SIM-SITU: a framework for the faithful simulation of in situ processing

Valentin Honoré, Tu Mai Anh Do, Loïc Pottier, Rafael Ferreira da Silva, Ewa Deelman, Frédéric Suter

SIM-SITU: A Framework for the Faithful Simulation of in situ Processing

Valentin Honoré*, Tu Mai Anh Do†, Loïc Pottier†, Rafael Ferreira da Silva‡, Ewa Deelman†, Frédéric Suter‡,*
* IN2P3 Computing Center, CNRS, Villeurbanne, France
† USC Information Sciences Institute, Marina del Rey, CA, USA
‡ Oak Ridge National Laboratory, Oak Ridge, TN, USA

***Abstract*—The amount of data generated by numerical simulations in various scientific domains led to a fundamental redesign of how the analysis and visualization of simulation outputs are performed. The throughput and capacity of storage subsystems have not evolved as fast as the computing power in extreme-scale supercomputers, making the classical post-hoc approach highly inefficient. In situ processing has then emerged as a solution in which simulation and data analysis/visualization are intertwined for better performance and greater interactivity.**

Determining the best *allocation*, i.e., how many resources to allocate to simulation and analysis respectively, *mapping*, i.e., where and at which frequency to run the analysis/visualization, and *data transfer mode* is a complex task whose performance assessment is crucial to the efficient execution of *in situ* processing. However, such a performance evaluation of different strategies usually relies either on directly running them on the targeted execution environments, which can rapidly become extremely time- and resource-consuming, or on resorting to simplified models of the components of an *in situ* application, which can lack of realism. In both cases, the validity of the performance evaluation is limited.

In this paper, we present SIM-SITU, a framework for the faithful performance evaluation of *in situ* processing strategies. We designed SIM-SITU to reflect the typical features of *in situ* processing systems. Thanks to its modular design, SIM-SITU has the necessary flexibility to easily and faithfully evaluate the behavior and performance of various allocation, mapping, and data transfer strategies. We illustrate the capabilities of SIM-SITU on a Molecular Dynamics use case. We study the impact of different strategies on performance and show how users can leverage SIM-SITU to determine interesting tradeoffs when adding analysis/visualization components to their application.

I. INTRODUCTION

The tremendous volumes of data generated by numerical simulations in various scientific domains such as nuclear engineering, climate modeling, biology, or astrophysics, led to a fundamental redesign of how the analysis and visualization of simulation outputs are performed. Due to the growing discrepancy between storage subsystems performance and computing power in extreme-scale supercomputers, moving large volumes of data from computational resources to disks may have a dramatic impact on performance [1].

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

This makes the classical post-hoc analysis/visualization, i.e., once all simulation outputs are stored, highly inefficient. To overcome these issues, the *in situ* processing approach came as solution that intertwines simulation and analysis/visualization to process data as it is generated. Then, the final outcome of the whole execution will be much smaller to store. A beneficial side effect if *in situ* processing has been to provide scientists with better insights about the evolution of the simulation as it runs, and thus better command-and-control options. Over the last decade, many software frameworks have been developed for efficient *in situ* processing, covering a broader scope than the initial and literal meaning of the *in situ* term [2].

A common challenge to these tools is to decide what is the best *allocation*, i.e., how many resources to allocate to the simulation and analysis/visualization components respectively, *mapping* — where and at which frequency run the data analysis/visualization, and *data transfer mode*, i.e., how to exchange data between the components. These are complex tasks whose performance assessment is crucial to the efficient execution of *in situ* processing. However, such a performance evaluation of different strategies usually relies either on directly running them on the targeted execution environments or on resorting to simplified models of the components of *in situ* applications. The former requires to tune the framework with regard to the hardware and software available on the target machine and thus can rapidly become extremely time- and resource-consuming, while the latter can lack of realism. In both cases, the validity of the performance evaluation is usually limited to a narrow set of configurations.

In this paper, we present **SIM-SITU**, a framework for the faithful performance evaluation of *in situ* processing systems. SIM-SITU builds on the popular SimGrid toolkit [3] and benefits of several key features of this versatile framework. SimGrid enables the simulation of large-scale distributed applications in a way that is accurate (via validated performance models), scalable (ability to run large scale instances on a single computer with low compute, memory, and energy footprints), and expressive (ability to simulate arbitrary platform, application, and execution scenarios). We designed SIM-SITU to reflect the typical structure of *in situ* applications with three distinct modules that respectively, (i) simulate the unmodified simulation component of the application; (ii) mimic the behavior of an underlying Data Transport Layer (DTL); and (iii) abstract the analysis/visualization component.

Thanks to this modular design, SIM-SITU has the necessary flexibility to easily and faithfully evaluate the behavior and performance of various combinations of *in situ* processing system features [2]. We illustrate the capabilities of SIM-SITU and study the impact of different allocation and mapping strategies on performance and show how users can leverage SIM-SITU to determine interesting tradeoffs when adding analysis/visualization components to their application.

The remaining of this paper is organized as follows. Section II presents the related work. In Section III, we describe the architecture of SIM-SITU and detail its different features and advantages. In Section IV, we use SIM-SITU to evaluate different strategies for *in situ* processing with a Molecular Dynamics (MD) application. Finally, Section V summarizes our contributions and presents some future research directions.

II. RELATED WORK

The performance evaluation of *in situ* processing, and in particular that of allocation and mapping strategies for both the simulation and data analysis/visualization components, is a complex and multi-parametric problem. Different approaches have been proposed in the literature to ascertain the performance gains brought by *on-node* or *off-node in situ* processing of data produced by a numerical simulation and determine the best deployment of the *in situ* application on a given target platform. We distinguish these approaches depending on whether they rely on actual experiments [4]–[8] or resort to simulation [9]–[11] to evaluate the performance of *in situ* processing. The former is intrinsically time- and resource-consuming while the latter may suffer from simplification biases when abstract versions of the components are developed.

To limit the cost of the experiments required to conduct performance evaluations, some works focused on the Data Transport Layer (DTL) that connects the simulation and the analysis/visualization components. In such cases, series of experiments are conducted either using the real application [4] or by leveraging data access traces to mimic the application behavior [7], [8]. Another approach consisted in reducing the experimental configuration space to a selected set of promising configurations. For instance, Malakar *et al.* tackled the allocation and mapping of *in situ* processing as an optimization problem expressed as a Mixed-Integer Linear Program [5], [6]. Solving this optimization problem results on a set of feasible *in situ* analyses whose performance has been assessed through experimentation in an actual platform. Lorhmann *et al.* leveraged surrogate models (i.e., proxy-applications) of expensive numerical simulation codes [9] to abstract application models. While this approach substantially reduces the cost of the experiments, it still captures the most important features of the considered application.

