DYFLOW: A flexible framework for orchestrating scientific workflows on supercomputers

Swati Singhal
University of Maryland
swati@cs.umd.edu

Alan Sussman
University of Maryland
als@cs.umd.edu

Matthew Wolf
Oak Ridge National Lab
wolfmd@ornl.gov

Kshitij Mehta
Oak Ridge National Lab
mehtakv@ornl.gov

Jong Youl Choi
Oak Ridge National Lab
choij@ornl.gov

Abstract

Modern scientific workflows are increasing in complexity with growth in computation power, incorporation of non-traditional computation methods, and advances in technologies enabling data streaming to support on-the-fly computation. These workflows have unpredictable runtime behaviors, and a fixed, predetermined resource assignment on supercomputers can be inefficient for overall performance and throughput. Inability to change resource assignments further limits the scientists to avail of science-driven opportunities or respond to failures. We introduce DYFLOW, a flexible framework that orchestrates scientific workflows on supercomputers based on user-designed policies. DYFLOW compartmentalizes orchestration stages into simplified constructs, and end-users can program and reuse them according to their workflow requirements through an easy-to-use interface. These constructs hide the intricacies involved in runtime management from end-users, for instance, procurement of information to understand the workflow state, assessment, and supervision of the runtime changes. DYFLOW is designed to work alongside existing workflow management systems and reuse the available (static) support for workflow management. We have integrated DYFLOW with an existing workflow management tool as a demonstration. With experiments performed on use cases from three types of scientific workflows and two different parallel architectures, we show that DYFLOW achieves the desired orchestration incurring a small cost to carry out the runtime changes.

Keywords

policy-driven workflow management, performance monitoring, user-defined dynamic workflows, *in situ* analysis, data-driven orchestration, resource adaptation, resilience

1 Introduction

Scientific workflows have typically contained a set of loosely coupled tasks – i.e., simulation, analysis, or visualization – interconnected via the filesystem. Technological and methodological advances have enabled scientific workflows to expand both in scale and complexity [6]. The use of *in situ* (or *in transit*) techniques, for instance, to filter, digest, and reduce data stream sizes, is becoming an increasingly popular option to overcome I/O bottlenecks. These techniques reduce disk storage requirements by managing dataflow between workflow tasks, employing in-memory staging (buffering), or node-to-node data transfers. Managing such complex workflows is challenging as workflow tasks often run concurrently with input/output dependencies, potentially affecting performance across workflow tasks and also system resource usage [8]. There are additional complications when the workflows themselves are composed of tasks derived from non-traditional HPC methods, such as machine learning and graph algorithms that exhibit unpredictable computation patterns.

On supercomputers, resource management support for user jobs is usually static, where resources are assigned once based on an initial resource requirement specification, thereby unable to accommodate dynamically changing requirements. Predetermining an efficient resource assignment becomes challenging for a workflow with changing runtime requirements, resulting in overprovisioning of resources or loss of workflow performance due to under-provisioning. Modern supercomputers provide abundant resources on a single node for workflow tasks to share and take advantage of data locality. Due to the unpredictable behavior of workflow tasks, scientists often hesitate to avail themselves of such opportunities. An orchestration service is hence desirable that can monitor workflows to adapt the resource assignments according to changing runtime needs of the workflow. Such a service can further enable scientists to respond to failure events and avail science-driven opportunities provided by on-the-fly analysis to improve overall experiment accuracy. Throughout this paper, we refer to the dynamic management of scientific workflows as the orchestration service.

Therefore, this paper presents DYFLOW – a flexible framework that can orchestrate scientific workflows based on user-designed policies to take advantage of the benefits of dynamic resource assignment. DYFLOW contains programmable constructs corresponding to different stages of dynamic management. These constructs are available to the user through an easy-to-use interface (XML) that enables the desired runtime management of a workflow without user involvement. For instance, DYFLOW can support science-driven functionality, improve resource assignments based on performance-driven events, and aid in providing resilience to workflow task failures. DYFLOW internally handles acquiring monitoring information (i.e. performance measurements) at scale from system resources and performs the processing required to plan the appropriate resource reassignment and other workflow actions at runtime. Furthermore, the framework executes the final plan of action and determines the appropriate resource assignments while ensuring that the workflow state remains consistent after the runtime reassignment.

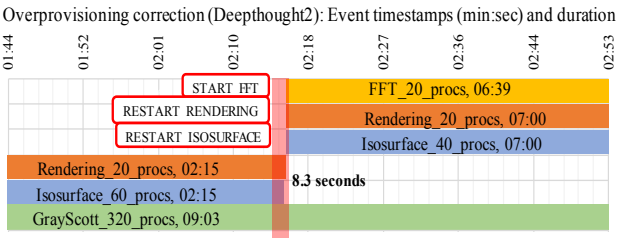

Figure 1: DYFLOW improves the throughput of an *in situ* workflow by rebalancing execution parameters in response to performance variations. The red vertical bars show the response window of DYFLOW for a dynamic event where the resources are taken from the running analysis tasks and used to launch additional analysis task to improve overall throughput.

DYFLOW uses services from existing workflow management systems for interacting with the system resource manager, setting the initial resource assignments, and applying the final actions on workflow tasks at runtime. It also utilizes support from application profilers, e.g., TAU [16], for acquiring real-time monitoring information. As a demonstration, we have implemented DYLOW using an existing workflow management system, Cheetah/Savanna [9]. We have tested the integrated system on two cluster configurations while applying our strategies to three scientific workflows with different runtime requirements. The results indicate that our framework accommodates and fulfills the dynamic needs of scientific workflows incurring a small cost to carry out the runtime changes.

As Figure 1 shows, DYFLOW improves the throughput of an *in situ* workflow at runtime by launching additional tasks and correcting resource over-provisioning. Dynamic changes are applied to meet the user expectation of keeping the runtime performance (i.e., average time per timestep) within the desired interval, ensuring that the experiment finishes on time and resources are well utilized.

