BeeFlow: a Workflow Management System for In situ Processing Across HPC and Cloud Systems

Jieyang Chen*, Qiang Guan‡, Zhao Zhang◇, Xin Liang*,
Louis James Vernon†, Allen McPherson†, Li-Ta Lo†, Patricia Grubel†, Tim Randles†,
Zizhong Chen*, and James Paul Ahrens†
*University of California, Riverside
‡ Kent State University
◇Texas Advanced Computing Center
†Los Alamos National Laboratory
{jchen098, xlian007, chen}@cs.ucr.edu,
qguan@kent.edu,
zzhang@tacc.utexas.edu,
{vernon, mcpherson, ollie, pagrubel, trandles, ahrens}@lanl.gov

***Abstract*—In this paper, we propose `BeeFlow` – an in situ analysis processing workflow management system across multiple platforms using Docker containers. `BeeFlow` can support both traditional workflows as well as workflows with in situ analysis. `BeeFlow` leverages Docker containers to provide a portable, flexible, and reproducible workflow management system across HPC and cloud platforms. We showcase how current in situ visualization workflows can apply `BeeFlow` with DOE production codes VPIC and Flecsale.**

***Keywords*-scientific workflow; high performance computing; cloud computing; container.**

I. Introduction

Modern scientific simulation and big data analytics workflows require sharing large amount of data between tasks. The data are usually stored in filesystems by producer tasks and later read out by consumer tasks. However, with increasing demands of solving more complicated problems and processing larger amounts of input data, sharing data between tasks via filesystems is inefficient. For example, simulation-analysis based workflow is one commonly used workflow in scientific computing. It basically comprises two parts: (1) simulation tasks, that are responsible for running simulations and dumping simulation output data and (2) analysis tasks, that are responsible for loading and analyzing simulation data, then present results to end users (e.g., statistic results, visualization images, etc.). In traditional simulation workflows (also known as pure offline dependency workflows [1]), analysis tasks need to wait for all simulation tasks to complete dumping all data to filesystems before processing data. However, as simulation problems become more complicated, the simulation data can reach hundreds of terabytes to petabytes. Storing and loading such large amounts of data on disks can significantly degrade overall performance of user's workflows and cause heavy burdens on filesystems. Recently, in situ analysis [1] was proposed to allow analysis tasks to run with simulation tasks side by side. It offers two benefits over traditional offline workflow: First, as simulations progress, they can send the simulation data to analysis tasks for real-time processing and output results simultaneously. The data can be transferred via either small-sized temporary or permanent data files on shared file systems or network communication between processes, which can greatly mitigate the heavy burdens on filesystems. Second, in situ workflow enables users to identify potential problems as the current simulations making processes, so that users can reconfigure, adjust, and restart simulations immediately to save time, get flexible real-time simulation control, or filter out less relevant data as needed to save computing resources. For instance, vector particle-in-call (VPIC) [2], [3], [4] and Flecsale [5] are two Department of Energy (DOE) simulation codes that are designed specifically for in situ workflow with real-time visualization using ParaView, so that users can make real-time decisions based on the simulation progress.

However, it is difficult to manually control the process of launching and debugging large-scale workflows that consist of hundreds of tasks with different kinds of dependency modes. Many workflow management systems have been developed to ease this complication by automating the launching process. Among all workflow management systems [6], [7], [8], [9], [10], to the best of our knowledge, none of them natively support launching workflows with in situ dependencies between tasks. To launch workflow with in situ analysis, current users need to manually launch each task and configure in situ dataflow between tasks. In most of the cases, tasks with in situ dependencies need to be launched in a time sensitive manner simultaneously, causing considerable difficulties to

[1]The publication has been assigned the LANL identifier LA-UR-17-27303.

current users if some tasks require complicated launching processes. The manual approach also costs users significant time and effort while manually re-launching the whole or partial workflow for debugging or reproducing experiments.

In this paper, we propose an in situ analysis enabled workflow management system – BeeFlow. Specifically, the main contributions of BeeFlow include:

- **In situ support:** BeeFlow supports in situ dependencies between tasks in addition to traditional offline dependencies. It allows users to define both in situ workflows [1] and offline workflows;
- **In situ dataflow and control design:** BeeFlow is designed to support most kinds of dataflow between tasks in modern in situ workflows, including filesystem-based sharing and network-based sharing. It also supports backward simulation control from analysis tools;
- **Container support:** BeeFlow is built based on the Build and Execution Environment (BEE) [11], a Docker container supported universal execution framework. So, BeeFlow uses Docker images for user application development. This is especially beneficial when deploying large-scale workflows on production systems because it can save considerable time and effort for application configurations and minimizes compatibility issues on target systems. Also, BEE-atified HPC application containers ensure BeeFlow can run on most production HPC and cloud computing environments;
- **Multiple platforms support:** Benefiting from the virtualized Docker environment, BeeFlow can be easily configured to run on multiple platforms and easily switched between HPC and cloud platforms (e.g., Amazon Web Services (AWS)) when needed.