Only a few recent attempts has leveraged simulation to enable the exploration of the *in situ* parameter space. Aupy *et al.* designed a numerical simulator [10] that measures evaluation metrics for scheduling decisions by solving optimization problems on resource allocation and partitioning for an *in situ* analysis set. The simulator used a predetermined set of parameters to study the impact of the *in situ* analyses that are scheduled on the performance of the entire *in situ* execution. In a previous work, we created a synthetic MD application based on the extrapolation of benchmarking performance of realistic MD engines [11]. This synthetic MD application replaced computational kernels by delays, hence did not perform any heavy computing operations.

To the best of our knowledge, this is the first work that proposes a faithful and scalable simulation framework that models the behavior of various combinations of features of *in situ* processing systems.

III. SIM-SITU ARCHITECTURE

The term "*in situ*" initially described a way for a *data analysis/visualization* component to consume and apply different routines on the scientific data periodically produced by a numerical *simulation* component directly where it is generated without moving it. In other words, the *simulation* and *analysis/visualization* codes run on the same computing resources and share a common memory space. However, the meaning of the *in situ* term has then evolved to describe the more generic concept of "processing data as it generated" and cover a broader range of possible execution scenarios. To address the confusion caused by this semantic evolution, a group of over fifty experts proposed a terminology to describe *in situ* processing systems [2]. They proposed six axes, and terms defining them, to better distinguish currently available *in situ* processing systems. Before detailing the architecture and design choices of the SIM-SITU framework, we briefly review these axes and their associated terms.

The **integration type** axis defines how the analysis and visualization routines are integrated into the simulation code. We focus on the **Application-aware** integration type, in which the simulation code explicitly calls a **multi-purpose API** to interact with the analysis/visualization component.

The data **access** axis by the analysis/visualization routines can either be **direct**, through **deep or shallow copy** of the data in a shared memory space, or **indirect**, when data is exchanged over the network or through a file system.

The **proximity** axis refers to deciding whether the analysis/visualization has be performed **on-node**, i.e., within a shared memory space and without data movement, **off-node**, e.g., dedicating a set of nodes on the same machine to analysis/visualization which implies data transfers over the network, or using **distinct computing resources**.

The **division of execution** axis specifies how compute resources are shared between the simulation and analysis/visualization components. A **space division** means that disjoint sets of resources are used simultaneously to generate and process data, while with a **time division** the same resources alternate between the generation and processing activities.

The **output type** axis describes which operations can be performed on the simulation data before it is processed by the analysis/visualization component. Three categories have been identified depending on whether the data size if reduced (**Subset**), increased (**Derived**), or remains the same (**Transform**).

The last axis is about **operation controls** and describes whether some **human-in-the-loop** interactions are needed before the simulation can resume or the analysis/visualization component runs in a **automatic** way. We only consider the latter in the design of SIM-SITU, but with the capacity to be **adaptive** and change the complexity of the analysis process as the simulation runs, if needed.

Figure 1 shows what could be the architecture of a generic *in situ* processing system according to this terminology. The objective of the proposed SIM-SITU framework is to enable the study of various combinations the available options for allocation, mapping, and data transfer. In the following we distinguish the following main components:

- The numerical *simulation* component that performs domain-specific computations, generally in an iterative process. This component periodically, i.e., after a predefined number of iterations, produces some scientific data;
- One or multiple *analysis/visualization* components that consume the data generated by the numerical simulation and may send back new data to that component;
- A *Data Transport Layer* (DTL), whose complexity depends on the degree of coupling between the two former components and the resources they are allocated to, is responsible for efficient data movements.

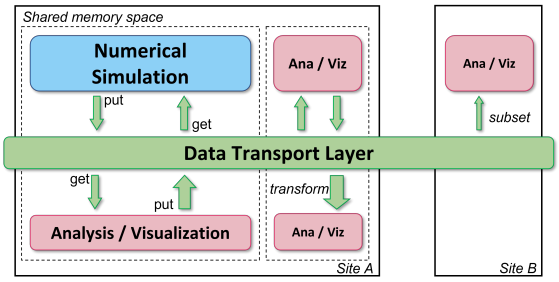

Fig. 1. Architecture of a generic *in-situ* processing system.

To faithfully simulate such a generic *in situ* processing system, SIM-SITU builds on several features of SimGrid [3], which is an open-source versatile framework for developing simulators of distributed applications executed on heterogeneous distributed platforms. One of the key strengths of SimGrid is to not trade accuracy for scalability. Its fast performance models have been theoretically and experimentally evaluated and validated [12] and make it possible to run large-scale instances on a single machine.

Numerical Simulation Component. To maximize the realism of the study of a parallel application, an appealing approach is to directly consider its unmodified code. This ensures that it does not only capture the computations executed on the different processes but also the exact communication pattern of the application. This approach is at the origin of SMPI [13] that comes with the SimGrid distribution.

SMPI allows to compile and run unmodified MPI programs written in C, C++, or FORTRAN. For instance, assuming the program is written in C, one simply compiles it with `smpicc`, instead of `mpicc`, and executes it with `smpirun`, instead of `mpirun`. This causes the code of the MPI application to be executed as is, but the MPI ranks actually execute as threads in a single process, and thus share the same address space. Each time an MPI function is called, control is handed off to SMPI where network operations are replaced by simulated delays. These delays are computed using the performance models at the core of SimGrid [14]. Each block of code in between two MPI calls is benchmarked on the machine where SMPI runs. This is possible because, with SMPI, MPI ranks execute as threads in mutual exclusion. These benchmarked execution times can then be scaled and simulated as compute delays that correspond to the compute speeds of the nodes in the simulated platform. In this way, SMPI simulates both communication and computation operations as computed delays.

The only difference with a classical MPI execution is that `smpirun` takes one extra command-line argument, `-platform`, which allows the user to describe the hardware platform on which the execution of the MPI program is to be simulated. This platform description specifies a network topology between compute nodes, where network links have specified latencies and bandwidths, and compute nodes have specified compute speeds and numbers of cores.

As part of its integration testing effort[1] SMPI simulates the execution of multiple proxy- and full-scale applications and HPC runtimes. It is thus a natural candidate in the design of SIM-SITU to handle the numerical simulation component in the context of *in situ* processing.

