batch and true streaming framework. The same code logic can be applied to batch and stream processing. Google Cloud Dataflow [13], which is based upon MillWheel, supports batch and true streaming.

For the programming model, Storm uses what we can call an **explicit dataflow model**; A direct graph is built from spouts and bolts which represent the source of data and the data manipulation logic respectively. These spouts and bolts are the nodes in the graph. The nodes are linked by predefined types of links such as shuffle or GroupByKey. The user can control the number of running instances of each node by specifying its *degree of parallelism*. On the other hand, Flink uses an **implicit dataflow model**; The user applies the desired transformations and operations on the data source. These transformations are later translated to a data flow graph. The implicit data flow model can greatly simplify the development and deployment of large-scale data workflows.

TensorFlow [9] is a large-scale Machine Learning framework. It can be deployed on heterogeneous hardware architectures, from mobile devices to a grid. A TensorFlow employs a dataflow programming model. The computation is described by a directed graph, which is composed of a set of nodes. Nodes represent the instantiation of an operation and have zero or more inputs and outputs. Tensors are arbitrary dimensionality arrays and represent the flowing data between the nodes. Tensors flow along edges in the graph from outputs to inputs. TensorFlow codes are translated to a data flow graph that can be visualized to the user. Tensors are animated while they flow between the nodes. This visual representation can greatly simplify the understanding of the whole workflow. TensorFlow can be also classified under the implicit

dataflow model category. But when the performance is a high priority, fine-tuned task placement requires an explicit description of the workflow. Implicit model is easier for the developer as the framework takes care about resource allocation but it might not guarantee the highest possible performance.

2.3 Map-Reduce in Scientific Domain

The benefits provided by the Map-Reduce paradigm in the web and business domain make it worth to try in the scientific domain. In Map-Reduce, users develop programs according to the programming API and the framework takes care of fault tolerance, load balancing, task management, and scheduling. These features have the same importance for in-situ frameworks. Implementing them allows scientists to focus on developing analysis codes without being concerned with deployment issues. Map-Reduce excels when dealing with data that can easily be sub-divided. Array-based scientific data have this property too adding another argument why Map-Reduce could be appropriate in HPC domain.

One of the problems faced with Map-Reduce when applied to analyze scientific data is the data format. Hadoop reads/writes the data from/to a HDFS. Scientific data are usually stored in NetCDF and HDF5 files. These file formats provide a logical view of the scientific data, which is usually represented as multidimensional arrays. Map-reduce with its simple byte stream data model cannot deal efficiently with these data formats. SciHadoop [17] integrates Hadoop with NetCDF library to allow processing of NetCDF data with Map-Reduce API. SciHadoop extends the Map-Reduce data model to handle scientific data taking their semantic into account. MRAP [25] is another example of Map-Reduce framework that provides access pattern to HPC analytics programs. MRAP enhances the Map-Reduce language with additional expressiveness that allows users to specify the logical semantics of their data.

HiMach [27] is Map-Reduce framework implementation to analyze Molecular Dynamics codes. It allows user program to conduct multiple rounds of reduce operations. The Key/Value pairs produced by the map phase and reduce phase (in case of multiple rounds) are saved to disk before executing the next phase. HiMach proposed that the Key should be thought as categorical identifier for a group of related values such as the identifier of the atoms to be analyzed. For the Value, it should carry a timestamp and the quantities of interest. The timestamp is used to reconstruct a valid series of frames. HiMach only supports post-mortem analysis.

SMART [29] is a sophisticated Map-Reduce like framework - written in C++11 - for in-situ analytics on many cores processors. It supports the two types of in-situ analysis: time-sharing and space-sharing. SMART achieves high performance compared to traditional Map-Reduce implementations by updating the reduction results in-place with no intermediate Key/Value pair emitted or stored; which removes the need for the expensive shuffling phase. Instead, SMART uses a two level combination phase. A local combiner maintains the reduction maps that accumulates the output of all processes running on the same node. A global combiner merges all these local reduction maps. Instrumenting an existing code to use SMART requires some code changes that are hard to maintain and done correctly for complex scenarios. For example, to run is-situ analytics in the space-sharing mode, the analysis should be launched from within the parallel OpenMP pragma. This requires a deep understanding of the code and non-straightforward modifications.

But the main question remains, to which extent the Map-Reduce model is suitable for scientific analytics. According to SMART, Map-Reduce can fit many applications as visualization algorithms, statistical analytics and feature analytics. It is specifically suitable to perform the embarrassingly parallel steps of analytics codes. In HiMach they noted that some Molecular Dynamics algorithms that exhibit temporal dependency between the frames of a trajectory should be redesigned to eliminate the strong data dependence between adjacent frames. In general, whether the Map-Reduce programming model is applicable depends on the data access pattern and algorithmic design of the analysis program.