We conduct evaluations that show BeeFlow has usability and performance similar to current scientific workflow management systems. We also showcase three scientific workflows using BeeFlow. The three workflows cover both in situ and offline workflows with two kinds of dataflow modes.

The rest of this paper is organized as follows: we introduce backgrounds in section II and give formal dependency definitions in section III. In section IV, we discuss the design details of BeeFlow. Case study and evaluation are discussed in section V. In section VI, we discuss other large scale scientific workflow management tools and how BeeFlow differs. Finally, we conclude in section VII.

II. Backgrounds

A. BEE

BEE [11] is a Docker-based containerization environment that enables HPC applications to run on both HPC and cloud computing platforms. BEE provides a unified user interface for automatic job launching and monitoring. BEE users only need to wrap their application in a standard Docker image and provide a simple BeeFile (job execution environment description) to run on BEE. Since the same standard Docker environment is provided across platforms, no user application modification is necessary. In addition, BEE solves the security constrain of HPC environment that cannot be addressed with current Docker daemon. In this work, we build BeeFlow based on BEE, so it naturally inherits all benefits of BEE. This allows us to build a unified workflow management system across multiple platforms.

B. In situ Analysis

In traditional scientific workflows, data are usually shared via file systems between tasks. However, as we are aiming to solve more complicated problems, workflow data can reach hundreds of terabytes to petabytes. It can be inefficient to store large amounts of data on disks for extended periods. One solution to mitigate this problem is in situ analysis [1], allowing data consuming tasks to run along with the data producing tasks. As data producing tasks make progress, they can send the intermediate data to data consuming tasks for real-time processing and output results simultaneously. The data can be transfered either via smaller temporary or permanent data files on shared filesystems or via network communication between processes. This type of workflow enables users to identify potential problems during current simulations and further reconfigure, adjust, and restart simulations to save time; get flexible real-time simulation control; or filter out less relevant data as needed to save computing resources. In order to launch in situ workflows, current users need to manually launch each task, which introduces much complications. However, none of the current workflow management systems natively support in situ workflows, therefore in this work, we propose to design a workflow launcher with in situ analysis support.

III. Dependency Definitions

Before we discuss the design details of BeeFlow, we first formally define two types of dependency that are supported in BeeFlow. Although the offline dependency has been extensively studied and supported in previous works [6], [7], [8], [9], [10], we include a formal definition here for clarification and comparison with in situ dependency.

A. Offline dependency

When we have two tasks A and B with B (offline) depending on A, then B can only be launched after A has finished all its work. As shown in **Fig. 1**, logically a synchronization point must to be placed between the finishing point of task A and the starting point of task B to enforce this dependency.

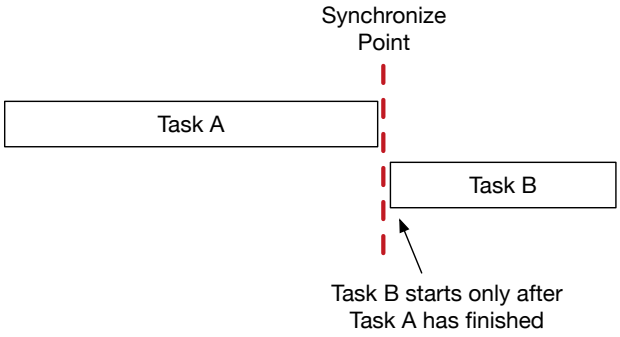


Fig. 1: Example of offline dependency

B. In situ dependency

When we have two tasks A and B with B (in situ) depending on A, then B can be launched only after A has started. As shown in **Fig. 2**, logically a synchronization point need to be placed in the beginning of task A to enforce this dependency. The amount of time that task B needs to wait after task A starts can be defined according to application requirements.

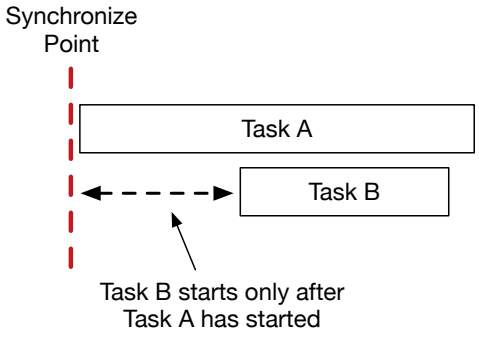


Fig. 2: Example of in situ dependency

IV. Design

Design of a scientific workflow management system usually follows the five-layer functional architecture model [6], [7], [8], [9], [10]. The five layers are:

- **Presentation Layer**: This layer is responsible for gathering workflow information from the user and providing configuration capabilities.
- **User Services Layer**: This layer is responsible for providing functionality (i.e., monitoring workflow status, etc.) to users.
- **WEP Generation Layer**: This layer is responsible for generating the Workflow Execution Plan (WEP) based on the workflow design provided by the user in the presentation layer.
- **WEP Execution Layer**: This layer is responsible for scheduling workflow tasks within the WEP.
- **Infrastructure Layer**: This layer is responsible for managing underlaying infrastructures and executing workflow tasks on them.

