In-Transit Molecular Dynamics Analysis with Apache Flink

Henrique C. Zanúz
Bruno Raffin
henrique.colao@gmail.com
bruno.raffin@inria.fr
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France

Omar A. Mures
Emilio J. Padrón
omar.alvarez@udc.gal
emilio.padron@udc.gal
Univ. Coruña, Facultade de Informática
15071 A Coruña, Spain

ABSTRACT

In this paper, an on-line parallel analytics framework is proposed to process and store *in transit* all the data being generated by a Molecular Dynamics (MD) simulation run using staging nodes in the same cluster executing the simulation. The implementation and deployment of such a parallel workflow with standard HPC tools, managing problems such as data partitioning and load balancing, can be a hard task for scientists. In this paper we propose to leverage Apache Flink, a scalable stream processing engine from the Big Data domain, in this HPC context. Flink enables to program analyses within a simple window based map/reduce model, while the runtime takes care of the deployment, load balancing and fault tolerance. We build a complete in transit analytics workflow, connecting an MD simulation to Apache Flink and to a distributed database, Apache HBase, to persist all the desired data. To demonstrate the expressivity of this programming model and its suitability for HPC scientific environments, two common analytics in the MD field have been implemented. We assessed the performance of this framework, concluding that it can handle simulations of sizes used in the literature while providing an effective and versatile tool for scientists to easily incorporate on-line parallel analytics in their current workflows.

1 INTRODUCTION

At the dawn of exascale computing, scientists are taking advantage more than ever of the evolving massively parallel computational resources available. Computational science simulations are increasingly generating more output data, due to the continuous improvements in the computational infrastructure and in the parallel numerical algorithms. Unfortunately, the I/O subsystem in HPC environments has not evolved at the same pace than processors and memories, so the storage and analysis of all that huge amount of data have started to become the real bottleneck in the scientist workflow.

Furthermore, there is an inherent need to simplify the creation of analysis code that can exploit all these heterogeneous resources. On the one hand, the need is to exceed the classic analytics pipeline based on a post-hoc approach to bypass the I/O bottleneck; and on the other hand, to get beyond a sequential analysis by leveraging the HPC infrastructure for enabling efficient analyses at large scale, but limiting the complexity of programming and deploying parallel algorithms at a relatively low level as required when relying on the classical HPC programming models (MPI/OpenMP, data partitioning and distribution…).

This paper explores an alternative approach based on Big Data frameworks. We use Apache Flink, a distributed streaming dataflow engine, to process *in transit* the data from the simulation. We leverage Flink high level stream processing programming model, and its runtime that takes care of the deployment, load balancing and fault tolerance. The analytics results are next written as soon as available in a scalable NoSQL database, using Apache HBase, a key-value database on top of HDFS. Both the results from the analytics and the raw data produced by the numerical simulation are stored in this database. Then, traditional post-hoc data processing based on batch analytics is also possible in our framework, as the output data from the simulation are stored after being analyzed in Flink. Besides, the same analysis kernels can be (re)used for the on-line in transit and the post-hoc data processing.

The numerical simulation used for the experiments is a Molecular Dynamics (MD) parallel mini-app, CoMD, that emulates the representative workloads of a real full scale MD simulation. At every given timestep, each CoMD process outputs the positions of all the atoms it simulates, giving what is commonly called an MD *trajectory*. Various analyses on these data are usually performed to extract meaningful knowledge.

The overall set-up consists in using multiple dedicated nodes of a cluster to run CoMD (simulation), Flink (analysis) and HBase (storage). Two common compute kernels for MD analytics were implemented to demonstrate the expressivity of Flink's programming paradigm: a position histogram of atoms and an atom neighbor computation based on a cutoff distance.

2 STATE OF THE ART

Traditional MD analysis tools associated with large MD simulation codes are mainly focused on post-hoc trajectory analysis. Gromacs [14] offers several trajectory analysis tools, for example. The included algorithms are mostly coded in C and only some of them are parallel. In terms of usability, each analysis is a different program. No framework for developing new analytics is provided, so whatever new analysis needed has to be coded from scratch. MDAnalysis [15] or VMD [13] are classical post-hoc trajectory analysis tools. They rely on MPI and/or OpenMP parallelization that are often complex to deploy at large scale and difficult to use efficiently by computational biologists. VMD has been extended to support in situ rendering [19].

Several in situ and in transit frameworks report experiments with MD simulations to demonstrate their scaling abilities. They all rely on HPC based approaches, mainly MPI+X, used in different ways [8–10, 22, 23]. To our knowledge these approaches mainly led to research prototypes and have not yet been adopted by scientists.

