Dynamic Heterogeneous Task Specification and Execution for In Situ Workflows

Orcun Yildiz,* Dmitriy Morozov,† Bogdan Nicolae,* and Tom Peterka*
*Argonne National Laboratory, IL 60439, USA, {oyildiz,bnicolae,tpeterka}@anl.gov
†Lawrence Berkeley National Laboratory, Berkeley, CA, dmorozov@lbl.gov

***Abstract*—Today's science campaigns consist of multiple tasks with wide-ranging data and computing requirements, and rarely are all the required capabilities found in current in situ workflow systems. In this work, we explore providing increased capabilities for scientific computing by bringing new capabilities to in situ workflows: a flexible interface for workflow specification, heterogeneous task placement, and dynamic changes to the workflow task graph. We evaluate our approach using materials science and cosmology use cases. Our results show that our approach (i) can save time and resources in science workflows exhibiting dynamic patterns by enabling dynamic workflow changes during their lifetime; (ii) enables easier specification of large-scale workflows consisting of subgraphs and ensemble computations; (iii) efficiently coordinates heterogeneous tasks by enabling free intermixing of time and space partitioning, thus resulting in time and space savings.**

***Index Terms*—HPC, In Situ Workflows, Dynamic Workflows, Heterogeneous Workflows**

I. Introduction

In recent years, we have witnessed increased complexity of scientific computing workflows coupled with a dramatically increasing gap between the computation and I/O capabilities of high-performance computing (HPC) systems. These trends make traditional manual postprocessing of scientific data generated by HPC applications infeasible. Automated in situ processing, i.e., in situ workflows, is the prominent alternative to the traditional post hoc approach, automating data analysis at simulation time while minimizing I/O.

One major capability lacking in current in situ workflow solutions is the ability to dynamically add and remove resources to accommodate changes in requirements of science applications [1]. For instance, new analysis tasks may need to be launched on-the-fly as materials simulations evolve because the discovery of a superconducting vortex may require additional analysis codes to track the feature [2].

Another required capability of in situ scientific workflows often lacking is the support for flexible intermixing of task placement and data transfer methods. Coupling of in situ tasks can take two different forms: (1) tasks executing sequentially on the same resource (called time partitioning) or (2) tasks executing concurrently on separate resources (called space partitioning). Both coupling types are used by the HPC community, with time-space tradeoffs between them. For example, users with a limited set of resources may opt for time partitioning, while users with strict performance requirements may choose space partitioning. Being able to switch seamlessly between coupling types is needed to support a variety of different science use cases. In particular, heterogeneity of different tasks in the same workflow can require mixing of time and space partitioning on a task-by-task basis.

Scientific workflows can be complex and large scale with ensembles consisting of multiple instances of repeated subgraphs of tasks. Hence, another required capability of in situ scientific workflows is to provide a flexible workflow specification interface to define such workflows. This interface should capture the global view of the entire workflow while being easy to use. Since workflow specification is closely tied to workflow execution, having a flexible interface can also facilitate efficient runtime execution. Also, the workflow interface needs to be scalable, handling small graphs of a few tasks such as a simulation coupled to a small number of analyses; to large graphs of high-throughput tasks such as an ensemble of subworkflows.

Motivated by these needs of today's science campaigns, in this work we present our approach to dynamic heterogeneous task specification and execution for in situ workflows. In order to address the above challenges, we combined some features from two existing in situ systems: Decaf [3] and Henson [4]; where Decaf is a parallel in situ dataflow system[1], and Henson is a cooperative multi-tasking system.[2] Our approach provides the following advantages:

- Flexible workflow specification interface to define complex scientific workflows.
- Support for both shared- and distributed-memory communication in concert with flexible intermixing of time- and space-partitioning task placement.
- Support for dynamic workflow changes depending on the requirements of the workflow tasks.

We evaluate our solution with two science use-cases. In the first case, motivated by materials science, we launch an ensemble of multiple molecular dynamics simulation instances to observe a rare nucleation event, requiring dynamic workflow changes driven by the detection of the event. The second problem is motivated by computational cosmology, where the workflow is a parameter space exploration consisting of four different tasks—synthetic particle generation, Voronoi tessellation, density estimation, and rendering—that are heterogeneous in their data and computation requirements. Our results

[1]https://github.com/tpeterka/decaf
[2]https://github.com/henson-insitu/henson

demonstrate that our approach provides new capabilities for in situ analysis such as flexible workflow interface, dynamic workflow changes, and heterogeneous task placement.

The remainder of this paper is organized as follows. Section II presents background and related work. Section III describes our methodology for increased capabilities for in situ scientific workflows. Section IV presents our experimental methodology, followed by experimental results in Section V. Section VI concludes the paper with a summary and a brief look at future work.