Data Transport Layer. Among the fifteen *in situ* processing systems characterized in [2], eleven follow the application-aware approach, i.e., rely on an API for the integration of analysis/visualization with a numerical simulation. Moreover, five of these systems focus on data movements between components and implement a data transport layer [15]–[19]. As this is considered as the most flexible, extensible, and efficient way to implement *in situ* processing, we decided to simulate such a DTL in SIM-SITU.

This implementation exposes a simple API to ease the integration of the DTL into application code. Following the Publish/Subscribe messaging model, the simulation component can *put* data into the DTL in an asynchronous way, and proceed with its computation. The analysis/visualization component(s) can then *get* this data and apply their own code to it. If the simulation component needs to retrieve the results of an analysis to continue, the same put/get mechanism can be used in the other direction. Such a feedback from the analysis/visualization component through the DTL can also be used to easily implement a time division of execution.

The DTL implementation of SIM-SITU also proposes to declare different *channels* to exchange different types of data or connect different components. For instance, two different channels can be used to implement a two-way exchange between the simulation and a data analysis component: one for the simulation data, the other for the analysis results.

[1] https://framagit.org/simgrid/SMPI-proxy-apps

Another advantage of such a simulated DTL is that it allows users to compare the performance of different data access modes, i.e., shallow or deep memory copy, transfer over the network, or using files, without having to modify the code to interact with the DTL. This becomes a configuration parameter of the different channels of the DTL.

To simulate the behavior of a shallow memory copy while respecting the flow dependencies between the producers and consumers, the DTL of SIM-SITU includes an implementation of the message queue that does not induce any advance of the simulated clock. This mode can also be used to artificially remove all the overhead caused by data movements and thus focus on the impact of different allocation and mapping schemes only on the compute part of *in situ* processing.

To simulate both deep memory copy and data transfer over the network, SIM-SITU leverages the *mailbox* concept used by SimGrid to implement inter-process communications. It acts as a rendez-vous point between a producer and a consumer processes. When both meet on that rendez-vous point, the actual communication starts. SimGrid mailboxes use a queue internally to store unmatched communications, i.e., when one side is waiting for the other, which ensures the respect of flow dependencies. An interesting feature of this mailbox concept is that the mapping of the producer and consumer processes determines the data access mode. If both are mapped on the same compute node, and thus sharing a memory space, the deep memory copy of data between *in situ* components can be simulated as a "on-node" communication through the node loopback. Conversely, if producer and consumer are on different nodes, the simulated communication will go over the interconnection network. This process allocation is given to SIM-SITU in a configuration file, thus allowing to change the data access mode without any code modification.

Analysis/Visualization Component(s). The data analysis and visualization routines that can be performed in conjunction with a numerical simulation are usually specific to a given scientific study or even to a given run of the application. Moreover, the complexity of these versatile components can vary greatly depending on what knowledge scientists want to extract from the simulation data. It can range from a simple computation of a variable derivative to help steer the simulation to a more complex parallel computation, or a full visualization of the current state of the simulation.

To enable the study of such diverse behaviors, we decided to abstract them in SIM-SITU using one or several SimGrid *actors*. A SimGrid actor corresponds to a simulated process that can consume some simulated *resources* (e.g., CPU time, network bandwidth, or storage space) by performing some simulated *activities* (e.g., executing a computation, communicating with another actor, or doing some I/O).

Isolating and abstracting the data analysis/visualization components within actors out of the MPI world offers a great flexibility to SIM-SITU. For instance, changing the proximity of the analysis/visualization from on-node to off-node is trivial with SIM-SITU. Indeed, a simulated actor can be started on any node of the simulated computing infrastructure, that execution location being specified at the creation of the actor, or predefined in a configuration file describing an initial deployment of the different actors. Moreover, it is possible to spawn new actors, stop existing ones, or even migrate them from one node to another while SimGrid is running. SIM-SITU could then be used to study complex scenarios where the analysis load evolves along time.

Users can act on many parameters when designing and executing their *in situ* experiments. Depending on what they want to observe or steer during the execution of the simulation component, the compute cost and complexity of the analysis/visualization routines, the volume of data to transfer from simulation to analysis, and the frequency of the analysis can change from one run to another. Combining these variable parameters leads to interesting questions such as "Is it more efficient to run frequent but light analyses or scarce but heavy ones?". Moreover, opting for a given configuration will have a direct impact on performance and may benefit of a particular allocation and mapping scheme. Abstracting the analysis/visualization as a combination of simulated activities executed by one or several actors, in a independent or coordinated way, allows SIM-SITU to easily reflect all the versatility of this important component of *in situ* processing.

The behavior of a generic and abstract analysis/visualization component in SIM-SITU can then be summarized as starting n actors ($n \geq 1$) that: (i) get some data from the DTL; (ii) simulate some computation; and (iii) may produce results which can be either put into the DTL or send to another actor. We can derive multiple scenarios from this simple structure.

SIM-SITU users can easily create sequential or parallel components, whose *mapping* on computing nodes is described in a file similar to the MPI hostfile. The complexity and computation and communication patterns of these actors is then expressed through the S4U API of SimGrid. In its first version, SIM-SITU comes with two analysis actor implementations. In the former, each spawned actor just performs a single *Execution* activity by calling the following method: `s4u::this_actor::execute(workload)`. This corresponds to simulating the execution of the amount of work given as parameter at the compute speed of the node the actor is mapped on. In the latter, n actors are involved in a coordinated analysis. Each actor splits the execution of its own workload in s steps, and at each step exchanges some data with the other actors through an All-to-All collective communication operation. Such a parallel analysis component allows us to investigate the perturbation of the simulation component caused by the communications of analysis/visualization component when network resources are shared.

The workload an actor has to compute is determined by the product of the values of three parameters: (i) the number of "elements" coming from the simulation components it gets from the DTL; (ii) an analysis cost per element; and (iii) and a compute scaling factor. This last parameter allows us to artificially increase the analysis cost to study what-if scenarios. Similarly, for the data movements between components, users

can specify an element size and a data transfer scaling factor to artificially change the transferred data size (i.e., simulate subset, transform, or derived output types). This workload parameter can also derive from a performance model [20].

Finally, it is possible to compose several actor types to form a more complex analysis workflow. For instance, to simulate the aggregation of some quantities individually produced by the analysis actors before sending back some metrics of interest to the simulation component, we implemented the *in situ* workflow depicted in Figure 2.