Computational biologists have considered the map/reduce model as an alternative to traditional HPC parallelization approaches to speed up the post-hoc MD trajectory analysis. Himach [20] was the first MD analysis framework to provide parallel execution capabilities inspired by Google's Map-Reduce. The authors initially considered Hadoop as a target, but due to its poor performance they developed a dedicated map/reduce framework based on MPI, Python and using VMD for the analysis kernels. Himach has a run-time responsible for assigning the tasks to the processors, coordinating the parallel I/O requests, storing and managing intermediate values (temporary key-value pairs) and orchestrating the data exchange between processes. Himach focuses on a temporal parallelization where each key-value is one simulation timestep with all atom positions. It shows a reasonable good performance and scalability, almost linearly until 32 MPI processes but degrading from that point due to I/O and communications.

On [17], the authors compare three general purpose task-parallel frameworks, Spark, Dask and RADICAL-Pilot, with respect to their ability to support post-hoc MD analytics on HPC environments. They also assess them in comparison to classical HPC MPI approaches. Their experiments show that Spark outperforms Dask when it comes to communication intensive tasks and iterative algorithms, due to the in-memory RDDs. Dask's low and high level APIs prove to be more versatile than Spark, although both have showed some limitations, requiring work-arounds to implement the analytics. Radical-Pilot proved to be more useful for coarse-grained task-level parallelism and when it is necessary to integrate other existing HPC analytics frameworks (such as MDAnalysis). Even though none of them outperformed their MPI counterpart, their easier programming paradigm and not so big performance gap still creates a strong case for using them for MD analytics.

If many tools have extended the map/reduce model for stream processing, very few works have attempted to combine this model with in situ analytics for large parallel simulations. The SMART [21] in situ framework proposes to rely on the map/reduce model to define in situ analysis. Implemented on top of MPI and OpenMP it outperforms Spark, but lacks features like support for in transit processing or fault tolerance. DataSpace [7] is not a map/reduce framework but adopts the key/value storage concept to automatically index data produced by parallel simulations and store them in the memory of in transit nodes. Analytics query this in-memory database to get the needed data. DataSpace relies on MPI.

Many stream processing frameworks designed for internet and IoT applications exist, Spark, Flink and Storm being the most visible ones. For sake of conciseness refer to surveys like [5] for comparative studies.

3 IN TRANSIT PARALLEL ANALYTICS FRAMEWORK FOR MD SIMULATIONS

The architecture of the proposed framework connects the CoMD simulation parallelized with MPI to Flink worker nodes using ZeroMQ. Flink executes the analysis scripts in parallel and in-memory as soon as a new timestep is received, then injecting results to the HBase distributed database, that takes care of storing the results using its local disks (see an example of an 8 nodes configuration in Figure 1). An additional node is used to run both the Flink job manager and the HBase master.

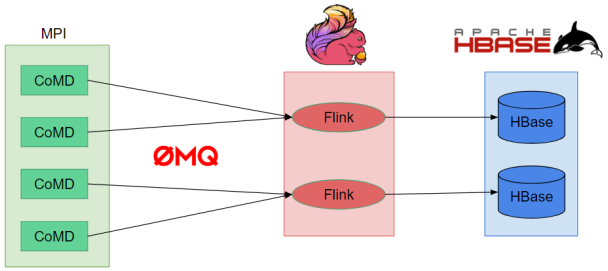


Figure 1: Diagram of the proposed framework deployed on 8 nodes: the MD simulation (CoMD) running in 4 nodes; 2 nodes executing the on-line analytics with Flink, receiving the data stream through ZeroMQ; and other 2 nodes in charge of storing data (raw data from the simulation and results from the analytics) in a distributed database (HBase).

3.1 Parallel Simulation

CoMD [11] is a *mini-app* for MD simulations, i.e. a simplified version of an actual MD application, sharing the same features and patterns in terms of operations, workload and work balance. We use the MPI version of CoMD, where each MPI updates the states of a fixed and different set of atoms at each timestep.

3.2 Distributed Stream Processing

Apache Flink [4] is an open source analytics engine supporting both stream and batch processing. Flink is a distributed, high-performing, highly availably and fault-tolerant framework. Flink is the result of the Stratosphere project [2], an open source platform for massively parallel Big Data analytics. It includes various level of optimizations like Parallelization Contracts (PACTs) [3]. To our knowledge Stratosphere/Flink has never been used for analyzing MD trajectories.

Flink stream processing model relies on the dataflow model as defined in [1]. It enables to continuously trigger the execution of map/reduce-like scripts [6] on windows of data regularly extracted from data streams. If the window operates on keyed data, i.e. data that are partitioned according to a key defined by the user, the operations are executed in parallel.