We have to take the best of the two worlds, by figuring out the best way to employ the Map-Reduce concept in the HPC environment. Map-Reduce offers great programming flexibility and handles many important deployment aspects behind the scene. Using Map-Reduce as it is, prevents the low-level optimization that brings the performance to its highest level. The Key/-Value concept can offers some flexibility to annotate the data, while the simple bytes stream data model of Map-Reduce is not sufficient for scientific data. The dataflow model with Key/-Value for the exchanged data offers the flexibility and performance to assemble and execute scientific workflows. The programming model alongside with a rich data model that describes the flowing data provides strong base building blocks for in-situ frameworks.

— 3 —

Architecture Design

3.1 Programming Model

Defining the correct programming model is the first challenge faced when designing in-situ frameworks. Scientists, who are not HPC experts, are likely to develop their own analysis pipeline and analysis codes. Thus, they need tools that are flexible while offering a reasonable abstraction from the execution context without incurring performance loss as much as possible.

The MPI programming model requires expertise to optimize for complex scenarios. It has many powerful features, but this power brings with it a complexity. It is best suitable for SPMD (single program, multiple data) programs. Individual components of a workflow could be, and usually are MPI programs, but treating all the components as one MPI application does not provide the required flexibility. In MPI model, a fixed set of worker processes is created on program initialization with no possibility to dynamically launch new processes. In Damaris as an example, the simulation and visualization are treated as a single MPI program. A complete recompilation is needed for any code changes. In addition, any failure breaks down the whole workflow.

Using a virtual distributed shared memory, DataSpaces for instance, has the advantage of simple API and the ability to couple applications not running at the same time. On the other hand, it does not lead to the desired performance as the data do not flow directly from the producer to the consumer. Using the staging node as a temporary buffer until the analysis to be scheduled can not provide the highest performance. In a work done by the major players in the in-situ domain [16], one of their conclusion emphasizes the overhead of using staging nodes for data buffering purpose, and suggest that a more optimal approach would be to process the data in a streaming fashion rather than buffer it.

In the dataflow model, the workflow is represented as a direct graph where the vertices represent computational components (simulation or analysis codes). These components are linked together by communication channels that are represented as edges in the dataflow graph. Data are transferred between components as messages. The components interact with the framework by publishing and reading messages. Messages carry the actual data and flow in graph between vertices. Numerical simulations are usually iterative, at each step, one or more messages could by published to the framework. At runtime, the graph is instantiated according to the execution context. The framework takes care of deploying the application on the target architecture, and of coordinating the analytics workflows with the simulation execution. FlowVR is an example of this type of frameworks. The dataflow model enables stream processing of the data which

offers the highest performance. The graph can be plotted, giving the user a visual representation that is easy to understand and to explain. Debugging and monitoring tools can be developed and integrated into a dataflow framework to track the data movement in the graph and spot the strugglers' components that cause a bottleneck during the data flow. In the dataflow model, the components could be developed and tested independently, and integrated later in the workflow without any modification to other components.

TensorFlow and Flink are the most recent frameworks for distributed Machine Learning and Big Data processing respectively. They adopted the dataflow model in their runtime engine. In addition to the high performance, TensorFlow can draw an animated graph that shows how the data are flowing in the graph. Flink provides a low overhead fault tolerance mechanism in the absence of failure. It is based on Chandy-Lamport algorithm for distributed snapshots. Regular processing keeps going, while checkpoints happen in the background.

We are convinced that this dataflow approach is both easy to understand and yet expresses enough concurrency to enable an efficient execution of scientific workflows. For this work, we rely on the dataflow model of FlowVR.

3.2 Augmenting the Workflow with Metadata

The dataflow model is concerned about the high-level abstraction of the components and the connections between them. It does not provide any description of the flowing data and does not provide any control over it. We show that exposing some information about the flowing data to the workflow-level gives more flexibility for the user to control the data movement. Moreover, it allows the framework to optimize data movements and thus improve performance.

Let consider one example related to the frequencies of data production and consumption. It is often the case that the analysis consumes the data in a lower frequency than the frequency when it is generated. In principle, discarding the extra messages by the analysis solves the problem. If we suppose that the simulation and analysis run on different nodes, the network bandwidth will be consumed with useless data movement. A more classical approach is to move the filtering to the simulation, the simulation now sends the messages needed for the analysis and discard the others. This solution solves the extra movement of data but it modifies the simulation code in a way not related to the simulation logic; now, the simulation should decide what and when it pushes data to the analysis. The simulation should be designed without any knowledge that an analysis will consume its data with different frequency. Let now consider that we need to perform another type of analysis running at a different frequency than the first one. The simulation should be modified again to take into account the frequency of the newly added analysis. Modifying the simulation code for each new analysis becomes unpractical and hard to maintain. If the filtering is moved back to the analysis to control their frequency, the network will be saturated with data that eventually will be discarded.