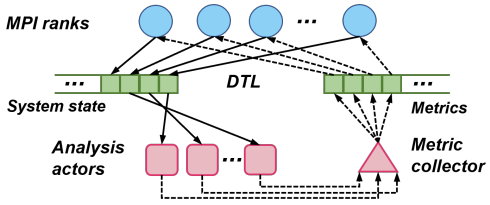

Fig. 2. Implementation of an *in situ* workflow with several actor types.

First, the MPI ranks running the simulation component put the data they have generated into the DTL in a fire-and-forget mode and immediately proceed with the next iteration of the main simulation loop. In this scenario, they will later block to retrieve the analysis results, after several iterations, i.e., before having to start a new analysis.

Each *analysis* actor runs an infinite loop in which it waits for data to analyze to be available in the DTL. When it is the case, the actor simulates the execution of the corresponding workload as described above. Then each actor asynchronously sends dummy results to the *metric collector* actor, and waits again for new data to be available in the DTL.

The *metric collector* actor simply waits for having received as many individual analysis results, or metrics, to accumulate as there are ranks executing the simulation component. As the number of analysis actors can be smaller than the number of MPI ranks, an actor can send more than one metric to the collector. Once all the metric values for a given analysis phase have been collected, this metric collector puts as many copies of the final analysis results into the DTL, so that each MPI rank can retrieve one set of metrics and pursue its execution of the simulation component.

Here, the DTL is organized around two distinct channels. The former stores the current system states sent by each of the MPI ranks to the analysis actors (plain arrows) while the latter stores the metrics computed by the analysis actors and aggregated by the metric collector that are sent back to the MPI ranks (dashed arrows). The communications between the analysis and metric collector actors rely on a standard SimGrid mailbox (dotted arrows), and are thus outside the MPI world.

This communication scheme allows us to decouple the needed synchronization between the simulation and the analysis/visualization components from that needed for the simulation itself. It also improves the flexibility of SIM-SITU by allowing users to start any number of analysis actors without any code modification.

IV. Evaluation of In-Situ Processing Scenarios with Sim-Situ

To illustrate the capacities and flexibility of SIM-SITU, we consider the application of *in situ* processing to Molecular Dynamics (MD) simulations. Studying the evolution of molecular systems at the atomic scale is one of the most prominent types of simulations currently running on extreme-scale systems. A reproducibility artifact for this paper is available online [21].

A. Experimental Setup

Application. More precisely, we relied for our experiments on the ExaMiniMD proxy-application [22], [23] which is part of the Exascale Computing Project Proxy App Suite v4.0 [24]. ExaMiniMD captures both the computation and communication schemes that are implemented in the classical MD code LAMMPS [25]. As other proxy-applications, ExaMiniMD shows a good balance between having a compact and manageable code and representing the main performance concerns of MD applications. ExaMiniMD belongs to the family of the all-atom MD simulations. It computes floating-point intensive pairwise atom-atom unbounded interactions over a certain period of time. The simulated system corresponds to a set of particles distributed in a 3D volume. The main loop computes the trajectories of the particles according to a Verlet time integration method and the short-range forces between particles as a Lennard-Jones potential. The parallelization of this MD problem follows a typical domain decomposition approach. Each MPI rank manages a sub-volume and a halo to exchange with its neighbors periodically. All the data exchanges in the simulation component rely on point-to-point MPI communications with asynchronous receives.

Experimental Platform. Our experiments were conducted on the *dahu* cluster of the Grid'5000 experimental testbed. This cluster consists of 32 nodes that comprise two Intel Xeon Gold 6130 CPUs with 16 cores each and 192 GiB of memory. These nodes are interconnected through a 10 Gb/s Ethernet network. We leverage an existing thorough calibration of the SMPI network model for this same cluster [26] to ensure our simulated results are accurate. This calibration runs a series of tests on a limited number of nodes to assess the performance of point-to-point communications and saturate a switch. It can then be used to extrapolate the size of a given cluster beyond its actual number of nodes.

We used the git version of ExaMiniMD, compiled with g++ v8.3.0 and linked to Kokkos v3.3.01 and OpenMPI v3.1.3. We used the `Serial` Kokkos device with one rank per core. To simulate ExaMiniMD, we relied on SimGrid v3.29. SIM-SITU is implemented as a shared library built against SimGrid and linked to ExaMiniMD at compile time.

B. Performance of the Simulation Component

In this section, we analyze the simulated execution of the unmodified code of the target application. This only requires minimal modification to the `Makefile` file to indicate that the compiler to use is `smpicxx`.

Additional, yet optional, modifications of the application code can be made to drastically reduce the execution time of the simulated version. SMPI offers to replace time-consuming computational parts of the simulated application by delays which are estimated by sampling the execution time of a given kernel or loop body either for a predefined number of times or until the standard deviation of the samples is under a given threshold. All the subsequent calls are then replaced by the average execution time of these samples. This sampling can be done either at a *local* scale, i.e., each MPI rank determines its own delay from the samples it executed, or at a *global* scale, i.e., the delay is determined from samples executed by all the MPI ranks. SMPI also provides a mechanism to reduce the application memory footprint by sharing memory allocation among simulated MPI ranks.

We used this feature on the most time-consuming kernel of ExaMiniMD that represents 69% of the execution time of the application [27]. This corresponds to a 1-line modification of the code to call the sampling macro with its parameters. We chose to run 150 samples with a standard deviation threshold of 0.002 in our experiments.

Cost. First, we compare the cost, in core × hours, of running a representative instance of ExaMiniMD with SIM-SITU to that of an actual run. Figure 3 shows the results of this comparison for different numbers of MPI ranks. We map an MPI rank per core and use a single Kokkos thread per MPI rank. Error bars show the standard deviation over five runs.

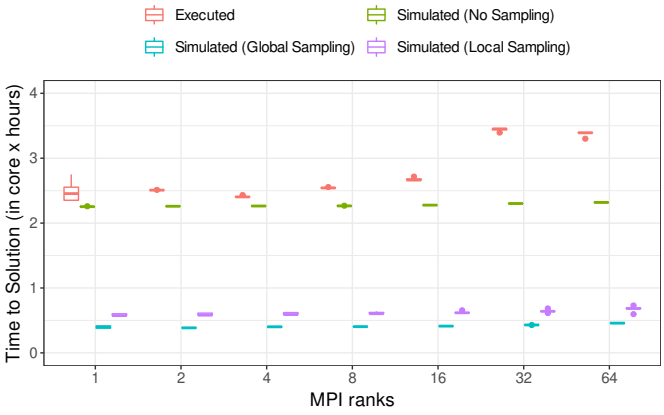

Fig. 3. Time to run or simulate (with or without kernel sampling) a representative instance of ExaMiniMD. Each rank runs a single Kokkos thread and is mapped on a different core.