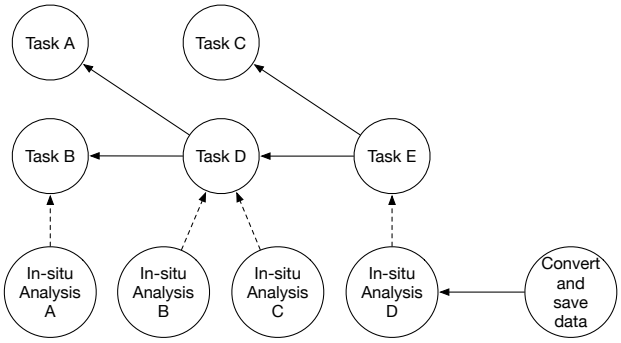


Fig. 3: Example of DAG with both offline and in situ dependency

A. Presentation Layer

Traditional offline-based dependency relations are usually represented using Directed Acyclic Graphs (DAGs). Tasks are represented as vertices and dependencies between them are represented as directed edges. We also choose DAG for dependency representation in `BeeFlow`. To incorporate in situ dependency into a DAG-based dependency representation, we extend DAG to use two types of directed edges to represent two types of dependencies: one for offline dependency (solid edges) and another for in situ dependency (dashed edges). We use a simplified example to show how the extended DAG represents workflow with both kinds of dependencies. In our example workflow, we have three different simulation tasks: `Task B`, `Task D`, and `Task E`. One simulation's input depends on another, so they have to be executed in order (i.e., offline dependency). Also, `Task D` and `Task E` need additional input, which need to be per-processed by `Task A` and `Task C` first (i.e., offline dependency). Meanwhile, we also want to monitor the progress of each simulation with in situ analysis tools. `Task D` also has two separate in situ analysis tasks monitoring two different parts of the output of `Task D`. Finally, we want to convert and save the in situ analysis data for `Task E` after its in situ analysis task (`In-situ Analysis D`) has finished. This workflow mixes both traditional offline dependencies and in situ dependencies. We can represent its dependencies using our extended DAG as in **Fig. 3**. Solid edges represent offline dependency and dashed edges represent in situ dependency.

DAGs are usually represented as a series of dependencies in workflow configuration files, the input files of workflow management tools. To represent both offline and in situ dependencies, we add an additional flag to indicate which mode is used. In our `Beeflow` management system, the description

file is defined using JSON format. For example, the DAG of our example workflow is represented as follows:

```
{
  "Task A": {
    "dependency_list":[]
  },
  "Task B": {
    "dependency_list": []
  },
  "Task C":{
    "dependency_list": []
  },
  "Task D":{
    "dependency_list": ["Task A", "Task B"],
    "dependency_mode": "Offline"
  },
  "Task E":{
    "dependency_list": ["Task C", "Task D"],
    "dependency_mode": "Offline"
  },
  "In-situ Analysis A":{
    "dependency_list": ["Task B"],
    "dependency_mode": "In-situ"
  },
  "In-situ Analysis B":{
    "dependency_list": ["Task D"],
    "dependency_mode": "In-situ"
  },
  "In-situ Analysis C":{
    "dependency_list": ["Task D"],
    "dependency_mode": "In-situ"
  },
  "In-situ Analysis D":{
    "dependency_list": ["Task E"],
    "dependency_mode": "In-situ"
  },
  "Convert and save data":{
    "dependency_list": ["In-situ Analysis E"],
    "dependency_mode": "Offline"
  }
}
```

B. User Services Layer

In this section, we discuss the design details of how we build `BeeFlow` on top of `BEE` to provide workflow launching and monitoring features for users.

When using the original `BEE` execution framework, users first need to start the `BEE Orchestration Controller` in the background. The orchestration controller is responsible for handling task launching requests. To launch tasks, users need to use the `BEE launcher` to send task launching requests to the `BEE Orchestration Controller`. However, the original design only allows one task to be launched at a time. To launch a multi-task workflow, users need to manually launch each task and carefully handle complicated dependencies between them.

To enable workflow launching features, we need to extend `BEE` to launch multiple tasks. The simplest and most straightforward design is to modify the `BEE launcher`, allowing it to group multiple task launching requests into one. Since the `BEE Orchestration Controller` was originally designed to handle one task at a time, tasks in a multiple task launching request will still be handled one by one, sequentially. So, it would significantly degrade launching efficiency, especially for tasks that will pull large Docker images. On the other hand, a more efficient way to enable multiple task support is to handle multiple task launching requests simultaneously, so they will not interfere or block each other. Inspired by the commonly used web server design, in which each client's request is handled by an individual thread, we modify `BEE Orchestration Controller` to handle multiple task launching requests following similar design principles.

