A Versatile Simulated Data Transport Layer for In Situ Workflows Performance Evaluation

Frédéric Suter
Oak Ridge National Laboratory, Oak Ridge, TN, USA
suterf@ornl.gov

Abstract—**In situ processing does not only allow scientific applications to face the explosion in data volume and velocity but also to address the time constraints of many simulation-analysis workflows by providing scientists with early insights about their applications at runtime. Multiple frameworks implement the concept of a data transport layer (DTL) to enable such in situ workflows. These tools are very versatile, directly or indirectly access the data generated on the same node, another node of the same compute cluster, or a completely distinct node, and allow data publishers and subscribers to run on the same computing resources or not. This versatility puts on researchers the onus of taking key decisions related to resource allocation and how to transport data to ensure the most efficient execution of their in situ workflows. However, domain scientists and workflow practitioners lack the appropriate tools to assess the respective performance of particular design and deployment options.**

In this paper we introduce a versatile simulated DTL designed to provide researchers with insights on the respective performance of different execution scenarios of in situ workflows. This open-source, standalone library builds on the SimGrid toolkit and can be linked to any SimGrid-based simulator. It facilitates the evaluation of the performance behavior, at scale, of different data transport configurations and the study of the effects of resource allocation strategies. We demonstrate the scalability, versatility, and accuracy of this simulated DTL by reproducing the execution of two synthetic benchmarks and of a real-world in situ workflow composed of an MPI application and a parallel data analysis. Results of simulations run on a *single core* show that the proposed library can simulate the interactions of tens of thousands of simulated processes deployed on two interconnected commodity clusters in a few seconds, and the execution by a thousand simulated processes of an in situ workflow in less than three minutes.

I. Introduction

In situ processing [1], originally defined as the capacity to analyze or visualize data as it is generated by scientific applications or instruments, is an alternative to the post hoc processing paradigm, in which analyses are performed only after an entire dataset has been produced. In situ processing allow applications in various scientific domains such as cosmology, nuclear engineering, climate modeling, or biology [2]–[5], to manage the explosion in data volume and velocity combined to the growing discrepancy between storage subsystems performance and computing power in extreme-scale supercomputers. It also addresses the time constraints of many simulation-analysis workflows [6] by providing researchers with early insights about the evolution of their applications at runtime, offering them additional command-and-control capacities (e.g., early termination, exploration of new parameters, additional analyses), and eventually reducing the time to science results.

The term "in situ" has evolved over the last decade to become an umbrella term covering approaches well beyond its initial definition. In situ workflows now facilitate data reduction, annotation, and transformation at different stages of their execution and heavily rely on an efficient data management layer to transparently optimize data operations [7].

This evolution led the in situ processing community to capture the different meanings of the "in situ" term in a comprehensive terminology [8]. This terminology defines a *data transport layer* (DTL) as an application-aware, multi-purpose interface system integrated with an application, that can directly or indirectly access data on the same node, another node of the same compute cluster, or a completely distinct node, and allow data publishers and subscribers to run on the same computing resources or not.

This complex and versatile definition of a DTL highlights a common challenge faced by users of the multiple frameworks that implement this definition [9]–[13]. They usually must determine how many resources to allocate to the components of an in situ workflow, where to allocate them, and how to transport data from one component to another. These decisions are key to efficient executions. However, domain scientists and workflow practitioners lack the appropriate tools to assess the respective performance of particular design and deployment options. They must either rely on crude back-of-the-envelope estimations that lack of realism or actually execute their workflows, usually at small scale, using different configurations to determine what could be the best options for runs at larger scale. This second approach can become time- and resource- consuming and may still fail to capture certain phenomena that only appear at a certain scale.

This paper introduces a *versatile simulated data transport layer* designed to help researchers with the performance evaluation of in situ workflows design and deployment options. We implement this simulated DTL as an open-source, standalone shared library [14], built on top of the SimGrid framework [15] that can be linked to any SimGrid-based simulator.

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). 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).

SimGrid provides accurate and scalable simulation models and low-level abstractions for simulating distributed computing infrastructures (e.g., commodity clusters, clouds, or high-performance computing systems) [16], MPI runtime systems [17], and distributed applications. Since 2001, SimGrid has been used in more than 650 publications [18]. However, its low-level simulation abstractions can make the implementation of simulators of complex systems labor-intensive [19]. With its higher level abstractions and programming interface, the proposed simulated DTL can reduce the development effort needed to evaluate the behavior and performance, at scale, of different data transport configurations and study the effects of resource allocation strategies on performance. We demonstrate its scalability, versatility, and accuracy by reproducing, in simulation, the execution of synthetic benchmarks used by actual DTL frameworks and of a real-world in situ workflow composed of a parallel application and a parallel data analysis that periodically exchange data through the DTL. Results of simulations run on a *single core* show that the proposed library can simulate the interactions of tens of thousands of simulated processes deployed on two interconnected commodity clusters in a few seconds, and the execution by a thousand simulated processes of an in situ workflow in less than three minutes.

The rest of this paper is organized as follows. Section II describes frameworks that implement a DTL, identify their common features, and reviews previous efforts in leveraging simulation to analyze the performance of in situ workflows. We detail our simulated DTL in Section III and evaluate its scalability, versatility, and accuracy in Section IV. Finally, Section V summarizes our work and outlines future work.

II. Related Work

Several frameworks have been developed to efficiently transport data within in situ processing workflows. Hereafter, we review their main characteristics and salient features. We also review the few works considering the simulation-based performance evaluation of the execution of in situ workflows.

The ADIOS [9] high-performance I/O framework exposes a publish/subscribe programming interface to allow scientific applications to explicitly describe the data they produce and when it is ready for output, and for analysis components to express what data they need. A key feature of ADIOS are its multiple engines that can either write/read directly to/from the storage system or stream data from the application to the memory of staging nodes where analysis and visualization components run, using different techniques.

The DataSpaces framework [10] exposes a virtual shared space to the components of in situ workflows to support dynamic and asynchronous data-intensive coupling patterns. Components can dynamically attach or detach themselves from this shared space during the execution of the workflow. DataSpaces relies on geometric descriptors (e.g., coordinates, or space filling curves) to identify regions of interest that data consumers can query. For data transport, DataSpaces initially relied on the DART data transport substrate [20] but has been since integrated as an ADIOS engine.

LowFive [11] implements a DTL as an HDF5 Virtual Object Layer plugin and is distributed as a standalone library. It can transport data from multiple producers to multiple consumers either through files or directly over the network via MPI. When the numbers of producers and consumers or the source and destination data layouts differ, LowFive can redistribute data while transporting it. When codes are already using HDF5 for their I/Os, they require no further modification to use LowFive as a data transport layer.

The CAPIO middleware framework [12] can transform file-based data transport in data-intensive workflows into streaming-based data transport, without having to modify the application code. The objective is to reduce the workflow execution time by avoiding expensive I/O operations on a shared file system. To this end, the I/O coordination language of CAPIO allows workflow designers to express additional semantics related to data dependencies through code annotations. Its runtime system then exploits this semantic information to generate the data streams accordingly.

