frontiers | Frontiers in High Performance Computing

TYPE Methods
PUBLISHED 20 November 2024
DOI 10.3389/fhpcp.2024.1472719

OPEN ACCESS

EDITED BY
Andre Merzky,
Rutgers, The State University of New Jersey,
United States

REVIEWED BY
Jay Lofstead,
Sandia National Laboratories (DOE),
United States
Dongfang Zhao,
University of Washington, United States

*CORRESPONDENCE
Orcun Yildiz
oyildiz@anl.gov

RECEIVED 29 July 2024
ACCEPTED 30 October 2024
PUBLISHED 20 November 2024

CITATION
Yildiz O, Morozov D, Nigmetov A, Nicolae B and Peterka T (2024) Wilkins: HPC *in situ* workflows made easy.
Front. High Perform. Comput. 2:1472719.
doi: 10.3389/fhpcp.2024.1472719

COPYRIGHT
© 2024 Yildiz, Morozov, Nigmetov, Nicolae and Peterka. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

Wilkins: HPC *in situ* workflows made easy

Orcun Yildiz[1]*, Dmitriy Morozov[2], Arnur Nigmetov[2], Bogdan Nicolae[1] and Tom Peterka[1]

[1]Argonne National Laboratory, Lemont, IL, United States, [2]Lawrence Berkeley National Laboratory, Berkeley, CA, United States

In situ approaches can accelerate the pace of scientific discoveries by allowing scientists to perform data analysis at simulation time. Current *in situ* workflow systems, however, face challenges in handling the growing complexity and diverse computational requirements of scientific tasks. In this work, we present Wilkins, an *in situ* workflow system that is designed for ease-of-use while providing scalable and efficient execution of workflow tasks. Wilkins provides a flexible workflow description interface, employs a high-performance data transport layer based on HDF5, and supports tasks with disparate data rates by providing a flow control mechanism. Wilkins seamlessly couples scientific tasks that already use HDF5, without requiring task code modifications. We demonstrate the above features using both synthetic benchmarks and two science use cases in materials science and cosmology.

KEYWORDS
HPC, *in situ* workflows, usability, ensembles, data transport, flow control

1 Introduction

In situ workflows have gained traction in the high-performance computing (HPC) community because of the need to analyze increasing data volumes, together with the ever-growing gap between the computation and I/O capabilities of HPC systems. *In situ* workflows run within a single HPC system as a collection of multiple tasks, which are often large and parallel programs. These tasks communicate over memory or the interconnect of the HPC system, bypassing the parallel file system. Avoiding physical storage minimizes the I/O time and accelerates the pace of scientific discoveries.

Despite their potential advantages, challenges for *in situ* workflows include the growing complexity and heterogeneity of today's scientific computing, which pose several problems that are addressed in this article. First, the workflow system should enable seamless coupling of user task codes, while providing a flexible interface to specify diverse data and computation requirements of these tasks. In particular, the workflow interface should support specification of today's complex workflows including computational steering and ensembles of tasks. Second, user tasks may employ a wide variety of data models. This heterogeneity of the data is even more evident with the growing number of AI tasks being incorporated in *in situ* workflows. The workflow system should provide a data model abstraction through which users can specify their view of data across heterogeneous tasks. Third, *in situ* workflows often include tasks with disparate data rates, requiring efficient flow control strategies to mitigate communication bottlenecks between tasks.

Another key factor is the usability of *in situ* workflows. The workflow systems should be easy to use while being able to express the different requirements of users. One common concern among users is the amount of required modifications to their task codes. Unfortunately, the current state of the art often requires changes to user codes, where users

manually need to insert workflow API calls into their codes to be able to run them within the *in situ* workflow system. Such code modifications can be cumbersome, depending on the level of such changes, and further impede adoption of workflow systems. Ideally, the same code should be able to run standalone as a single executable and as part of a workflow.

Driven by the needs of today's computational science campaigns, we introduce Wilkins, an *in situ* workflow system with the following features:

- Ease of adoption, providing scalable and efficient execution of workflow tasks without requiring any task code changes.
- A flexible workflow description interface that supports various workflow topologies ranging from simple linear workflows to complex ensembles.
- A high-performance data transport layer based on the rich HDF5 data model.
- A flow control mechanism to support efficient coupling of *in situ* workflow tasks with different rates.

We demonstrate the above features with both synthetic experiments and two different science use cases. The first is from materials science, where a workflow is developed for capturing a rare nucleation event. This requires orchestrating an ensemble of multiple molecular dynamics simulation instances coupled to a parallel *in situ* feature detector. In the second use case, the *in situ* workflow consists of a cosmological simulation code coupled to a parallel analysis task that identifies regions of high dark-matter density. These tasks have disparate computation rates, requiring efficient flow control strategies.

The remainder of this paper is organized as follows. Section 2 presents background and related work. Section 3 explains the design and implementation of Wilkins. Section 4 presents our experimental results in both synthetic benchmarks as well as two representative science use cases. Section 5 concludes the paper with a summary and a look toward the future.

2 Background and related work

We first provide a brief background on *in situ* workflows. Then, we present the related work by categorizing *in situ* workflows according to their workflow description interfaces and data transfer mechanisms.

2.1 *In situ* workflows

Scientific computing encompasses various interconnected computational tasks. *In situ* workflow systems have been developed over the years by the HPC community to automate the dependencies and data exchanges between these tasks, eliminating the need for manual management. *In situ* workflows are designed to run within a single HPC system, launching all tasks concurrently. Data transfer between these tasks is done through memory or interconnect of the HPC system instead of the physical storage. Representative of such systems include ADIOS (Boyuka et al., 2014), Damaris (Dorier et al., 2016), Decaf (Yildiz et al., 2022), ParaView Catalyst (Ayachit et al., 2015), SENSEI (Ayachit et al., 2016), and VisIt Libsim (Kuhlen et al., 2011).

2.2 Workflow description interfaces

Most *in situ* workflow systems use a static declarative interface in the form of a workflow configuration file to define the workflow. For instance, Decaf (Yildiz et al., 2022) and FlowVR (Dreher and Raffin, 2014) workflow systems use a Python script for workflow graph description, while ADIOS (Boyuka et al., 2014), Damaris (Dorier et al., 2016), and VisIt Libsim (Kuhlen et al., 2011) all use an XML configuration file. Similarly, Wilkins provides a simple YAML configuration file for users to describe their workflows. Some workflow systems choose to employ an imperative interface. Henson (Morozov and Lukic, 2016), a cooperative multitasking system for *in situ* processing, follows this approach by having users directly modify the workflow driver code.

Alternatively, workflows can be defined implicitly using a programming language such as Swift/T (Wozniak et al., 2013), which schedules tasks according to data dependencies within the program. While Swift/T can handle complex workflows, users need to organize and compile their code into Swift modules.

One important aspect of workflow description interfaces is their extensibility while maintaining simplicity. In particular, the workflow interface should allow users to define complex scientific workflows with diverse requirements, ideally with minimal user effort. One example is ensemble workflows where there can be numerous workflow tasks and communication channels among them. Even though ensembles of simulations today are often executed offline in a preprocessing step prior to analyzing the results, there is a growing trend to execute them *in situ* in order to dynamically adapt the search space, save storage, and reduce time to solution. For instance, there are some *in situ* systems that are specifically designed for this type of workflows. Melissa (Schouler et al., 2023) is a framework to run large-scale ensembles and process them *in situ*. LibEnsemble (Hudson et al., 2021) is a Python library that supports *in situ* processing of large-scale ensembles. DeepDriveMD (Brace et al., 2022) is a framework for ML-driven steering of molecular dynamics simulations that couples large-scale ensembles of AI and HPC tasks. There are also domain-specific coupling tools in certain scientific fields that support ensemble simulations, such as the climate community with CESM (Kay et al., 2015) and E3SM (Golaz et al., 2022). While we also support ensembles in Wilkins, our workflow description interface is domain-agnostic and generic, which is not specifically tailored to a particular category of workflows, such as ensembles.

2.3 Data transfer mechanisms

One key capability of workflows is to automate the data transfers between individual tasks within the workflow. Data transfer mechanisms vary among *in situ* workflows, but shared memory and network communication are the most common data transfer mechanisms.

In *in situ* workflows where tasks are colocated on the same node, shared memory can offer benefits by enabling zero-copy communication. VisIt's Libsim and Paraview's Catalyst use shared-memory communication between analysis and visualization tasks, operating synchronously with the simulation within the same address space. Henson is another workflow system that supports shared-memory communication among colocated tasks on the same node. This is achieved by dynamically loading the executables of these tasks into the same address space.