Our work provides two research contributions; (1) an abstraction that compartmentalizes the dynamic management stages into simplified constructs that support state-of-the-art-workflows, and (2) an easy-to-use interface that enables scientific end users to program and reuse these constructs. DYFLOW's compartmentalization empowers users with controls to dictate fine-grained operational semantics for different stages of the runtime. Together with the focus on ease-of-use, this work aims to provide a generic, user-accessible platform for identifying, managing, and capitalizing on runtime events at scale that can generate end-user benefits such as improved workflow performance and throughput, early result validation, and failure recovery. This work further provides a platform for exploring new ways of conducting scientific studies on supercomputers.

Extensive research in workflow management and task scheduling has resulted in numerous systems that focus on different aspects of managing workflows, and we have built DYFLOW leveraging these prior systems and their insights. In addition to the scientific workflow community, we also draw substantial guidance from cloud/enterprise service orchestration runtimes. Elastic scaling of resources to handle performance fluctuations is a critical capability in today's cloud stacks, yet they have proven difficult to incorporate into traditional, batch-oriented scientific workflows. By staying focused on the dynamic management components that are most relevant for scientific end users, DYFLOW offers a platform for further study of the connections between content-driven *in situ* scientific workflow control and the quality-of-service service compositions of cloud-based systems.

The organization of the paper is as follows. Section 2 discusses the details of the DYFLOW framework. Section 3 discusses the implementation used as a demonstration for the experiments and the XML interface. Section 4 showcases experiments performed on two HPC systems executing three scientific workflows with different runtime requirements to demonstrate the benefits of using DYFLOW. Section 5 described related work, and Section 6 discusses potential directions for future work.

2 DYFLOW framework

DYFLOW is a conceptual model that compartmentalizes dynamic management into four stages; **Monitor**, **Decision**, **Arbitration** and **Actuation**. All these stages exist simultaneously and function continuously on the input received from the previous stage. The first stage is the Monitor that gathers runtime data from the running workflow tasks that is needed to identify dynamic events and construct metric values. The changes in the metric values pass on to the Decision stage. The second stage is the Decision that assesses the metric values to identify if an event of interest has occurred and then determine the actions needed in response to the event. The selected actions are passed on to the Arbitration stage. The third stage is Arbitration that constructs a plan of action that is feasible and consistent with the workflow specifications on receiving the input from Decision. The last stage is Actuation that executes the final plan of action received from Arbitration. For each of these stages, DYFLOW exposes features that enable users to express events that can change the behavior of the workflow at runtime through the actions in response to those events.

2.1 Monitor

The Monitor stage defines the data to procure for runtime assessment, the input method to employ for real-time procurement of this data, and the translation operations necessary to convert the procured data into metrics for identifying events of interest. This stage allows users to define different monitoring requirements through sensors that support abstract features. These features provide the set of commands to users for expressing wide-ranging needs that vary from simple metrics like the maximum memory consumed by a task to complex metrics computed from workflow measurements. Monitor features include:

Source type: Depending on the workflow, the required information could be organized in a specific format and available through a given medium. Source type determines how data of interest is generated and exchanged at runtime for a sensor. For instance, the desired data can be generated by an online profiler, the running task or system, and is available through a database service, a streaming service, or files.

Preprocessing: Preprocessing operations distill the data before it is processed into the desired metric. It is useful when the input read from each process is sizeable, for instance, a vector or multi-dimensional array.

Group-by and reduction: These operations dictate metric formulation. Group-by collects the data from running tasks and organizes it based on the granularity, while the reduction operation

summarizes the grouped information into a metric. A granularity-based grouping enables expressing metrics from the collected data that can capture events in different scopes. For instance, the physical memory usage can have two metrics: one that keeps track of the physical memory used by a task on each compute node used, while the other keeps track of the overall physical memory used by the task to assess the memory usage pattern. Some examples of granularity levels include node-task, task-level, node-workflow, and workflow-level. The node-task granularity groups data from every process belonging to the same workflow that shares the compute node. The node-workflow granularity performs the same grouping for all the processes belonging to the workflow. With task-level granularity, the groups define the data from all the processes belonging to the same task, while with workflow-level granularity, the groups define data from all the tasks belonging to the same workflow.

***Join*:** A sensor can join its output with another sensor to compute a complex metric that relies on multiple data inputs. For instance, Instructions Per Cycle (IPC), a metric used for measuring CPU performance, is computed by dividing the number of instructions completed by the number of CPU cycles used.

Sensors act as portable functions invoked using inputs that vary across workflow tasks and architectures. For example, workflow tasks have different variable names representing the desired information, or the hardware counter information used for defining metrics can differ across architectures. The Monitor stage manages the background activities of the user-defined sensors to ensure correctness. These include setting (or resetting) connections to input streams or databases when the workflow tasks start (or restart), gathering the sensor outputs, and sending the information to the Decision stage for evaluation and updating sensors about changes performed on the workflow at runtime.

2.2 Decision

Once a metric is defined, a set of guidelines must be determined that clarifies what evaluation criteria should be employed to capture the events of interest from the metric values. Should the evaluation be based on the instantaneous values or observing the values over a period of time? What actions should be taken in response to the events at runtime? And, how frequently should the evaluation be performed? The Decision stage allows users to define policies that provide abstract features that simplify setting these guidelines and supporting a broad range of policies. Decision guidelines include:

Sensor(s) to use: Defines the sensor output(s) to employ for the policy with the desired granularity level(s).

History and pre-analysis: The policy could maintain a history of sensor outputs, like a sliding window of a specified size, and perform a preliminary analysis to capture a pattern. For instance, to identify the events based on the running average rather than the last observed value of the IPC metric.

Evaluation condition: The evaluation condition compares the input against a threshold, and the result determines if an event of interest has occurred. The evaluation could use the instantaneous or pre-analyzed output from a single sensor or a value derived from a set of sensor outputs.

Suggested action: The suggested actions represent the high-level operations applied to one or more tasks in response to the event of interest. These high-level operations are concise and easy to understand as they encapsulate different low-level operations required to perform the desired action. For example, a SWITCH operation represents the following low-level operations; signaling a running task to stop, estimating resources for launching the replacement task, acquiring the required resources, and initiating the replacement task if enough resources are available. Other examples of high-level actions include ADDCPU, RMCPU, STOP, START, and RESTART. These correspond to increasing or decreasing the number of CPUs assigned to the task to increase or decrease the number of processes, stopping a running task, and starting a task or restarting the current task. Each high-level operation supports additional parameters to guide the action, e.g., the desired number of CPUs to increase or decrease or user settings to apply (i.e., using a shell script) before starting or restarting tasks.