Maestro [13] is a memory- and data-aware middleware for data orchestration within workflows. At its core, Maestro manages a pool of resources to which workflow components can contribute by submitting or requesting data objects. These objects encapsulate data and metadata that provide hints about how to efficiently transport data. The data management layer of Maestro relies on libfabric to move, re-layout, redistribute, or copy data across workflow components.

For in situ visualization, tools such as Paraview Catalyst [21] allow users to build pipelines and instrument their code to feed these pipelines with data as they are produced.

All these frameworks share common features: (i) a publish/-subscribe programming interface that allows the components of in situ workflows to fully delegate data management to an external framework; (ii) rich metadata providing the data transport layer with information about the type, shape, and distribution of a given piece of data; and (iii) different transport methods ranging from the use of files on a shared storage space, to direct (deep or shallow) copies in a shared memory space through memory-to-memory transfers over the network. We used these features as a foundation for the development of the proposed versatile simulated data transport layer.

Only a few works have leveraged simulation to analyze the performance of in situ workflows. Aupy *et al.* designed a numerical simulator [22] to evaluate scheduling decisions for a set of in situ analyses by solving optimization problems on resource allocation and partitioning. This simulator uses a predetermined set of parameters to study the effect of the scheduled in situ analyses on the performance of the entire workflow. However, its underlying simulation models are simplistic and ignore network contention. The proposed simulated DTL benefits from SimGrid's validated simulation models [23] and can thus produce more realistic simulation results. Do *et al.* created a synthetic Molecular Dynamics (MD) application [24] by extrapolating benchmark results and used this lightweight placeholder in their performance evaluation. This tool, which is not available online, is thus limited to the study of MD in situ workflows. Our open-source DTL allows to develop simulators of any type of producer and consumer applications, at different levels of detail. In a previous work [25] we showed the utility of simulation-based performance assessments when designing in situ workflows. However, this proof-of-concept implementation lacked of some important features now provided by the simulated DTL, such as the support of dynamic connection and disconnection of publishers and subscribers, file-based transport, or complex data redistribution patterns when streaming data.

III. Versatile Simulated Data Transport Layer

A. Overview and Terminology

A data transport layer can is the central component in a publish-subscribe system, that connects applications that produce data objects to those which consume them, moves these data objects, and keeps track of metadata. Fig. 1 shows a generic data transport layer, yet representative of actual frameworks, and introduces the terminology used in this paper.

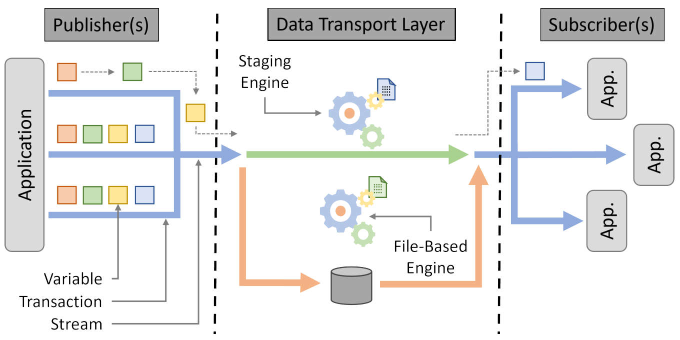


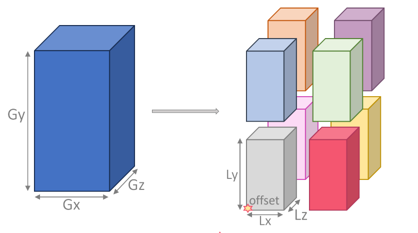
Fig. 1. Overview and terminology of a generic Data Transport Layer.

On the publisher side, one or more applications (e.g., large-scale physics simulations) produce data, represented by multiple *variables*, and publish them to the DTL within a *transaction* using a specific *stream*. On the subscriber side, one or more applications (e.g., analysis and visualization components) subscribe to specific streams to retrieve the data they need from the DTL. The DTL itself exposes multiple *engines* in charge of the actual data transport from the publisher(s) to the subscriber(s). These engines either rely on *files* stored on a file system or use *staging* techniques to move data in memory or over the network.

The proposed simulated DTL builds on the low-level abstractions exposed by the SimGrid toolkit [15]. Typical SimGrid-based simulators are composed of multiple *actors* (i.e., simulated processes) that launch *activities* (i.e., computations, network communications, or I/O operations) defined by an amount of work to do (e.g., bytes to read, write, or transfer, compute operations to perform) on *resources* (i.e., CPUs, network links, or disks). The simulation models at the core of SimGrid determine the respective completion date of these activities to make the simulated time advance. We combined these low-level abstractions to expose higher-level concepts and ease the writing of simulators of in situ processing.

B. Variables and Transactions

At the core of the simulated DTL is the data transported from publishers to subscribers. Many simulation-analysis workflows involve parallel MPI codes as data producers. These codes manipulate multidimensional arrays distributed over multiple ranks. We adopt this data structure, used by popular DTL frameworks, as the basis of our *variable* abstraction. Fig. 2 shows how to define a 3-dimensional array, distributed to eight MPI ranks in $2 \times 2 \times 2$ grid, as a self-descriptive tuple.



Variable: < name, {Gx, Gy, Gz},☆, {Lx, Ly, Lz}, element size >

Fig. 2. Description of a 3-dimensional array distributed over eight MPI ranks as a tuple composed of the name of the variable, its global dimensions, the local part owned by each rank, defined by an offset and an element count, and the size of the elements stored in this variable.

This tuple stores the name of the variable (that is unique to a given stream), the global dimensions of the multidimensional array (G_x, G_y, and G_z) and, for each rank, the local part (L_x, L_y, and L_z) owned by that rank after decomposition and distribution, and a 3D-offset (represented by a star) that indicates where the local array is positioned in the global array. Finally, the tuple stores the size of the elements in the array.

Fig. 3 shows how to describe such variables in a simulator, using the DTL interface. To describe scalar variables, a simplified version of the `define_variable()` function only requires the variable name and element size as parameters.

```
// Definition of a 3D variable to publish to the DTL.
// Each producer owns a chunk of 128x128x256 doubles.
auto v = stream->define_variable("3D-Var",
    {128 * nx, 128 * ny, 256 * nz}, // Global size
    {128 * x, 128 * y, 256 * z},    // Local offsets
    {128, 128 , 256},               // Local counts
    sizeof(double));                // Element size
```

Fig. 3. Definition of a 3D variable distributed over multiple producers.

The simulated DTL does not actually store or transport the contents of the multidimensional array represented by a variable. Internally, only the distribution of the variable and its local and global sizes matter. The distribution is used to determine I/O or communication patterns when transporting the variable while the size is used to compute the simulated cost of transport. The DTL stores this information as metadata while it is active. These metadata enable the simulation of execution scenarios in which actors subscribe to data after their production in a *post-hoc* fashion.