When the workflow tasks are located on separate nodes within the same system, data can be transferred between the tasks using the system interconnect. This approach enables efficient parallel communication by eliminating the need for the parallel file system. Decaf (Yildiz et al., 2022) is a middleware for coupling parallel tasks *in situ* by establishing communication channels over HPC interconnects through MPI. Similarly, Damaris (Dorier et al., 2016) uses direct messaging via MPI between workflow tasks to exchange data. Wilkins also adopts this approach to provide efficient parallel communication between workflow tasks.

Some workflow systems opt to use a separate staging area when moving the data between the tasks instead of direct messaging. This approach is often called data staging; it requires extra resources for staging the data in an intermediate location. Systems such as DataSpaces (Docan et al., 2012), FlexPath (Dayal et al., 2014), and Colza (Dorier et al., 2022) adopt this approach, where they offload the data to a distributed memory space that is shared among multiple workflow tasks. Other approaches such as DataStates (Nicolae, 2020, 2022) retain multiple versions of datasets in the staging area, which enables the tasks to consume any past version of the dataset, not just the latest one.

While these *in situ* solutions offer efficient data transfers by avoiding physical storage, they share a common requirement for modifications to task code. For instance, Decaf and DataSpaces both use a put/get API for data transfers which needs to be integrated into task codes. On the other hand, Wilkins does not require any changes to task codes if they already use HDF5 or one of the many front-ends to HDF5, such as HighFive (BlueBrain, 2022), h5py (Collette, 2013), NetCDF4 (Rew et al., 2004), SCORPIO (Krishna, 2020), or Keras (Gulli and Pal, 2017).

3 Methodology

Wilkins is an *in situ* workflow system that enables heterogenous task specification and execution for *in situ* data processing. Wilkins provides a data-centric API for defining the workflow graph, creating and launching tasks, and establishing communicators between the tasks.

3.1 Overall architecture

Figure 1 shows an overview of Wilkins and its main components, which are data transport, data model, workflow execution, and workflow driver. At its data transport layer, Wilkins uses the LowFive library (Peterka et al., 2023), which is a data model specification, redistribution, and communication library implemented as an HDF5 Virtual Object Layer (VOL) plugin.

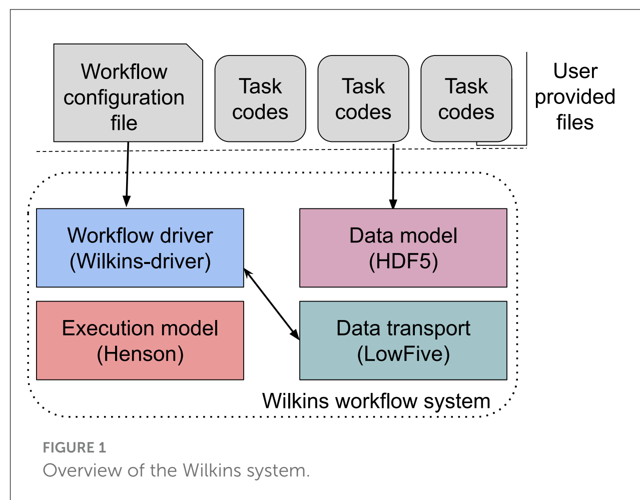

FIGURE 1
Overview of the Wilkins system.

LowFive can be enabled either by setting environment variables or manually constructing a LowFive object, via the LowFive API, in the user task codes. Wilkins adopts the former approach to have task codes with no modifications.

To execute the workflow tasks, Wilkins relies on Henson's execution model, where user task codes are compiled as shared objects (Morozov and Lukic, 2016). At the workflow layer, Wilkins has a Python driver code, where all the workflow functions (e.g., data transfers, flow control) are defined through this code. This Python driver code is generic and provided by the Wilkins system, and users do not need to modify this code.

At the user level, users only need to provide the workflow configuration file and the constituent task codes. Linking the task codes as shared objects is often the only required additional step to use Wilkins.

3.2 Data-centric workflow description

Wilkins employs a data-centric workflow definition, where users indicate tasks' resource and data requirements using a workflow configuration file. Rather than specifying explicitly which tasks depend on others, users specify input and output data requirements in the form of file/dataset names. By matching data requirements, Wilkins automatically creates the communication channels between the workflow tasks, and generates the workflow task graph as a representation of this workflow configuration file. Wilkins supports any directed-graph topology of tasks, including common patterns such as pipeline, fan-in, fan-out, ensembles of tasks, and cycles.

Users describe their workflow definition in a YAML file. Listing 1 shows a sample YAML file representing a 3-task workflow consisting of one producer and two consumer tasks. The producer generates two different datasets—a structured grid of values and a list of particles—while the first and second consumer each require only the grid and particle datasets, respectively. Users describe these data requirements using the *inport* and *outport* fields in the YAML file. While the sample YAML file in Listing 1 uses full names for the file and dataset names, it is also possible to use matching patterns

```
tasks:
  - func: producer
    nprocs: 3
    outports:
      - filename: outfile.h5
        dsets:
          - name: /group1/grid
            file: 0
            memory: 1
          - name: /group1/particles
            file: 0
            memory: 1
  - func: consumer1
    nprocs: 5
    inports:
      - filename: outfile.h5
        dsets:
          - name: /group1/grid
            file: 0
            memory: 1
  - func: consumer2
    nprocs: 2
    inports:
      - filename: outfile.h5
        dsets:
          - name: /group1/particles
            file: 0
            memory: 1
```

Listing 1 Sample YAML file for describing a 3-task workflow consisting of 1 producer and 2 consumers.

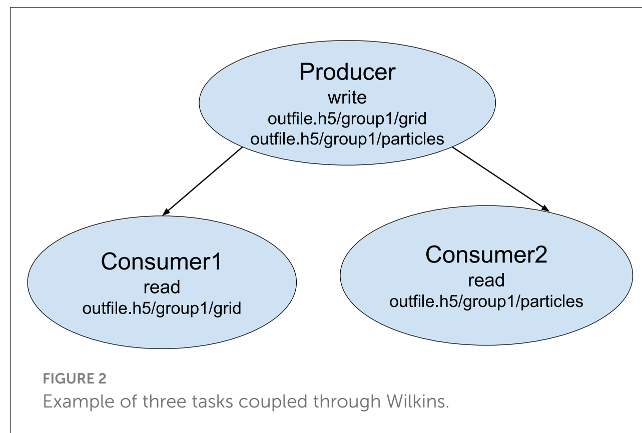

FIGURE 2
Example of three tasks coupled through Wilkins.

(e.g., **.h5/particles* can be used instead of *outfile.h5/particles*). Based on these requirements, Wilkins creates two communication channels: one channel between the producer and the first consumer for the grid dataset, and another channel between the producer and the second consumer for the particles dataset. In these channels, tasks will communicate using LowFive, Wilkins' data transport library, either through MPI or HDF5 files. Users can select the type of this communication in the YAML file by setting *file* to 1 for using files or by setting *memory* to 1 for using MPI. For instance, this example uses MPI in both of the communication channels between the coupled tasks. Figure 2 illustrates the workflow consisting of these three tasks coupled through Wilkins.

For the resource requirements of the tasks, users indicate the number of processes using the *nprocs* field. Wilkins will assign these resources to the tasks and launches them accordingly. The execution model of Wilkins is described in Section 3.5.

3.2.1 Defining ensembles

Ensembles of tasks have become prevalent in scientific workflows. For instance, one common use case is to run the same simulation with different input parameters in hopes of capturing a rare scientific event (Yildiz et al., 2019). Other examples of ensembles arise in AI workflows performing hyperparameter optimization, or for uncertainty quantification (Meyer et al., 2023). Such ensembles are often large-scale, requiring the orchestration of multiple concurrent tasks by the workflow system.

One question is how to specify an ensemble of tasks in a workflow configuration file. As there are often many tasks in an ensemble, we cannot expect users to list them explicitly. Instead,

```
tasks:
  - func: producer
    taskCount: 4 #Only change needed to
    define ensembles
    nprocs: 3
    outports:
      - filename: outfile.h5
        dsets:
          - name: /group1/grid
            file: 0
            memory: 1
  - func: consumer
    taskCount: 2 #Only change needed to
    define ensembles
    nprocs: 5
    inports:
      - filename: outfile.h5
        dsets:
          - name: /group1/grid
            file: 0
            memory: 1
```

Listing 2 Sample YAML file for describing an ensemble of tasks with a fan-in topology.