Evaluation frequency: Every policy has a defined frequency to decide when to trigger the evaluation condition. Evaluation frequency helps in avoiding events that have transitory effects.

Like sensors, policies act as portable and reusable functions. The inputs to these policies vary with different workflows and tasks. For example, evaluation thresholds or the tasks to which the policy action would apply can differ across the monitored tasks.

2.3 Arbitration

The Arbitration stage is a complex stage of dynamic management that determines which actions – if any – will be applied to modify the current state of the workflow. This stage ensures that the final set of actions are feasible and consistent with the workflow specification. Arbitration screens the high-level operations suggested by the Decision stage resolving conflicts, inconsistencies, and dependencies. For instance, conflict results when one policy suggests stopping a task while another suggests increasing the number of processes for the same task. Similarly, some high-level operations depend on additional operations to ensure consistency. For instance, whenever a running task is terminated or restarted, all the (tightly) coupled dependent tasks need to be signaled. Resolution of conflicts results in filtering out a set of high-level actions and deferring others. To construct a plan of action, Arbitration maps the filtered high-level operations to low-level operations. These low-level operations represent the function calls understood by a resource manager or underlying resource management service.

Resource management is the primary responsibility of Arbitration as a feasible final plan is dependent on the available resources. Hence, it maintains recent information about the resources that include the total allocated resources, resource health, and the current resource assignment to workflow tasks. Arbitration issues requests for additional resources whenever necessary and resolves conflicts and incompatibilities among low-level operations when resources are insufficient to meet all requirements. For instance, if tasks A and B want to increase their number of processes while the available resources cannot allow both operations, then one of these would be denied. A final executable plan with revised resource assignments consists of all the selected low-level operations sequenced in the order in which to apply them. Ordering is required to avoid execution inconsistencies. For example, if any operation reduces the

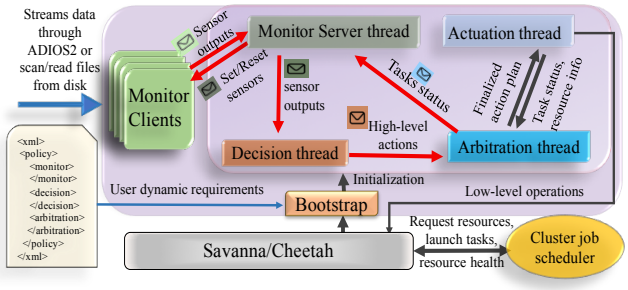

Figure 2: Overview of DYFLOW implementation as an extension to the Cheetah/Savanna workflow service. Arrows represent the exchange of data using JSON messages (in red), function call (in black) or file/stream reads (in blue).

number of processes of a task releasing resources, it should precede others that use those resources.

Arbitration provides users with the flexibility to define rules that guide the plan of action.

***Policy priorities*:** Assign priorities to policies according to their relevance. This helps in resolving conflicting high-level actions.

***Task priorities*:** Assign priorities to tasks according to their relevance. This helps in resolving conflicting low-level operations.

***Task inter-dependencies*:** Determine the dependent tasks and their parent tasks and if the dependency is tight (i.e., the dependent task runs concurrently with the parent and gets data via an *in situ* medium) or loose (i.e., the dependent task runs uncoupled from the parent and gets data via disk). This information helps in identifying dependent operations.

2.4 Actuation

This stage implements all the low-level operations invoked by Arbitration in the final plan of action. These low-level operations serve as a plugin to any static service that interacts directly with the cluster resource manager and launches workflow tasks on the compute node. Having a pluggable Actuation stage allows the DYFLOW model to be portable across cluster architectures and build on the services supported by the existing workflow management systems. Some examples of such low-level operations include starting a task with a resource assignment(start_task_with_resources), sending signals to a task (signal_*_task), terminating a task (stop_task), requesting or releasing extra resources from cluster resource manager(request_resources, release_resources), and enquiring resources' health (get_resource_status).

3 DYFLOW Implementation

The design of a model implementation of DYFLOW is shown in Figure 2. Our implementation extends the functionality of an existing workflow management service, Cheetah/Savanna. The Cheetah and Savanna tools are a Department of Energy CODAR (Co-designing of Online Data Analysis and Reduction) project [9, 13] effort to investigate various resource allocation trade-offs as part of broader co-design studies. Cheetah is a composition tool used to specify the workflow; Savanna is a runtime environment that runs on launch/service cluster nodes, communicates with the cluster scheduler, allocates the required resources, and spawns the workflow tasks on the allocated resources. Cheetah/Savanna incorporates the orchestration functionality of DYFLOW as an external

Algorithm 1 Arbitration Protocol

function ARBITRATION($A_{sugg}, R_{free}, R_{asgn}, T_{pri}, D_{pri}, T_{dep}, T_{waiting}$)
  $A_{filter} \leftarrow$ resolve conflicts in A_{sugg} using D_{pri}
  $A_{total} \leftarrow$ (get dependent actions for A_{filter}) $\cup A_{filter}$
  $S_{op} \leftarrow$ Get low level operations for A_{total}
  $N_{des} \leftarrow$ Calculate required resources from S_{op}
  while $N_{des} > Count(R_{free})$ **do**
    $R_{rel} \leftarrow$ Find the lowest priority running task (and any dependent tasks) using T_{pri} and T_{dep} that can shed resources
    if $Count(R_{rel}) > 0$ **then**
      $S_{op} \leftarrow$ add operation to stop task(victim) (and dependents) in S_{op}
      add victim(and dependents) to $T_{waiting}$
      $R_{free} \leftarrow R_{free} \cup R_{rel}$
    else
      Select the least significant operation from A_{total} that acquire resources based on T_{pri}. Update S_{op} and N_{des}.
    end if
  end while
  while $N_{des} < Count(R_{free})$ and a task (with highest priority) from $T_{waiting}$ can be started **do**
    Update S_{op}, $T_{waiting}$ and N_{des}.
  end while
  $op_{final} \leftarrow$ Order operations in S_{op}
  $R_{reasng} \leftarrow$ Determine new resource assignment for op_{final}
end function

Python library and its modules implement the different dynamic management stages.