Simulated actors can publish, or subscribe to, one or more variables within a *transaction*. This logical construct, inspired by the notion of *step* of the ADIOS framework, delimits the interactions between an actor and the DTL and enables the synchronization between publishers and subscribers.

When a simulated actor starts a new transaction on a stream, the DTL makes it wait for the completion of any in-flight data transport activity from the previous transaction on that stream.

Actors that subscribe to a variable can also, before beginning a new transaction, *select* a specific subset of the multidimensional array this variable represents (e.g., to focus on a smaller region of interest or adapt the decomposition and distribution of the variable to subsequent data processing). Fig. 4 shows such a selection, made on the subscriber side. Four actors subscribe to the 3D variable defined in Fig. 2 and select blocks of 2D slices, along the Z-dimension.

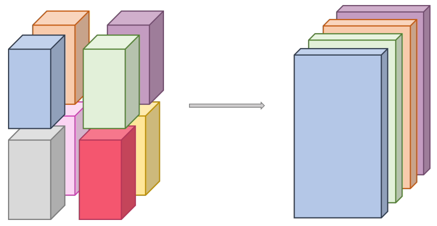


Fig. 4. Example of selection made on the subscriber side. Four actors subscribe to a variable published by eight actors with a different distribution. When the DTL streams this variable, this implies a $M \times N$ data redistribution.

This example, which corresponds to the data production and consumption patterns of the simulation-analysis in situ workflow considered in Section IV-D, also illustrates that the simulated DTL can express, and supports the combination of, such drastically different data access patterns.

During its execution, a simulated actor can perform several transactions. Indeed, in many in situ processing workflows (e.g., Particle-in-Cell codes in plasma physics or MD simulations), data is periodically produced, transported, and analyzed to monitor the progress of an iterative computation (e.g., to measure densities and currents within a plasma or follow the trajectories of atoms and molecules), or just saved to keep regular checkpoints. For any variable, the DTL keeps as metadata which actor(s) published it, in which transaction(s). This allows subscriber(s) to select specific transaction(s) when retrieving data from the DTL.

C. Streams

The *Stream* abstraction represents a connection between a simulated actor and the DTL, through which data transport operations occur, and acts as a *variable* factory. The publishers define the variables a stream has to transport. Each publisher provides global and local information about the variable as shown in Fig. 3. On the subscribers side, actors first have to inquire about a variable (i.e., to know its shape and size) before being able to retrieve it from the DTL. The `inquire_variable(name)` method of the Stream class returns a Variable object. Actors can also obtain a list of the names of the variables associated to a stream. Finally, opening a stream creates a specific *Engine* to actually handle data transport. We describe the currently available engines and transport methods in the next section.

D. Engines and Transport Methods

The *Engine* abstraction is the base interface through which the DTL interacts with the simulated communication or I/O subsystems in charge of the simulation of data movement or storage. We consider the two types of engine supported by most DTL frameworks: *file-based* engines, that write and read data to and from storage and *staging* engines that stream data from the memory of publishers to that of subscribers.

Engines are attached to streams. A simulated actor can thus adapt the type of engine to the purpose of each individual stream. For instance, one will create a stream with a file-based engine to store application checkpoints and another stream with a staging engine to transfer data from one analysis component to another. The type of engine to use can be specified either at the creation of a stream or in an external configuration file passed as argument when creating the DTL.

An engine is then associated to a specific *transport method* that further specifies how data is written to and read from a file system or streamed from one workflow component to another. This separation between engine and transport method allows users to switch between multiple implementations of the same service without having to modify the code of their in situ workflow simulator: Changing of transport method simply amounts to modifying a configuration parameter of the stream.

For file-based engines, the default transport method consists in having each publisher simulate the writing, for each transaction, of its own share of a variable in a distinct simulated file located on a specified simulated storage space. When a subscriber requests a (selection of a) variable with a different access pattern, the DTL first computes which files contain the different pieces of the requested variable and then simulates the corresponding read operations of these files, wherever they are virtually stored. The simulation of these I/O operations is delegated to the file system module of SimGrid [26] that exposes high-level abstractions for the simulation of file-system operations on local and remote storage.

To create a file-based engine, users of the DTL must specify where to store the simulated files. This is done by passing as argument to the `Stream::open()` method a string which contains the location and name of the targeted file system and a path to a specific directory. This information can also be stored in a separate configuration file, which means that users can test different scenarios (e.g., using a local or remote file system) without having to modify the code of their simulator.

The location of the file system has a direct impact on the simulation of I/Os by SimGrid's file system module. If the DTL accesses a remote file system, a write (resp. read) operation implies the simulation of a network communication before (resp. after) the simulation of the corresponding I/O operation on a storage device.

The DTL exposes two transport methods for staging engines. The first method simulates both memory copy and network transfer while ensuring the respect of flow dependencies. Whether a data copy or transfer is simulated depends on the respective mapping of the publisher and subscriber on computing resources. If both run on the same node, they virtually share a memory space, and the DTL simulates a deep memory copy—as an intra-communication whose performance can be configured in description of the simulated platform. Otherwise, it simulates a network communication.

To implement this, we leverage SimGrid's *mailbox* abstraction which acts as a rendez-vous point between actors. Only when two actors meet on such a rendez-vous point, the simulation of a memory copy or data transfer starts.

The second transport method provides users with a "what if an ideal transport existed?" baseline for their performance evaluation studies, i.e., all the data exchanges made through the DTL take zero time. This method leverages another abstraction exposed by SimGrid to simulate inter-process communications: *Message queues* have the same semantic and purpose as mailboxes, ensuring the respect of control and flow dependencies, but do not induce any simulated time.

When streaming data, a $M \times N$ data redistribution among M publishers and N subscribers may be necessary. The exact redistribution pattern is automatically determined by the DTL in three steps: (i) when a publisher *puts* a variable into a stream, it asynchronously waits for data requests (using zero-simulated-cost message queues) from any subscriber that opened that stream; (ii) when a subscriber *gets* (a subset of) this variable from the stream, it computes which publishers own pieces of its local view of the variable and send them each a request to put the corresponding pieces, defined by offsets and element counts, in dedicated mailboxes (resp. message queues); and (iii) when publishers end their transaction, they asynchronously put the requested pieces in these mailboxes (resp. message queues). The DTL then simulates the corresponding data exchanges, and may possibly force actors to wait for their completion when a new transaction starts.

E. Simulated In Situ Workflows

Resorting to simulation for the performance evaluation of in situ processing workflows allows users to capture complex, dynamic, and transient performance behaviors (e.g., network and I/O contention or dependencies between processes) that may cause unexpected waiting times and to abstract the computational complexity of their various workflow components.

The choice to build a simulated DTL on top of SimGrid is motivated by the fact that it not only provides a fast simulation kernel and validated network models [23] but also allows developers of simulators to combine different programming models (e.g., data-flow applications represented as directed acyclic graphs, communicating sequential processes, or MPI codes) within the same simulator [15], [27].