Moving the frequency control to the workflow level is the key for an efficient solution that also brings more flexibility for the scientists. We propose to annotate the flowing data with a *key*. The key carries a description information about the actual data i.e. metadata. The key is designed to be more than an integer number or an identifier; it is a composite key that is represented as a tuple with named and typed parts. Having tuples enables to easily combine different types of metadata like iteration step and producer id (MPI process rank for a parallel simulation). If a single integer is required at some point, for instance to store the data in a Key/Value store, the tuple can be hashed to a single integer. Now, the frequency control can

be implemented as a predicate that returns `true` in order to process the message and `false` to discard it. This predicate is defined on the workflow level. It is implemented as a stateless function taking as input parameters the key where each key component is identified by its name. This predicate is attached to the communication channel between the simulation and analysis.

For example, suppose that the simulation annotates its data with a key that has one part named `(timestep)`. The value of the key is incremented by one for each newly produced frame. We assume that the analysis cannot process more than one message each ten. The filtering predicate is implemented as `lambda timestep: timestep%10 == 0`. By moving the frequency control to a higher level, the simulation and analysis codes are no longer concerned with it. It is the framework responsibility to receive all the messages from the simulation and to forward the ones that satisfy the provided predicate to the analysis. Changing the frequency can be done by modifying the predicate in the configuration script without any recompilation for the workflow components. Contrary to the previous solution, a recompilation should be done to either the simulation or the analysis after each modification. Notice that with this approach, the framework has the flexibility to choose where to execute it to optimize data movements.

Decorating the data with a composite key can provide control beyond the frequency control. The key can hold the partition number of the process that data belong to. Simulations usually run in parallel and the simulation space is divided between the parallel processes. The analysis could be applied on a subspace of the simulation space. In an analogous solution to the previous frequency control problem, the best way to perform this partitioning is to define the partition number in the key as `(partition)`. If we suppose that a parallel simulation run on 4 processes and we want to analyze just the fourth part, the predicate will be implemented as `lambda partition: partition == 4`.

More advanced selections that are related to the simulated domain can also be applied. For example, in a molecular dynamics simulation of a membrane protein, the objective of the analysis is to count the number of potassium ions that permeate through an assembly of atoms that make a channel. The analysis tracks the potassium ions and counts the times they pass through the channel. The analysis is applied only to the potassium ions and not concerned about other atoms. Transferring all the atoms to the analysis is not efficient as the potassium ions account for fewer than 1% of all the atoms. For this case, the key can hold the type of atoms as `(type)`. The simulation output in each time step multiple messages with different value of the key according to the type of the atoms. A filter like `lambda type: type == "potassium"` can be included in the workflow.

As the key is composite, more than one filtering criteria and be employed in the same predicate. For example, the key could be composed of two parts `(timestep, partition)` and the filter is implemented as `lambda timestep, partition: timestep%10 == 0 and partition == 4`.

3.3 Data Model

With a key, we annotate the data in a way to perform selection and filtering on it. The message is either discarded or forwarded. The actual data will be transferred as a single block. The framework treats the data as a raw bytes array. Knowing more about the structure of data allows the framework to perform more sophisticated movements of the data.

For example, in a weather forecasting simulation, the output consists of many variables such as temperature, wind speed, humidity and precipitation. To compute the average temperature

over a day, just the temperature is needed by the analysis; other information is irrelevant to the analysis. Sending the complete simulation output will be inefficient. On the other hand, modifying the simulation to send the temperature does not preserve the modularity of the simulation, as the simulation is now conceded with issues not related to the simulation itself. Relying on keys is not a sustainable option as it requires splitting the different types of data into different messages.

We propose to define a schema for the exchanged data. The schema is a hierarchical representation of the data structure. It can describe the data produced by the simulation and needed by the analysis. The schema gives names to the fields of the data structure. These names are further described by their types and sizes (in case of arrays). With these semantic information about the data, the framework can performs *schema matching*; just the fields needed by the analysis are extracted from the simulation and sent to the analysis.

Going back to the weather forecasting example, the simulation declares the structure of its output by a schema. Similarly, the analysis declares the required fields e.g. the temperature, in another schema. The framework now has the necessary information to extract and transfer just the data needed by analysis.

In addition to the main simulation output, usually, simulations output other types of data as monitoring data or Checkpoint/Restart data. For a more control over these data, they could be described also by a schema. In similar way, analysis could accept inputs other than their main input such as control information. In a complex workflow or in an analytics pipeline, the analysis components are also sources of data. This creates the need to distinguish between the input schema and output schema.