We can see that the number of core × hours needed to solve this problem instance remains stable as the number of MPI ranks increases, i.e., the actual execution completes faster with higher rank counts. SIM-SITU runs on a single core and takes the same time to complete whatever the number of MPI ranks used. This time is commensurate to that of the actual execution, but uses much less computing resources. Activating the kernel sampling, either local or global, in SIM-SITU reduces the time to solution by a factor of 5, thus results can be obtained in about 25-30 minutes instead of 2.5 hours.

Execution vs. Simulated time. Then, we compare the time returned by SIM-SITU to that of an actual run of ExaMiniMD and assess the impact of kernel sampling. Figure 4 shows that SIM-SITU correctly reflects the performance evolution trend of the simulated application with less variability and a reasonable error. Activating the kernel sampling, be it local or global, slightly degrades the accuracy of SIM-SITU. However, this degradation remains stable as the number of ranks increases and can thus easily be taken into account when assessing the performance of a given *in situ* processing strategy.

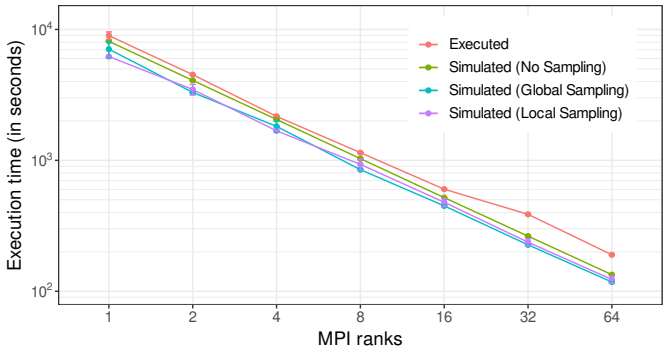

Fig. 4. Execution vs. simulated time of ExaMiniMD (with or without kernel sampling) when varying the number of MPI ranks.

On larger core counts, scaling up to the full size of the target platform (i.e., 32 nodes and 1,024 cores) and for a larger problem instance, Figure 5 shows that the accuracy of the local sampling version drops from 512 cores while the versions with global sampling and without sampling remain accurate.

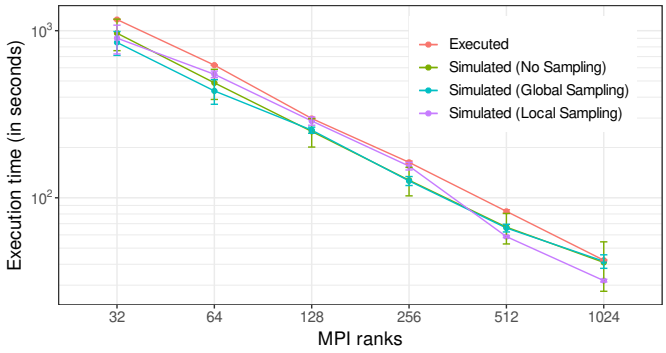

Fig. 5. Execution vs. simulated time of a larger instance of ExaMiniMD (with or without kernel sampling) when varying the number of MPI ranks.

C. Assessing the Performance of In Situ Processing Scenarios

In this section, we describe some of the many *in situ* processing scenarios that can be simulated with the proposed SIM-SITU framework. Each of the following subsections corresponds to a design choice faced by users who want to couple a numerical simulation to analysis/visualization routines. We show how the flexibility of SIM-SITU easily allows users to investigate different performance tradeoffs by simply changing some configuration parameters.

On-node processing: Space vs. Time Division of Execution. One of the first question to answer when coupling an analysis/visualization component to a numerical simulation is to determine whether the analysis can be done inline with the simulation, i.e., keeping data in place in memory and pausing the simulation to perform the analysis, or should be offloaded to other computing resources, i.e., staging data and allowing the simulation to continue its execution.

Such a decision is an intrinsically multi-parametric choice as it depends on the duration of the analysis component, the amount of data to exchange between the components, and the time between two analyses. Thus, it translates into a complex optimization problem whose main objective is often to avoid blocking the progress of the simulation component.

To illustrate how SIM-SITU can be leveraged to obtain objective and faithful performance indicators and guide the design of potentially complex *in situ* workflows, we consider the following experimental scenario where a user has only access to a single node with 32 cores to execute a MD simulation and perform some analysis (using the non-coordinated version of the analysis actors) and has to choose between a space division of execution (i.e., simulation and analysis components can run concurrently on 16 cores each) or a time division of execution (i.e., simulation and analysis components alternate their executions using 32 cores). We start from a baseline scenario in which the respective overall execution times of the simulation and analysis components are well balanced using a space division of execution. Then, we scale (up and down) the amount of analysis to perform and show in Figure 6 the impact of this modification on the active and idle times of each component and the overall execution time. For the sake of simplicity, we ignored the time to stage data to the memory of the cores dedicated to the analysis with a space division of execution by using the message queue implementation of SIM-SITU's DTL.

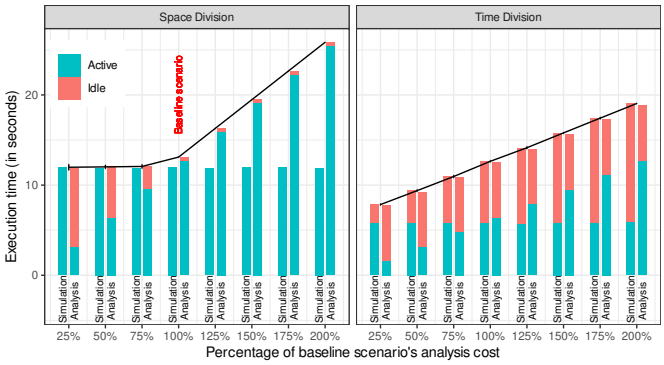

Fig. 6. Evolution of the active and idle times of the simulation and analysis components and of the overall execution time (black solid line) with the amount of analysis to perform, under space and time divisions of execution.

Scaling down the amount of analysis to perform causes the overall execution time to be dominated by the duration of the simulation component. In such configurations, i.e., for 25% to 75% of the baseline cost, opting for a time division of execution leads to shorter overall execution times, as more resources can be allocated to the simulation component and inline the analysis remains affordable.