II. Background and Related Work

In recent years, in situ workflows have gained popularity in the HPC community. In situ workflows run within a single HPC system, and the data exchange is done through memory or the supercomputer interconnect during the scheduled execution of a job [5]. Here, we review related work in task placement, data communication, task scheduling, and dynamic aspects.

Because our focus is in situ processing, we present background and related work for in situ workflows, and we do not cover distributed or cloud-based workflows in this section. We refer the reader to several survey papers presenting taxonomies of representative distributed workflow systems [6], [7].

A. Task Placement

We can characterize in situ solutions according to the task placement mode that they support. For instance, Libsim [8] employs the time partitioning mode for performing in situ analysis and visualization for the VisIt visualization tool [9]. Similarly, Catalyst [10] exposes the simulation data in situ to ParaView [11], and the in situ processing happens on the same resources as the simulation. On the other hand, FlexPath [12] provides a publish/subscribe model to exchange data between parallel codes running on separate resources. DataSpaces [13] provides a distributed memory space for enabling space partitioning support, where workflow tasks can both push data into this space and retrieve from it.

Some approaches support both time and space partitioning. For example, ADIOS [14], originally designed for I/O staging, can now couple tasks through an I/O interface with different task placement modes. Users usually set the task placement mode in a configuration file. While we also support both time and space partitioning placement modes in a transparent way to the user, we support freely intermixing these different task placement modes in the same workflow.

Some works study the impact of the task placement mode efficiency. Kress et al. [15] compare the efficiency of time and space partitioning modes for scientific visualization in terms of time to solution and total cost. They indicate that none of these task placement methods is superior in all scenarios, and the best choice depends on many factors such as type and scalability of the visualization algorithms.

B. Data Communication Mechanisms

In situ workflows run within a single HPC system, and communicate over shared memory or through the interconnect of the machine. VisIt's Libsim and Paraview's Catalyst libraries support shared-memory communication between analysis and visualization tasks running synchronously with the simulation, in the same address space. Damaris/Viz [16] splits the MPI communicator of the simulation to allocate dedicated cores for performing visualization tasks. Messages between simulation and visualization tasks are allocated in a shared-memory buffer. On the other hand, Decaf [3] is a middleware for coupling parallel tasks in situ by creating communication channels over HPC interconnects through MPI. Some in situ solutions offload the data to a distributed memory space that is shared among multiple workflow tasks. DataSpaces and FlexPath implement such a communication mechanism using a publisher/subscriber model.

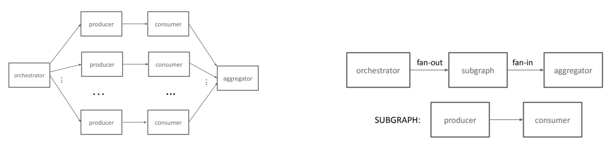
Some systems can support several data communication mechanisms by employing in situ tools with different data exchange methods. SENSEI [17], an in situ system designed with tool portability in mind, allows for multiple communication methods by interfacing with different in situ tools. While we also support several communication mechanisms including shared- and distributed-memory communication, our approach is generic and is not tailored for a specific category of applications, such as visualization.

C. Task Scheduling

The majority of HPC in situ tools schedule workflow tasks onto resources by relying on a static configuration file for the workflow specification. The static nature of task scheduling in these tools makes it difficult to support complex scientific workflows with task graphs that evolve over time. One exception is Henson [4], [18], a cooperative multitasking system for in situ processing that uses shared objects and coroutines as its main abstractions, together with a built-in scripting language. Through these features, Henson provides extra flexibility in scheduling the workflow tasks. Workflows can also be specified using a programming language, such as Swift/T [19], which schedules tasks based on the data dependencies in the program. Swift/T can support complex workflows; however, the user code has to be organized and compiled into Swift modules.

D. Dynamic Workflow Changes

This aspect of in situ workflows remains largely unexplored, with few research efforts. One recent work is the Flexpath publish/subscribe system, which can accommodate analysis task arrivals/departures. However, the experiments in Dayal et al. [12] demonstrate only static 2- or 3-task linear pipelines with a fixed number of MPI ranks per task. Melissa [20] is a parallel client/server architecture for the analysis of ensembles, where independent simulation groups can connect dynamically to the parallel server when they start. This system is limited to changing the number of simulation instances, not performing generic changes to the workflow task graph