The launching workflow of our extended `BEE Orchestration Controller` is as follows: First, when the `BEE Orchestration Controller` receives task launching requests and related task information, instead of starting the launching process immediately on the main thread, the `BEE Orchestration Controller` now creates several new `BEE` task threads and stores all related task information to the thread object including status indicator variables for querying their launching and execution progress later. Then, BEE task threads are launched. The `BEE Orchestration Controller` uses a hash table to store all task threads with the task name as the key for locating tasks quickly. At anytime, users can launch a separate monitoring tool to keep track of the status of all the tasks they are running. In addition, as we will discuss in later sections, we install event-based listeners to each `BEE` task thread to enforce task dependencies during task launching. The overall framework of our extended BEE Orchestration Controller is shown in **Fig. 4**. We create a special launcher, `BeeFlow Composer`, dedicated for launching workflows with multiple tasks.

C. WEP Generation Layer

The next step is to generate WEPs, which include all task-related preparation before actual execution. Specifically, besides creating `BEE` task threads as mentioned in the previous section, we also need to provision each `BEE` task thread to enforce the workflow logic specified by users while launching tasks.

First, we need to get the task launching order from our previously defined JSON formatted workflow dependency description file. For traditional pure offline dependency mode, we can get the task launching order using Topology Sort from a series of dependencies. To incorporate both offline and in

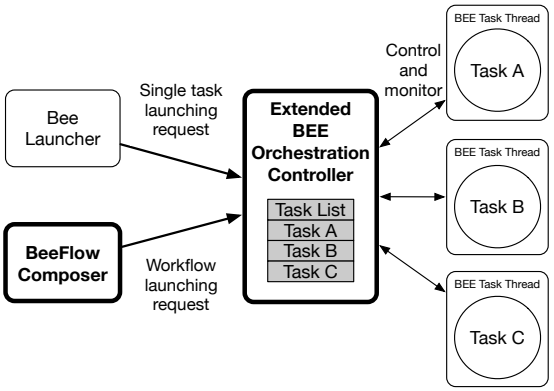


Fig. 4: Extended BEE Orchestration Controller with multiple task handling capabilities

situ dependency mode, we find that it is hard to use Topology Sort, since in situ dependency mode allows both tasks to be started at the same time and cannot be described as a series of launching orders. Instead we propose to use primitive synchronization elements in Python to enforce this kind of mixed launching order. During the creation of each `BEE` task thread, we create two Python event objects: one for sending out signals when the task starts and another for sending out signals when the task finishes. Other tasks can listen for these signals to enforce dependencies.

Algorithm 1 shows the algorithm used to add Python event objects for enforcing workflow logic. For each task in the `Beeflow` workflow description file, we check if it is in some other tasks' dependency list. If so, the dependent task is set to wait for the event signals from those tasks according to the dependency type.

Algorithm 1 `Beeflow` launching logic

```
Require: Task List (TL)
for task in TL do
    create two events: start_event, end_event;
    for task2 in TL do
        if task in task2.dependency_list then
            if task2.dependency_mode == "in situ" then
                task2.add_wait_for (start_event);
            end if
            if task2.dependency_mode == "offline" then
                task2.add_wait_for (end_event);
            end if
        end if
    end for
end for
```

Fig. 5 shows how Python event objects are added to our previous workflow example to enforce launching orders. Each rectangle represents a simplified launching source code of each task. $S*$ and $E*$ represent event objects for starting and ending the current task. The `wait for` function will block the current task's launching process until all of the events it is waiting for have received signals. Notice `Task A` to `Task E` are on the offline workflow, so each one of them needs to wait for the end signal sent from certain tasks. The four in situ analysis processes are in in situ dependency mode, so they wait for the start signals of corresponding simulation tasks. Finally, `Convert and save data` in the end must wait for the completion of `In-situ Analysis D`; so, it waits for its end signal.

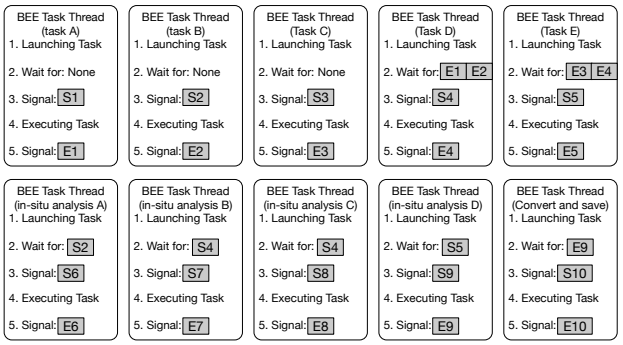


Fig. 5: Example of applying events synchronization primitive to enforce workflow logics