When the analysis cost is scaled up to become dominant, we observe different consequences. With a time division, each component has to wait for the other and is often idle, but both complete at the same time. With a space division, each component needs more time to execute its workload, as it runs on less resources, but the simulation component can finish earlier and release its resources. Moreover, the duration of the simulation component is not impacted by the volume of analysis in this configuration, which makes it more predictable. However, if the completion time of both components is the performance metric to optimize, a time division of execution should be favored.

More factors can influence the choice of division of execution which are not included in this simple illustrative study. For instance, alternating between components with a time division can cause cache trashing and thus impact the performance of both components. When both simulation and analysis perform intense communications, a space division may lead to competition for the network and also degrade the performance of both components if no care is given to the mapping of the analysis actors. In that particular case, the coordinated analysis actors of SIM-SITU can be used to estimate the effects of such network interference and determine the best mapping for the different components.

On-node processing: Impact of Simulation to Analysis Core Allocation Ratio. When the analysis/visualization component is executed on the same resources as the simulation component with a space division of execution, a key performance parameter is to determine a good *simulation to analysis core allocation ratio* R, defined as the number of cores allocated to the simulation component over the number of cores allocated to the data analytics components [28]. As our target cluster has 32 cores per node, we consider 5 values for this ratio as shown in Table I. Then, we run simulations for 1, 2, 4, and 8 nodes (i.e., 32, 64, 128, and 256 cores).

TABLE I
CONSIDERED SIMULATION TO ANALYSIS CORE ALLOCATION RATIOS.

R	# simulation cores	# analysis cores
1	16	16
3	24	8
7	28	4
15	30	2
31	31	1

Then we consider a user who would like to perform a constant amount of analysis during the execution of their simulation and know, for a given number of cores, what would be the most efficient simulation to analysis core allocation ratio to use. This user can act on two parameters to execute the desired amount of analysis: the *frequency* and the *cost* of one execution of the analysis/visualization component. For instance, if the main simulation loop is executed 8,000 times and 400 units of

analysis have to be performed, (at least) four (frequency, cost) configurations can be envisioned: (20, 1), (200, 10), (500, 25), and (1000, 50). The (500, 25) configuration means that 25 units of analysis work are performed every 500 iterations. Thanks to the flexibility of SIM-SITU, varying the analysis cost simply amounts to changing the value of the computing scaling factor parameter.

To compare the performance of the different (frequency, cost) configurations, we define an *efficiency ratio* η as:

$$\eta = 1 - \frac{I}{m}, \tag{1}$$

where I is the sum of the idle times experienced by both the simulation and analysis/visualization components, i.e., when one component is waiting for the other to proceed with its execution, and m is the makespan, or completion time, of the overall *in situ* processing.

Figure 7 shows the achieved efficiency for these four combinations of frequency and analysis costs and two core-allocation ratios ($R = 15$ and $R = 31$). The other ratios lead to lower efficiency for any core count and are thus not displayed for the sake of readability.

It shows some interesting trends and tradeoffs. We can see that the (frequency, cost) configuration that leads to the best efficiency is not the same for every core count. As the number of cores available for the execution of the *in situ* processing increases, it appears to be more efficient to reduce the frequency and increase the cost of the analysis/visualization component. This is confirmed by the trends of the (20, 1) and (200, 10) configurations with $R = 31$ whose efficiency steadily decreases with the increase of the number of cores. For these configurations, the analysis actors do not have enough work to process and thus are idle most of the time. As the total core count grows, more analysis actors are started, hence amplifying this phenomenon. A similar trend can be seen for larger analysis cost, but the tipping point where the efficiency starts to drop is for larger core counts.

We also observe a generally decreasing trend for the efficiency as the number of cores grows with $R = 15$, but

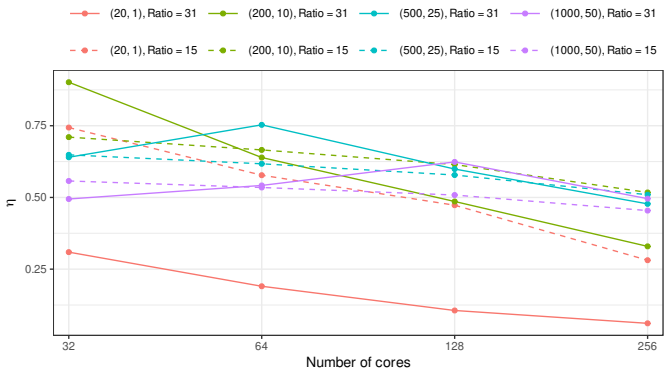

Fig. 7. Efficiency of ExaMiniMD *in situ* workflow in four (frequency, cost) configurations for two core allocation ratios.

over a narrower range. Moreover, the (200, 10) configuration appears to be consistently achieving good efficiency for this core allocation ratio, for all total core counts. This better stability might be preferred by users when they have to adapt their executions to the number of currently available cores they have access to and do want to risk to loose efficiency by selecting the wrong configuration.

Figure 8 shows a different view of the (1000, 50) configuration, i.e., the evolution of the active and idle times of the simulation and analytics components when the core allocation ratio and the total number of cores increase[2].

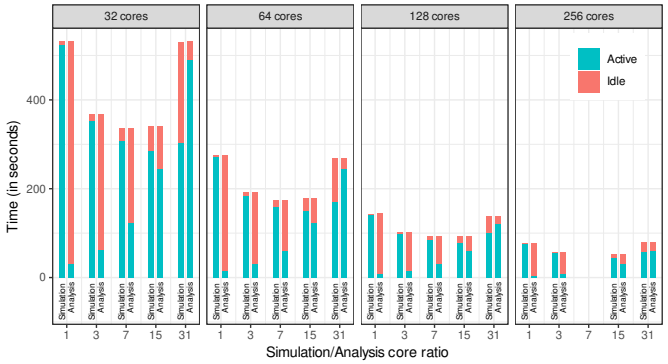

Fig. 8. Evolution of the active and idle times of the simulation and analytics components when increasing the core allocation ratio and the total number of cores for the (1000, 50) scenario.

In this scenario, we can see that the respective active times of the simulation and analysis/visualization component follow opposite trends. For small core allocation ratios, the execution time is completely dominated by the time to execute the simulation. Then, the time to execute the analysis component increases linearly with the ratio, as less cores are allocated to execute the same amount of analysis, until a tipping point is reached where the simulation waits for analysis ($R = 31$).