Wilkins provides an optional *taskCount* field, where users can indicate the number of task instances in an ensemble. With this one extra field of information in the YAML file, Wilkins allows specification of various workflow graph topologies with ensembles of tasks including fan-in, fan-out, M to N, or combinations of those. Wilkins automatically creates the communication channels between the coupled ensemble tasks, without users having to explicitly list such dependencies thanks to its data-centric workflow description. Listing 2 shows a sample YAML file for describing ensembles with a fan-in topology, where four instances of a producer task are coupled to two instances of a consumer task.

Figure 3 illustrates how Wilkins performs ensemble coupling of producer-consumer pairs in a fan-in topology with four producer and two consumer instances. For each matching data object, Wilkins creates a list of producer task indices and a list of consumer task indices. Wilkins then links these producer-consumer pairs by iterating through these indices in a round-robin fashion, as shown in Figure 3.

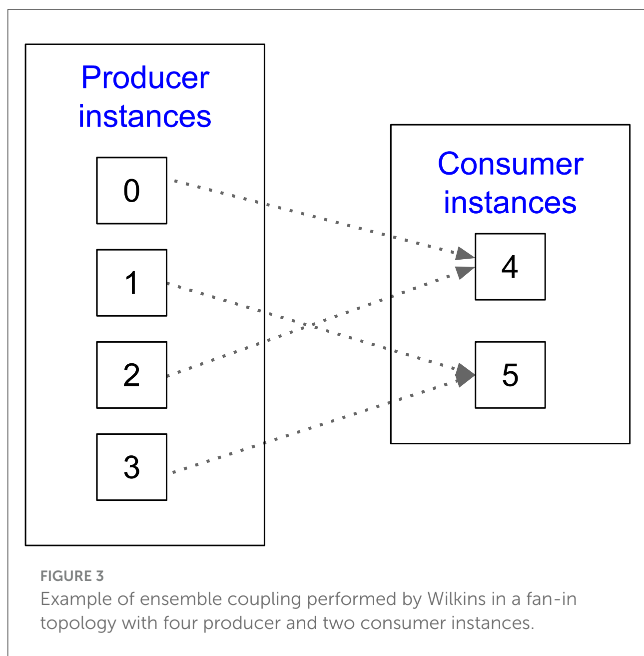

FIGURE 3
Example of ensemble coupling performed by Wilkins in a fan-in topology with four producer and two consumer instances.

3.2.2 Defining subset of writers

Despite the advantages of parallel communication between the processes of workflow tasks, some simulations opt to perform serial I/O from a single process. For instance, the LAMMPS molecular dynamics simulation code first gathers all data to a single MPI process, and then this process writes the output serially (Plimpton et al., 2007).

To support such scenarios with serial or partially parallel writers, we introduce an optional *io_proc* field in the workflow configuration file. Users simply can indicate the number of writers in addition to the number of processes for the producer task. Then, Wilkins will assign this set of processes (starting from process 0) as I/O processes, while the remaining processes will only participate in the task execution (e.g., simulation) without performing any I/O operations.

This feature is implemented in the workflow driver code, which first checks whether a producer process is an I/O process based on the workflow configuration file. If so, the Wilkins driver creates a LowFive object and sets its properties (e.g., memory, file) in order for this process to participate in the data exchange. Local communicators and intercommunicators between the tasks provided to the LowFive object only involves I/O processes, and other processes do not participate in these communicator creation, which is handled by Wilkins.

3.3 Workflow driver

The Wilkins runtime, *Wilkins-driver*, is written in Python and serves as the main workflow driver to execute the workflow. *Wilkins-driver* orchestrates all the different functions within the workflow (e.g., launching tasks, data transfers, ensembles, flow control) as specified by the users in the workflow configuration file. Users do not need to modify the *Wilkins-driver* code to use any of the Wilkins capabilities.

Wilkins-driver first starts by reading the workflow configuration file to create the workflow graph. Based on this file, it creates local communicators for the tasks and intercommunicators between the interconnected tasks. Then, *Wilkins-driver* creates the LowFive plugin for the data transport layer. Next, it sets LowFive properties such as whether to perform data transfers using memory or files. After that, several Wilkins capabilities are defined, such as ensembles or flow control if they are specified in the configuration file. *Wilkins-driver* also checks whether there are any custom actions, which can be specific to particular use cases. We detail in Section 3.5 how users can specify custom actions through external Python scripts. Ultimately, *Wilkins-driver* launches the workflow.

3.4 Data model and data transport library

Wilkins employs the data model of the LowFive (Peterka et al., 2023) library. HDF5 (Folk et al., 2011) is one of the most common data models, and as a VOL plugin, LowFive benefits from HDF5's rich metadata describing the data model while affording users the familiarity of HDF5.

In its data transport layer, Wilkins uses the data redistribution components of LowFive, which enables data redistribution from M to N processes. Wilkins allows coupled tasks to communicate both *in situ* using in-memory data and MPI message passing, and through traditional HDF5 files if needed. Users can select these different communication mechanisms via the workflow configuration file.

Building on the base functionality of LowFive, Wilkins adds several new capabilities related to data transport. Wilkins extends the LowFive library by developing a callback functionality. In Wilkins, we use these callbacks to provide additional capabilities such as flow control. For example, we can exchange data between coupled tasks at a reduced frequency, rather than exchanging data at every iteration. Another scenario is custom callbacks, where users can define custom actions upon a specific I/O operation such as dataset open or file close. We will see examples of such callbacks in the next subsections.

3.5 Execution model

In a Wilkins workflow, user task codes can be serial or parallel; they can also have different languages such as C, C++, Python, or Fortran. Wilkins executes the user codes as a single-program-multiple-data (SPMD) application, thus having access to the MPI_COMM_WORLD across all ranks. Wilkins partitions this communicator and presents restricted MPI_COMM_WORLDs to the user codes, relying on Henson's PMPI tooling to make this transparent. This way the user codes see only their restricted world communicator, and the user codes are still written in a singular standalone fashion using this world communicator, as if they were its only users. Wilkins manages the partitioning of the global communicator into different local communicators, one for each task, as well as the intercommunicators connecting them. This process is entirely transparent to the users. Users only need to compile their codes as shared objects to execute them with Wilkins.

10.3389/fhpcp.2024.1472719

Although we build on Henson's execution model, Wilkins adds several new features not provided by Henson, such as ensembles and flow control. Also, Wilkins provides a flexible workflow description interface which is declarative, while Henson has an imperative interface. We discuss the tradeoffs between these interfaces in Section 3.5.2.

As a Wilkins workflow is an SPMD application, it is submitted to the HPC system job scheduler (e.g., Slurm) as a single batch job. Resource (e.g., CPU, GPU) and process placement is handled by the job scheduler—independent from Wilkins. Job schedulers provide users various options to optimize their process and resource placement strategies.

```
dw_counter = 0
def custom_cb(vol, rank):
    #after dataset write callback
    def adw_cb(s):
        global dw_counter
        dw_counter = dw_counter + 1
        if dw_counter % 2 == 0:
            #serving data at every 2
            dataset write operation
            vol.serve_all(True, True)

    vol.set_after_dataset_write(adw_cb)
```

Listing 3 Sample custom action script that can be provided by the user.

3.5.1 Support for different consumer types

In today's scientific workflows, we can categorize tasks into three types: (i) producers such as HPC simulations that generate data periodically, (ii) consumers such as analysis or visualization tasks that consume these data, and (iii) intermediate tasks such as data processing that are both producers and consumers in a pipeline. Moreover, for the tasks that consume data, we can have two different types:

3.5.1.1 Stateful consumer

Such consumers maintain state information about the previous executions (e.g., timesteps). For instance, particle tracing codes need to keep information on the current trajectory of a particle (Guo et al., 2017) while advecting the particle through the next step.

3.5.1.2 Stateless consumer

Such consumers do not maintain any information regarding their previous executions, as each run of a stateless task is entirely independent. For example, a feature detector code used in the analysis of molecular dynamics checks the number of nucleated atoms in each simulation timestep to determine whether nucleation is happening (Yildiz et al., 2019), with no relationship to previous timesteps.

Wilkins' execution model supports all these different task types, including both stateful and stateless consumers, transparent to the user. Wilkins first launches all these tasks as coroutines. Once the consumer tasks are completed, Wilkins uses a LowFive callback to query producers whether there are more data to consume. Producers respond to this query with the list of filenames that need to be consumed, or an empty list if no more data will be generated (all done). With this query logic, Wilkins handles both stateful and stateless consumer types. A stateful consumer is launched once and runs until completion for the number of timesteps or iterations as defined by the user. On the other hand, stateless consumers are launched as many times as there are incoming data to consume.

3.5.2 Support for user-defined actions

There can be scenarios that require custom workflow actions such as time- or data-dependent behaviors. For instance, users can request to transfer data between tasks only if the data value exceeds some predefined threshold. The *Wilkins-driver* code that executes the workflow is generic and does not support such custom actions by default.