Bootstrap: This module parses the XML file with user orchestration specifications of the workflow and initiates threads corresponding to the Monitor, Decision, Arbitrator modules providing them with essential information. For instance, the Monitor module receives the sensor information while the Decision module gets the policies details. All communications between the service threads occur through shared queues and JSON [1] formatted messages. The Actuation module is a wrapper for the plugin inside Savanna that executes all the low-level operations.

Monitor: This module is a client-server service. A client(s) is a hybrid MPI and Python threads-based service that can run on a compute node or launch node of a cluster. The server service runs on the launch node within the DYFLOW library and connects to the client(s) using PyZMQ [2]. Flexibility to launch multiple clients on the compute or launch nodes benefits the Monitor to address requisite scaling needs. Running the server on the launch node ensures its availability in events of computing resource failures. The server manages the client(s): starts (or restart) client(s) with the sensors along with the tasks to monitor, updates the client(s) whenever the runtime status of monitored tasks is changed, filters the out of order messages from the client(s), and sends updates from the client(s) to the Decision module. A client(s) manages and executes the sensors by connecting to workflow tasks, collecting the monitoring data, and sending the sensor outputs to the server. Our implementation supports sensors that can stream user data through ADIOS2, or stream data generated by the TAU [16] profiler using ADIOS2, scan disks for files, and read files. TAU is an online profiler that collects performance data via code instrumentation and event-based sampling. ADIOS2[3] is a state-of-the-art unified I/O framework that encompasses a variety of transformations and transport methods, including file I/O and other *in situ* methods for task coupling (e.g., Sustainable Staging transport (SST)). We employ SST transport in

[1] JavaScript Object Notation(JSON):https://www.json.org/json-en.html
[2] Python ZeroMQ website: https://zeromq.org/languages/python
[3] ADIOS2 website: https://csmd.ornl.gov/software/adios2

```
<monitor>
  <sensors>
    <sensor id="PACE" type="TAUADIOS2">
      <group-by> <group granularity="task" reduction-operation="MAX"/>
      </group-b> </sensor>
  </sensors>
  <monitor-tasks>
    <monitor-task name="Isosurface" workflowId="GS-WORKFLOW"  info-source="tau-iso.bp.*">
      <use-sensor sensor-id="PACE" info="looptime">
        <parameter key="info-type" value="double"/>
      </use-sensor>
    </monitor-task>
    … … …
</monitor>
```

Figure 3: XML illustrating a sensor for tracking main iteration time using code instrumentation support from the TAU profiler

ADIOS2 to enable both dynamic connections between workflow tasks and high-performance data movement.

Decision: This module screens incoming sensor messages(s) for out-of-order updates and maps them to the policies. Each policy uses these updates to trigger evaluation at defined frequency intervals; otherwise, the updates are either discarded or stored to maintain history. Policy responses (if any) are collected and sent as a single JSON message to the Arbitration module.

Arbitration and Actuation: Arbitration uses the protocol described in Algorithm 1 to finalize runtime modifications. The protocol has two limiting factors: (1) MPI-based tasks that depend on inter-and intra-task communication cannot grow and shrink without restart, and (2) resource manager support for on-demand resource allocation and de-allocation is not commonplace on supercomputers.

The protocol takes the following inputs; the set of suggested actions (A_{sugg}), the allocated free (healthy) resources (R_{free}), the allocated (healthy) resources assigned to tasks (R_{asgn}), task priorities (T_{pri}), task dependencies (T_{dep}), decision priorities (D_{pri}), and a list of tasks waiting to acquire resources ($T_{waiting}$). It outputs an action plan with an ordered set of low-level operations (op_{final}) with the revised resource assignment (R_{reasgn}).
The protocol begins with conflict resolution across high-level operations (utilizing decision priorities) and filtering the suggestion set (A_{filter}). The types of conflicts resolved include: *STOP-START*, *STOP-RESTART*, or *RMCPU-ADDCPU*. Next, it identifies dependent operations for the filtered suggestions (through task dependencies) and finalizes the set of high-level operations (A_{total}). The set, A_{total}, is then mapped to low-level operations, and an initial plan of action (S_{op}) is determined along with computing the additional resources (i.e., CPU cores) required (N_{des}) to execute the plan.

When free resources are not available to satisfy the additional resource request, a running task with the lowest priority becomes the victim. The victim task relinquishes the resources (represented by the set R_{rel}) and waits in a queue for computation resources to become available. If a victim task is not available, the lowest priority operation requesting additional resources gets discarded from the plan. This process repeats until the available resources can meet the requirements of the revised plan. On the other hand, when resources are freed by the plan, the waiting list tasks ($T_{waiting}$) are provided the opportunity to start with preference given to high priority tasks. Finally, the operations in the finalized plan are ordered (e.g., STOP, RMCPUS proceeds START, ADDCUPS), and the revised resource assignment is determined. Once the protocol completes, the Arbitration module waits for the Actuation module

```
<decision>
  <policies>
    <policy id="INC_ON_PACE" >
        <eval operation="GT" threshold="36" />
        <sensors-to-use> <use-sensor id="PACE" granularity="task" /></sensors-to-use>
        <action> ADDCPU </action>
        <history window="10" operation="AVG "/>
        <frequency seconds="5"/> </policy>
    <policy id="DEC_ON_PACE" >
        <eval operation="LT" threshold="22" />
        <sensors-to-use> <use-sensor id="PACE" granularity="task"/>  </sensors-to-use>
        <action> RMCPU </action>
      … … …
  </policies>
  <apply-on workflowId=" GS-WORKFLOW">
    <apply-policy  policyId="INC_ON_PACE" assess-task name="Isosurface">
      <act-on-tasks> Isosurface</act-on-tasks>
      <action-params> <param key="adjust-by" value="20"/> </action-params>
      … … …
  </apply-on>
</decision>
```

Figure 4: XML illustrating policies for changing the number of CPUs when the pace of the task is outside a desired interval.

```
<arbitration>
  <rules>
    <rule-for workflowId="GS-WORKFLOW">
      <task-priorities>
        <task-priority name="GrayScott" priority="0"/>
        … … …
      <task-dependencies workflowId="GS-WORKFLOW">
        <task-dep name="Isosurface" type="TIGHT" parent="GrayScott"/>
        … … …
  </rules>
</arbitration>
```

Figure 5: XML illustrating arbitration rules for determining task dependencies and prioritizing task and decision policies.

to execute the plan. If the Actuation module returns successfully, the Arbitration module discards new decision messages for a sufficient time, allowing the workflow state to settle down after the changes.