For instance, SimGrid's SMPI programming interface [17] allows for the simulation of full-fledged, unmodified parallel MPI applications as data producer and/or data consumer. If these applications are already interfaced with a data management framework to handle their I/Os, enabling their simulated execution with the proposed DTL amounts to replace the sections of code interacting with these data management frameworks, which are usually well identified and isolated, by calls to the simulated DTL.

It is also possible to define actors that mix simulated activities (e.g., computations) to MPI calls to mimic the execution flow of an MPI application without having to emulate the exact computations it performs. The simulation of such mock MPI applications, as shown in Fig. 5, is much faster but can still capture complex communication patterns.

```
1  static void simulation_main(int argc, char** argv) {
2    MPI_Init();
3    int nranks, rank;
4    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
5    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7    double size   = std::stod(argv[1]);
8    double l_size = size / sqrt(nranks);
9
10   auto dtl = DTL::connect();// Connect to the DTL
11
12   // Add a ''Data'' stream  using a ''File'' engine
13   auto s = dtl->add_stream("Data")
                 ->set_engine_type("File")
                 ->set_transport_method("File");
14   // Define a 2D array of chars
15   auto v = s->define_variable("V", {size, size},
                                {size*rank, size*rank},
                                {l_size, l_size},
                                sizeof(char));
16   // Open the stream in ''Subscribe'' mode
17   auto e = s->open("cluster:file_system:/working_dir/",
                      Stream::Mode::Publish);
18
19   // Allocate a shared data buffer for MPI
20   void* data = SMPI_SHARED_MALLOC(size * size);
21
22   for (int it = 0; it < 100; it++) {
23     // Compute a GFLOP
24     sg4::this_actor::execute(1e9);
25     // Perform an all-to-all collective communication
26     MPI_Alltoall(data, size * size, MPI_CHAR, data,
                    size * size, MPI_CHAR, MPI_COMM_WORLD);
27     sg4::this_actor::execute(500e8); // More computation
28
29     // publish data to the DTL
30     e->begin_transaction();
31     e->put(v, v->get_local_size());
32     e->end_transaction();
33   }
34
35   e->close();
36   DTL::disconnect();
37   SMPI_SHARED_FREE(data);
38   MPI_Finalize();
39 }
```

Fig. 5. Example of code for a synthetic MPI application that mixes actual MPI calls to simulated computations and data publication to the DTL.

Each actor involved in this parallel application first connects itself to the DTL (line 10), creates a stream (line 13), defines a variable (line 15), and creates an file-based engine by opening the stream (line 17). The core of this mock simulation is the for loop (lines 22 to 33) in which each actor computes a billion floating point operations, performs an all-to-all collective communication, performs more computation, and finally initiates a transaction with the DTL to publish its share of the variable. After a hundred iterations of that loop, each actor closes the engine (line 35) and disconnects itself from the DTL (line 36).

Fig. 5 also illustrates how SMPI features can be used to reduce the memory footprint of the simulated execution. Here, actors allocate (line 20) and free (line 37) a single buffer shared across all actors instead of allocating a distinct buffer each.

Fig. 6 illustrates the capacity to develop simulated actors at a higher level of abstraction. It shows the code of a generic analysis actor that subscribes to the same stream as in Fig. 5 (line 3), inquires about the variable V (line 6), and opens the stream in *Subscribe* mode to obtain a handler on the file-based engine (line 9). The core of this generic analysis is the loop in which the actor does a transaction to retrieve the current contents of variable V from the DTL (lines 13-15) and then performs a computation amounting to a thousand floating point operations per element (line 17). Finally, the actor closes the engine (line 20) and disconnects itself from the DTL (line 21).

```
1  static void analysis() {
2    auto dtl = DTL::connect()
3    auto s = dtl->add_stream("Data");
4
5    // Obtain metadata for variable ‘‘V’’
6    auto V = s->inquire_variable("V");
7
8    // Open the stream in ‘‘Subscribe’’ mode
9    auto e = s->open("cluster:file_system:/working_dir/",
                      Stream::Mode::Subscribe);
10
11   for (int i = 0; i < 10 ; i++) {
12     // Get the latest transaction for variable ‘‘V’’
13     e->begin_transaction();
14     e->get(V);
15     e->end_transaction();
16     // Compute 1e3 floating point operations per element
17     sg4::this_actor::execute(V->get_local_size() * 1e3);
18   }
19
20   e->close();          // Close the engine
21   DTL::disconnect();   // Disconnect from the DTL
22 }
```

Fig. 6. Example of code for a generic simulated analysis actor that iteratively retrieves data from the DTL and performs some computation.

F. Workflow Simulation and DTL Life Cycle

Fig. 7 shows how to create and run a simulation-analysis in situ workflow using the DTL to exchange data between the simulated MPI application described in Fig. 5 and the generic analysis process in Fig. 6 that runs on a single core.

```
1  int main(int argc, char** argv) {
2    sg4::Engine e(&argc, argv);
3    e.load_platform("./platform_description.so");
4
5    auto analysis_host    = e.get_host_by_name("node-0");
6    auto simulation_hosts =
        e.get_hosts_from_MPI_hostfile("./hostfile");
7
8    // Create the data transport layer
9    DTL::create("./DTL_config_file.json");
10
11   // Start a simulated MPI code instance run by
        multiple actors
12   SMPI_app_instance_start("simulation",
        simulation_main, simulation_hosts, argc, argv);
13
14   // Create a single in situ analysis actor
15   e.add_actor("analysis", analysis_host, analysis_main);
16
17   // Run the simulation
18   e.run();
19   return 0;
20 }
```

Fig. 7. Example of the `main` function of a simulator of a simple in situ workflow in which a parallel MPI application periodically publishes to the DTL data to be processed by a single subscribing analysis component.

This code first creates the SimGrid engine in charge of the execution of the simulation (line 2) and loads a shared library that describes the simulated platform (line 3). Then, it assigns roles to hosts (i.e., simulated compute nodes) in that platform. The analysis is executed on `node-0` (line 5) while where the MPI ranks will run is defined in a regular MPI hostfile (line 6).

The DTL is created by calling `DTL::create()` (line 9). This function can take as an optional argument a JSON configuration file that describes the different streams to be created during the simulation each with a name, engine type, and transport method. An instance of the MPI application in Fig. 5 (line 12) and the analysis actor in Fig. 6 (line 15) are then created before running the simulation (line 18).

In situ processing workflows involving more than two components can be simulated using the same principles as in Fig. 5 to 7: (i) each component is either an MPI code or a simulated actor; (ii) each component creates one or more streams to publish and/or subscribe to data; and (iii) the streams create the flow dependencies between components and form the workflow.

A common in situ processing scenario is that some analyses or visualization are only needed when certain conditions are met. In such cases, a new process is spawned, subscribes to some variables, and analyzes or visualizes data. The DTL has been designed to enable the development of simulators in which actors can connect to or disconnect from the DTL at any time. The DTL thus remains active from its creation until the end of the simulation when it is automatically destroyed. Internally, the DTL is implemented as a server daemon process that answers connection and disconnection requests from the simulated actors and maintains the set of active connections.