A Flink script is transformed into a dataflow graph, taking into account the parallelization opportunities supported by the different operators being used. At execution, one TaskManager in each Flink worker node performs the operations over the input data. Each TaskManager provides one task-slot per core in the node, usually called sub-tasks. At specific points of the execution, the streams might be split into stream partitions that can be distributed or re-assigned to other sub-tasks. In opposite to a MPI based programming, the user does not need to explicitly control the distribution of data and computation.

3.3 Key-value Store

Apache HBase is an open-source distributed database based on Google proprietary *BigTable*. It is a non-relational database that

runs on top of a distributed file system, commonly the Hadoop File System (HDFS). HBase can be seen as a distributed, sparse, persistent and multidimensional sorted map, indexed by a row key, a column key and a timestamp.

The worker nodes in HBase are called RegionServers. Each RegionServer contains an arbitrary number of regions. Each region is responsible for storing rows of one specific table, based on an interval of row-keys. The actual content of the rows are stored in HFiles on the underlying HDFS File System. An HBase master node coordinates the RegionServers and assigns their row-key intervals.

3.4 Putting the Pieces Together

Two connectors are needed in Flink to glue all the components together and get the whole system as depicted in Figure 1, with Flink receiving the data stream from CoMD, then processing the data and sending the raw data and the analysis results to the storage nodes. On the one hand, a source connector is used for the streaming data ingestion via ZeroMQ, and on the other one a sink connector is needed for dumping all data on the database.

The ZeroMQ [12] (or 0MQ) asynchronous message library is used to connect the parallel simulation with Flink. ZeroMQ is datatype agnostic and provides its own sockets, which are able to send and receive atomic messages over different transport layers, such as TCP, multi-cast, inter-process and in-process communications. In this work, we rely on the 'Push-Pull' pattern.

Currently, Flink lacks an official ZeroMQ source connector (it was available in times of Stratosphere, but was not migrated to Flink due to license issues), so we have developed our own connector [16]. The connector enables parallel connections: each CoMD process directly connects to a different thread on Flink nodes, these threads being distributed on the different Flink nodes and handling data to the local Flink sub-tasks. From Flink programming interface point of view, this is seen as a single stream that is actually distributed, effectively enabling parallel processing even before this data streams be keyed.

Flink includes an official HBase connector. So implementing the sink was more straightforward. The trajectory raw data are stored in one table where the rows are identified by a *"MPI Rank_Timestep"* key, so as to favor a partitioning of data based on their MPI rank. As most of the time queries need the data for a full timestep, this scheme reduces the risk for hotspots. The results from the analytics, way smaller than the raw trajectory, are stored in another table.

4 EXPERIMENTAL RESULTS

This whole environment runs on an homogeneous cluster with nodes having 2 processors Intel Xeon E5-2630 v3, with 8 Cores/CPU (i.e. a total of 16 cores per node), 128 GB, 2 x 558 GB HDD as storage and a 10 Gbps Ethernet network.

The first set of experiments used 8 simulation nodes (125 MPI CoMD processes, applying a $5 \times 5 \times 5$ 3D space partitioning), 1 Flink node with 16 task-slots and 1 HBase node. CoMD simulated a 32 million atoms model, which represents in number of atoms about half of the HIV virus molecular structure [18].

The amount of streamed and processed data is controlled through the frequency of the timesteps output by CoMD: from every timestep down to one every 20 iterations. These trajectories are fully stored in HBase a couple of minutes after CoMD finishes. To limit data buffering throughout the workflow, low heap memory limits were imposed on Flink and HBase: 6 GB for Flink TaskManagers and 30 GB for HBase RegionServers.

These experimented saving frequencies are high compared to the standard of MD simulations. Real MD simulations iterate at high speed (250 it/s or more is common). Only a fraction of these timesteps are usually saved (1/500 or even less) to limit the trajectory size and the impact on the simulation performance [8].

Three different on-line analysis scenarios were used for each experiment: a simple raw data injection, with no analysis at all; a low-demanding data processing that computes a position histogram of the atoms; and a high demanding analytics that computes the list of neighbors for each atom, i.e. at a distance lower than a given cutoff value. Histogram computation and neighbor identification are common analysis patterns for MD analytics [17]. The Flink pseudocodes for both algorithms are shown in Figure 2.

ZeroMQ ensures that messages stay in order and that none is lost. But we have to make sure that data processed together contain all and only the atoms from the same timestep. For that purpose we rely on Flink *time event windows*. The simulation timestep value contained in every message sent by CoMD is used as watermark. Flink splits the data streams according to tumbling windows of size one, i.e. once all messages for the next timestep have been received, ensuring the integrity of the data processed for each window.

For both algorithms we rely on Morton indexing (z-order curve), used to partition the space for the position histogram or as acceleration data structure when searching for neighbors.