(a) End-to-end workflow (b) Simplified view of an end-to-end workflow

Fig. 1. Ensemble of workflow graphs with normal and simplified views.

```
import networkx as nx

# Subgraph definition
subgraph = nx.DiGraph()
subgraph.add_node("producer", start_proc=0, nprocs=4, ...)
subgraph.add_node("consumer", start_proc=4, nprocs=1, ...)
subgraph.add_edge("producer", "consumer", ...)

N = 3 #multiplicity of the subgraph

# End-to-end workflow definition
w = nx.DiGraph()
w.add_node("orchestrator", start_proc=0, nprocs=1, ...)
w.sub_graph(subgraph, N)
w.add_node("aggregator", start_proc=16, nprocs=1, ...)

# Fan-out and fan-in declaration between
# multiple subworkflows and orchestrator/aggregator tasks
w.fan_out(orchestrator, subgraph, N, ...)
w.fan_in(subgraph, aggregator, N, ...)

wf.processGraph("workflow.json")
```

Listing 1. Snippet of workflow configuration file defining an ensemble of workflows.

such as adding/deleting other tasks. One way of bringing dynamic features to in situ workflows is to combine them with distributed-area workflows that support dynamic workflow changes. Recent work [21] employs Decaf workflows as single tasks of a PyCOMPSs distributed workflow [22], extending Decaf in situ workflows with dynamic features. Being a distributed workflow designed for clouds and grids, however, PyCOMPSs has performance limitations such as data exchange through files and serial completion of dependent jobs, which we want to avoid when executing in situ on HPC platforms.

III. Methodology

In this section, we discuss how our approach addresses these challenges: workflow graph description, heterogeneous task placement, and workflow dynamics.

A. *Workflow Graph Description*

The first challenge is how to specify workflows, in particular complex and large-scale ones. Our approach borrows Decaf's Python API, where users can define the different nodes and edges of a workflow task graph and the number of processes assigned to each workflow task. This workflow specification is done prior to the launch of the workflow, and read upon initialization. In Section III-C, we will see how we can modify this workflow graph at runtime to support workflows with task graphs that evolve over time.

Specifying large-scale workflows such as an ensemble of tasks shown in Figure 1(a) is a challenge when describing the graph, because such workflows have too many tasks to list explicitly in a workflow configuration file. Also, users may want to vary the number of the ensemble instances, making it desirable to represent each instance as a single entity that is repeated some number of times. In our Python API, we introduce a subgraph feature to represent each ensemble instance, and to describe the end-to-end workflow as a composition of these subgraphs with other tasks in the workflow.

Figure 1(b) shows this simplified view of an end-to-end workflow graph that the user specifies in a Python script, rather than explicitly describing all the tasks and edges in Figure 1(a). Listing 1 is a snippet of workflow configuration file defining an ensemble of workflows using this simplified view. The user first defines the subgraph by indicating its nodes, edges, and resource requirements. Second, the user indicates the number of ensemble instances. Third, the user describes the end-to-end workflow graph, by defining other tasks in this workflow, and declaring the communication channels between ensemble instances and these tasks (e.g., fan-out, fan-in). For instance, the orchestrator task can send different configuration parameters to multiple instances (fan-out), and the aggregator task can collect information from these instances (fan-in). Then, we generate the end-to-end workflow specification based on this simplified view.

B. *Heterogeneous Task Placement*

Scientific workflows usually consist of tasks that are heterogeneous in their data and computation requirements. Such tasks can favor different task placement modes (time or space partitioning) depending on their requirements; hence, there is a need for flexibility in task placement. We support both time and space partitioning modes, and can freely intermix these different task placement modes transparently in the same workflow.

We automatically detect the task placement mode based on the location of the tasks, described in the workflow configuration file. Specifically, users provide the starting process rank of the task together with its name, number of processes, and its arguments when defining the task in the configuration file. Our approach uses the starting process rank to detect the placement mode of the tasks: time partitioning when the coupled tasks have the same starting process, and space partitioning when the tasks have a different starting process.

Our approach uses the most direct communication mechanism among the coupled tasks: shared memory when time partitioned, and message passing when space partitioned.

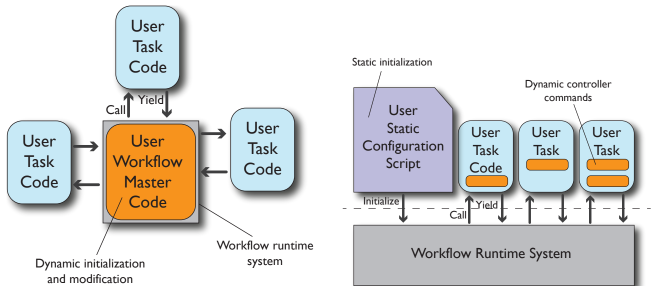
For shared-memory communication, we inherit Henson's execution model, where individual codes are compiled as shared objects. This allows us to load multiple position-independent shared objects in the same address space, letting them access each others' memory directly. Therefore, tasks can exchange data by simply passing pointers to each other, with zero copies and no changes to their code.