IV. Experimental Evaluation

This section presents the results of different experiments to assess the scalability, versatility, and accuracy of the proposed simulated data transport layer. All the simulation runs have been executed on a single core of an Intel Xeon Platinum 8380 processor at 2.30 GHz running an Ubuntu 24.04. The DTL v0.1 library [14] is built on top of SimGrid v4.0 [28] and its file system simulation module FSMod v0.3 [26]. The source code of the different benchmarks used in this section, as well as all the scripts to run the experiments and produce the presented graphs are available online [29].

A. Simulated Platform

For this experimental evaluation, we consider a simulated platform composed of two homogeneous clusters. The first cluster, dedicated to the simulation component, comprises 256 96-core nodes while the second cluster dedicated to analyses has 128 48-core nodes. Each core of these clusters can respectively process 11×10^9 and 6×10^9 floating point operations per second. These clusters have the same internal network topology: Nodes are connected through 1 Gb/s private network links to a 10 Gb/s crossbar switch. The two clusters are interconnected through a single 20 Gb/s network link.

TABLE I
PARAMETERS OF THE DATASPACES SCALABILITY EXPERIMENT.

Number of Publishers	64	128	1,024	2,048	4,096	8,192
Publisher distribution	$4 \times 4\times 4$	$4 \times 4 \times 8$	$8 \times 8 \times 16$	$8 \times 16 \times 16$	$8 \times 32 \times 16$	$16 \times 32 \times 16$
Number of Subscribers	8	16	128	256	512	1,024
Subscriber distribution	$4 \times 1\times 2$	$4 \times 1 \times 4$	$8 \times 2 \times 8$	$8 \times 4 \times 8$	$8 \times 8 \times 8$	$16 \times 8 \times 8$
Variable global size	2 GB	4 GB	32 GB	64 GB	128 GB	256 GB

Each node of the first cluster has a scratch space of 1 TB on a SSD disk whose read and write bandwidths are 560 MB/s and 510 MB/s respectively. Both clusters share access to a 100 TB file system whose read and write bandwidths are 180 MB/s and 160 MB/s respectively. The first cluster is connected to this remote file system through a 20 Gb/s network link, while the second cluster has a slower connection at 10 Gb/s.

B. Scalability Study – Simulation Time

To assess the capacity of the DTL to answer concurrent insert and retrieve requests at scale, we simulate the execution of the scalability experiment used to evaluate DataSpaces [10]. This weak-scaling synthetic benchmark involves from 64 to 8,192 publishers that send, through the DTL, a 3-dimensional array of double precision elements to 8 to 1,024 subscribers. Each publisher is assigned a $128 \times 128 \times 256$ region of the array, injecting 256 MB into the DTL. Each subscriber retrieves a $128 \times 512 \times 512$ region. Table I summarizes the distribution of publishers and subscribers along each dimension.

We simulate two in situ execution scenarios with different proximity [8]: *On-node* and *distinct resources*. In the on-node scenario, 64 publishers and 8 subscribers run on the same node (with more nodes involved for the larger configurations). Data transport thus occurs from memory to memory with a staging engine, or using the local scratch space of each node with a file-based engine. In the second scenario, publishers run on the first cluster while subscribers run on the second cluster. In that configuration, staging transport implies network communications between the two clusters, while file-based transport relies on the shared storage space. In both scenarios, we create a stream per compute node and execute one and ten transactions with a 1-second delay between two transactions. Fig. 8 shows the measured average *simulation times* over ten executions for the on-node scenario and Fig. 9 for the scenario with distinct computing resources allocated to the analysis.

We observe similar performance for both scenarios. The simulated DTL can process concurrent insertion and retrieve requests submitted by more than ten thousands simulated actors within a single transaction in less than one second.

We also see that the simulation time increases sub-linearly with the number of transactions, which further illustrates the good scalability of the simulated DTL and its capacity to handle complex data transport scenarios in a reasonable time. The faster increasing simulation time of staging engines when actors performs multiple transactions comes from the additional internal complexity of having subscribers requesting data pieces to publishers.

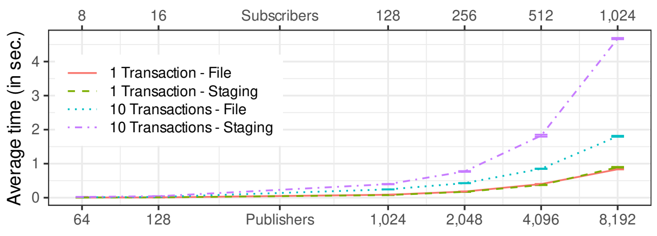


Fig. 8. Average simulation time of the weak scaling benchmark in the *on-node* scenario for 1 or 10 transactions with a File or Staging engine. The bottom scale of the X axis shows the number of actors publishing data while the top scale shows the number of subscribers retrieving data from the DTL.

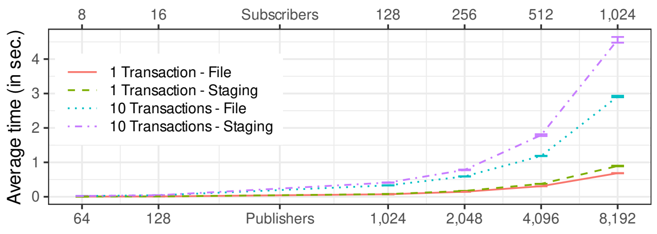


Fig. 9. Simulation times of the weak scaling benchmark in the *distinct resources* scenario, for 1 or 10 transactions with a File or Staging engine.

C. Scalability Study – Simulated Time

To further assess the scalability of the simulated DTL, we also simulate the execution of a scalability test used to evaluate LowFive [11], focusing now on the simulated time. This synthetic benchmark is composed of two parallel applications that respectively produce and consume two variables: A regular grid of 64-bit unsigned integers and a list of particles, each particle being represented by three 32-bit floats. Each publisher produces one million grid points and one million particles, thus publishing a total of 19 MB to the DTL. The publishers to subscribers ratio is 3:1. Each subscriber thus retrieves three million grid points and three million particles, or 57 MB of data from the DTL. Table II summarizes the different configurations used in this second weak scaling experiment.

We simulate three in situ execution scenarios: The on-node and distinct resources scenarios as in the previous section and an additional *off-node* scenario in which the subscribers run on the same compute cluster as the publishers but on different nodes. The difference between the on-node and off-node scenarios is that for the latter, the DTL must transport data over the internal network of the compute cluster when

TABLE II
PARAMETERS OF THE LOWFIVE SCALABILITY EXPERIMENT.

#Actors	#Pub.	#Sub.	#Particles & #Grid Points	Data size (in GB)
4	3	1	3M	0.06
16	12	4	12M	0.22
64	48	16	48M	0.99
256	192	64	192M	3.54
1,024	768	256	768M	14.34
4,096	3,072	1,024	3,072M	55.88
16,384	12,288	4,096	1,288M	223.51

using the staging engine and via the shared file system when using the file-based engine. Moreover, there are not enough resources on the first cluster to execute the largest case with 16,384 actors in the off-node scenario. As the simulated time is deterministic, we run each simulation scenarios only once. Fig. 10 shows the obtained results.