User Interface: We choose XML for the user interface because it is portable and easy to use and extend. The XML contains sections corresponding to the Monitor, Decision, and Arbitration stages. The monitor section defines the sensors and the workflow tasks to monitor using the sensors [4]. The decision section sets the policies and the workflow tasks for which these policies will perform the assessment. The arbitration section sets the rules for the workflow that corresponds to setting priorities and dependencies.

The monitor section, demonstrated in Fig. 3, sets a sensor, PACE, to track the time spent in the main loop of the workflow tasks. This information is generated through the TAU code instrumentation facility and collected in real-time using ADIOS2. The sensor returns a metric representing the time taken to complete an iteration, or timestep, of the task. The metric is the maximum of values received from all the processes of the monitored workflow task. The example further illustrates how the sensor is configured for monitoring a workflow task, *Isosurface*, with the details of the variable to be read.

The example for the decision section, shown in Fig. 4, defines two policies that act on the output of the PACE sensor. One policy increases the number of CPUs of a task if the average time per timestep is more than a threshold value, and the other decreases the number of CPUs when the average time per timestep is less than a threshold value. The policies maintain a running average of the sensor output using the latest 10 values. The policies evaluate the sensor output every 5 seconds.

Finally, the example for the arbitration section, shown in Fig. 5, demonstrates the rules that set the priority value for a workflow

[4]Detailed semantics of the DYFLOW XML can be found at https://github.com/swatisgupta/DYFLOW

Table 1: A single run configuration of XGC1 and XGCa

TASK(S)		SETTING	Summit	Deepthought2
XGC1	*XGCa*	PROCESSES	192 (14 PER NODE)	192 (4 PER NODE)
XGC1	*XGCa*	THREADS PER PROCESS	10	10
XGC1	*XGCa*	TIMESTEPS PER RUN	100	100
XGC1	*XGCa*	PARTICLES PER PROCESS	250K	250K

task, GrayScott, and specifies that the task $Isosurface$ has a tightly coupled dependency on $GrayScott$.

4 Experiments

We showcase examples from three scientific workflows highlighting some of the dynamic capabilities achievable through DYFLOW. In the examples, DYFLOW flexibly enables modification of the workflow state in response to science-driven events, re-assignment of computation resources in response to performance-driven events, and recovery from failure. Our experiments test DYFLOW based on the model implementation that builds on Savanna/Cheetah. To show the costs incurred by DYFLOW, we conducted these experiments on a standard Unix cluster and a state-of-the-art high-end supercomputer. The results show that DYFLOW accommodates varied dynamic workflow requirements and incurs a low cost to carry out the desired changes to the workflow execution.

4.1 Clusters

Summit: A high-end supercomputer with 4,608 nodes, where each node consists of 2 IBM Power9 CPUs (i.e., 42 cores with each core is 4-way hyper-threaded), 6 NVIDIA Volta GPUS, 512 GB of DDR4 memory and additional 96 GB of High Bandwidth Memory (HBM2). All the nodes are interconnected using Mellanox EDR 100*G* InfiniBand.

Deepthought2: A standard Linux cluster with 448 nodes, where each node has 20 cores (with 2 hardware threads/core) and 128 GB of DDR3 memory running at 1866 MHz. Each node has dual Intel Ivy Bridge E5-2680v2 processors running at 2.80 GHz and the nodes are interconnected with Mellanox FDR Infiniband.

4.2 Use cases

This section describes the three use cases: XGC1-XGCa, Gray-Scott, and LAMMPS. These use cases are specifically selected to represent workflows based on different types of scientific simulation techniques. For instance, XGC1-XGCa is an exemplar of the workflows that synthesize particle-in-cell computations; Gray-Scott is a MiniApp that represents workflows that synthesize mesh-based fluids; the LAMMPS use case exemplifies workflows that synthesize a combination of both particles and mesh-based computations.

XGC1-XGCa coupling based fusion simulation:
XGC1 [12] and XGCa are gyrokinetic particle-in-cell codes developed to study complex multi-scale simulations of turbulence and transport dynamics of the fusion processes in state-of-the-art fusion reactors, called Tokamaks, including D3D, JET, KSTAR, and the next-generation ITER reactors. XGC1 is highly complex and computation-intensive software that often takes several days to simulate a short time interval of fusion reactions in the reactors. On the other hand, XGCa uses a simplified physical model that can simulate fusion reactions for a longer physical time within a fixed amount of wall clock time. A complete simulation of Tokamak reactors requires a femtosecond-scale resolution, which is very expensive to complete in a reasonable time frame with XGC1; therefore, scientists have to resort to a coarser-scale in the micro- to millisecond range in practice. An alternative employed to maintain the precision of the fully converged simulation is alternating the simulation between XGC1 and XGCa such that XGCa pushes the simulation forward at a faster rate [11]. The scientists choose the alternation frequency to enable the experiment to moves forward quickly in simulation time with confidence that the statistics (if not exact values) of the result will be the same as that produced with XGC1 alone.

Gray-Scott based reaction-diffusion online analysis:
Gray-Scott is a mathematical model that simulates reaction-diffusion systems and is used to study chemical species that can produce a variety of patterns (stripes, spots, periodicity) often seen in nature. There are many applications found in biology, geology, physics, ecology, etc. that undergo similar chemical reactions, and Gray-Scott can be employed as a simplified system to represent them. Hence there are also a variety of concurrent data analysis functions that may be useful, based on the target of study.