For each component a data schema describes its input ans output data. The schema is represented a tree structure. The root of the tree with the name `schema` has two optional children; `in` and `out`. They represent the information related to the input and output respectively. Each of it, has one or more child. Each child is referenced by a **tag** and has in turn two children: the `key` and the `value`. The key represents the key used in the workflow level. The value represents the schema of the actual data that is transmitted between components.

This tree structure can be implemented in any text format. Each component needs to declare its schema in an accompanied text file. The framework can statically read and parse these files to generate the rules of data extraction and forwarding.

The underlying mechanism for the data movement should respect the data's semantic. For example, simulation and analysis programs may not run on the same number of processing units, this requires the exchanged data to be split, merged and redistributed in an efficient way without breaking the data's semantic. Using a schema, the structure of each variable is known and can be extracted from the output message. This allows the framework to automatically split and merge simple data structures as one-dimensional arrays. Also, the user can implement its own split and merge routine for sophisticated distributions or to handle complex structures.

Explicitly declaring the I/O data schema of a component leads to many benefits. It allows the framework to automatically extract the required fields by the analysis from the complete simulation output. Also, it allows the framework to perform data type checking between the linked components. This checking can detect type errors during compile time preventing many errors at runtime. Moreover, the schema serves as an interface that eases the coupling between independently developed components. As the I/O data of the component is self-described, integrating new component is a workflow is simple as adding some line to the configuration script. In addition, explicitly declaring the schema allows the framework to do type checking to ensure

the compatibility between connected components. Furthermore, this information can be used to optimize task placement. Task placement is the mapping of the computational components to the available resources as computation nodes. As the framework has a complete knowledge about the source of the data needed by each component, it could search for a mapping that minimize the data movement.

3.4 Data Distribution

When running in a parallel context, many decisions should be taken to ensure the integrity of the exchanged data. Messages could be merged or split. These operations should preserve the data structure and do not have any effect on the workflow components. The considered patterns are depicted in Figure 3.1.

Suppose that a parallel simulation runs two processes and it is connected to a single process analysis. These two processes of the simulation generate messages with the same schema but with different values for the Key/Value pair. The user can decide whether to keep the simulation output as two separate messages or to merge them in one message. The default behavior is to keep all the messages without any merging. Merging is useful when we want the analysis to receive the simulation output as one message. In case of merging, the framework can merge the data according to their schema. In this case, the keys should be merged by a routine provided by the user to create the key of the new message.

On the other hand, suppose that the analysis runs in parallel with two processes and the simulation run with one process. The simulation output could be forwarded to the two analysis processes and each process decides which part of the message to handle. This method is neither efficient nor flexible. Because the message is transfered twice and the analysis code will be concerned about handling just part of the received message. A better approach is to split the simulation output message into two messages and forward each message to one analysis process. A split routine to be supplied by the user is required in that case. In addition, the keys on the new messages should be specified; it does not have to be similar to the original key.

For a more complex scenario, the simulation and the analysis run as parallel programs and with different number of processes, N and M respectively. This configuration is called a MxN data distribution. A merger followed by a splitter can implement this pattern but it is not the most efficient way to do it. A better approach is to develop an ad-hoc pattern that can take advantage of schemas when appropriate to save on code development. A high MxN construct could be provided by the framework that can be tuned with some parameter to fit the user case.

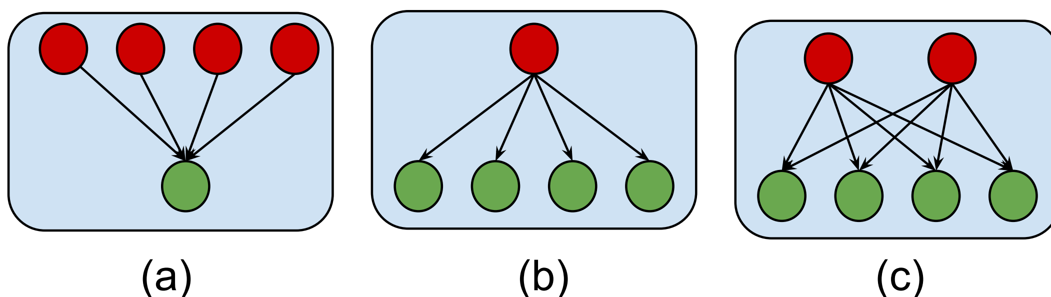


Figure 3.1: Data Distribution: (a) represents a merge pattern (b) represents a split pattern and (c) represents a MxN pattern

— 4 —

Prototype

In this chapter, we present a prototype implementation of the proposed programming and data model. Our prototype relies on a revisited version of FlowVR and Conduit. We conclude this chapter with a detailed example that clarifies the presented prototype.