For the position histogram, the data from the messages are first split (locally with a *flatmap*), associating with each atom the corresponding z-index. Because the data stream is parallel (one data source per MPI process) this operation is executed in parallel on Flink workers. Then a *keyby* shuffles the data so that all atoms belonging to the same z-cell are gathered together on the same Flink worker. This operation is combined with a reduction to count the number of atoms per cell. Again this shuffling/reduce operation is performed in parallel.

For the neighbor search, we use a z-indexing defined according to the cutoff radius. A *flatmap* associates with each atom a z-index for its z-cell but also for each z-cell it is neighbor of. This leads to duplicate the atom data 27 times for a 3D simulation like CoMD. Then the data are shuffled to gather all atoms with the same z-index together through the *keyby*, leading to gather all atoms belonging to the same neighborhood together. Then, a local operation (*apply*) computes the neighbor list for each atom.

The first experiment looked at analyzing the stability of the framework to cope with a state-of-the-art input pressure. Since MD simulations might run for extensive periods and generate huge trajectories, we want to be sure the system can reach and sustain a steady state, with non-increasing demand for buffering resources along time. Therefore, we tested our workflow with different frequencies of data outputs.

The maximum sustainable throughput is obtained at 71 MB/s (or 8.875 MB/s/node) when CoMD sends data every 11 iterations, for the direct data injection (Flink simply forwards the data to HBase) and the histogram computation. For both experiments CoMD produces

Alg. 1: Position histogram

```
CoMDStream.FlatMap:
  Emit foreach atom 1 tuple: <Timestep,BucketIndex,1>
.KeyBy(BucketIndex)
.Window by Timestep
.Sum(2)
```

Alg. 2: Neighbor identification

```
CoMDStream.FlatMap:
  Emit foreach atom+neighbor_cell 1 tuple: <Atom,CellIndex>
.KeyBy(CellIndex)
.Window by Timestep
.Apply(Select_Neighbor_Atoms)
```

Figure 2: Analysis kernels used in the experiments

63 GB of data eventually stored into HBase with an additional 500 KB for storing the histogram results.

Figure 3 shows some interesting data about the behavior of the system with the maximum experimented sustainable throughput (Figure 3a) and with the minimum unsustainable one (Figure 3b). In both subfigures, the graph on the left shows the cumulated number of messages processed throughout time at each point of the system. In this graph, three cumulative histograms are overlaid. The orange one counts the number of messages generated by CoMD; the blue one, the number of messages received by Flink and the green one, the number of messages handled by HBase. In the Figure 3a CoMD produces data every 11 iterations. The system is able to handle the data at the same rate it is produced by the simulation: CoMD generation, Flink processing and HBase ingestion histograms are almost perfectly superimposed. On the other hand, setting CoMD to produce one output every 10 iterations (Figure 3b) leads to a growing gap between the CoMD histogram and the almost matching Flink and HBase histograms. After about 200s Flink and HBase start to lag behind CoMD. After investigation, it appears that HBase is not able to cope with the data pressure. HBase imposes back pressure to Flink very quickly (both histograms almost match) that in turn imposes back pressure on ZeroMQ. The different buffers involved in this chain from HBase to ZeroMQ (ZeroMQ maintains buffers on the sender and receiver) get filled but are large enough to avoid impacting CoMD execution that proceeds at the same pace. After 750s CoMD finishes to produce messages while ZeroMQ, Flink and HBase keep on working for some extra time to empty the buffers. The right curves show that HBase accumulates data into buffers and flushes them at a regular pace.

The neighbor computing analysis leads to a significantly lower sustainable throughput of 1.10 MB/s. The bottleneck is clearly the overload of Flink as the data are replicated 27 times and need to be shuffled when keyed. We are exploring alternative algorithms that hopefully will achieve significantly better performance, addressing the issue of data duplication and optimizing the neighbor computing.

We also ran a weak scalability test, scaling both the workload and the number of processors, running the histogram computation on a 4 times bigger set-up, with 512 CoMD processes on 32 nodes and 64 cores for Flink and HBase (4 nodes each). The number of atoms in the simulation was also multiplied by 4, generating a trajectory of 256 GB. The maximum sustainable throughput achieved reached 100 MB/s, only 40% more than with a four time smaller configuration.

For situating these performance according to execution in a more traditional HPC context let consider the results of [8], running a Gromacs MD simulation on a supercomputer. They test different in situ processing scenarios with FlowVR, saving simulation outputs to disk or performing some analytics. The closest scenario where all data produced are saved to disk, the simulation produces about 35 MB/s of data when running on 256 cores, and 75 MB/s when running on 512 cores. This is below the sustainable bandwidth we achieved with our set-up (this comparison should obviously be considered with care as it is not a direct comparison of the very same simulation running on the same machine).