For this study, we used several data analysis tasks, the most computationally intensive of which is a 3D Fast Fourier Transform (*FFT*) of the output arrays from the Gray-Scott model. Some of the other analyses are inexpensive to compute, such as computing the norm of a set of output vectors (*PDF_Calc*), while others are complicated and can change in computational complexity based on the data (e.g., *Isosurface* and *Rendering* compute and render the iso-surfaces of the output vectors). This combination of very regular and highly variable analyses means it is easy for a user to make poor resource allocation decisions that lead to either under-or over- provisioning depending on what analysis process(es) are used for a particular scientific study.

LAMMPS based Molecular Dynamics online analysis:
LAMMPS[5] is a prominent molecular dynamics simulation code, used for applications ranging from engineering nano-materials to designing new alloys to exploring protein-folding. The great variety of uses and user communities means that there is intense interest in building and sharing tools for online analysis and management of data that can be customized to each research team's needs. We focus on a scenario where a set of tools are integrated for analyzing solids as they break and melt under stress. In particular, LAMMPS is coupled with three analysis processes that compute the radial distribution function (*RDF_Calc*), perform common neighbor analysis (*CNA_Calc*), and compute central symmetry (*CS_Calc*).

4.3 Managing workflow tasks in response to science- or data-driven events

We demonstrate the utilization of DYFLOW to orchestrate science-driven events by employing the loosely coupled workflow with two tasks, XGC1 and XGCa, to simulate fusion reaction. The tasks run alternately, each for a fixed number of simulation timesteps until they jointly complete the desired total number of timesteps. For correctness, the experiment relies on the error assessment of XGCa so that XGC1 takes over the simulation whenever the error accumulation is high.

Two sensors and two policies were defined to address the dynamic requirements, as shown in the sample XML in Figure 7. The

[5]LAMMPS website: https://lammps.sandia.gov

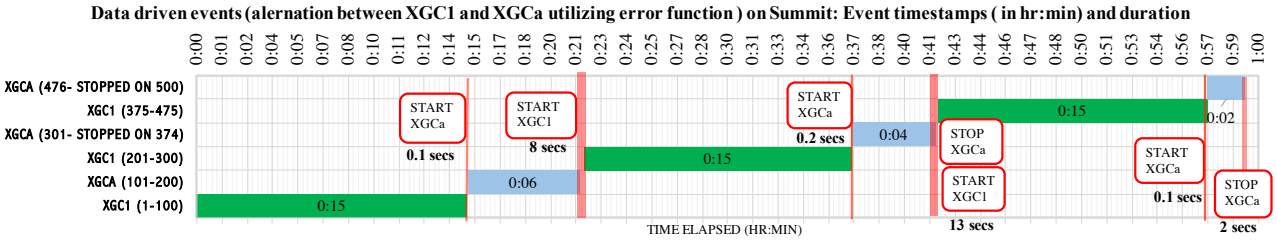

Figure 6: Gantt-chart showing the experiment performed on Summit for the XGC1-XGCa workflow to demonstrate running iterative experiments and terminating tasks based on runtime events.

first sensor, *NSTEPS*, tracks progress, i.e., the number of global timesteps completed during the simulation. Both XGC1 and XGCa write an output file once a fixed number of global simulation timesteps complete; therefore, the source type for this sensor is *'DISKSCAN'*. The metric for this sensor computes the maximum number of timesteps completed by the workflow. The second sensor, ERROR, is defined to compute the error in the output from XGCa. The XGCa output is available in ADIOS2 format, so the source type for this sensor is *'ADIOS2'*. The properties of the fusion simulation output that could define an error estimation function is ongoing research by fusion scientists, so the definition of this sensor is incomplete.

To register the two dynamic events, we set three policies as follows. *RESTART_UNTIL_COND* starts XGC1 (or XGCa) if the output of *NSTEPS* of the workflow is less than 500 for XGCa (or XGC1). *STOP_ON_ COND* stops XGC1 (or XGCa) if the output of *NSTEPS* is greater than 500 for XGCa (or XGC1). *SWITCH_ON_COND* stops XGCa and starts XGC1 when high error accumulates in the simulation output generated by XGCa. Instead of using the ERROR sensor, we use the *NSTEPS* sensor output at task granularity to generate a proxy error condition. The proxy error condition causes termination of XGCa after the 374th timestep of the simulation completes, based on the assumption that error accumulation can be high after that many timesteps. Before starting XGC1 a user script, restart-xgc.sh, runs to set XGC1 inputs to restart from the last saved output of XGCa. Both tasks have priority 0 (the highest priority) since they run alternately. Further, we prioritize the policies so that *STOP_ON_COND* has the highest priority as it signifies experiment completion, and *SWITCH_ON_COND* is preferred over *RESTART_UNTIL_COND* to resolve conflicts.