4.1 The Conduit Library

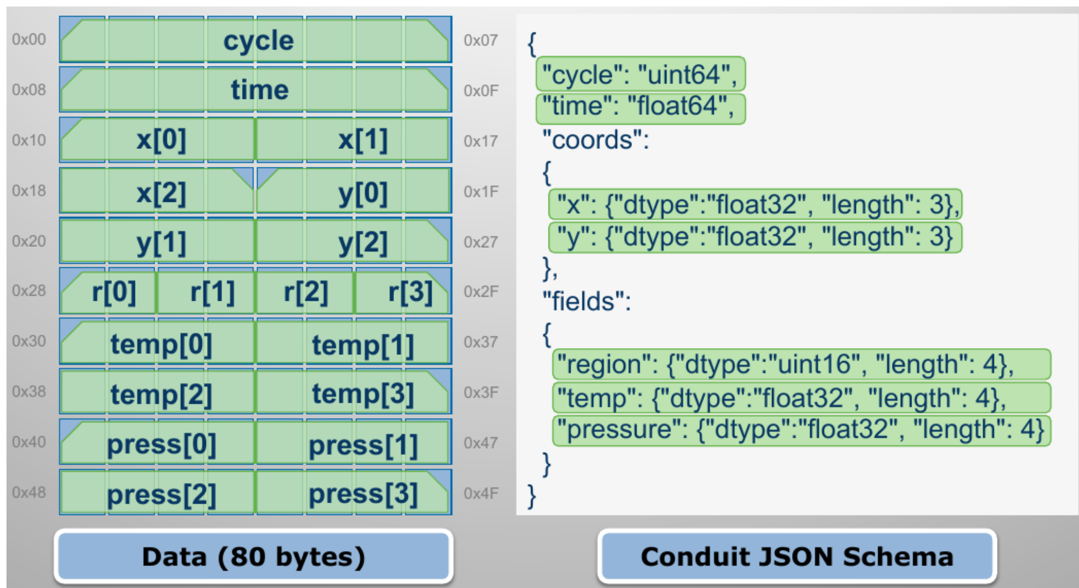


Figure 4.1: Conduit schema

Conduit is a software library that describes and manipulates raw bytes arrays described by a Conduit schema. The schema alongside the backed array is represented in Conduit as a `Node`. Conduit's `node` provides a map interface to access and manipulate the underlying data structure. Figure 4.1 shows a data structure based on an array of 80 bytes and its schema represented by Conduit. For our data model, we adopt the Conduit syntax for describing the Value schemas.

4.2 Data Schema

As discussed in the previous chapter, the data schema of a component is represented as tree structure. The root element contains the input and output schemas. The output schema contains the Key/Value pairs that could be produced by the component. Conversely, the input schema contains the the Key/Value pairs that are expected to be accepted by the component. The data schema is represented in JSON because of its minimal syntax compared to XML, and to ease integration with the underlying data representation library, i.e. Conduit. A template of the schema is shown in listing 4.1. Each component should accompanied by a text file that contains the JSON representation of its data schema.

```
{"schema": {
  "in": {
    "input-tag-1": {
      "value": { ... }
      ,"key": { ... }
    },
    "input-tag-2": {
      "value": { ... }
      ,"key": { ... }
    }
    },"out": {
      "output-tag-1": {
        "value": { ... }
        ,"key": { ... }
      },
      "output-tag-2": {
        "value": { ... }
        ,"key": { ... }
      }
    }
  }
}
```

Listing 4.1: A template of the data schema

Conduit library allows setting and retrieving variables with specified type and fixed size. For example, when dealing with arrays, the schema should contain their actual lengths. But when describing the output schema, the lengths of the arrays are not known till run time. To handle the problem of declaring arrays size statically, we introduce variables with a $ prefix in the output schema. The programming API allows to substitute them with the actual value at runtime. For input schema, the length of the arrays can be specified by a $*$ to match any size, or it can be hard coded if other array lengths are not acceptable.

For the key part, the key is represented as a tuple. The tuple parts have names and types. We use the Conduit schema to describe the key in the schema file. Just primitive types are allowed in the key schema.

4.3 Programming API

The programming API is inspired by FlowVR and consists of the following functions:

- `void wait()` : Blocks until a Conduit node is received.

- `Node get(string tag)` : Reads a Conduit Node with the specified tag. The tag should be declared as input tag in the data schema. If it is not available returns `NULL`.

- `void put(string tag, Node output)` : Publishes a Conduit Node with the specified tag. The tag should be declared as output tag in the data schema.

- `bool available(string tag)` : Checks the readiness of a particular schema of the input. It is useful when a module might accept more than one type of schemas for inputs.

- `Node allocate_node(string tag, vector<pair<string, int>> sizes)` : Creates a Node structure and allocates its memory. It accepts the name of schema to create and a list of the values of the variables if exist. If no arguments are provided, an empty Node is created.

We try to keep the programming API a minimal as possible to facilitate using it by the users.