For distributed-memory communication, we create parallel communication channels over MPI, an association of a producer, a consumer, and a communication object to exchange data between the producer and the consumer. Producers and consumers are parallel programs, each with their own MPI communicator. We create an additional communicator between the producer and the consumer for the data exchange. Messages are passed in parallel, point-to-point from producer ranks to consumer ranks, without aggregating at the root of either the producer or the consumer, and redistributing data between different numbers of producer/consumer processes.

C. Dynamic Workflow Changes

Scientific workflows may exhibit complex dynamic patterns, requiring the capability to perform workflow changes at run time. Such workflow changes may include adding and deleting tasks and scheduling the placement of workflow tasks with respect to time and space partitioning and shared or distributed memory. To support such dynamic workflow changes, we designed a *controller* module.

A controller module is programmable by the user just like any other workflow task, and it can be either a separate task or incorporated into existing workflow tasks. The controller provides a simple API to reconfigure the workflow depending on the changes in the requirements of the workflow tasks.



(a) Global-level API (b) Task-level controller API

Fig. 2. Illustration of two possible ways of supporting dynamic workflow changes.

We investigated two options in designing the API for the controller: global- or task-level. Figure 2 highlights the difference between these options. The first one is a global-level API that enables defining the workflow programmatically at the workflow specification level (i.e., by programming the workflow master code). With the global-level API, the workflow engine is exposed to the user via the workflow master code, giving the user maximum flexibility for dynamic workflow changes. However, we decided that this also introduces additional complexity for users to manage the control logic for such changes. In particular, this control logic involves defining the interactions between workflow tasks and the runtime engine at the global level when the workflow is reconfigured. Examples of such interactions include checkpointing or restarting workflow tasks, and reconfiguring communication links between workflow tasks.

The second solution, that we ultimately elected, is a task-level API implemented as a controller module. Here, the workflow is initially defined by a static Python script and later modified via the API of the controller. We opted for the controller API since it hides the control logic from the user by providing a set of predefined commands (e.g., *changeTask*, *addTask*) for managing interactions between workflow tasks and the runtime engine.

The controller API enables three different dynamic workflow changes: i) looping until the desired solution is reached, ii) redistribution of MPI ranks among the tasks of the workflow graph, and iii) adding or deleting tasks to the workflow graph. Workflow tasks are initially launched based on the static workflow configuration file, as described in Section III-A. Upon a modification request to the workflow graph, the controller first checks whether these changes leave the workflow in a valid state (e.g., correct total number of processes is maintained). If the request is valid, the controller then broadcasts the new workflow graph to the other workflow tasks. Workflow execution then continues with this new workflow graph. This control logic continues until the user issues a *shutdown* call via the controller API.

IV. Experiments

Our experiments were conducted on the Bebop cluster at Argonne National Laboratory, which has 1,024 computing nodes. We employed nodes belonging to the Broadwell partition. The nodes in this partition are outfitted with 36-core Intel Xeon E5-2695v4 CPUs and 128 GB of DDR4 RAM. All nodes are connected to each other by an Intel Omni-Path interconnection network.

A. Science Use Cases

1) Materials Science: The materials science problem we study is nucleation as a material cools and crystallizes; in this case water freezing, but the same workflow applies to nucleation in many other material systems. Understanding the mechanisms and kinetics of crystallization is key to understanding a wide range of natural and technological systems. Nucleation, however, is a stochastic event that requires a large number of molecules to elucidate its kinetics. Capturing nucleation in simulations is difficult, especially during the early stages when only a few atoms have crystallized.

One way to simulate nucleation is to run many instances of small simulations. Given its stochastic nature, nucleation may be observed in only a few of these simulation instances. Manually managing such ensemble workflows can be burdensome for scientists who need to run and analyze many simulation instances. Our approach allows us to fully automate this science pipeline, where scientists can define the task graph using our subgraph API, and use the controller API to steer the workflow based on the results thus far. The result is a dynamic workflow graph, with as many instances of simulation and analysis tasks as are needed to detect a rare nucleation event.

Figure 3 shows this workflow graph, where each simulation instance consists of the LAMMPS [23] molecular dynamics model coupled to a parallel diamond structure detector developed in [21]. Detection of nucleated atoms requires identifying the different phases of ice: hexagonal, cubic, and amorphous/liquid. The controller task checks whether one of the simulation instances exceeded the required minimum number of nucleated atoms, as detected by the diamond

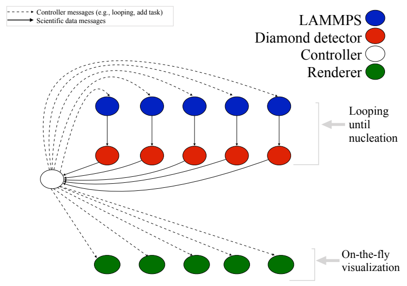


Fig. 3. Dynamic workflow graph driven by the controller, where looping is continued until there is a successful nucleation event or a maximum number of iterations is reached.