D. WEP Execution Layer

Once task threads are constructed and configured with dependency enforcement event objects, they are ready for launching. The launching is handled by `BEE`. Depending on users' configuration, workflow tasks can run on HPC systems or AWS platforms. Since events objects are integrated into each task, no modification is necessary to `BEE` to run workflow-related tasks, and `BEE` is not aware of the dependencies between tasks. So, `BEE` schedules `BeeFlow` tasks the same as regular tasks (i.e., as if no dependencies exist) and workflow dependencies can be enforced by each task. This design simplifies the launching process and eliminates overhead brings by task progress monitoring for dependencies enforcement purposes as in other workflow tools.

E. Infrastructure Layer

1) Execution: Basically, each task is executed by `BEE` on different platforms specified by users. To provide a unified environment across platforms, `BEE` uses a Docker container as it's execution environment. On HPC platforms, `BEE` uses its universal back-end `BEE-VM` to run Docker. On AWS, `BEE` can directly use BOTO API to launch pre-built Docker-enabled instances then run the user applications within the Docker container.

2) DataFlow Design: In workflow, data are usually produced by some tasks and then consumed by other tasks. This is the reason dependencies exist. For in situ analysis, the analysis tools sometimes control the simulation processes. In this section, we discuss the dataflow design in `BeeFlow` to support these functionalities.

First, we design the traditional disk-based dataflow between tasks in `BeeFlow` where one task writes output to files on disk and another reads the files as input data. We assume file names are predefined in both tasks in order to allow data sharing between tasks, we need to make sure both tasks can access the same working directory or a synchronized directory. In `BEE`, the task applications run in Docker containers, and data is mounted with two steps: (1) a shared local directory (on HPC systems) or shared EFS (on AWS) is mounted to a pre-configured directory in BEE-VM or BEE-AWS. (2) the pre-configured directory is mounted into the working directory in the Docker container. In order to share the same working directory between two dependent tasks, we only need to ensure that they are mounting the same local directory or EFS, which can be configured before launch in `BEE`.

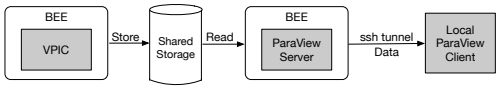
The second kind of dataflow is socket-based dataflow, in which data is transfered between tasks using TCP/UDP sockets. As a simulation task progresses, it sends out data to in situ analysis tools for processing. Later, as shown in the Flecsale workflow of our case study, as the simulation makes progress and sends out data, users can employ local ParaView clients to visualize simulation progression. Socket connections can be easily established when user tasks are running on bare metal systems. However, when using `BEE`, the execution environment is isolated between tasks. Modification to the network configuration of the two-layer virtulization is difficult. So instead we propose to use the existing SSH connections to establish tunnel connections between tasks, since it requires minimum modification to `BEE` and avoids all complicated configurations by users. Moreover, the tunnel connection not only allows data transfer from simulation application to in situ analysis tools, it also allows backward simulation control from the in situ analysis tool (e.g., ParaView Catalyst [12], [13]).

V. Case Study

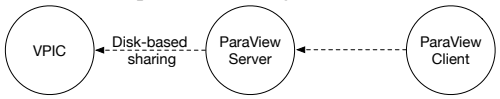
In this section, we first introduce three commonly used scientific workflows that vary from in situ workflow to offline workflow. Then, we evaluate and compare several different workflow launching approaches by trying to adopt the three workflows.

A. Workflows in Case Study

1) VPIC: Vector Particle-In-Cell (VPIC) is a general purpose particle-in-cell simulation code for modeling kinetic plasmas in multiple spatial dimensions [2], [3], [4]. In our customized VPIC workflow, we perform turbulence simulation using VPIC with in situ analysis using ParaView. ParaView [12], [13] is an open-source, multi-platform data analysis and visualization application. ParaView users can quickly build visualizations to analyze their data using qualitative and quantitative techniques. The ParaView tool in our workflow has two parts: ParaView Server and ParaView Client. ParaView Server runs on the same computing system as the scientific application and is responsible for gathering and rendering visualization data from VPIC. ParaView Client runs on the users' local machines and is responsible for displaying visualized scientific data by connecting with ParaView Server.

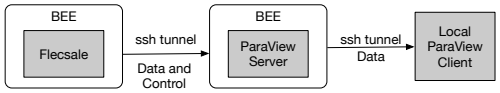


(a) Components and logic in VPIC workflow

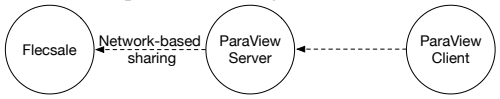


(b) DAG dependency graph in VPIC workflow

Fig. 6: VPIC workflow



(a) Components and logic in Flecsale workflow



(b) DAG dependency graph in Flecsale workflow

Fig. 7: Flecsale workflow

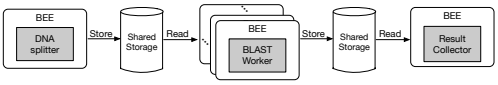
In our customized VPIC workflow, we use ParaView Server to perform disk-based in situ analysis, then, via the network, transfer visualized results to the local ParaView Client. The overall workflow is shown in **Fig. 6a**. The abstracted DAG is represented as **Fig. 6b**.