5 CONCLUSION AND DISCUSSION

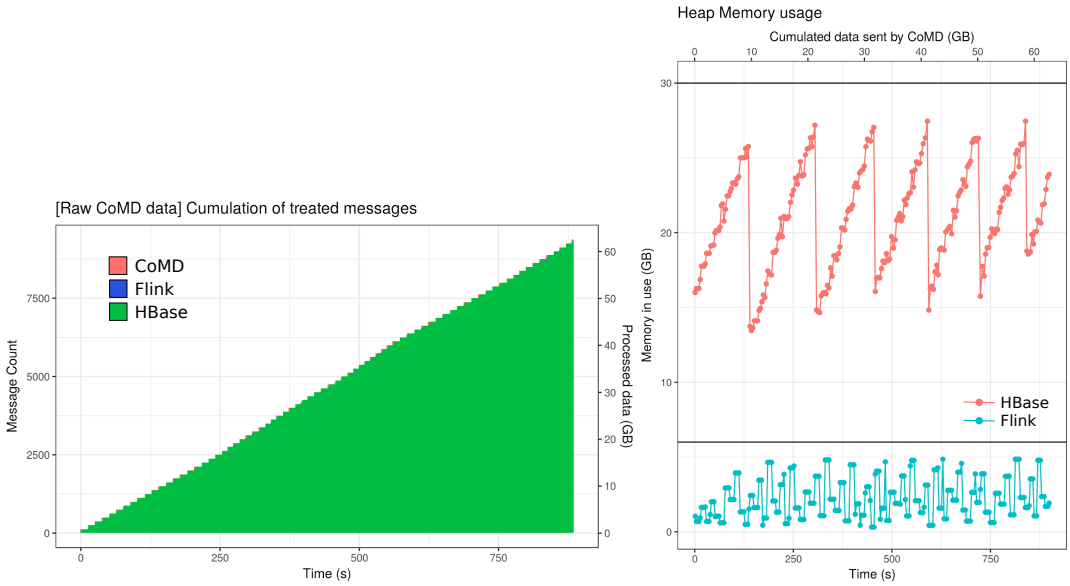
This paper investigates the HPC & BigData convergence for the on-line analysis of parallel simulation outputs. We presented early results using Flink to process in transit data produced by a parallel MD simulation and store the raw data from the simulation as well as the analytics results to HBase. We managed to achieve a sustainable throughput of 71 MB/s for 125 simulation processes (resp. 100 MB/s with 512 simulation processes) while saving 63 GB of raw results to HBase (resp. 256 GB) and computing a histogram. It also revealed performance issues when scaling or with the neighbor computations, but that should not be considered definitive as this early work leaves room for many possible improvements.

We are currently working on refining the analytics algorithms, using different Flink operators for increasing performance, moving to a different machine with InfiniBand network and SSD disks. We are also considering using a different storage. HBase was chosen as being standard in the Big Data domain, but other storage systems like Cassandra are known to have significantly better write performance.

This work also shows that Flink offers both a high level programming model easy to master, and an advanced runtime discharging the user of many time consuming aspects such as data communication, data partitioning, load balancing and fault tolerance. The programmer does not have to control data localization, i.e. to know explicitly the data each node keeps. He/She only has to control data aggregation by relying on data indexing with keys. Once the data are stored on HBase, the user can also query and process these data post-hoc without changing of programming environment. An HPC based approach would require more expertise and development time, often for the benefit of better performance. We discussed our approach with a few computational biologists that find it very appealing to trade some performance for saving on human expertise and time.

ACKNOWLEDGMENTS

Experiments ran on the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, several Universities and other organizations. This work is partially



Dispatched messages **Memory usage**

(a) Sustainable throughput: 8.875 MB/s/node (for a total of 71 MB/s)

Dispatched messages **Memory usage**

(b) Unsustainable throughput: 9.75 MB/s/node (for a total of 78 MB/s)

Figure 3: Analyzing the stability of the framework for two different throughputs with the Histogram analytics

supported by the Ministry of Economy and Competitiveness of Spain, Project TIN2016-75845-P (AEI/FEDER, UE), and from the Galician Government under the Consolidation Program of Competitive Research Units (Competitive Reference Groups, ED431C 2017/04).

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. FernÃąndez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*

23, 6 (2014), 939–964.