Table 1 shows the initial setup for the experiments on both clusters. **Summit:** Figure 6 shows the individual timestamps and durations of the workflow tasks and all the events. The timestamps in the figure are relative to the start of the experiment. The green bars show the times XGC1 runs, the blue bars show the times XGCa runs, and the red intervals show the dynamic adjustment period. On average, XGC1 runs $2.5x$ slower than XGCa to simulate 100 timesteps. The simulation starts with XGC1 while XGCa waits in the queue due to their loosely coupled dependency. Because of *RESTART_UNTIL_COND*, XGCa starts three times with the same resources when XGC1 terminates as resources become available. The response time to finalize the plan and execute it for these events is $\approx$ 0.1-0.2 seconds. Similarly, XGC1 starts when XGCa finishes (when 200 global simulation steps complete). The response time, in this case, is 8 seconds - 4 seconds of this time is due to the

```
<sensors>
  <sensor id="NSTEPS" type="DISKSCAN">
      <group-by> <group granularity="task" reduction-operation="MAX"/>
                 <group granularity="workflow" reduction-operation="MAX" />
      </group-by> </sensor>

   <sensor id="ERROR" type="ADIOS2">
       <preprocess operation="..." />
       <group-by> <group granularity="task" reduction-operation=" ... "/> </group-by>
... ... ...

<policies>
   <policy id="RESTART_UNTIL_COND" >
       <eval operation="LT" threshold="500" />
       <sensors-to-use> <use-sensor id="NSTEPS" granularity="workflow"/> </sensors-to-use>
       <action> START </action>
      <frequency seconds="5"/> </policy>

    <policy id="SWITCH_ON_COND " >
       <eval operation="EQ" threshold="374" />
       <sensors-to-use> <use-sensor id=" NSTEPS" granularity="workflow" /> </sensors-to-use>
       <action> SWITCH </action>
       <frequency seconds="5"/> </policy>

    <policy id="STOP_ON_COND" >
        <eval operation="GT" threshold="500" />
          <sensors-to-use> <use-sensor id=" NSTEPS" granularity="workflow" /> </sensors-to-use>
          <action> STOP </action>
        <frequency seconds="5"/> </policy>
    ... ... ...
    <apply-policy policyId="RESTART_UNTIL_COND" assess-task="XGCA">
          <act-on-tasks> XGC1 </act-on-tasks>
         <action-params> <param key="restart-script" value="restart-xgc1.sh"/> </action-params>
         </apply-policy>

   <apply-policy policyId="SWITCH_ON_COND" assess-task="XGCA">
          <act-on-tasks> XGC1 </act-on-tasks>
     ... ... ...
    <apply-policy policyId="STOP_ON_COND" assess-task="XGCA">
          <act-on-tasks> XGCA </act-on-tasks>
... ... ...

<rules>
    <rule-for workflowId="FUSION-WORKFLOW">
         <policy-priorities>
            <policy-priority name="STOP_ON_COND" priority="0"/>
            <policy-priority name="RESTART_UNTIL_COND" priority="1"/>
... ... ...
```

Figure 7: XML example illustrating the user specification for switching on error and restarting the experiment for the desired number of timesteps.

delay enforced by frequency settings of the policy in evaluating the sensor output. The time to start XGC1 is greater than that of XGCa due to running the user script. Because of *SWITCH_ON_COND*, XGCa stops after completing 74 steps (i.e., 374 global simulation steps complete at this point) with a response time of $\approx$ .13 seconds. From *STOP_ON_COND*, XGCa stops after completing 502 global timesteps with a response time of 2 seconds. **Deepthought2:** In a similar experiment on Deepthought2 (graph not shown), the response times as follows: $0.8 - 0.2$ seconds for starting XGCa, 11 seconds for staring XGC1, 24 seconds for switching to XGC1 from XGCa, and 42 seconds to stop XGCa.

Without DYFLOW, the simulation completes only using XGC1 and takes approximately 25% more time on each cluster.

Table 2: Initial configuration for *Gray-Scott* workflow that results in resource under- provisioning

TASK	SETTING	Summit	Deepthought2
GRAY-SCOTT	PROCESSES	340 (34 PER NODE)	320 (16 PER NODE)
GRAY-SCOTT	GRID/PROCESS	$42 \times 140 \times 175$	$88 \times 88 \times 140$
ISOSURFACE	PROCESSES	20 (2 PER NODE)	20 (2 PER NODE)
RENDERING	PROCESSES	20 (2 PER NODE)	20 (1 PER NODE)
FFT	PROCESSES	20 (2 PER NODE)	20 (1 PER NODE)
PDF_CALC	PROCESSES	20 (2 PER NODE)	20 (1 PER NODE)
GRAY-SCOTT	TOTAL STEPS	50	50
ALL	TIME LIMIT	30 MINS	35 MINS

4.4 Managing resource distribution in response to performance-driven events

This section focuses on the performance concerns of tightly coupled workflows where tasks communicate data through in-memory streams and may affect each others' runtime behavior via changes in data flow at runtime. Using the Gray-Scott workflow, we demonstrate how to utilize DYFLOW to respond to performance-driven events at runtime. Modern profilers, like TAU [16], enable users to access different types of performance measurements that can be collected indirectly via system support or directly through code instrumentation. For example, system-generated information includes memory footprint, CPU utilization, and network bandwidth, while code instrumentation can measure time spent in various code sections. For the experiments in this section, the time taken to complete an iteration (or a single iteration of the outermost loop in the application) represents the pace at which the tasks are progressing.

Using the above performance metric, we performed experiments capturing two types of runtime events; (a) Under-provisioning: a task is assigned fewer resources than required, slowing the overall workflow performance (for instance, the simulation task waits for analysis tasks to read data for a timestep), such that the experiment may not finish in the allotted time, and (b) Over-provisioning: a task is assigned more resources than required, so there are times when resources are under-utilized.

Figures 3 to 5 show sample XML for such an experiment. To set the monitoring metric we define a sensor, PACE, which reads the TAU-generated information using code instrumentation. The generated information represents the time spent in each iteration and is available in real-time through ADIOS2. To identify the under- and over-provisioning events, we set two policies, *INC_ON_PACE* and *DEC_ON_PACE*, that increase or decrease the number of CPUs assigned to any monitored task (by 20) if the average pace is slower or faster than the desired values (i.e., thresholds) respectively. The average is computed over a sliding window of 10 values to avoid decisions based on a single timestep.

For the threshold for *INC_ON_PACE*, we used a value of 36 seconds based on the desire that the workflow complete 50 timesteps in 30 minutes; therefore, the tasks spend a maximum of 36 seconds per timestep. If any workflow task takes more time per timestep, then it needs more processes to complete the experiment on time. For *DEC_ON_PACE*, the threshold value is 24 seconds which utilizes a variant percentage, such that if the task is more than a third faster than the maximum time per time step (i.e., two thirds of 36, or 24), then it can use fewer resources. Further, in our experiments, the task priorities are assigned high to low (0 to 4) in the following order; *Gray-Scott*, *Isosurface*, *Rendering*, *FFT*, *PDF_Calc*. The priorities indicate the relevance of these tasks over others. For rectifying the effects of runtime performance of the analysis tasks

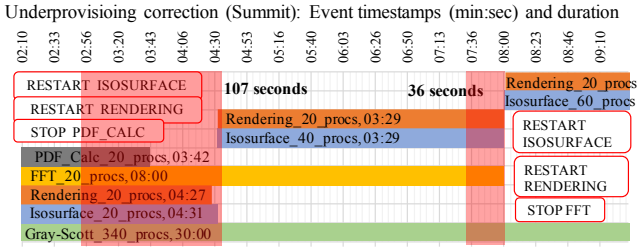

Figure 8: Gantt-chart showing the experiment performed on Summit with the Gray-Scott workflow to demonstrate correcting under-provisioning of resources along with the response times of DYFLOW

on the simulation to maximize the simulation performance, these policies are only applied to the analysis tasks. We did not have to set policy priories in the XML as the policies cannot conflict for the same workflow task.