To support such custom actions, we explored two options: (i) allowing users to modify the Wilkins driver code directly (similar to the workflow systems with imperative interfaces such as Henson) or (ii) letting users define these custom actions in an external Python script, which the Wilkins runtime incorporates. These options have tradeoffs with respect to usability and extensibility. While the first option, an imperative interface, provides more extensibility by exposing the workflow runtime to the user, it introduces additional complexity as users would need to be familiar with the *Wilkins-driver* code. We opted for a declarative interface, and decided that adopting the second option, defining external custom actions, would be a convenient middle ground between imperative and declarative interfaces. In our design, users define custom actions in a Python script using callbacks, and these callbacks allow imperative customization within an otherwise declarative interface.

Listing 3 shows a sample Python script representing the custom actions requested by the user. For instance, consider a scenario where the producer task performs two dataset write operations, but the analysis task only needs the second dataset. Without modifying the producer task code, the user can provide this script to Wilkins, which will then perform data transfer between tasks after every second dataset write operation. In this script, user simply defines this custom action (*custom_cb*) in a callback at after dataset write (*adw_cb*), which delays the data transfer until the second occurrence of dataset write operation. We will see more examples of the use of callbacks in flow control and in the high-energy physics science use case.

3.6 Flow control

In an *in situ* workflow, coupled tasks run concurrently, and wait for each other to send or receive data. Discrepancies among task throughputs can cause bottlenecks, where some tasks sit idle waiting for other tasks, resulting in wasted time and resources. To alleviate such bottlenecks, Wilkins provides a flow control feature, where users can specify one of three different flow control strategies through the workflow configuration file:

- **All**: This is the default strategy in Wilkins when users do not specify any flow control strategy. In this strategy, the producer

waits until the consumer is ready to receive data. A slow consumer can result in idle time for the producer task.

- **Some**: With this strategy, users provide the desired frequency of the data exchange in order to accommodate a slow consumer. For instance, users can specify to consume data every N iterations, where N is equal to the desired frequency (e.g., 10 or 100). This strategy provides a tradeoff between blocking the producer and consuming at a lower frequency.
- **Latest**: In this strategy, Wilkins drops older data in the communication channel and replaces them with the latest timestep from the producer once the consumer is ready. This strategy can be useful when the problem is time-critical, and scientists prefer to analyze the latest data points instead of older ones.

Specifying the flow control strategy requires adding only one extra field of information to the configuration file, *io_freq*, where users can set the above strategies by specifying $N > 1$ for the *some* strategy, 1 or 0 for *all*, and -1 for the *latest* strategy.

Wilkins enforces these different strategies for flow control using LowFive callbacks, transparent to the user. For instance, consider a simple workflow consisting of a faster producer coupled to a slower consumer, using the *latest* flow control strategy. In LowFive, the producer serves data to the consumer when the producer calls a file close operation. When the *latest* strategy is in place, Wilkins registers a callback before the file close function, where the producer checks whether there are any incoming requests from consumers before sending the data. If there are requests, then data transfer happens normally (the same as no flow control strategy). However, if there are no requests from the consumer, then the producer skips sending data at this timestep and proceeds with generating data for the next timestep. This process continues until the producer terminates. All these steps are part of LowFive and Wilkins, and are transparent to the user.

Such a flow control mechanism allows Wilkins to support heterogeneous workflows consisting of tasks with different data and computation rates.

4 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. We installed Wilkins on the Bebop machine through the Spack package manager (Gamblin et al., 2015), where Wilkins is available online as open-source project.[1]

4.1 Synthetic experiments

We perform four different sets of experiments. For the first experiment, we use the code developed in Peterka et al. (2023) to couple a producer and a consumer task that communicate using LowFive, without a workflow system on top. Then, we measure the overhead of Wilkins as a workflow system compared with that scenario. Second, we evaluate the flow control feature of Wilkins. Third, we demonstrate Wilkins' capability of supporting complex ensembles. Lastly, we compare Wilkins with ADIOS2, one of the representative systems from the state-of-the-art.

For the synthetic benchmarks, we follow the approach used by Peterka et al. (2023). For the first two sets of experiments, we have a linear 2-node workflow coupling one producer and one consumer task. In the ensemble experiments, we vary the number of producer and consumer instances representing various workflow topologies.

We generate synthetic data containing two datasets: one is a regular grid comprising 64-bit unsigned integer scalar values, and the other one is a list of particles, where each particle is a 3-d vector of 32-bit floating-point values. Per producer process, there are 10^6 regularly structured grid points and 10^6 particles. Each grid point and particle occupies 8 and 12 bytes, respectively. Consequently, the total data per producer process is 19 MiB. We report the average times taken over 3 trials.

4.1.1 Overhead of Wilkins compared with LowFive

In this overhead experiment, we perform a weak scaling test by increasing the total data size proportionally with the number of producer processes. The producer generates the grid and particles datasets, and the consumer reads both of them. We allocate three-fourths of the processes to the producer, and the remaining one-fourth to the consumer task. For this overhead experiments, we also use larger data sizes with 10^7 and 10^8 grid points and particles per MPI process. Table 1 shows the number of MPI processes for each task and total data sizes.

Figure 4 shows the time to write/read grid and particles datasets between the producer and consumer tasks in a weak scaling regime. We can observe that LowFive and Wilkins execution times are similar for all data sizes. The difference between using LowFive standalone and with Wilkins at 1K processes is only 2%. We consider this overhead negligible, considering the additional capabilities Wilkins brings as a workflow system compared with LowFive, which is only a data transport layer.

4.1.2 Flow control

In these experiments, we use 512 processes for both producer and consumer tasks. We use the sleep function to emulate the computation behavior of tasks. For the producer task, we use 2 seconds sleep. For the consumer task, we emulate three different slow consumers as 2×, 5×, and 10× slow consumers by adding 4, 10, and 20 s sleep to the consumer tasks. The producer task runs for a total of 10 timesteps generating grid and particles datasets. We employ three different flow control strategies: (i) *all*—producer task serving data at every timestep, (ii) *some*—producer task serving data at every 2, 5, or 10 timesteps, and (iii) *latest*—producer task serving data when the consumer signals that it is ready. For the *some* strategy, we run with $N = 2$ for the 2× slow consumer, $N = 5$ for the 5x slow consumer, and $N = 10$ for the 10x slow consumer. Table 2 shows the completion time of the workflow under these

1 Wilkins is available at https://github.com/orcunyildiz/wilkins/.

TABLE 1 Number of MPI processes for producer and consumer tasks and the total data size exchanged between them.

Workflow size (procs)	Producer size (procs)	Consumer size (procs)	Total data size (10^6/proc) (GiB)	Total data size (10^7/proc) (GiB)	Total data size (10^8/proc) (GiB)
4	3	1	0.06	0.6	6
16	12	4	0.22	2.2	22
64	48	16	0.99	9.9	99
256	192	64	3.54	35.4	354
1,024	768	256	14.34	143.4	1,434

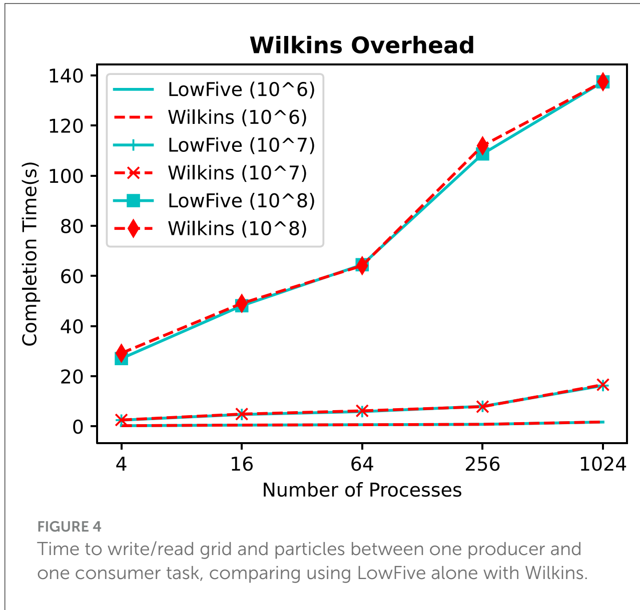

FIGURE 4
Time to write/read grid and particles between one producer and one consumer task, comparing using LowFive alone with Wilkins.

TABLE 2 Completion time for the workflow coupling a producer and a (2×, 5×, and 10×) slow consumer under different flow control strategies.