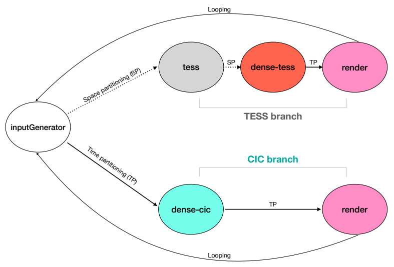


Fig. 4. Workflow graph for density estimation in gravitational lensing with two different density estimators.

structure detector. Looping continues until there is a successful nucleation event or a maximum number of iterations is reached. When the number of nucleated atoms exceeds a user-specified threshold, the controller launches the renderer tasks in order to visualize the nucleation. The renderer tasks are launched only when there are enough nucleated atoms. Each of the LAMMPS and diamond detector tasks in Figure 3 are parallel MPI programs written in C++, while the controller task is a serial C++ program. Each of the visualization tasks are serial Python programs.

2) Cosmology: The second use case is motivated by cosmology; in particular, density estimation in gravitational lensing for investigating dark matter. Density estimation is a transformation from discrete particle data to a continuous density function defined over a 3D or 2D field. In this use case, we consider two different density estimators: cloud in cell (CIC) and tessellation (TESS) [24]. CIC estimates the density directly from the particle data while TESS first performs a Voronoi tessellation before computing the density. The latter results in more accurate density estimate, but comes at a higher cost due to the expensive computation of Voronoi tessellations.

Different estimators with cost-accuracy tradeoffs raise the question of which one to use when computing lensing simulations, given problem size and computational budget [24]. We automated the decision-making process by performing a parameter sweep for different problem configurations (number of particles, cutoff constant) using different density estimators.

Figure 4 shows this workflow graph for density estimation with two different branches, one for each density estimator,

```
import networkx as nx
w = nx.DiGraph()

# Task declaration
w.add_node("inputGenerator", start_proc=0,  nprocs=1, ...)
w.add_node("dense-cic", start_proc=0,  nprocs=1, ...)
w.add_node("tess", start_proc=1, nprocs=4, ...)

# Dataflow declaration
w.add_edge("inputGenerator", "dense-cic", ...)
w.add_edge("inputGenerator", "tess", ...)

wf.processGraph("workflow.json")
```

Listing 2. Snippet of workflow configuration file with two different placement modes.

CIC and TESS. The *inputGenerator* task generates a synthetic dataset to mimic characteristics of actual data in computational cosmology simulations. The tessellation and density estimator (*tess*, *dense-cic* and *dense-tess*) codes are built on the Tess library [25], which in turn is built on the DIY data-parallel programming model [26]. The *render* task generates a 2D density image by projecting from a 3D density volume. In this parameter sweep, the tessellation and density estimator codes are parallel MPI C++ programs, while the *inputGenerator* and the *render* tasks are serial Python programs.

One advantage of our approach is that workflow tasks are agnostic to their placement mode, since we can automatically detect the task placement mode based on the workflow configuration file. Listing 2 presents a sample workflow configuration generating different placement modes for different density estimators. For instance, we detect the coupling mode of *inputGenerator* and *dense-cic* as time partitioning since they have the same starting process rank, creating a shared-memory communication channel between these tasks. On the other hand, a distributed-memory communication channel is created between *inputGenerator* and *tess* tasks because they are running on separate resources. This intermixing of task placement requires no change in the task code; the user simply modifies the workflow configuration file, which generates the workflow graph.

V. Results

A. Dynamic Workflow Changes

To evaluate our capability to enable dynamic workflow changes, we studied the nucleation problem, where we ran multiple instances of simulation and analysis, as shown in Figure 3. Here, we show the benefits of reallocating resources (MPI ranks) between simulation and analysis tasks depending on the simulation behavior.

Redistributing resources among the workflow tasks: Scientific workflows display dynamic behaviors during the course of their execution, and resources may need to be redistributed among the workflow tasks in response to these changes. When simulating nucleation, only a few atoms crystallize for much of the simulation's run time; hence, the analysis of early time steps requires less computation than the later ones. Ideally, we would like to reallocate resources between simulation and analysis tasks once the simulation starts producing interesting results such as nucleation.