[3] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. 2011. MapReduce and PACT-Comparing Data Parallel Programming Models. In *Proc. of Datenbanksysteme fÃijr Business, Technologie und Web (BTW 2011)*. 25–44. http://stratosphere.eu/assets/papers/ComparingMapReduceAndPACTs_11.pdf

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/issue1.htm

[5] Subarna Chatterjee and Christine Morin. 2018. Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks. In *Proc. of 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 143–152. https://doi.org/10.1109/CCGRID.2018.00029

[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[7] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. DataSpaces: an Interaction and Coordination Framework forÂăCoupled Simulation Workflows. *Cluster Computing* 15, 2 (2012), 163–181. https://doi.org/10.1007/s10586-011-0162-y

[8] Matthieu Dreher and Bruno Raffin. 2014. A Flexible Framework for Asynchronous in Situ and in Transit Analytics for Scientific Simulations. In *Proc. of 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 277–286. https://doi.org/10.1109/CCGrid.2014.92

[9] M. Dreher, K. Sasikumar, S. Sankaranarayanan, and T. Peterka. 2017. Manala: A Flexible Flow Control Library for Asynchronous Task Communication. In *Proc. of 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Vol. 00. 509–519. https://doi.org/10.1109/CLUSTER.2017.31

[10] Laurent Colombet Estelle Dirand and Bruno Raffin. 2018. TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics. In *Proc. of SupercomputingAsia (SCAsia) 2018*. Singapore.

[11] ExMatEx: Exascale Co-Design Center for Materials in Extreme Environments. 2016. CoMD: Classical molecular dynamics proxy application. https://github.com/ECP-copa/CoMD Online; accessed 2018-08-06.

[12] Pieter Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.

[13] William Humphrey, Andrew Dalke, and Klaus Schulten. 1996. VMD: Visual Molecular Dynamics. *Journal of Molecular Graphics* 14, 1 (1996), 33–38. https://doi.org/10.1016/0263-7855(96)00018-5

[14] Erik Lindahl, Berk Hess, and David Spoel. 2001. GROMACS 3.0: A package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling* 7 (2001), 306–317. https://doi.org/10.1007/s008940100045

[15] Naveen Michaud-Agrawal, Elizabeth J Denning, Thomas Woolf, and Oliver Beckstein. 2011. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry* 32 (2011), 2319–2327. https://doi.org/10.1002/jcc.21787

[16] Omar A. Mures. 2016. ZeroMQ connector for Apache Flink. https://github.com/omaralvarez/flink-zeromq Online; accessed 2018-08-06.

[17] Ioannis Paraskevakos, André Luckow, George Chantzialexiou, Mahzad Khoshlessan, Oliver Beckstein, Geoffrey C. Fox, and Shantenu Jha. 2018. Task-parallel analysis of molecular dynamics trajectories. *CoRR* abs/1801.07630 (2018). https://arxiv.org/abs/1801.07630

[18] Juan R. Perilla and Klaus Schulten. 2017. Physical properties of the HIV-1 capsid from all-atom molecular dynamics simulations. *Nature Communications* 8 (19 07 2017), 15959 EP –. http://dx.doi.org/10.1038/ncomms15959

[19] John E Stone, Peter Messmer, Robert Sisneros, and Klaus Schulten. 2016. High performance molecular visualization: In-situ and parallel rendering with EGL. In *Proc. of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Vol. 2016. NIH Public Access, 1014–1023. https://doi.org/10.1109/IPDPSW.2016.127

[20] Tiankai 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. 2008. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *Proc. of 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2008.5214715

[21] Yi Wang, Gagan Agrawal, Tekin Bicer, and Wei Jiang. 2015. Smart: A MapReduce-like Framework for In-situ Scientific Analytics. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 51, 12 pages. https://doi.org/10.1145/2807591.2807650

[22] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. 2013. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 78, 12 pages. https://doi.org/10.1145/2503210.2503279

[23] F. Zheng, H. Zou, G. Eisnhauer, K. Schwan, M. Wolf, J. Dayal, T. A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. 2013. FlexIO: I/O middleware for Location-Flexible Scientific Data Analytics. In *Proc. of 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. 320–331. https://doi.org/10.1109/IPDPS.2013.46

A ARTIFACT DESCRIPTION APPENDIX: IN-TRANSIT MOLECULAR DYNAMICS ANALYSIS WITH APACHE FLINK

A.1 Abstract

This section describes the artifacts of this work submitted to ISAV'18. We describe the environment and the context of the experiments presented in Section 4. We explain how we compile and execute CoMD and how we deploy and execute the in transit analytics pipeline proposed in the paper.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithms:** Morton indexing (z-order curve), position histogram, neighbor identification based on a cutoff distance
- **Programs:** C binary, Java jars, Python scripts, CoMD, ZeroMQ, JZMQ, Flink, HBase
- **Compilation:** C, MPI, Java, Maven
- **Data set:** Generated by CoMD
- **Run-time environment:** Flink and HBase appropriately configured with access to JAVA_HOME. Java bindings for ZeroMQ installed (JZMQ).
- **Hardware:** Grid 5000's paravance cluster; each node: 2 x Intel Xeon E5-2630 v3, 8 cores/CPU, 128 GB, 2 x 558 GB HDD, 2 x 10 Gbps
- **Publicly available?:** Not yet

A.2.2 How software can be obtained (if available). The framework we are presenting in this paper is not publicly available yet, but will be open-sourced in a near future. The whole software stack will be included in a git repository, including an appliance to ease the setup and deployment. For more information, feel free to send a request at emilio.padron@udc.gal.