Strategy	Completion time (2×) (s)	Completion time (5×) (s)	Completion time (10×) (s)
All	51	111.7	211.7
Some	31.2	35	44.9
Latest	33.5	38	45.8

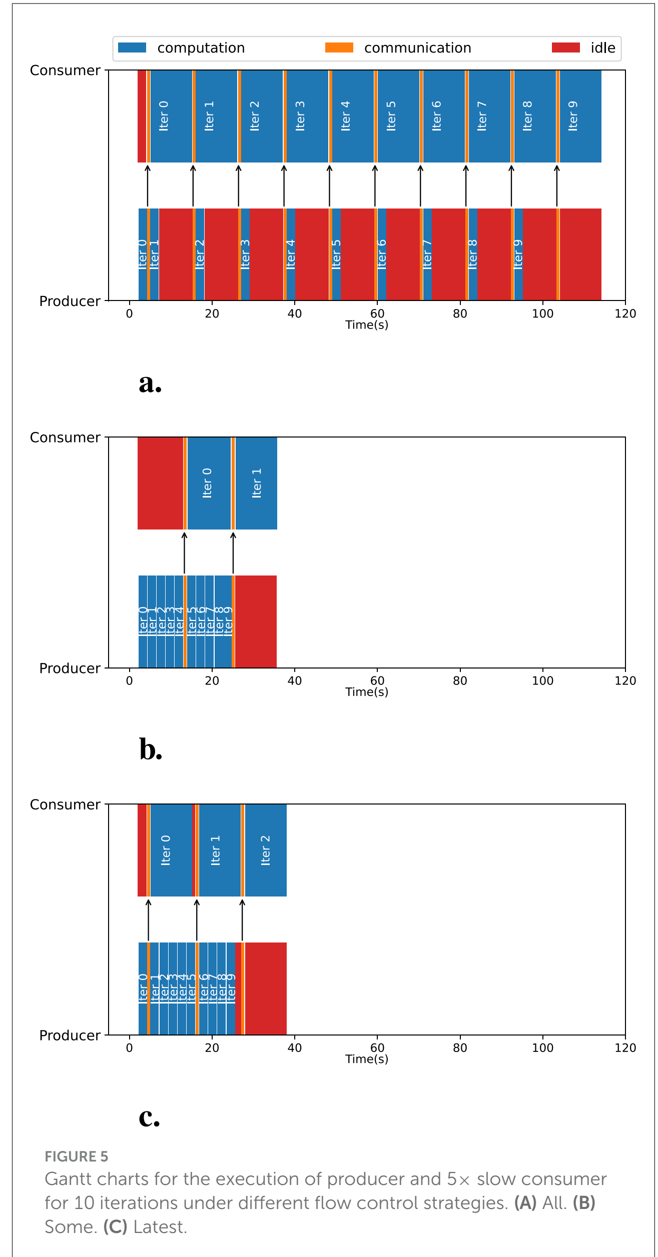

FIGURE 5
Gantt charts for the execution of producer and 5× slow consumer for 10 iterations under different flow control strategies. **(A)** All. **(B)** Some. **(C)** Latest.

different strategies for each consumer task with different rates. We observe that using the *some* and *latest* flow control strategies results in up to 4.7× time savings. As expected, time savings are larger for the workflow with the slowest consumer (10× sleep). Time savings gained with the flow control strategies are due to the fact that the producer task does not have to wait for the slow consumer at every timestep, and can continue without serving to the next timestep when using the *some* and *latest* flow control strategies.

To further highlight the reduction in idle time for the producer, we illustrate the timeline for the execution of producer and 5× slow consumer under different flow control strategies in Figure 5. Blue bars represent the computation, while red ones represent the idle time for workflow tasks. We show the data transfer between tasks with an arrow and orange bars. With *all* flow control strategy, we can see that the producer task has to wait for the slow consumer at every timestep for the data transfer, resulting in significant idle time. In contrast, with the *some* and *latest* flow control strategies,

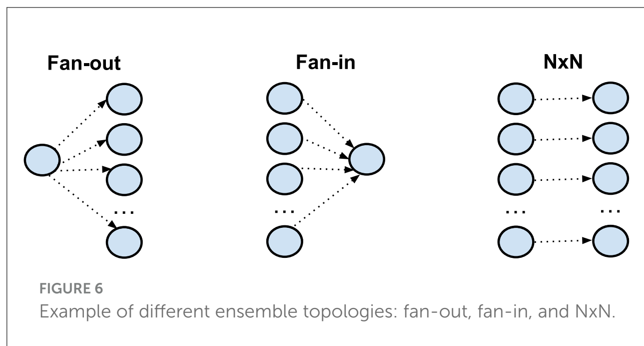

FIGURE 6
Example of different ensemble topologies: fan-out, fan-in, and NxN.

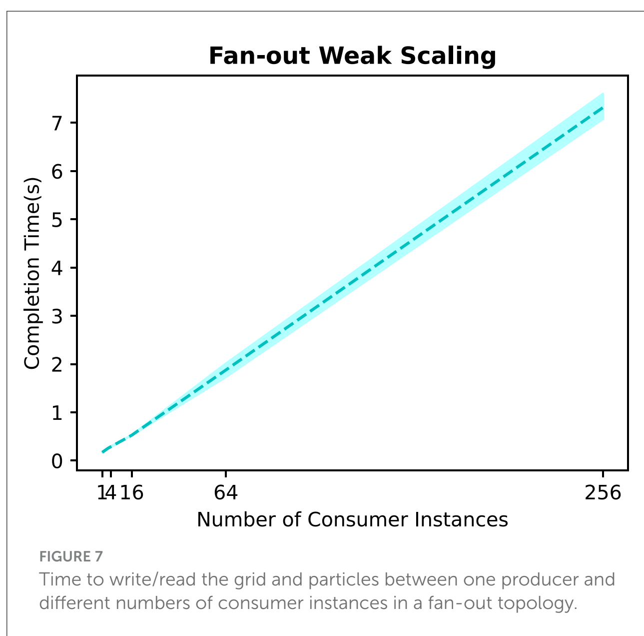

FIGURE 7
Time to write/read the grid and particles between one producer and different numbers of consumer instances in a fan-out topology.

these idle times are avoided where the producer task has to wait for the consumer only at the end of the workflow execution.

4.1.3 Scaling of ensembles

In these ensemble experiments, we use two processes for both producer and consumer instances. We vary the number of these instances to represent three different ensemble topologies: fan-out, fan-in, and NxN. Examples of these topologies are shown in Figure 6.

First, we analyze the time required to write/read the grid and particles between a single producer and different numbers of consumer instances in a fan-out topology. Figure 7 shows the results, where we use 1, 4, 16, 64, and 256 consumer instances. We can see that total time increases almost linearly with the number of consumer instances. For example, while the completion time is around 0.6 s with 16 consumer instances, this time increases to 8.2 s with 256 consumer instances. This is due to the fact that the data transfers between the producer and consumer instances happen sequentially, with each consumer instance processing different data points (e.g., the producer task can send different configuration parameters to each consumer instance).

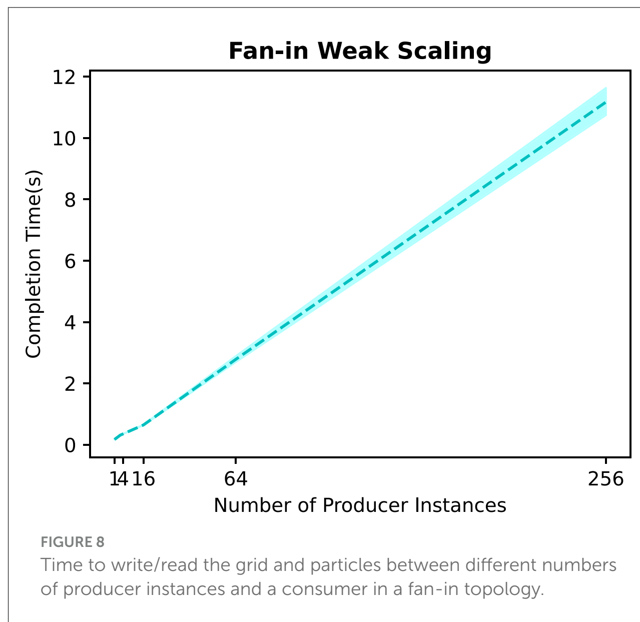

FIGURE 8
Time to write/read the grid and particles between different numbers of producer instances and a consumer in a fan-in topology.

Next, we evaluate Wilkins' support for the fan-in topology by varying the number of producer instances. Figure 8 shows the results. Similarly to fan-out results, we see that total time increases almost linearly with the number of producer instances as the consumer has to read from each producer instance.