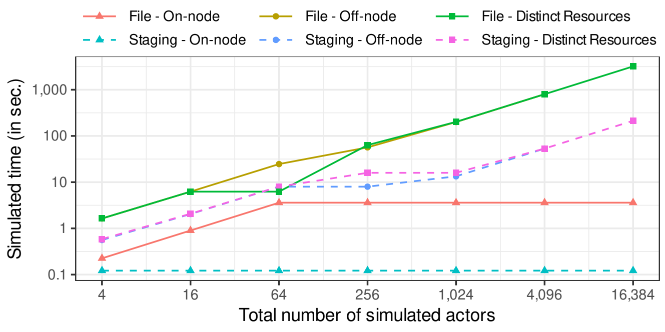


Fig. 10. Simulated time to transport grid and particles between one producer and one consumer applications using either a File-based or Staging engine and different in situ configurations, in a weak scaling regime.

In the *Staging – On-node* scenario, the ratio between publishers and subscribers and the size of the data published by an actor remain the same as we increase the scale of the benchmark. The simulated time is thus negligible and corresponds to the memory copy of 57 MB of data.

Simulations also capture that, in the *File – On-node* scenario, the transport time does not increase anymore once nodes are fully used (i.e., running 48 publishers and 16 subscribers). From that point the amount of data locally written to and read from the scratch space of a node remains constant.

We observe a similar behavior in the *Staging – Off-node* and *Staging – Distinct Resources* scenarios. However, as the number of nodes increases, more inter-node communications are needed to transport data, thus creating network contention and causing a further increase of the simulated time.

Finally, the *File – Off-node* and *File – Distinct Resources* scenarios lead to similar performance once more than one node is involved. However, the configuration with 64 actors corresponds to a sweet spot for the *File – Distinct Resources* with enough actors to fully use the nodes and less contention in the first cluster to access the share storage as the subscribers retrieve data from the second cluster.

D. Versatility Study – Simulation-Analysis In Situ Workflow

To illustrate the versatility of the DTL and its capacity to simulate real-world simulation-analysis in situ workflows, we consider the Gray-Scott application used in the ADIOS tutorials [9]. This reaction-diffusion model is a 3D 7-point stencil code that models the $U+2V \longrightarrow 3V$ chemical reaction, where U and V are two chemical components in a solution. This reaction consumes U and produces V and both substances diffuse over time in the solution.

The actors that execute the Gray-Scott iterative numerical simulation are organized in a 3D-grid. At each time step, they compute the current concentrations of the two chemical components, store them into two uniformly distributed 3D variables and exchange the boundaries of the data region they own with their neighbors. The current states of these variables are published to the DTL every 10 time steps for analysis.

The analysis consists in a parallel application in which each actor retrieves 2D slices of the variables U and V along the Z-dimension from the DTL. For each slice, it computes a probability density function (PDF) to determine the concentration of each chemical component in the solution in every point. It then creates and publishes two new 2D variables to the DTL. The first dimension has as many elements as the actor has slices of the 3D variables and the second dimension stores the density values in each of the bins used to compute the PDF.

The two components of this simulation-analysis in situ workflow being MPI applications, they can be compiled and run in a simulated mode with SimGrid without any modification. To integrate our simulated DTL, we had to replace the calls to ADIOS by equivalent calls to the DTL. These calls are isolated from the core of the Gray-Scott simulation and PDF calculation analysis, as it is usually the case in in situ workflows that rely on a data transport layer. Moreover, thanks to the semantic proximity between the DTL and ADIOS programming interfaces (i.e., same Variable and Engine abstractions, Stream vs. IO, Transaction vs. Step), these modifications are straightforward and amount for 0.5% of the codes composing the workflow. We believe that integrating the DTL into existing in situ workflows already using a data transport layer should require minimal development effort to enable a simulated execution.

The only modifications made to the non I/O related part of the Gray-Scott application code were to drastically reduce the simulation time and the memory footprint of the simulation. To this end, we leveraged two features offered by the SimGrid's SMPI programming interface. We introduced a call to the `SMPI_SAMPLE_LOCAL` macro in each of the three nested loops at the core of the application to go over all the elements in variables U and V. This macro makes each simulated actor sample the time taken to emulate the execution of the compute kernel captured by the macro for a predefined number of iterations and then replace this kernel by a simulated delay equal to the average of the sampled times. We opted for the sampling of $log_2(local\_dim)$ iterations along each dimension with no significant effect observed on the simulated time.

TABLE III
PARAMETERS OF THE ADIOS VERSATILITY EXPERIMENT.

#Compute nodes	1	2	4	8	16
#Publishers	64	128	256	512	1,024
Pub. distribution	4^3	8×4^2	$8^2 \times 4$	8^3	16×8^2
GB / Pub. / Trans.	4	2	1	0.5	0.25
#Subscribers	4	8	16	32	64

Additionally, we used the `SMPI_SHARED_MALLOC` macro introduced in Fig. 5 to have all actors allocating a single shared buffer instead of allocating a distinct buffer each.

We analyze the respective performance of six in situ scenarios in a strong scaling regime. We simulate the execution of a hundred time steps of the Gray-Scott application and of ten PDF calculations for a $256 \times 256 \times 256$ domain. Every ten time steps, the DTL transports 256 GB of data to be analyzed for a total of about 2.5 TB of data published over the entire lifecycle of the simulation. Table III describes five configurations with 1 to 16 compute nodes and 64 publishers and four subscribers per node. Fig. 11 shows the obtained simulated execution times. For each scenario, the black line denotes the simulated time achieved when using a staging engine with the transport method that uses SimGrid's message queues. This allows us to distinguish the time taken by the simulation of the Gray-Scott application and the analysis from the time spent to transport data between these two workflow components. The analysis component still relies on a file-based engine to export its outputs, as there is no consumer of these data in this workflow.

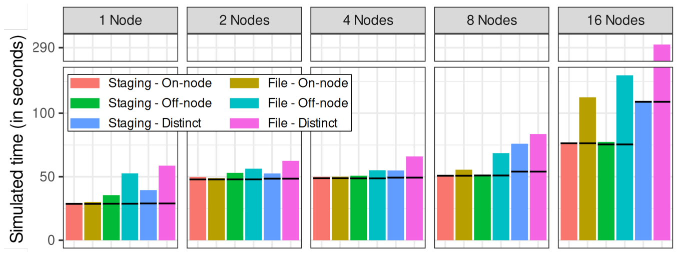


Fig. 11. Simulated time of the execution of the Gray-Scott simulation-analysis in situ workflow using either a File-based or Staging engine and different in situ configurations, in a strong scaling regime. Black lines denote the simulated time achieved when using a staging engine with the transport method that uses SimGrid's message queues.