2) Flecsale: Flecsale [5] is another workflow similar to the previous VPIC workflow. Flecsale is a computer software package developed for studying problems that can be characterized using continuum dynamics, such as fluid flow. In this workflow, we build Flecsale with Catalyst so that it can enable network-based in situ analysis with simulation control from ParaView. The connection is established using the previously mentioned SSH tunnel. The workflow is shown in **Fig. 7a**. We construct the DAG for this Flecsale workflow as in **Fig. 7b**.

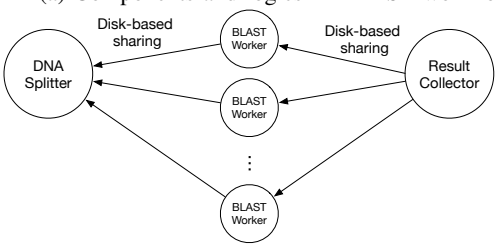
3) BLAST: In our third workflow evaluation, we choose a traditional BLAST workflow for DNA sequence matching. BLAST is a biological application aimed to find similarities between biological sequences [14]. In the BLAST workflow, first a DNA splitter divides the large-sized input DNA

sequences into several partitions. Then, it launches several BLAST workers to work on one partition each. Finally, when all workers have done their job, a result collecting worker gathers all the results from each worker and merges them into one output file. The workflow is shown in **Fig. 8a**.

The dependencies in this workflow are all offline. The DNA splitter has no depending task, so it can start as soon as we start the workflow. All workers must wait for the splitter, so this is an offline dependency. The result collector needs to wait for all workers, so this is also an offline dependency. All intermediate data are shared through files on disk, so all dependencies are using disk-based sharing. The DAG of this workflow is shown in **Fig. 8b**.



(a) Components and logics in BLAST workflow



(b) DAG dependency graph in BLAST workflow

Fig. 8: BLAST workflow

B. Evaluation Methodology

We evaluate `BeeFlow` by comparing it with a manual launch approach and another container-based workflow tool – MakeFlow [15]. The reason we compare `BeeFlow` with MakeFlow instead of other workflow management tools is that MakeFlow has the most similar functionalities to `BeeFlow`. We use MakeFlow in the default way as shown in their official use case examples [16]. We also compare to the manual approach, because it is the only method, other than `BeeFlow`, that we know can launch an in situ workflow. We focus on evaluating three aspects: in situ analysis support, usage complexity, and usage time cost.

The evaluation is conducted on our testbed cluster – Darwin. It has a 'Galton' node partition with KVM enabled. Each node has two 8-core Intel Ivy Bridge E5-2650 v2 processors with 251 GB RAM. For the cloud system, we use Amazon Web Services EC2. The results on AWS are similar to the HPC system results, so we omit results of AWS due to page space limit.

C. Evaluation Results

1) in situ Analysis Support: We first evaluate the in situ analysis support of different workflow launching approaches.

TABLE I: Usage Complexity Comparison for VPIC, Flecsale and BLAST Workflows

Approaches	BeeFlow	Manual	MakeFlow
VPIC Workflow			
Deploying Applications	6	88	N/A
Configuring Workflow	2	0	N/A
Launching Workflow	1	5	N/A
Flecsale Workflow			
Deploying Applications	6	86	N/A
Configuring Workflow	2	0	N/A
Launching Workflow	1	5	N/A
BLAST Workflow			
Deploying Applications	6	10	10
Configuring Workflow	3	0	m+2
Launching Workflow	1	m+2	1

Although MakeFlow is similar to `BeeFlow`, it does not natively support in situ analysis. It only supports offline dependency as specified in each line of its workflow description files. The manual approach brings the freedom to support any kind of dependencies, including in situ analysis. However, it also costs users considerable time and effort to manually launch each task following each dependency rule. `BeeFlow` natively supports in situ analysis workflow as well as offline workflow. Its automation feature minimizes the usage complexity and time cost. Both `BeeFlow` and the manual approach can launch VPIC, Flecsale, and BLAST workflows. MakeFlow, on the other hand, only supports launching the BLAST workflow.