Lastly, we evaluate the time required to write/read the grid and particles between different number of producer and consumer instances in an NxN topology. Figure 9 displays the results, where we use 1, 4, 16, 64, and 256 instances for both producer and consumer tasks. Unlike the fan-out and fan-in topologies, we can observe that the time difference is minimal when using different numbers of ensemble instances. This different behavior is expected as we have a one-to-one relationship between producer and consumer instances in an NxN topology, allowing data transfers to occur in parallel unlike the sequential process in fan-out and fan-in topologies. The slight increase in the total time can be attributed to the increased network contention at larger scales.

4.1.4 Comparison with ADIOS2

Next, we compare Wilkins with ADIOS2, a popular framework for coupling scientific codes through different transport layers. This experiment uses the Sustainable Staging Transport (SST) engine of ADIOS2, which allows direct connection of data producers and consumers via ADIOS2's put/get API. In particular, we configure ADIOS2's SST engine to use the MPI transport layer.

Figure 10 shows the results comparing Wilkins with ADIOS2. Wilkins consistently outperforms ADIOS2 at all scales. Moreover, in terms of usability, ADIOS2 requires task code modifications to use its put/get API when coupling scientific codes. In this synthetic benchmark, we wrote 10 additional lines of code using ADIOS2's API. Although the learning curve was not steep for such code modifications for a synthetic benchmark, similar modifications may be difficult in legacy codes with complex software stacks. On the other hand, Wilkins does not require any task code modifications.

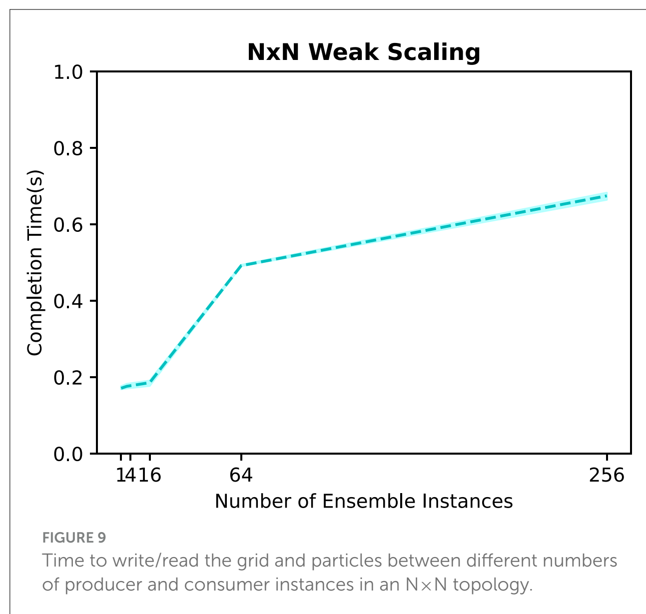

FIGURE 9
Time to write/read the grid and particles between different numbers of producer and consumer instances in an N×N topology.

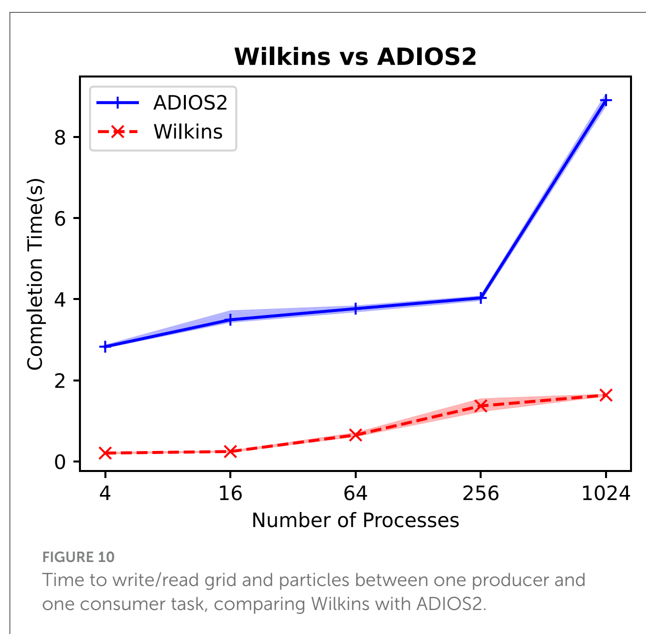

FIGURE 10
Time to write/read grid and particles between one producer and one consumer task, comparing Wilkins with ADIOS2.

ADIOS2 also has a support for HDF5-VOL, which would enable coupling codes without any code changes. However, their support is limited to only file-based data transport, and it lacks several operations required for our synthetic benchmarks, such as hyperslabs and memory space reading/writing (ADIOS2, 2024).

Lastly, ADIOS2 also provides a runtime configuration file, not unlike Wilkins, where users can switch between transport modes. One major difference is that while the Wilkins configuration file specifies a workflow graph, ADIOS2 only focuses on the I/O transport options. As a result, ADIOS2 does not support some of the features required for today's scientific workflows, such as ensembles and different flow control strategies.

```
tasks:
  - func: freeze
    taskCount: 64 #Only change needed to
    define ensembles
    nprocs: 32
    nwriters: 1 #Only rank 0 performs I/O
    outports:
      - filename: dump-h5md.h5
        dsets:
          - name: /particles/*
            file: 0
            memory: 1
  - func: detector
    taskCount: 64 #Only change needed to
    define ensembles
    nprocs: 8
    inports:
      - filename: dump-h5md.h5
        dsets:
          - name: /particles/*
            file: 0
            memory: 1
```

Listing 4 Sample YAML file for describing the molecular dynamics workflow for simulating nucleation with 64 ensemble instances.

4.2 Science use cases

We use two representative science use cases in materials science and cosmology to further demonstrate Wilkins applicability to actual scientific workflows. In both of these experiments, no changes were made to the workflow task codes, which further validates the ease-of-use of Wilkins. Here, we do not compare further with other systems such as ADIOS2, where we would have to modify the task codes.

4.2.1 Materials science

Nucleation occurs as a material cools and crystallizes, e.g., when water freezes. Understanding nucleation in material systems is important for better understanding of several natural and technological systems (Chan et al., 2019; Greer, 2016; Gettelman et al., 2012). Nucleation, however, is a stochastic event that requires a large number of molecules to reveal its kinetics. Simulating nucleation is difficult, especially in the initial phases of simulation when only a few atoms have crystallized.

One way scientists simulate nucleation is to run many instances of small simulations, requiring an ensemble of tasks where simulations with different initial configurations are coupled to analysis tasks. In this workflow, we couple a LAMMPS molecular dynamics simulation (Plimpton et al., 2007) *in situ* with a parallel feature detector that finds crystals in a diamond-shaped lattice (Yildiz et al., 2019). To create the ensemble, we use N instances for both simulation and analysis tasks in an N×N topology. To define these ensemble tasks, we only need to add the *taskCount* information to the workflow configuration file. Listing 4 shows the configuration file for this molecular dynamics workflow with 64 ensemble instances.

In LAMMPS's I/O scheme, all simulation data are gathered to rank 0 before they are written serially. This undermines Wilkins' capacity for efficient parallel communications. On the other hand,

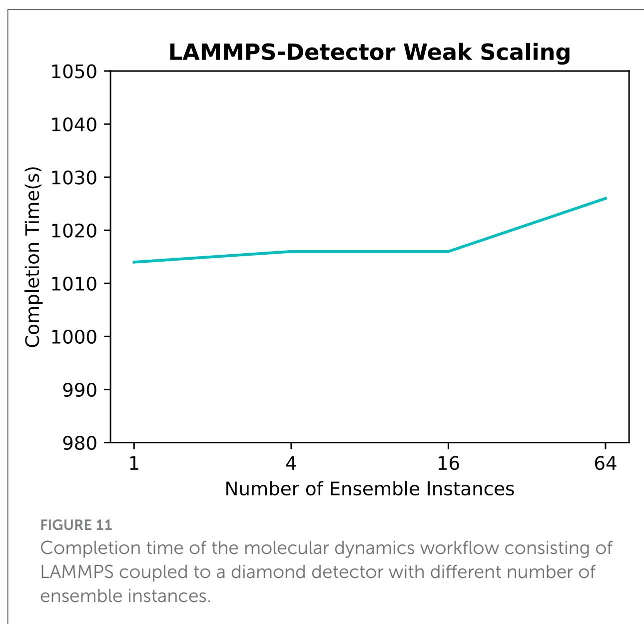

FIGURE 11
Completion time of the molecular dynamics workflow consisting of LAMMPS coupled to a diamond detector with different number of ensemble instances.

this demonstrates the applicability of the subset writers feature of Wilkins, where we only need to set the number of writers (i.e., *io_proc*) to 1 in the configuration file, as shown in Listing 4. Furthermore, LAMMPS supports writing HDF5 files, therefore, no modifications are needed to execute LAMMPS with Wilkins; we only had to compile LAMMPS as a shared library with HDF5 support.