To investigate the benefits of resource reallocation, we compared the workflow completion time and overall resource usage of the nucleation problem with static allocation versus dynamic reallocation. We ran 5 instances of LAMMPS and the diamond detector concurrently. In the static case, each LAMMPS instance used 28 MPI processes, and each detector used 7 processes. For dynamic reallocation, we start with 34 processes for LAMMPS and one process for the diamond detector; and we reallocate resources to 28 LAMMPS and 7 diamond detector processes once number of nucleated atoms exceeds a desired threshold.[3]

We used a controller module as a separate task, and dedicated one process to the controller for managing the control of the workflow and triggering dynamic reallocation based on the simulation behavior.

We ran LAMMPS for 5,000,000 time steps with a water model composed of 4,360 atoms, and we performed the diamond structure analysis every 10,000 iterations. Table I shows the workflow completion time and overall resource usage for static and dynamic allocation of workflow tasks. With dynamic allocation, MPI processes are adjusted at time step 4,000,000, with LAMMPS reducing from 34 to 28 processes, and the diamond detector increasing from 1 to 7 processes. The results demonstrate that dynamic reallocation saves 10% time and resources. This dynamic resource reallocation does not require any changes to the workflow task codes, and we rely on the checkpoint and restart ability of LAMMPS when performing the resource reallocation. For simulations without such an ability, existing checkpoint tools such as VeloC [27] can be used for checkpoint and restart.

Allocation scheme	**Workflow completion time**	**Resource requirements**
Static allocation	4,650 seconds	226.0 core-hours
Dynamic reallocation	4,170 seconds	203.9 core-hours

TABLE I
WORKFLOW COMPLETION TIME AND OVERALL RESOURCE USAGE FOR STATIC AND DYNAMIC ALLOCATION OF WORKFLOW TASKS WHEN STUDYING NUCLEATION.

To further investigate the simulation behavior under these different allocation schemes, we plot in Figure 5 the LAMMPS throughput (timesteps per second) during its different phases. We note that the LAMMPS throughput is lower in the beginning for both approaches due to the several initialization steps such as energy minimization of the system. We observe that dynamic allocation outperforms the static allocation on average by 15% until resource reallocation happens at time step 4,000,000 (at 80% of the simulation lifetime). The benefits of dynamic reallocation depends on several factors such as the time when the workflow is reconfigured and the amount of resources that are redistributed among the workflow tasks. Next, we perform two sets of experiments to highlight the impact of these factors.

[3]We performed a profiling study to find the optimal number of processes for the diamond detector with two different behaviors of the simulation: before and during nucleation.

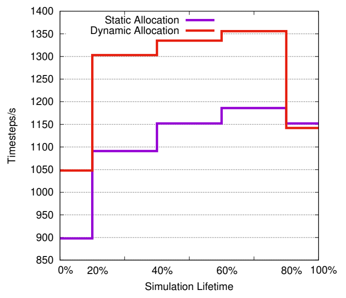


Fig. 5. LAMMPS throughput during different phases for static and dynamic allocation schemes.

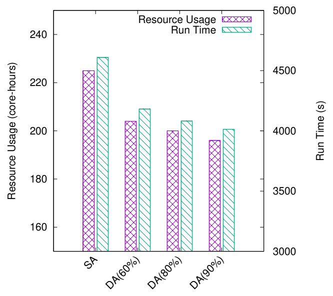


Fig. 6. Resource usage and completion time of static and dynamic allocation, with resource reallocation happening at different times during the simulation.

a) Impact of workflow reconfiguration time: We study the relationship between the dynamic allocation and the time when the workflow is reconfigured. Instead of performing the resource reallocation once the number of nucleated atoms exceeds a desired threshold, we configured the controller module to trigger the resource reallocation at three different times: 60%, 80%, and 90% of the simulation lifetime.

Figure 6 shows the resource usage and workflow completion time of static and dynamic allocation, with resource reallocation happening at different times during the simulation. As expected, we see that the resource usage and workflow completion time decreases if the workflow is reconfigured later in simulation's lifetime. For example, dynamic reallocation at 90% of the simulation lifetime saves 5% more time and resources compared with reallocating at 60% of the simulation lifetime. This is because nucleation happens relatively late in the simulation lifetime, highlighting the fact that dynamic reallocation can have larger benefits when interesting phenomena (e.g., nucleation) occur in only a small duration of the simulation lifetime.

b) Impact of resource requirements of the workflow tasks: The optimal amount of resources (re)allocated to workflow tasks depends on the resource requirements of each task. To study the impact of the resource requirements of the workflow tasks on the dynamic reallocation, we set two different resource distributions between LAMMPS and the diamond detector. In the first scenario, we assigned 17 processes to both LAMMPS and the diamond detector statically (SA(1:1)). In the second scenario, we assigned 28 processes to LAMMPS, and 7 processes to the diamond detector statically (SA(4:1)). For dynamic reallocation, we started with 34 processes for

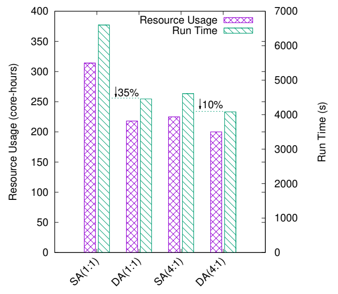


Fig. 7. Resource usage and completion time of static and dynamic allocation, with different resource distribution between LAMMPS and the diamond detector.