2) Usage Complexity: We then compare the usage complexity of each approach. There are many aspects that affect the usage complexity of an approach. The usage complexity is affected by the total number of command lines that need to be specified by users and the complexity of constructing/-filling configurations files, the total number of configuration sections that need to be configured. So, we define the Usage Complexity as:

$$Usage\ Complexity = Number\ of\ Command\ Lines + Number\ of\ Configuration\ Sections$$

We evaluate the usage complexity of several main operations:

- **Deploying Applications**: Only needed when applications in the workflow are first used or are updated between tests;
- **Configuring Workflow**: Only needed when workflow is defined or updated;
- **Launching workflow**: Needs to be done for each test.

We first evaluate the usage complexity of lunching VPIC workflow using `BeeFlow` and the manual approach. **Table I** shows the usage complexity for launching the VPIC workflow on Darwin. For deploying applications, among the three applications, only VPIC and ParaView Server need to be configured. `BeeFlow` uses Docker for deployment, so it only needs

a few simple configurations in its job description files. The manual approach, on the other hand, requires users to build and configure applications from scratch, which introduces extra complication and efforts. On Darwin, it requires 60 commands to deploy ParaView Server and 28 commands to deploy VPIC. For configuring workflows, `BeeFlow` requires two commands for configuring its workflow description files. The manual approach does not need any workflow configuration. Finally, to launch the workflow, only one command is necessary for `BeeFlow`. Five commands are needed to launch manually.

Second, we compare the usage complexity of launching the Flecsale workflow using `BeeFlow` and the manual approach. As shown in **Table I**, similar to VPIC workflow, `BeeFlow` has much lower usage complexity than the manual approach for deploying applications with little extra workflow configurations.

Third, we compare the usage complexity of launching BLAST workflow using different approaches. Depending on the user's configuration on the number of workers in this workflow, the total number of tasks varies. **Table I** shows the usage complexity to launch BLAST workflow on Darwin, in which m represents the number of workers. For deploying the application, Docker-based `BeeFlow` has lower usage complexity than the manual approach and MakeFlow since they both need manual application deployment. The manual approach does not need workflow configuration, while MakeFlow needs $m + 2$ configuration lines (one for each task). `BeeFlow`, on the other hand, features automation that only needs one configuration for m worker, so it only needs three total configurations. Finally, due to lack of automation, the manual approach has the highest usage complexity when launching the workflow.

3) Usage Time Cost: In this section, we evaluate the time cost of launching each workflow using each approach. The timing results are shown in **Table II**. We time the deployment of each task in the three workflows on Darwin and the time cost for users to launch each workflow, including wait time due to task dependencies. The results show, due to the Docker container support, `BeeFlow` has the shortest application deployment time. This is especially beneficial if target applications require complex and time consuming configurations. For example, the ParaView tool, used by both VPIC and Flecsale workflows, requires at least 30 minutes to build and takes even longer when configured with more features enabled. On the other hand, `BeeFlow` only needs several minutes to pull Docker images. Although MakeFlow also supports Docker containers, the Docker daemon is usually not supported in current production HPC software stacks, including our testbed Darwin. So, we still need to manually deploy each application when using MakeFlow.

To launch each workflow, both `BeeFlow` and MakeFlow feature automation, so only several seconds are needed to initialize workflow tools and launch each workflow (users do not need to wait). For the manual approach, users need to wait for each task to become ready, and then launch the next task. For example, in the VPIC workflow, users need to first start the VPIC simulation manually, then wait for 34 seconds before they can launch ParaView for in situ analysis. The lack of automation can cost users extra time.

TABLE II: Time cost of launching VPIC, Flecsale and BLAST Workflows

Approaches	BeeFlow	Manual	MakeFlow
VPIC Workflow			
Deploying Applications	2m25s	78m31s	N/A
Launching Workflow	3s	34s	N/A
Flecsale Workflow			
Deploying Applications	2m5s	36m20s	N/A
Launching Workflow	3s	1m5s	N/A
BLAST Workflow (two workers)			
Deploying Applications	40s	2m22s	2m22s
Launching Workflow	3s	4m44s	3s

VI. Related Work

In this section, we discuss and compare several commonly used scientific workflow management systems. Those have been intensively used by various fields, including biology, computational engineering, astronomy, and climate modeling. Most of them share similar five-layer functionality architecture, so that they have similar features. However, each of them still has some different designs and functionalities as discussed in each of the following subsections.

A. Pegasus

Pegasus [8] is a commonly used scientific workflow management system that has been adopted in many scientific research fields. Similar to `BeeFlow`, Pegasus features portability and stability on different infrastructures including HPC and cloud platforms. It also features, data transfer support across different execution partitions and fault-tolerance. On the other hand, since `BeeFlow` aims to support in situ analysis, either involving multiple application-specific file generation or network/in-memory data transfer, we choose to let users handle the file transfers between tasks or simply use our shared storage or SSH tunnel options. As for fault tolerance, many in situ analysis enabled scientific applications feature in-application fault tolerance or checkpoint/restoration, so we leave handling fault tolerance to the application. Moreover, in the presentation layer Pegasus uses XML-format DAG for user workflow input, on the other hand, `BeeFlow` uses JSON-formatted configuration files, providing similar usability. Finally, in the user service layer, both Pegasus and `BeeFlow` provide workflow status monitoring. However, Pegasus does

not support Docker images as user input, so potential software compatibility issues need to be handled by Pegasus users.