4.4 Configuration Script

When working with dataflow model, a description of the workflow is needed. A Python script is used to define the components and the links between them. Scripts are shown to be more powerful in this task than files, such as XML. Scripts are a complete programming language, thus make it easier to describe conditions and loops, which are very important when defining large and complex workflows.

In the configuration script, computational components are represented as `modules`. As in FlowVR, a configuration script is used to describe the workflow as a direct graph of connected modules. Modules are first defined and then linked together with optional predicate. A direct link between two modules means that the data will flow from the source to the destination. It is actually a direct link. Changing this file does not require any recompilation for the modules. This allows rapid testing for many workflow configurations to find the one that produces the highest performance for the current hardware architecture. For example, changing the number of parallel processes or on which nodes and cores the components are mapped could be done easily and without any recompilation.

The basic API functions to build the workflow are:

- `Module(identifier, cmdLine, hosts, cores)` : Creates a Module object with a unique identifier, the command line to run this module, and the hosts and cores where the process will run.

- `connect(src, dest, tag, predicate)` : Creates a link between two modules, going from the source to the destination and carrying the data with the tag of the source. It takes an optional predicate which could be applied on the Key of the data. The predicate parameters names should match the variables names in the key schema.

4.5 Development Status

The prototype is developed relying on FlowVR. For the data schema, a library is developed to deal with the data related issues. The data schema files of the components are parsed and a

schema matching is performed between each connected output and input schema. The library raises warning messages when any mismatch is found, for example the data needed by the analysis does not exist in the output schema of the simulation. The library implements the methods that instantiate the schema of a Key/Value pair according to the provided variables at runtime. When a program runs as a parallel program, the data produced by the processes could be merged into one schema. On the other hand, a message could be split to be distributed to multiple processes. Merging and splitting routines that handle one dimension arrays are implemented. Other methods related to data distribution are not implemented yet. Wrapper classes have been developed to translate the high-level API to the lower level FlowVR calls. The translation of the configuration script to a dataflow graph is not yet implemented.

4.6 Example

We go through an example to clarify the previously described concepts. In this example, we have a molecular dynamics simulation that produce atoms related data and an analysis that accepts as input this type data. We start first by defining the I/O data of the simulation and analysis. The simulation has two types of output: the simulation output represented by the tag `sim-data` data and the Checkpoint/Restart data represented by the tag `CR`. The analysis declares that it accepts two arrays of position and type of the atoms and produces the density and state for elections. The analysis also accept control information under the tag `control-info` that is used to send events to the analysis while it is running. The schema file of the simulation is presented in code-listing 4.2 while code-listing 4.3 presents the analysis schema file.

```
1  {"schema": {
2    "out": {
3        "sim-data": {
4          "value": {
5            "length": "uint32",
6            "atoms": {
7              "position": {"dtype":"float64", "length": "$atoms_length"},
8              "velocity": {"dtype":"float64", "length": "$atoms_length"},
9              "type": {"dtype":"uint16", "length": "$atoms_length"}
10           }
11         },
12         "key": {
13           "timestep":"uint64", "partition":"uint8"
14         }
15       },
16       "CR": {
17         "value": {
18           "raw": {"dtype":"uint8", "length": "$CRN"}
19         },
20         "key": {
21           "CR-ID":"uint64", "partition":"uint8"
22         }
23       }
24     }
25   }
26 }
```

Listing 4.2: The simulation data schema

```
{"schema": {
  "in": {
    "sim-data": {
        "atoms": {
          "position": {"dtype":"float64", "length": "*"},
          "type": {"dtype":"uint16", "length": "*"}
        }
      },
      "control-info": {
          "event": "uint8"
        }
    }
  ,"out": {
    "electrons-data": {
      "value": {
          "electrons": {
            "density": {"dtype":"float32", "length": "$electrons_length"},
            "state": {"dtype":"uint8", "length": "$electrons_length"}
          }
        }
      ,"key": {"timestep":"uint64", "range":"uint8"}
      }
    }
  }
}
```

Listing 4.3: The analysis data schema

We notice that the tags `sim-data` of the simulation and analysis are not exactly similar. The analysis tag represents a subset of the variables declared by the simulation. The extraction of the needed variables is done automatically by the framework.

The next step is to include the framework API calls into the code base of the simulation and analysis. Traditional I/O API calls should be replaced with the framework equivalent calls. Consequently, the framework is now responsible for the I/O channels.