We also see that some "sweet spots" can be found, typically for $R = 15$, where the active times of the simulation and analysis/visualization components are both efficient and well balanced. It is interesting to note that it is in contradiction with the efficiency metric that indicates a better efficiency with a core allocation ratio of 31 from 64 cores.

These results illustrate how SIM-SITU can be used to determine a good core-allocation ratio for a given cost of analysis and number of nodes and that it is important for users of the SIM-SITU framework to leverage all the metrics the tool can provide them while configuring their *in situ* processing .

On-node vs. Off-node processing: Impact of data staging. Another design choice faced by users of *in situ* workflows is to decide of the mapping of the resources allocated to the data analytics component. In other words, the question is: "would it better to adopt an on-node strategy, i.e., mapping

[2]The lack of values for 256 cores and $R = 7$ is due to a crash of ExaMiniMD for this particular instance with 224 MPI ranks (w/ or w/o SIM-SITU). It seems to come from a badly handled division by 0 in ExaMiniMD's code.

the analytics resources on the same nodes as the simulation resources, or an off-node strategy, i.e., dedicating some node(s) to the analytics?". The former has the advantage of minimizing the cost of data exchanges between simulation and analytics thanks to a shared memory space, but the scattering of the analysis resources over multiple nodes may hinder the performance of this component (e.g., communication intensive analysis/visualization routine). Conversely, the latter benefits of having all the analysis located on a single, or small number of dedicated nodes, but induces a larger communication overhead to exchange data with the simulation component.

To illustrate how SIM-SITU can help users to evaluate the relative performance of on-node and off-node proximity schemes, we consider the following scenario. The simulation component still corresponds to the main loop of the target application. The analysis/visualization component now corresponds to a routine that involves all the analysis resources and whose performance is impacted by the number of nodes onto which these resources are allocated, i.e., its execution time increases with the resource scattering. Finally, the user can decide of the volume of data produced by the simulation to transfer to the analysis.

Simulating such a performance study is made easy by the features of SIM-SITU. Switching from an on-node to an off-node mapping simply amounts to change the analysis hostfile, while changing the volume of transferred data or the performance profile of the analysis/visualization routine can be done by modifying some parameter values. Figure 9 shows the evolution of the execution time of the simulation component when the volume of data to exchange with the analysis component is scaled up to a thousand times. The applications is executed on 16 nodes and two execution modes are considered. The on-node mapping uses a core allocation ratio of 15, i.e., two cores per node are allocated to the analytics component while a full node is dedicated to the analysis in the off-node mapping.

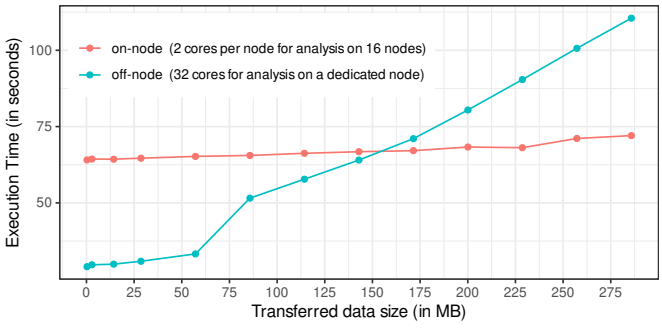

Fig. 9. Evolution of the execution time of the simulation component when the volume of data transfer is scaled up in the on-node and off-node execution modes with 16 nodes and $R = 15$.

We can see that the scattering of the analysis resources across nodes makes the on-node execution mode less efficient than the off-node execution mode when a small amount of data is exchanged. However, as we increase this volume of transferred data, the execution time of the off-node mode starts to increase linearly, while the on-node execution mode is only slightly impacted as the data exchanges are done through a shared memory space. Such studies may help users in the design of their analysis/visualization component by showing them where lies the tipping point between off-node and on-node mappings for a given configuration of their *in situ* processing.

V. CONCLUSION AND FUTURE WORK

Analyzing or visualizing the data produced by large-scale numerical simulations as they are produced is an appealing alternative to the classical *post-hoc* approach that is more and more impacted by the increasing discrepancy between the relative performance of computing and storage subsystems in extreme-scale supercomputers. However, the development of such *in situ* processing raises several challenging questions, such as "what *amount* of analysis can be done and at which *frequency*?", "how many *resources* can be taken off of the execution of the simulation to execute the analysis?", or "Is it better to perform *on-node* or *off-node* analysis/visualization?".

Determining answers to these questions that do not cause the performance of *in situ* processing to be worse than the classical "simulation then analysis" approach usually falls down to evaluating the performance of different allocation, mapping, and data movement strategies for different input configurations. However, the state-of-the-art on the performance evaluation of *in situ* processing shows that it relies either on time- and resource-consuming experiments on a limited set of scenarios or on the execution of abstracted versions of the initial applications that may lack of realism.

In this paper, we introduced the SIM-SITU framework, a generic framework for *in situ* processing based on the popular SimGrid toolkit. The modular design of SIM-SITU faithfully captures the features of state-of-the-art *in situ* processing systems. We illustrated its capacities on a Molecular Dynamics use case. With only a few minor code modifications, we showed how SIM-SITU could be used to study different execution scenarios of *in situ* processing and highlight important performance tradeoffs.

As part of our future work, we plan to further demonstrate the capacities of SIM-SITU by investigating more allocation and mapping strategies on different use-case applications. We will particularly focus on off-node processing where nodes are dedicated to analysis/visualization. Studying such strategies would be a first step in evaluating the impact of data transfers and network performance on *in situ* processing. We also plan to extend the capacities and realism of SIM-SITU by developing more complex versions of the Data Transport Layer component that mimic the behavior of popular implementations such as ADIOS [15], DataSpaces [18], or Dimes [19]. The objective is to provide SIM-SITU users with the capacity to easily select which flavor of the DTL they want to use for their *in situ* processing. Finally, we plan to leverage SIM-SITU to carry out performance evaluations in scenarios that would be hardly possible to evaluate through actual

experiments on supercomputers. For instance, the necessity of running series of simulations in ensembles broadens the range of feasible *in situ* configurations and raises new allocation and scheduling challenges. Moreover, the modularity of the SIM-SITU framework offers enough flexibility to envision the online evaluation of scheduling decisions in the context of an adaptive sampling process [29].