The best performance is achieved when both components of the workflow run on a single node. In the *On-node* scenario, the overhead of data transport is negligible as the staging engine directly transports data from memory-to-memory while the file-based engine leverages the fast local storage. In the *Off-node* scenario, the slight cost of additional intra-node communications between the simulation and the analysis is reflected in the simulated execution time of the staging engine. Similarly, the simulation captures the extra cost of using the remote shared storage instead of the local scratch space by the file-based engine. Finally, the *distinct* scenario reflects the additional cost of moving data between compute clusters.

For all scenarios, the overall execution time increases as soon as the Gray-Scott application runs on more than one node, reflecting the effect of the transition from intra-node to inter-node MPI communications. From two to eight nodes, the simulated execution times for the on-node scenario and the off-node scenario with the staging engine do not increase as the cost of internal MPI communications of the Gray-Scott application dominates while the overall amount of data exchanged among actors does not decrease with the number of actors. When we run the analysis on the second compute cluster, we observe the increasing effect of network contention.

When we deploy the in situ workflow on 16 nodes, we observe a global performance degradation as MPI communications become limited by the crossbar switch backplane for all scenarios. In the distinct resources scenario the bandwidth of the network link between the two clusters also becomes a bottleneck and amplifies this performance degradation.

Finally, we observe longer simulated times when using a filed-based engine and the shared remote file-system (i.e., in the off-node and distinct resources scenarios), due to the additional cost of I/O, with an increasing performance gap with the staging engine as we increase the number of nodes.

To conclude this evaluation of the performance of the DTL when it simulates a real-world simulation-analysis in situ workflow that involves two MPI parallel applications and large amounts of data to transport, we analyze the time to execute such a simulation. Fig. 12 shows that we can simulated the largest execution scenario with 16 nodes that folds and emulates the execution of more than a thousand MPI ranks on a single core, in less than three minutes. This further shows that simulation-based performance assessments can help to the design and execution planning of in situ workflows.

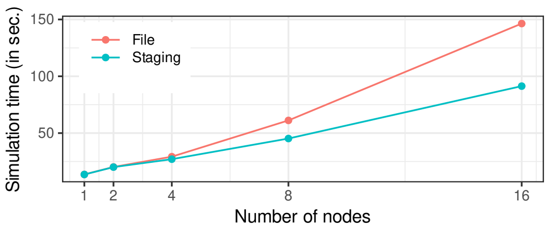


Fig. 12. Time to simulate the Gray-Scott simulation-analysis in situ workflow when increasing the number of nodes, and thus the number of simulated actors.

E. Accuracy Study – Real vs. Simulated Executions of a Simulation-Analysis In Situ Workflow

This paper mainly focuses on showing how one can implement simulators of in situ processing workflows that rely on a data transport layer with moderate development effort and how they can capture the discrepancies between different execution scenarios at scale. However, a typical concern with simulation is accuracy, i.e., how representative simulated executions are of real-world executions. Maximizing the accuracy of a simulator usually requires a careful calibration of the simulation models' parameter values [30] (e.g., computing speed, network and disk bandwidth) that we leave out of the scope of this paper.

To show that simulators using the DTL can reproduce the performance behavior of in situ processing workflows, we compare real and simulated executions of a small instance of the Gray-Scott in situ workflow. We execute the Gray-Scott application on 16 cores of an Intel Xeon Platinum 8380 at 2.30 GHz CPU and the PDF calculation on two cores of the same node. The real execution uses ADIOS (v2.9.1) and streams data between workflow components. The simulated execution uses the DTL with a staging engine.

Fig. 13 shows the Gantt charts of the real (top) and simulated (bottom) executions. In each figure, the black parts correspond to the time spent in the DTL. In this configuration the data consumer is waiting for the data producer. We can see that the simulation does not only matches the actual overall execution time, but also those on the ten individual steps and the interactions with the data transport layer. We computed the relative error between actual and simulated time by averaging the duration of individual steps across ranks first and then distinguishing the Gray-Scott simulation, the PDF calculation, and the time spent in the DTL by the analysis processes (this time is negligible in on the publishing side). These relative errors are 2.44%, 8.48%, and 15.96% respectively.

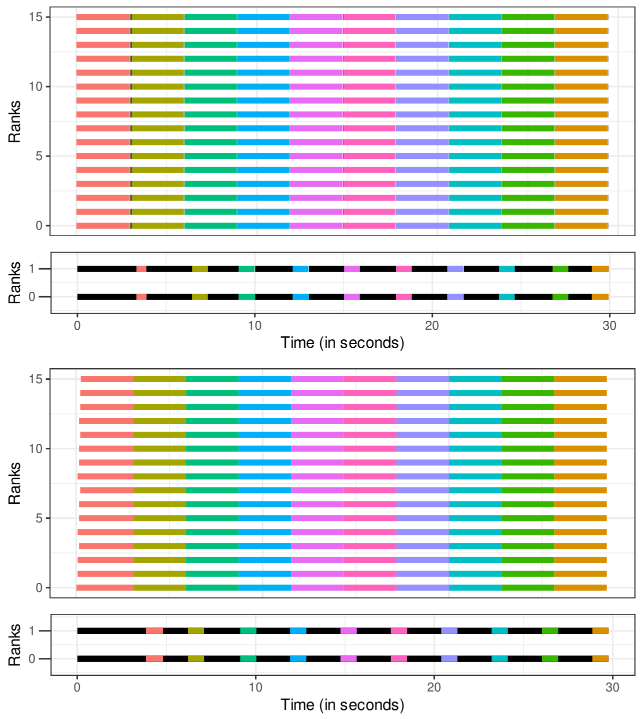


Fig. 13. Gantt chart of a real (top) and simulated (bottom) executions of ten steps the Gray-Scott simulation-analysis in situ workflow using ADIOS. Sixteen MPI ranks solve the Gray-Scott simulation and two ranks compute PDFs on 2D-slices. Each color represents a step, while black parts show the time spent in the DTL.

This experiment, while simple, shows that SimGrid/SMPI accurately captures the performance behavior or real-world MPI application while the simulated DTL ensure the respect of flow dependencies within in situ workflows and can simulate data exchanges as enabled by frameworks such as ADIOS.

V. Conclusion and Future Work

The capacity to analyze the data generated by large-scale applications as it is produced that is offered to domain scientists by in situ processing frameworks, and data transport layers in particular, presents many advantages in terms of dynamic control of the application and reduction of the time to science results. However, these frameworks leave scientists with the challenging task to determine what is the most efficient execution scheme of their in situ processing workflows and do not provide them with the necessary tools to assess the respective performance of numerous configuration and resource allocation options. To address these issues, we introduced in this paper an open-source versatile simulated data transport layer to enable the development of simulators of in situ processing workflow executions and the scalable evaluation of multiple execution scenarios. Our main design guideline was to combine the low-level abstractions of SimGrid to expose higher level concepts and API, as easy to program as actual DTL frameworks, hence lowering the entry barriers to developing simulators of in situ processing. The simulated DTL also implements and hides to users all the complexity of handling dynamic actor connections/disconnections, concurrent accesses to shared structures, complex data redistribution operations, and the conversion of simple put/get operations into complex I/O patterns or data redistribution.