In these experiments, we use 32 processes for each LAMMPS instance, and eight processes for each analysis task. We run LAMMPS for 1,000,000 time steps with a water model composed of 4,360 atoms, and we perform the diamond structure analysis every 10,000 iterations. To conduct this experiment, we vary the number of ensemble instances from 1 up to 64. Figure 11 shows the completion time under these scenarios. The results demonstrate that Wilkins can support execution of different number of ensemble instances without adding any significant overhead, in particular when there are a matching number of consumer instances in an N×N configuration. For example, the difference in completion time between a single instance and 64 ensemble instances is only 1.2%.

In terms of ease-of-use, no changes were made to the simulation or the feature detector source code to execute inside Wilkins, and to launch multiple instances in an ensemble, only one line was added to the producer and consumer task descriptions in the YAML workflow configuration file.

4.2.2 High-energy physics

The second use case is motivated by cosmology; in particular, halo finding in simulations of dark matter. The *in situ* workflow consists of Nyx (Almgren et al., 2013), a parallel cosmological simulation code, coupled to a smaller-scale parallel analysis task called Reeber (Friesen et al., 2016; Nigmetov and Morozov, 2019) that identifies high regions of density, called halos, at certain time steps.

```
def nyx(vol, rank):
    #after file close callback
    def afc_callback(s):
        if rank != 0:
            #other ranks, serving data
            vol.serve_all(True, True)
            vol.clear_files()
        else:
            if vol.file_close_counter
            % 2 == 0:
                #rank 0, serving data
                vol.serve_all(True, True)
                vol.clear_files()
            else:
                #rank 0 broadcasting files
                to other ranks
                vol.broadcast_files()

    #before file open callback
    def bfo_cb(s):
        if rank != 0:
            #other ranks receiving
            files from rank 0
            vol.broadcast_files()
    #setting the callbacks in the VOL plugin
    vol.set_after_file_close(afc_callback)
    vol.set_before_file_open(bfo_cb)
```
Listing 5 User action script provided by the user for enforcing the custom HDF5 I/O mechanism of Nyx.

AMReX (Zhang et al., 2019), a framework designed for massively parallel adaptive mesh refinement computations, serves as the PDE solver of Nyx simulation code, as well as providing I/O, writing the simulation data into a single HDF5 file. Reeber supports reading HDF5. As these user codes already use HDF5, no modifications were needed to execute them with Wilkins.

Ideally, a code utilizing parallel I/O would perform the following sequence of I/O operations. It would collectively create or open a file once from all MPI processes. This would be followed by some number of I/O operations, in parallel from all MPI processes. Eventually the file would be closed, again collectively from all MPI processes. LowFive is designed for this pattern, initiating the serving of data from a producer task to a consumer task upon the producer closing the file and the consumer opening the file. This assumes that the file close and file open occur exactly once, from all MPI processes in the task, as described above.

However, not all simulation codes perform I/O in this way, and Nyx is not the only code that violates this pattern. For various reasons—often related to poor I/O performance when accessing small amounts of data collectively—Nyx and other codes often employ patterns where a single MPI process creates or opens a file, performs small I/O operations from that single process, closes the file, and then all MPI processes re-open the file collectively for bulk data access in parallel. The file is opened and closed twice, the first time by a single MPI process, and the second time by all the processes in the task.

Such custom I/O patterns, which vary from one code to another, break the assumptions in LowFive about when and how

10.3389/fhpcp.2024.1472719

```
tasks:
  - func: nyx
    nprocs: 1024
    actions: ["actions", "nyx"]
    outports:
      - filename: plt*.h5
        dsets:
          - name: /level_0/density
            file: 0
            memory: 1
  - func: reeber
    nprocs: 64
    inports:
      - filename: plt*.h5
        io_freq: 2 #Setting the some flow
        control strategy
        dsets:
          - name: /level_0/density
            file: 0
            memory: 1
```

Listing 6 Sample YAML file for describing the cosmology workflow.

to serve data from producer to consumer. Fortunately, there is an elegant solution to incorporating custom I/O actions such as above. We added to the LowFive library custom callback functions at various execution points such as before and after file open and close. The user can program custom actions into those functions, e.g., counting the number of times a file is closed and delaying serving data until the second occurrence. In Wilkins, those custom functions are implemented by the user in a separate Python script, so that the user task code is unaffected.

Listing 5 shows the custom functions used in this cosmology use case, where there are two custom callback functions at after file close (*afc_cb*) and before file open (*bfo_cb*). In the after file close callback, process 0 broadcasts the data to other processes at the first file close, and serves data to the consumer at the second time, while other processes serve data upon (the one and only) file close. In the before file open callback, all processes other than 0 receive the data from process 0.

The user provides this script for custom actions and indicates it in the YAML file by setting the optional actions field with the name of this script file and the defined custom user function (i.e., *actions*:[“*actions*”, “*nyx*”]). Listing 6 shows the configuration file for this cosmology workflow.

Depending on the timestep, number of MPI processes, number of dark matter particles, number of halos, and density cutoff threshold, Reeber can take longer to analyze a timestep than Nyx takes to compute it. To prevent idling of Nyx and wasting computational resources while waiting for Reeber, we make use of the flow control strategies in Wilkins.

In these experiments, we use 1,024 processes for Nyx and 64 processes for Reeber. The Nyx simulation has a grid size of 256^3, and it produces 20 snapshots to be analyzed by Reeber. For this experiment we intentionally slowed Reeber down even further by computing the halos a number of times (i.e., 100), making the effect of flow control readily apparent. This allowed us to run Nyx with a smaller number of processes and for a shorter period of time, saving computing resources. We employ two different flow control strategies: (i) *all*: Nyx serving data at every timestep and (ii) *some*: Nyx serving data at every n timesteps, in this case we vary n as $n = 2$, $n = 5$, and $n = 10$. Table 3 shows the completion time of the workflow under these different strategies. Similarly to the synthetic experiments, we observe that using the *some* flow control strategy results in up to 7.7× time savings compared with the *all* strategy.

TABLE 3 Completion time for the cosmology workflow under different flow control strategies.

Strategy	Completion time (s)
All	5,421
Some ($n = 2$)	2,754
Some ($n = 5$)	1,084
Some ($n = 10$)	702

In terms of ease-of-use, we added one *actions* line and one *io_freq* line to the vanilla YAML configuration file in order to take advantage of custom callbacks and flow control, and made no changes to Nyx or Reeber source code in order to work with Wilkins. The only other required user file is the action script, which is a short Python code consisting of <25 lines.

5 Conclusion

We have introduced Wilkins, an *in situ* workflow system designed with ease-of-use in mind for addressing the needs of today’s scientific campaigns. Wilkins has a flexible data-centric workflow interface that supports the definition of several workflow topologies ranging from simple linear workflows to complex ensembles. Wilkins provides efficient communication of scientific tasks through LowFive, a high-performance data transport layer based on the rich HDF5 data model. Wilkins also allows users to define custom I/O actions through callbacks to meet different requirements of scientific tasks. Wilkins provides a flow control mechanism to manage tasks with different data rates. We used both synthetic benchmarks and two representative science use cases in materials science and cosmology to evaluate these features. The results demonstrated that Wilkins can support complex scientific workflows with diverse requirements while requiring no task code modifications.

Several avenues remain open for future work. Currently, Wilkins uses a static workflow configuration file, and cannot respond to dynamic changes in the requirements of scientific tasks during execution. We are currently working on extending Wilkins to support dynamic workflow changes such as adding/removing tasks from a workflow and redistribution of resources among workflow tasks. Here, we will also investigate whether using external services such as Mochi (Ross et al., 2020) and Dask (Rocklin, 2015) services would help Wilkins enable workflow dynamics. We are also collaborating closely with domain scientists to engage Wilkins in more science use cases. In particular, we are exploring use cases that couple HPC and AI applications, which can further demonstrate the usability of Wilkins in heterogeneous workflows.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

OY: Conceptualization, Data curation, Methodology, Software, Writing – original draft. DM: Conceptualization, Writing – review & editing. AN: Writing – review & editing. BN: Writing – review & editing. TP: Conceptualization, Funding acquisition, Writing – review & editing.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract numbers DE-AC02-06CH11357 and DE-AC02-05CH11231, program manager Margaret Lentz.