LAMMPS and one process for the diamond detector; and we reallocated resources either evenly (DA(1:1)) or to 28 LAMMPS and 7 diamond detector processes (DA(4:1)) once number of nucleated atoms exceeds a desired threshold.

Figure 7 shows the resource usage and workflow completion times for static and dynamic allocation under these scenarios. We observe that the benefit of dynamic reallocation is larger when a higher amount of resources is redistributed between LAMMPS and the diamond detector. For example, when these tasks have the same resource requirements, DA(1:1) saves 35% time and resources compared with the SA(1:1). On the other hand, these savings are 10% when DA(4:1) compared with SA(4:1). This is because more resources are reallocated when analysis tasks have higher resource requirements, making dynamic reallocation more beneficial.

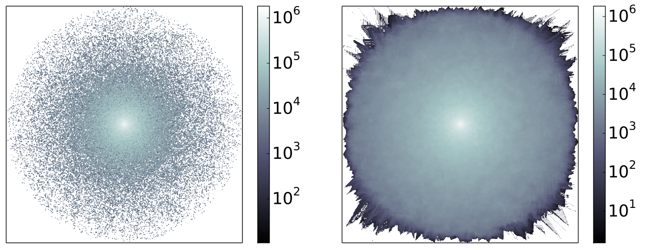
B. Free Mixing of Time and Space Partitioning

We performed a parameter sweep for the density estimation with TESS and CIC branches by varying the following parameters: number of particles and the cutoff constant. We used $10^4, 10^5$, and 10^6 particles estimated onto a 512x512 output grid, and we employed 1, 10, and 100 as the cutoff constant. These parameter ranges resulted in 9 different parameter sweep instances for each density estimator. Figure 8 shows the density image for CIC and TESS density estimators in one of these parameter sweep instances. As expected, we see that TESS achieves better accuracy compared with CIC, but this comes with 5 times higher performance cost. By automating the parameter exploration, our approach allows users to determine the appropriate density estimator depending on their needs and computational budget.

Approach	End-to-end execution time	Storage requirements
Our approach	435 seconds	3 MB
Traditional (post hoc)	725 seconds	1.2 GB

TABLE II
TIME AND STORAGE TAKEN BY DIFFERENT APPROACHES FOR RUNNING THE PARAMETER SWEEP.

Comparison with the traditional (post hoc) approach: Table II shows that automating the parameter sweep saves time and storage, compared with the traditional way of manually running each workflow task in sequence. The tessellation task has the longest execution time, hence, it dominates the parameter sweep. We reduced the end-to-end execution time 40% by running the density estimation concurrently with tessellation. We obtained further time savings due to running the CIC branch in parallel with the TESS branch. Additional savings can be gained by employing more branches in the parameter sweep, for different density estimators, or for different combinations of parameters.



(a) Cloud in cell (CIC) (b) Tessellation (TESS)

Fig. 8. Density images computed from 10^6 particles estimated onto a 512x512 output grid with two different density estimators.

Placement mode	End-to-end execution time	Tessellation time
Time partitioning	1170 seconds	540 seconds
Space partitioning	650 seconds	540 seconds

TABLE III
TIME TAKEN BY DIFFERENT PLACEMENT MODES FOR RUNNING THE TESS BRANCH.

Our approach also reduces the I/O time by communicating through shared memory or MPI instead of files between the workflow tasks. This results in significant storage savings, where we only need to store the rendered image files, requiring only 3 MB storage space. On the other hand, using files for communication requires storing the particle positions for density estimation. This occupies 1.2 GB of storage, 400 times higher than our approach.

Impact of the placement mode: Workflow tasks used in this parameter sweep have different requirements, in particular *tess* and *dense-cic*. *Tess* has two times higher memory footprint than *dense-cic*, and *dense-cic* has a much shorter execution time. Hence, *dense-cic* can be coupled in time partitioning mode, while the high time and memory cost of *tess* makes the space partitioning mode more suitable for it.