A.2.3 Hardware dependencies. The only hardware dependency is a cluster with nodes connected through an Ethernet network. Grid 5000* (G5K), a testbed for experiments on high performance computing, distributed systems, big-data and cloud, was used in this work. G5K infrastructure is composed of several clusters on several sites throughout France and Luxembourg. In all the experiments, all the nodes used were equal and part of one single cluster, in order to avoid introducing another parameters that can alter the results. The specifications of the nodes used for all the experiments are: 2 x Intel Xeon E5-2630 v3, 8 cores/CPU, 128 GB, 2 x 558 GB HDD, 2 x 10 Gbps.

A.2.4 Software dependencies. Our software stack is composed of:

- CoMD 1.1. We use a customized version to provide a stream data via ZeroMQ
- ZeroMQ library, version 4.1.4
  ZeroMQ java bindings (JZMQ), version 3.1.0
- Apache Flink, version 1.3.3
  Flink's Garbage Collector, version G1GC
- Apache HBase, version 1.2.6
- Apache Hadoop, version 2.7.6
- Oracle Java, version 1.8
- GNU Gcc, version 4.9

Our experiments were executed on a Debian 8 "Jessie" Operating System, inside a Kameleon appliance (more on this in A.7).

A.2.5 Datasets. The data stream generated by CoMD is used to feed the analytics pipeline.

A.3 Installation

- Get and compile (or use a binary package provided by a GNU/Linux distribution) ZeroMQ version 4.1.4 or above and the ZeroMQ java bindings (JZMQ), version 3.1.0 (the last version available).
- Install Oracle Java, version 1.8.
- Install Apache Hadoop, Apache Flink and Apache HBase, appropriately configured with access to JAVA_HOME.
- Get and compile a modified version of CoMD to provide the data stream via ZeroMQ. Compile it with MPI support (default option in the makefile under CoMD/src-mpi).
- Using Maven, create a Jar bundle with the last version of the analytics to be executed. Include the JZMQ classes in the Jar.

A.4 Experiment workflow

(1) Launch Flink & HBase

```
% ${FLINK_DIR}/bin/start-cluster.sh
% ${HBASE_DIR}/bin/start-hbase.sh
```

(2) Create two HBase tables with the following commands

```
% echo "create 'flink_ingestion', 'data', \
  {SPLITS => ['0','1','2','3','4','5','6','7','8','9']}" \
  | ${HBASE_DIR}/bin/hbase shell -n
% echo "create 'flink_analytics', 'data', \
  {SPLITS => ['0','1','2','3','4','5','6','7','8','9']}" \
  | ${HBASE\_DIR}/bin/hbase shell -n
```

(3) Execute the Jar package with the analytics in Flink using Flink CLI

```
% ${FLINK_DIR}/bin/flink run -d -p ${FLINK_TASKS} \
  -c org.inria.flink.${FLINK_EXP_KERNEL} \
  ${EXP_DIR}/FlinkCoMD_Experiments.jar 3000 2 9000 \
  ${TMP_DIR} ${COMD_PROCS}
```

Key parameters:
- `FLINK_TASKS`: Number of Flink sub-tasks per Flink node.
- `FLINK_EXP_KERNEL`: Flink analytics to execute.
- `COMD_PROCS`: Number of CoMD MPI processes.

(4) Execute the modified version of CoMD to generate the data stream to be analyzed

```
% mpirun -n ${COMD_PROCS} ${COMD_DIR}/bin/CoMD-mpi \
  --nSteps ${TIMESTEPS} --printRate ${OUTPUT_RATE} \
  --xproc ${XPROCS} --yproc ${YPROCS} --zproc ${ZPROCS} \
  --nx ${XCELLS} --ny ${YCELLS} --nz ${ZCELLS} --portNum 2 \
  --port 9000 --hostDir ${TMP_DIR}/ --hwm 3000
```

Key parameters:
- `XPROCS`, `YPROCS` & `ZPROCS`: Number of CoMD MPI processes for each dimension. Constraint: `XPROCS` × `YPROCS` × `YPROCS` = `COMD_PROCS`.
- `XCELLS`, `YCELLS` & `ZCELLS`: Number of CoMD unit cells per dimension.