B. Kepler

Kepler is a scientific workflow management system based on the Kepler project [7], [17]. Like `BeeFlow`, Kepler can also utilize both HPC and cloud computing resources. Kepler also allows users to plug in different execution models into the workflow. This feature can potentially support in situ analysis dependencies. However, it requires intensive development effort, and special dataflow in in situ workflow can be hard to adopt. For the presentation layer, Kepler integrates a graphical workbench as the user interface. Each task in Kepler is called an actor, a highly reusable module for workflow design. After designing the workflow, Kelper uses another component named director to generate an executable workflow then launches the workflow with either static or dynamic scheduling [18], [19]. Kepler also implemented three kinds of system level fault tolerance mechanisms.

C. Taverna

Taverna [20] is a scientific workflow management system specifically designed for biological experiments. Like Kepler, Taverna has a graphical user interface in the presentation layer to help users design and input their workflow. This user interface also serves as a monitor for workflow status. Taverna can be installed on either users' local machines or web servers and can utilize both HPC and cloud computing resources. However, Taverna does not have the flexibility to integrate different execution models to support in situ workflows.

D. Chiron

Chiron [21] a data-intensive scientific workflow that exploits a database approach. In its presentation layer, it allows users to define workflows with an algebraic or SQL expression. Then each expression will be converted into six basic operations: Map, SplitMap, Reduce, Filter, SRQuery, and MRQuery. In the user services layer, workflow monitoring and steering features are provided. Like `BeeFlow`, Chiron can also utilize HPC and cloud platforms. The cloud computing part is done by Scicumulus [22], [23]. Similar to other workflow management systems we discussed, Chiron does not support Docker containers, thus users need to handle compatibility issues. in situ workflows are also not supported in Chiron.

VII. Conclusions

In this work, we first address the importance of the new in-situ analysis workflows. Next, we proposed `BeeFlow`, an in-situ analysis enabled workflow management system across HPC and cloud platforms with Docker support. We showed how we designed and optimized `BeeFlow` in the five-layer functionality architecture and evaluated the usability and performance. Moreover, we showcased three commonly used scientific workflows on `BeeFlow`. Finally, we compared `BeeFlow` with current existing workflow systems in multiple aspects and show that `BeeFlow` can be easily adopted to launch modern in-situ workflows in our case studies. Comparisons also showed that `BeeFlow` brings much better usage complexity and user time cost compared with manual approach and similar results compared to existing commonly used workflow tools.

References

[1] C. Sewell, K. Heitmann, H. Finkel, G. Zagaris, S. T. Parete-Koon, P. K. Fasel, A. Pope, N. Frontiere, L.-t. Lo, B. Messer *et al.*, "Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[2] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[3] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation a," *Physics of Plasmas*, 2008.

[4] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," in *Journal of Physics: Conference Series*, 2009.

[5] "Flecsale," Available on-line at: https://github.com/laristra/flecsale.

[6] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, 2015.

[7] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. Mock, "Kepler: Towards a grid-enabled system for scientific workflows," in *the Workflow in Grid Systems Workshop in GGF10-The Tenth Global Grid Forum, Berlin, Germany*, 2004.

[8] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, 2005.

[9] E. Ogasawara, J. Dias, V. Silva, F. Chirigati, D. Oliveira, F. Porto, P. Valduriez, and M. Mattoso, "Chiron: a parallel engine for algebraic scientific workflows," *Concurrency and Computation: Practice and Experience*, 2013.

[10] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Services, 2007 IEEE Congress on*, 2007.

[11] "Bee," Available on-line at: https://github.com/lanl/bee.

[12] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *The Visualization Handbook*, 2005.

[13] U. Ayachit, "The paraview guide: a parallel visualization application," 2015.

[14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, 1990.

[15] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012.

[16] "Makeflow examples," URL: https://github.com/cooperative-computing-lab/makeflow-examples.

[17] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, 2004.

[18] M. Bux and U. Leser, "Parallelization in scientific workflow management systems," *arXiv preprint arXiv:1303.7195*, 2013.

[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, 2006.

[20] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble, "Taverna, reloaded," in *International conference on scientific and statistical database management*, 2010.

[21] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[22] D. de Oliveira, E. Ogasawara, F. Baião, and M. Mattoso, "Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 2010.

[23] D. Oliveira, E. Ogasawara, K. Ocaña, F. Baião, and M. Mattoso, "An adaptive parallel execution strategy for cloud-based scientific workflows," *Concurrency and Computation: Practice and Experience*, 2012.