To highlight the impact of the placement mode on the TESS branch (upper part of the workflow graph in Figure 4), we measured the end-to-end execution time for a total of 10 runs with 10^6 particles estimated onto a 512x512 grid. Table III shows that coupling *tess* with *inputGenerator* and *dense-tess* in time partitioning is 45% slower compared with coupling it in space partitioning. This is because we can hide the times required for *inputGenerator* and *dense-tess* tasks by running *tess* asynchronously in space partitioning. By enabling free intermixing of time and space partitioning, our approach can efficiently coordinate these heterogeneous tasks in a single workflow.

VI. Conclusion

Today's scientific workflows are becoming increasingly complex with heterogeneous tasks and varying data and computation requirements. In this work, we demonstrated that we can support complex science problems, where our approach brings new capabilities to in situ solutions such as flexible workflow definition, support for free mixing of time and space partitioning, and dynamic workflow changes.

In the future, we plan to extend our support for dynamic workflow changes. First, we will explore the integration of external inputs (e.g., human- or AI-in-the-loop) to the workflow system for steering the workflow at run time. A second direction that we will investigate involves global workflow resource changes, which would require resizing the entire workflow, rather than reallocating a fixed set of resources among the tasks. One might want to upscale or downscale the total workflow resources depending on several factors, such as system failures or workload changes. Here, checkpointing tools can enable resizing the workflow, and coordination of such tools with in situ workflow systems is another possible direction of future research.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contracts number DE-AC02-06CH11357 and DE-AC02-05CH11231. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] M. Dorier, O. Yildiz, T. Peterka, and R. Ross, "The challenges of elastic in situ analysis and visualization," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2019, pp. 23–28.

[2] H. Guo, T. Peterka, and A. Glatz, "In situ magnetic flux vortex visualization in time-dependent ginzburg-landau superconductor simulations," in *2017 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 2017, pp. 71–80.

[3] M. Dreher and T. Peterka, "Decaf: Decoupled dataflows for in situ high-performance workflows," Argonne National Laboratory, Argonne, IL (United States), Tech. Rep., 2017.

[4] D. Morozov and Z. Lukic, "Master of puppets: Cooperative multitasking for in situ processing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 285–288.

[5] T. Peterka, D. Bard, J. C. Bennett, E. W. Bethel, R. A. Oldfield, L. Pouchard, C. Sweeney, and M. Wolf, "Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources," *The International Journal of High Performance Computing Applications*, p. 1094342020913628, 2020.

[6] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of grid computing*, vol. 3, no. 3-4, pp. 171–200, 2005.

[7] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future generation computer systems*, vol. 25, no. 5, pp. 528–540, 2009.

[8] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 101–109. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109

[9] S. Ahern, E. Brugger, B. Whitlock, J. S. Meredith, K. Biagas, M. C. Miller, and H. Childs, "Visit: Experiences with sustainable software," *arXiv preprint arXiv:1309.1796*, 2013.

[10] 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 *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ACM, 2015, pp. 25–29.

[11] J. Ahrens, B. Geveci, and C. Law, "36 paraview: An end-user tool for large-data visualization," *The Visualization Handbook*, p. 717, 2005.

[12] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 246–255.

[13] 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.

[14] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[15] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Comparing the efficiency of in situ visualization paradigms at scale," in *International Conference on High Performance Computing*. Springer, 2019, pp. 99–117.

[16] M. Dorier, R. R. Sisneros, T. Peterka, G. Antoniu, and D. B. Semeraro, "Damaris/Viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, Atlanta, United States, Oct. 2013, conference. [Online]. Available: https://hal.inria.fr/hal-00859603

[17] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. Bethel, "The SENSEI generic in situ interface," in *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*. IEEE Press, 2016, pp. 40–44.

[18] E. Lohrmann, Z. Lukić, D. Morozov, and J. Müller, "Programmable in situ system for iterative workflows," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Klusáček, W. Cirne, and N. Desai, Eds. Springer International Publishing, 2018, vol. 10773, pp. 122–131.

[19] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 95–102.

[20] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin, "Melissa: Large scale in transit sensitivity analysis avoiding intermediate files," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 61.

[21] O. Yildiz, J. Ejarque, H. Chan, S. Sankaranarayanan, R. M. Badia, and T. Peterka, "Heterogeneous hierarchical workflow composition," *Computing in Science & Engineering*, 2019.

[22] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[23] S. Plimpton, P. Crozier, and A. Thompson, "Lammps-large-scale atomic/molecular massively parallel simulator," *Sandia National Laboratories*, vol. 18, p. 43, 2007.

[24] T. Peterka, H. Croubois, N. Li, E. Rangel, and F. Cappello, "Self-adaptive density estimation of particle data," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S646–S666, 2016.

[25] T. Peterka, D. Morozov, and C. Phillips, "High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 997–1007.

[26] D. Morozov and T. Peterka, "Block-Parallel Data Analysis with DIY2," 2016.

[27] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 911–920.