Code-listing 4.4 presents the simulation that act on two arrays of floating point numbers and an array of unsigned short integers. Before proceeding to the next iteration, an empty Node is created. The key and value are filled with the appropriate values. Then, the Node is published to the framework.

```
  float64* atom_position = new float64[nb_atoms];
  float64* atom_velocity = new float64[nb_atoms];
  uint16* atom_type = new uint16[nb_atoms];

  for (int t=0; t<MaxSteps; t++) {

    simulate(atom_position, atom_velocity, atom_type);

    Node n = allocate_node();
    n["key"]["timestep"] = t;
    n["key"]["partition"] = rank;
    n["value"]["length"] = length;
    n["value"]["atoms/position"].set_float64_ptr(atom_position, nb_atoms);
    n["value"]["atoms/velocity"].set_float64_ptr(atom_velocity, nb_atoms);
    n["value"]["atoms/type"].as_uint16_ptr(atom_type, nb_atoms);

```

```
17      put("sim-data", n);
18
19      if (do_checkpoint) {
20          Node cr = allocate_node();
21          cr["key"]["CR-ID"] = get_checkpoint_id();
22          cr["key"]["partition"] = rank;
23          cr["value"]["raw"].as_uint8_ptr(get_checkpoint_data());
24          put("CR", cr);
25      }
26  }
```

Listing 4.4: The simulation

In the previous code-listing, the `set_*_ptr` method performs a deep copy of its argument variable. To avoid the data duplication and the extra copy operation, we create the Node with a specified tag `sim-data`. Now, the Node could provide a direct write pointer to its underlying allocated array. In this way, arrays can be filled directly as shown in the code-listing 4.5.

```
1   for (int t=0; t<MaxSteps; t++) {
2
3     vector<pair<string, int> > sizes;
4     sizes.push_back(pair<string, int>("atoms_length", nb_atoms));
5     Node n = allocate_node("sim-data", sizes);
6
7     float64* atom_position = n["value"]["atoms/position"].as_float64_ptr();
8     float64* atom_velocity = n["value"]["atoms/velocity"].as_float64_ptr();
9     uint16* atom_type = n["value"]["atoms/type"].as_uint16_ptr();
10
11    simulate(atom_position, atom_velocity);
12
13    put("sim-data", n);
14
15    if (do_checkpoint) {
16        vector<pair<string, int> > sizes;
17        sizes.push_back(pair<string, int>("CRN", get_checkpoint_data_size()
    ));
18        Node cr = allocate_node("CR", sizes);
19        cr["key"]["CR-ID"] = get_checkpoint_id();
20        cr["key"]["partition"] = rank;
21        get_checkpoint_data(cr["value"]["raw"].as_uint8_ptr());
22        put("CR", cr);
23    }
24  }
```

Listing 4.5: The simulation without data replication

On the analysis side, the module waits for incoming messages. When a message is received, the wait returns and the loop is executed. In this step, we do not know whether the received message is the actual analysis input or the control information. A check for the availability of a specific tag should be used. After getting the appropriate Node, its fields and structure can be accessed. This logic is shown in code-listing 4.6.

```
1 while (wait()) {
2   if (available("sim-data")) {
3     Node n = get("sim-data");
4     float64* atom_position = n["value"]["atoms/position"].as_float64_ptr();
5     uint16* atom_type = n["value"]["atoms/type"].as_uint16_ptr();
```

```
6     analyze(atom_position, atom_type);
7   }
8   if (available("control-info")) {
9     Node n = get("control-info");
10    uint8 event = n["value"]["event"]
11    handle(event);
12  }
13 }
```

Listing 4.6: The analysis

The last step is to assemble the simulation and the analysis in the same workflow. A Python script is used to describe the involved modules and the connections between them. Let assume that that the simulation and analysis are not a parallel programs and they run on different nodes. First, the simulation module is defined, the command line to run the module is provided, and the hosts are specified. Another module for the analysis is also defined. Then, the two modules are linked with the data tag `sim-data` and a predicate. The framework discards the messages that do not satisfy the supplied predicate. The arguments of the predicate are the Key components of the exchanged data. An example of a configuration file is presented in code-listing 4.7.

```
1 simulation = Module("simulation", cmdLine = "./simulation.cpp -n 20", \
2   hosts = ["cluster-node-1"])
3
4 analysis = Module("analysis", cmdLine = "./analysis.cpp", \
5   hosts = ["cluster-node-2"])
6
7 connect(src=simulation, dest=analysis, tag="sim-data", \
8   predicate = lambda timestep, partition: timestep%10==0 and partition<2)
```

Listing 4.7: Configuration file

The framework applies the predicate on the same node where the data is generated. Just the messages that do satisfy the supplied predicate will be transmitted to their destination.

4.7 Discussion

In the previous example, we show how the proposed models fit the in-situ context. Exposing a composite key to a higher level gives more control over the exchanged messages. Message filtering could be done by the framework as soon as possible. Describing the data structure of the data allows the framework to do fine-grained data extraction. When the needed data structure of the analysis is known, the framework can move just the required data. The API is kept minimal to minimize the modifications to the source code. The proposed prototype could be integrated with other dataflow framework and it is not related to FlowVR.