Acknowledgments

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.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

ADIOS2 (2024). *ADIOS2 HDF5 API Support Through VOL*. Available at: https://adios2.readthedocs.io/en/latest/ecosystem/h5vol.html (accessed April 15, 2024).

Almgren, A. S., Bell, J. B., Lijewski, M. J., Lukić, Z., and Van Andel, E. (2013). Nyx: a massively parallel amr code for computational cosmology. *Astrophys. J.* 765:39. doi: 10.1088/0004-637X/765/1/39

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., et al. (2015). "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), 25–29. doi: 10.1145/2828612.2828624

Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., et al. (2016). "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), 40–44. doi: 10.1109/ISAV.2016.013

BlueBrain (2022). *HighFive - HDF5 header-only C++ Library*. Available at: https://github.com/BlueBrain/HighFive (accessed April 15, 2024).

Boyuka, D. A., Lakshminarasimham, S., Zou, X., Gong, Z., Jenkins, J., Schendel, E. R., et al. (2014). "Transparent *in situ* data transformations in adios," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (IEEE), 256–266. doi: 10.1109/CCGrid.2014.73

Brace, A., Yakushin, I., Ma, H., Trifan, A., Munson, T., Foster, I., et al. (2022). "Coupling streaming ai and hpc ensembles to achieve 100-1000× faster biomolecular simulations," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE), 806–816. doi: 10.1109/IPDPS53621.2022.00083

Chan, H., Cherukara, M. J., Narayanan, B., Loeffler, T. D., Benmore, C., Gray, S. K., et al. (2019). Machine learning coarse grained models for water. *Nat. Commun.* 10:379. doi: 10.1038/s41467-018-08222-6

Collette, A. (2013). *Python and HDF5: Unlocking Scientific Data*. O'Reilly Media, Inc.

Dayal, J., Bratcher, D., Eisenhauer, G., Schwan, K., Wolf, M., Zhang, X., et al. (2014). "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), 246–255. doi: 10.1109/CCGrid.2014.104

Docan, C., Parashar, M., and Klasky, S. (2012). Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Comput.* 15, 163–181. doi: 10.1007/s10586-011-0162-y

Dorier, M., Antoniu, G., Cappello, F., Snir, M., Sisneros, R., Yildiz, O., et al. (2016). Damaris: addressing performance variability in data management for post-petascale simulations. *ACM Transact. Parallel Comp.* 3:15. doi: 10.1145/2987371

Dorier, M., Wang, Z., Ayachit, U., Snyder, S., Ross, R., and Parashar, M. (2022). "Colza: enabling elastic *in situ* visualization for high-performance computing simulations," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE), 538–548. doi: 10.1109/IPDPS53621.2022.00059

Dreher, M., and Raffin, B. (2014). "A flexible framework for asynchronous *in situ* and in transit analytics for scientific simulations," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (IEEE), 277–286. doi: 10.1109/CCGrid.2014.92

Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 36–47. doi: 10.1145/1966895.1966900

Friesen, B., Almgren, A., Lukić, Z., Weber, G., Morozov, D., Beckner, V., et al. (2016). *In situ* and in-transit analysis of cosmological simulations. *Comp. Astrophys. Cosmol.* 3, 1–18. doi: 10.1186/s40668-016-0017-2

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., De Supinski, B. R., et al. (2015). "The spack package manager: bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. doi: 10.1145/2807591.2807623

Gettelman, A., Liu, X., Barahona, D., Lohmann, U., and Chen, C. (2012). Climate impacts of ice nucleation. *J. Geophys. Res.* 117:17950. doi: 10.1029/2012JD017950

Golaz, J.-C., Van Roekel, L. P., Zheng, X., Roberts, A. F., Wolfe, J. D., Lin, W., et al. (2022). The doe e3sm model version 2: overview of the physical model and initial model evaluation. *J. Adv. Model. Earth Syst.* 14:e2022MS003156. doi: 10.1029/2022MS003156

Greer, A. (2016). Overview: application of heterogeneous nucleation in grain-refining of metals. *J. Chem. Phys.* 145:211704. doi: 10.1063/1.4968846

Gulli, A., and Pal, S. (2017). *Deep Learning With Keras*. Packt Publishing Ltd.

Guo, H., Peterka, T., and Glatz, A. (2017). "*In situ* magnetic flux vortex visualization in time-dependent ginzburg-landau superconductor simulations," in *2017 IEEE Pacific Visualization Symposium (PacificVis)* (IEEE), 71–80. doi: 10.1109/PACIFICVIS.2017.8031581

Hudson, S., Larson, J., Navarro, J.-L., and Wild, S. M. (2021). libEnsemble: a library to coordinate the concurrent evaluation of dynamic ensembles of calculations. *IEEE Transact. Parall. Distrib. Syst.* 33, 977–988. doi: 10.1109/TPDS.2021.3082815

Kay, J. E., Deser, C., Phillips, A., Mai, A., Hannay, C., Strand, G., et al. (2015). The community earth system model (CESM) large ensemble project: a community resource for studying climate change in the presence of internal climate variability. *Bull. Am. Meteorol. Soc.* 96, 1333–1349. doi: 10.1175/BAMS-D-13-00255.1

Krishna, J. (2020). *Scorpio—Parallel i/o Library*. Available at: https://e3sm.org/scorpio-parallel-io-library/ (accessed April 15, 2024).

Kuhlen, T., Pajarola, R., and Zhou, K. (2011). "Parallel *in situ* coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*.

Meyer, L., Schouler, M., Caulk, R. A., Ribés, A., and Raffin, B. (2023). "High throughput training of deep surrogates from large ensemble runs," in *SC 2023-The International Conference for High Performance Computing, Networking, Storage, and Analysis* (ACM), 1–14.

Morozov, D., and Lukic, Z. (2016). "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), 285–288. doi: 10.1145/2907294.2907301

Nicolae, B. (2020). "DataStates: towards lightweight data models for deep learning," in *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference* (Nashville, TN), 117–129.

Nicolae, B. (2022). "Scalable multi-versioning ordered key-value stores with persistent memory support," in *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium* (Lyon), 93–103. doi: 10.1109/IPDPS53621.2022.00018

Nigmetov, A., and Morozov, D. (2019). "Local-global merge tree computation with local exchanges," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–13. doi: 10.1145/3295500.3356188

Peterka, T., Morozov, D., Nigmetov, A., Yildiz, O., Nicolae, B., and Davis, P. E. (2023). "Lowfive: *in situ* data transport for high-performance workflows," in *IPDPS'23: The 37th IEEE International Parallel and Distributed Processing Symposium*. doi: 10.1109/IPDPS54959.2023.00102

Plimpton, S., Crozier, P., and Thompson, A. (2007). Lammps-large-scale atomic/molecular massively parallel simulator. *Sandia Natl. Lab.* 18:43.

Rew, R. K., Ucar, B., and Hartnett, E. (2004). "Merging NetCDF and HDF5," in *20th Int. Conf. on Interactive Information and Processing Systems*.

Rocklin, M. (2015). "Dask: parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference* (Citeseer), 130–136.

Ross, R. B., Amvrosiadis, G., Carns, P., Cranor, C. D., Dorier, M., Harms, K., et al. (2020). Mochi: composing data services for high-performance computing environments. *J. Comput. Sci. Technol.* 35, 121–144. doi: 10.1007/s11390-020-9802-0

Schouler, M., Caulk, R. A., Meyer, L., Terraz, T., Conrads, C., Friedemann, S., et al. (2023). Melissa: coordinating large-scale ensemble runs for deep learning and sensitivity analyses. *J. Open Source Softw.* 8:5291. doi: 10.21105/joss.05291

Wozniak, J. M., Armstrong, T. G., Wilde, M., Katz, D. S., Lusk, E., and Foster, I. T. (2013). "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), 95–102. doi: 10.1109/CCGrid.2013.99

Yildiz, O., Dreher, M., and Peterka, T. (2022). "Decaf: decoupled dataflows for *in situ* workflows," in *In Situ Visualization for Computational Science* (Springer), 137–158. doi: 10.1007/978-3-030-81627-8_7

Yildiz, O., Ejarque, J., Chan, H., Sankaranarayanan, S., Badia, R. M., and Peterka, T. (2019). Heterogeneous hierarchical workflow composition. *Comp. Sci. Eng.* 21, 76–86. doi: 10.1109/MCSE.2019.2918766

Zhang, W., Almgren, A., Beckner, V., Bell, J., Blaschke, J., Chan, C., et al. (2019). AMReX: a framework for block-structured adaptive mesh refinement. *J. Open Source Softw.* 4:1370. doi: 10.21105/joss.01370