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] R. Gerber, J. Hack, K. Riley, K. Antypas, R. Coffey, E. Dart, T. Straatsma, J. Wells, D. Bard, S. Dosanjh, I. Monga, M. E. Papka, and L. Rotman, "Crosscut report: Exascale Requirements Reviews," 2018. [Online]. Available: https://www.osti.gov/biblio/1417653

[2] H. Childs, S. Ahern, J. Ahrens, A. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier, S. Dutta, J. Favre, T. Fogal, S. Frey, C. Garth, B. Geveci, W. Godoy, C. Hansen, C. Harrison, B. Hentschel, J. Insley, C. Johnson, S. Klasky, A. Knoll, J. Kress, M. Larsen, J. Lofstead, K.-L. Ma, P. Malakar, J. Meredith, K. Moreland, P. Navrátil, P. O'Leary, M. Parashar, V. Pascucci, J. Patchett, T. Peterka, S. Petruzza, N. Podhorszki, D. Pugmire, M. Rasquin, S. Rizzi, D. Rogers, S. Sane, F. Sauer, R. Sisneros, H.-W. Shen, W. Usher, R. Vickery, V. Vishwanath, I. Wald, R. Wang, G. Weber, B. Whitlock, M. Wolf, H. Yu, and S. Ziegeler, "A Terminology for in situ Visualization and Analysis Systems," *IJHPCA*, vol. 34, no. 6, pp. 676–691, 2020.

[3] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *JPDC*, vol. 74, no. 10, pp. 2899 – 2917, 2014.

[4] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics," in *Proc. of the 27th IEEE International Symposium on Parallel and Distributed Processing*, Boston, MA, 2013, pp. 320–331.

[5] P. Malakar, V. Vishwanath, T. Munson, C. Knight, M. Hereld, S. Leyffer, and M. E. Papka, "Optimal Scheduling of In-Situ Analysis for Large-Scale Scientific Simulations," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX, Nov. 2015.

[6] P. Malakar, V. Vishwanath, C. Knight, T. Munson, and M. E. Papka, "Optimal Execution of Co-analysis for Large-Scale Molecular Dynamics Simulations," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, Nov. 2016, pp. 702–715.

[7] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar, "Adaptive Data Placement for Staging-Based Coupled Scientific Workflows," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX, Nov. 2015, pp. 1–12.

[8] P. Subedi, P. E. Davis, and M. Parashar, "Leveraging Machine Learning for Anticipatory Data Delivery in Extreme Scale In-situ Workflows," in *Proc. of the IEEE International Conference on Cluster Computing*, Albuquerque, NM, Sep. 2019, pp. 1–11.

[9] E. Lohrmann, Z. Lukić, D. Morozov, and J. Müller, "Programmable In Situ System for Iterative Workflows," in *Proc. of the 21st Workshop on Job Scheduling Strategies for Parallel Processing*, 2018, pp. 122–131.

[10] G. Aupy, B. Goglin, V. Honoré, and B. Raffin, "Modeling High-Throughput Applications for In Situ Analytics," *IJHPCA*, vol. 33, no. 6, pp. 1185–1200, 2019.

[11] T. M. A. Do, L. Pottier, S. Caíno-Lores, R. Ferreira da Silva, M. A. Cuendet, H. Weinstein, T. Estrada, M. Taufer, and E. Deelman, "A Lightweight Method for Evaluating In Situ Workflow Efficiency," *Journal of Computational Science*, vol. 48, p. 101259, 2021.

[12] P. Velho, L. M. Schnorr, H. Casanova, and A. Legrand, "On the Validity of Flow-Level Tcp Network Models for Grid and Cloud Simulations," *ACM TOMACS*, vol. 23, no. 4, Dec. 2013.

[13] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, and F. Suter, "Simulating MPI applications: the SMPI approach," *IEEE TPDS*, vol. 18, no. 8, pp. 2387–2400, 2017.

[14] P. Bédaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, and B. Videau, "Toward Better Simulation of MPI Applications on Ethernet/TCP Networks," in *Proc. of the 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Denver, CO, 2013.

[15] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[16] M. Larsen, A. Woods, N. Marsaglia, A. Biswas, S. Dutta, C. Harrison, and H. Childs, "A Flexible System for in Situ Triggers," in *Proc. of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'18)*, Dallas, TX, 2018.

[17] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations," *ACM Transactions on Parallel Computing*, vol. 3, no. 3, oct 2016.

[18] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[19] F. Zhang, T. Jin, Q. Sun, M. Romanus, H. Bui, S. Klasky, and M. Parashar, "In-memory Staging and Data-Centric Task Placement for Coupled Scientific Simulation Workflows," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, p. e4147, 2017.

[20] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs, "Performance Modeling of in Situ Rendering," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, 2016.

[21] V. Honoré, T. M. A. Do, L. Pottier, R. Ferreira da Silva, E. Deelman, and F. Suter, "Reproducibility artifact for the "sim-situ : A framework for the faithful simulation of in-situ processing" paper," Jul 2022. [Online]. Available: https://doi.org/10.6084/m9.figshare.20416008

[22] A. Thompson and C. Trott, "A Brief Description of the Kokkos implementation of the SNAP potential in ExaMiniMD," Office of Scientific and Technical Information, Tech. Rep. 1409290, Nov. 2017.

[23] "ExaMiniMD Proxy Application GitHub Repository," [Online] https://github.com/ECP-copa/ExaMiniMD, May 2021.

[24] O. Aaziz, C. Vaughan, J. Cook, J. Cook, J. Kuehn, and D. Richards, "Fine-Grained Analysis of Communication Similarity between Real and Proxy Applications," in *Proc. of the 10th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Denver, CO, Nov. 2019.

[25] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[26] T. Cornebize, "High Performance Computing: towards better Performance Predictions and Experiments," Ph.D. dissertation, Université Grenoble-Alpes, Grenoble, France, Jun. 2021.

[27] D. Richards, O. Aaziz, J. Cook, H. Finkel, B. Homerding, T. Juedeman, T. McCorquodale, Peter an Mintz, and S. Moore, "Quantitative Performance Assesment of Proxy Apps and Parents," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-750182, Apr. 2018.

[28] P. Malakar, T. Munson, C. Knight, V. Vishwanath, and M. Papka, "Topology-Aware Space-Shared Co-Analysis of Large-Scale Molecular Dynamics Simulations," in *Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, Dallas, TX, Nov. 2018.

[29] E. Hruska, V. Balasubramanian, H. Lee, S. Jha, and C. Clementi, "Extensible and Scalable Adaptive Sampling on Supercomputers," *Journal of Chemical Theory and Computation*, vol. 16, no. 12, pp. 7915–7925, 2020.