— 5 — Conclusion

In-situ analytics is not limited to process the data while it is in memory and bypass the disk. It brings radical changes to the design of scientific workflows in the HPC domain. In addition to performance gain, in-situ frameworks provide more flexibility to scientists and more control over their experiments. We discussed how using the dataflow model as a programming model with a data model that annotates the data with semantic information can achieve both performance and flexibility. We demonstrated that decorating the data in a dataflow model with a named key provides more flexibility and preserves the modularity of the independently developed components in a scientific workflow. Going one step further by describing the structure of the flowing data allows the framework to optimize data movement between the simulation and dependent analysis components.

The next step in this work is to finish the implementation of the developed prototype. First, the translation of the configuration file to a dataflow graph has to be implemented and the predicate should be executed implicitly by the framework. Second, Complicated data distribution patterns such as MxN distribution could be provided to the user as high-level programming constructs. Finally, an exhaustive testing and performance evaluation should be performed to measure the overhead compared to the low-level FlowVR with no data model.

With the proposed programming and data model, task placements strategies can be further explored. We aim to develop new resource allocation algorithms to compute a mapping of tasks on helper cores and staging nodes taking into account data movements with minimal hints from the programmer. For example, knowing the schema of the data can be used to infer the total number of bytes to represent this data. The volume of data requested by a component could be used to decide whether to run it on a helper core or on a staging node.

In the future, we would consider some non-functional requirements of the in-situ frameworks as fault tolerance. Fault tolerance is often ignored because of its added overhead. Simulations may run for weeks and months and failures become the norm not the exception. In addition to that, nowadays, there is a tendency to run scientific simulations in the Cloud where fault tolerance becomes a primary concern. We need a fault tolerant framework that can handle unexpected failures and scheduled maintenance periods.

Bibliography

[1] Apache Flink: an open source platform for distributed stream and batch data processing. https://flink.apache.org/index.html. 9

[2] Conduit: a scientific data exchange library for hpc simulations. http://software.llnl.gov/conduit/. 5

[3] Lustre: a scalable, high-performance file system. http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf. 5

[4] Visit users manual. https://wci.llnl.gov/simulation/computer-codes/visit. 6

[5] The visualization toolkit. http://www.vtk.org/. 6

[6] The netcdf users guide. http://www.unidata.ucar.edu/software/netcdf/netcdf-4/newdocs/netcdf.pdf, 2008. 5

[7] Hdf5 user's guide. http://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UsersGuide.PDF, 2011. 5

[8] Synergistic challenges in data-intensive science and exascale computing. http://science.energy.gov/~/media/40749FD92B58438594256267425C4AD1.ashx, 2013. 1

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 9

[10] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM. 6

[11] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. 01 2005. 6

[12] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013. 9

[13] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015. 9

[14] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *Proceedings of Europar 2004*, Pisa, Italia, August 2004. 7

[15] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 25–29, New York, NY, USA, 2015. ACM. 6

[16] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. 13

[17] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: Array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 66:1–66:11, New York, NY, USA, 2011. ACM. 10

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. 8

[19] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. ACM. 7

[20] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In *2012 IEEE International Conference on Cluster Computing*, pages 155–163, Sept 2012. 6

[21] Matthieu Dreher and Bruno 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, United States, May 2014. IEEE Computer Science Press. 8

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. 5

[23] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM. 6

[24] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002. 5

[25] Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. Mrap: A novel mapreduce-based framework to support hpc analytics applications with access patterns. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 107–118, New York, NY, USA, 2010. ACM. 10

[26] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm at twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. 9

[27] T. Tu, C.A. Rendleman, D.W. Borhani, R.O. Dror, J. Gullingsrud, M.O. Jensen, J.L. Klepeis, P. Maragakis, P. Miller, K.A. Stafford, and D.E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008. 10

[28] V. Vishwanath, M. Hereld, and M.E. Papka. Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 9–14, Oct 2011. 7

[29] Yi Wang, Gagan Agrawal, Tekin Bicer, and Wei Jiang. Smart: A mapreduce-like framework for in-situ scientific analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 51:1–51:12, New York, NY, USA, 2015. ACM. 10

[30] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. 9

[31] Brad Whitlock, Jean M. Favre, and Jeremy 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, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association. 6

[32] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. PhD thesis, EECS Department, University of California, Berkeley, Feb 2014. 9

[33] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. 9

[34] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Predata: preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010. 6

[35] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 78:1–78:12, New York, NY, USA, 2013. ACM. 7

[36] Fang Zheng, Hongbo Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, Tuan-Anh Nguyen, Jianting Cao, H. Abbasi, S. Klasky, N. Podhorszki, and Hongfeng Yu. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 320–331, May 2013. 7