We highlighted the main characteristics of this simulated data transport layer. First, it has the its capacity to capture the main features of existing data transport frameworks: It adopts a publish/subscribe paradigm, relies on self-descriptive data, and exposes multiple data transport techniques. Second, our experiments demonstrated its scalability: It can simulate the interactions through the DTL of tens of thousands of simulated processes deployed on two commodity clusters in only a few seconds). Third, we showed its versatility: It can simulate data producer and consumer applications at different levels of abstraction, ranging from very abstract, easy-to-develop simulators to the combination of full-fledged MPI applications in an in situ simulation-analysis workflow. Finally, we showed its capacity to accurately reflect the performance behavior of different design and deployment choices.

As future work, we will extend the simulation capabilities of the DTL by implementing optimization techniques in the engines: I/O operations can be aggregated at node level to reduce pressure on the file system; data reduction operators can be applied before publishing to the DTL; and buffers can be added to staging engines to mitigate differences in data production and consumption rates. We also plan to add support of fault tolerance mechanisms as SimGrid can simulate (transient) failures of compute nodes, network links, or disks. Finally, we will further assess the capacity of the simulators built on the DTL to provide better insights into different scheduling or resource allocation strategies. To this end, we will develop simulators for different use cases to obtain more results and further demonstrate the potential impact of the simulated DTL on the design of in situ processing workflows.

Acknowledgments

This research is partially supported by U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number "ERKJ414 – Resilient Federated Workflows in a Heterogeneous Computing Environment" and Laboratory Directed Research and Development Strategic Hire funding No. 11134 from Oak Ridge National Laboratory, provided by the Director. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

[1] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report," *Computer Graphics Forum*, vol. 35, no. 3, pp. 577–597, Jun. 2016.

[2] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, and M. Day, "In Situ and In-transit Analysis of Cosmological Simulations," *Computational Astrophysics and Cosmology*, vol. 3, no. 4, 2016.

[3] M. Kim, T. Evans, S. Klasky, and D. Pugmire, "In Situ Visualization of Radiation Transport Geometry," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017, p. 7–11.

[4] S. Dutta, N. Klein, L. Tang, J. D. Wolfe, L. V. Roekel, J. J. Benedict, A. Biswas, E. Lawrence, and N. Urban, "In Situ Climate Modeling for Analyzing Extreme Weather Events," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2021, p. 18–23.

[5] T. M. A. Do, L. Pottier, R. Ferreira da Silva, S. Caíno-Lores, M. Taufer, and E. Deelman, "Accelerating Scientific Workflows on HPC Platforms with In Situ Processing," in *Proceedings of the IEEE/ACM 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022.

[6] E. Dart, J. Zurawski, C. Hawk, B. L. Brown, and I. Monga, "ESnet Requirements Review Program Through the IRI Lens: A Meta-Analysis of Workflow Patterns Across DOE Office of Science Programs," US Department of Energy (USDOE), Tech. Rep., 11 2023.

[7] R. Ferreira Da Silva, R. Badia, V. Bala, D. Bard, P.-T. Bremer, I. Buckley, S. Caino-Lores, K. Chard, C. Goble, S. Jha, D. S. Katz, D. Laney, M. Parashar, F. Suter, N. Tyler, T. Uram, I. Altintas *et al.*, "Workflows Community Summit 2022: A Roadmap Revolution," [Online] https://zenodo.org/record/7750670, 2023.

[8] 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," *International Journal of High Performance Computing and Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[9] W. 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.

[10] 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.

[11] T. Peterka, D. Morozov, A. Nigmetov, O. Yildiz, B. Nicolae, and P. Davis, "LowFive: In Situ Data Transport for High-Performance Workflows," in *Proc. of the 37th IEEE International Parallel and Distributed Processing Symposium*, 2023, pp. 985–995.

[12] A. Martinelli, M. Torquati, M. Aldinucci, I. Colonnelli, and B. Cantalupo, "CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows," in *Proc. of the 30th IEEE International Conference on High Performance Computing, Data, and Analytics*, 2023.

[13] C. Haine, U.-U. Haus, M. Martinasso, D. Pleiter, F. Tessier, D. Sarmany, S. Smart, T. Quintino, and A. Tate, "A Middleware Supporting Data Movement in Complex and Software-Defined Storage and Memory Architectures," in *Proc. of the Fourth International Workshop on Interoperability of Supercomputing and Cloud Technologies*, ser. Lecture Notes in Computer Science, vol. 12761. Springer, 2021, pp. 346–357.

[14] DTLMod, "Data Transport Layer Module for SimGrid," [Online] Available: https://github.com/simgrid/DTLMod/releases/tag/v0.1, 2025.

[15] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Lowering Entry Barriers to Developing Custom Simulators of Distributed Applications and Platforms with SimGrid," *Parallel Computing*, vol. 123, p. 103125, 2025.

[16] ——, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899 – 2917, 2014.

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

[18] "SimGrid's Use in Research Publications," https://simgrid.org/usages.html, 2023.

[19] G. Kecskemeti, S. Ostermann, and R. Prodan, "Fostering Energy-Awareness in Simulations Behind Scientific Workflow Management Systems," in *Proc. of the 7th IEEE/ACM International Conference on Utility and Cloud Computing*, 2014, pp. 29–38.

[20] C. Docan, M. Parashar, and S. Klasky, "Enabling High-Speed Asynchronous Data Extraction and Transfer using DART," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 9, p. 1181–1204, 2010.

[21] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization," in *Proc. of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2015, p. 25–29.

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

[23] 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 Trans. on Modeling and Computer Simulation*, vol. 23, no. 4, 2013.

[24] T. Do, L. Pottier, S. Caíno-Lores, R. Ferreira da Silva, M. 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.

[25] V. Honoré, T. A. Do, L. Pottier, R. Ferreira da Silva, E. Deelman, and F. Suter, "Sim-Situ: A Framework for the Faithful Simulation of in situ Processing," in *Proc. of the 18th IEEE International eScience Conference*, Salt Lake City, UT, 2022.

[26] FSMod, "File System Module for SimGrid," [Online] Available: https://github.com/simgrid/file-system-module/releases/tag/v0.3, 2025.

[27] "SimGrid "Frankenstein" example simulator," [Online] Available: https://github.com/simgrid/simgrid_frankenstein, 2024.

[28] SimGrid v4.0, "The 'This one is 4 you' Release," [Online] Available: https://github.com/simgrid/simgrid/releases/tag/v4.0, 2025.

[29] F. Suter, "Experimental Artifact," [Online] Available: https://doi.org/10.6084/m9.figshare.28872509, 2025.

[30] J. McDonald, M. Horzela, F. Suter, and H. Casanova, "Automated Calibration of Parallel and Distributed Computing Simulators: A Case Study," in *Proc. of the 25th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2024, pp. 1026–1035.