(5) Collect experimental results (see A.5)

A.5 Evaluation and expected result

By performing the procedure above, the experiment is launched. Once CoMD finishes and all the data is persisted in HBase, it is possible to obtain (from the lines in HBase) the timestamps of creation, ingestion in Flink and storage in HBase for all the messages sent by CoMD. Computing a cumulation histogram counting the number of each type of event throughout the experiment execution results on the graphs we have shown in this paper.

We consider a certain throughput value to be stable when no gap between the three curves (messages created, ingested by Flink and stored) is observable in any of the repetitions made. Otherwise, it is classified as unstable.

A.6 Experiment customization

By keeping the same scripts for the experiments, it is still possible to customize them by tuning a wide set of parameters (as shown in A.4), such as: the number of atoms computed; the number of nodes dedicated for each role (simulation, analysis, storage); the number of iterations the simulation computes; the output interval…

These are some of the specific experiment invocations we have used in this paper:

- Position histogram analytics, 32 million atoms (63 GB), 71 MB/s (8.875 MB/s/node), i.e. one timestep is analyzed every 11 iterations.
  Setup:
  - 8 simulation nodes with 125 CoMD MPI processes
  - 1 Flink node with 16 task-slots
  - 1 HBase node

```
% ${FLINK_DIR}/bin/flink run -d -p 16 -c org.inria.flink.Histogram \
  ${EXP_DIR}/FlinkCoMD_Experiments.jar 3000 16 9000 ${TMP_DIR} 125

% mpirun -n 125 --host ${HOST\_LIST} ${COMD_DIR}/bin/CoMD-mpi \
  --nSteps 825 --printRate 11 --xproc 5 --yproc 5 --zproc 5 \
  --nx 200 --ny 200 --nz 200 --portNum 16 --port 9000 \
  --hostDir ${TMP_DIR}/ --hwm 3000
```

- Position histogram analytics in 4x the scale, 32×4 million atoms (252 GB), 100 MB/s (6.25 MB/s/node), i.e. one timestep is analyzed every 30 iterations.
  Setup:
  - 16 simulation nodes with 512 CoMD MPI processes
  - 4 Flink nodes with 64 task-slots
  - 4 HBase nodes

```
% ${FLINK_DIR}/bin/flink run -d -p 64 -c org.inria.flink.Histogram \
  ${EXP_DIR}/FlinkCoMD_Experiments.jar 3000 64 9000 ${TMP_DIR} 512

% mpirun -n 512 --host ${HOST\_LIST} ${COMD_DIR}/bin/CoMD-mpi \
  --nSteps 2250 --printRate 30 --xproc 8 --yproc 8 --zproc 8 \
  --nx 320 --ny 320 --nz 320 --portNum 64 --port 9000 \
  --hostDir ${TMP_DIR}/ --hwm 3000
```

A.7 Notes

To enable experiment reproducibility, we relied on different software tools that enabled us to control the full deployment and execution of experiments, from the OS up to the last lever software used. Due to the strong coupling that these tools has with G5K and its batch scheduler, OAR**, we maintain a specific git repository with all the deployment stuff, in addition to the main repository with the Flink analysis kernels. This repository will be open-sourced as well, along with the main one.

The use of these tools is not actually a strong requirement to deploy and run the experiments, but it is recommended mainly by two reasons:

- The first one is building the system image that will be used on the experiment nodes, allowing us to precisely control what is installed on them. This is done through the use of Kameleon Image Builder***, a tool that allows the creation on systems images based on scripts.
- The second reason is to completely automatize the experiments workflow. For that a python script was made. This script uses Execo****, a G5K python API that allows running remote commands over SSH and lower level commands, such as requesting and deploying nodes. In more details, this python script was used to set-up the configuration of the software stack, to configuring the clusters for Flink, HBase and CoMD, to launch the experiments and to recover and log the interesting data from the experiments in a result file. This script receives as input a YAML file which describes which experiments will be executed, their parameters and the amount of nodes dedicated for each role in this environment.

Regarding the execution of the experiments, this python script was executed on a separate node from the ones used on the set-up. This node runs the regular G5K system image, not the ones customized by us. After being launched, this script is the one in charge of reserving the master and slave nodes for the experiments. As explained on the paper, the slave nodes were used to run the CoMD MPI processes, the Flink's TaskManagers (workers) and HBase's RegionServers (workers). The master node is used to launch the MPI processes on the other nodes, run Flink's JobManager and HBase's Master process. Even though the master node was assigned with several tasks, those task were not compute-intensive. In addition, it has not showed any symptom of being overloaded during our experiments, staying with low memory and CPU usages.

A.8 References

* Grid 5000 (G5K): https://www.grid5000.fr
** OAR: https://github.com/oar-team/oar
*** Kameleon: https://github.com/oar-team/kameleon
**** Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. Using the EXECO toolbox to perform automatic and reproducible cloud experiments. In 1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013, Bristol, United Kingdom, December 2013. IEEE.