We discuss the under-provisioning scenario in detail. In all our experiments (including the experiments in other sections), Arbitration processes the suggested actions only after 2 minutes from the start of the experiment to ensure all the running tasks have made some progress. For a similar reason, it discards all the suggested actions for 2 mins after the running workflow is modified.

Table 2 provides the initial configuration for the under-provisioning experiment on Summit and Deepthought2. **Summit:** Figure 8 shows all the dynamic events, giving the timestamps and durations. Figure 9 shows the average time per timestep as Decision receives them. Gray-Scott was started along with all the analysis tasks; *Isosurface*, *Rendering*, *PDF_Calc* and *FFT*. After 2 mins into the experiment, Arbitration considers the suggestion from policy *INC_ON_PACE* to increase the number of processes for all the analysis processes – the average time per timestep was above the threshold of 36 seconds. Arbitration only enables the action to increase the number of processes of *Isosurface* from 20 to 40 by acquiring the extra resources from *PDF_Calc*. Due to the runtime dependency on *Isosurface*, *Rendering* was also restarted. Arbitration took 107 seconds to finalize the plan and wait for Actuation to finish executing it.

After waiting for 2 mins, Arbitration again considered the actions suggested. At this time, policy INC_ON_PACE suggests increasing the number of processes for the analysis as the average time per timestep was above the threshold of 36 seconds. Only the action to increase the processes of *Isosurface* from 40 to 60 processes is enabled by acquiring the extra resources from *FFT_Calc*. As previously noted, *Rendering* was restarted due to its runtime dependency on *Isosurface*. Arbitration took 36 seconds to finalize the plan and wait for Actuation to apply the plan. After these changes, the average time per timestep for all the tasks was within the desired interval. **Deepthought2:** In a similar experiment on Deepthought2 (graphs not shown), *Isosurface* was restarted by acquiring resources from *PDF_Calc* and *FFT_Calc* while *Rendering* was restarted due to its runtime dependency. The time to finalize the plan of action and execute it was 87 seconds. The response times significantly reduce on both clusters if the tasks are not allowed to terminate gracefully (i.e., the time to finish the current timestep after receiving a kill signal). Without using DYFLOW, the experiment exceeds the allocation time limit, and the workflow tasks terminate prematurely due to timeout (requiring 10-12% additional time to finish).

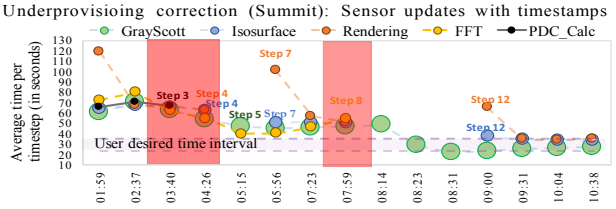

Figure 9: Average time per timestep information obtained from the Gray-Scott workflow tasks used by DYFLOW to improve performance on Summit.

```
<sensors>
   <sensor id="STATUS" type="ERRORSTATUS">
      <group-by> <group granularity="task" reduction-operation="FIRST"/> </group-by>
   </sensor>
 ... ... ...
<policies>
   <policy id="RESTART_ON_FAILURE" >
      <eval operation="GT" threshold="128" />
      <sensors-to-use> <use-sensor id="STAUS" granularity="task"/> </sensors-to-use>
      <action> RESTART </action>
       <frequency seconds="5"/>
... ... ...
```

Figure 10: XML example illustrating the user specification for restarting tasks on failure.

4.5 Managing workflow state in response to failures

Large-scale workflows sometimes desire the capability to respond to failure events. Different runtime failures can affect the workflow tasks, for instance, hardware failures such as a node or network failure or software failures such as memory corruption due to software errors. With *in situ* workflows, another type of failure is losing timestep information when the tasks reset, i.e., stopping and restarting at runtime (as evident in Fig. 9), or buffer overwrites when buffer capacity is exceeded. Failure diagnosis can be difficult, specifically in the case of software-generated or dataflow-related failures. Identifying the cause of failure in these scenarios requires deep analysis. Therefore, the focus of this section is demonstrating how DYFLOW can helps workflow achieve resilience to some types of hardware failures.

Arbitration continually collects the status of allocated resources from Actuation, which (indirectly) relies on the underlying job scheduler to provide this information. During resource reassignment, Arbitration ensures the exclusion of problematic resources. In the experiment, we show a form of resilience to node failure(s) in the cluster.

We use the molecular dynamics *in situ* workflow where the simulation is tightly coupled and co-located with three analysis tasks at runtime. Figure 10 shows the sample XML to enable the workflow tasks to restart after failure. To become aware of failures, we define a sensor, STATUS, that reads the error files generated by job schedulers when tasks fail to get the error number returned. The metric is computed at task granularity and returns the error number read by the first MPI process (rank 0). From the cluster scheduler's view, Savanna is the job script; therefore, we read the exit status saved by Savanna after the workflow task completes or fails. To detect failure, we define a policy, *RESTART_ON_FAILURE*, that restarts the workflow tasks whenever the error number is greater than 128 (the standard exit codes for system signals). A user can provide a script to run before the restart to ensure that the tasks resume correctly. Further, the task priorities are assigned high to

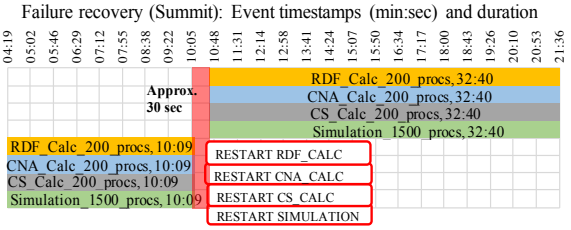

Figure 11: Gantt-chart showing the experiment performed on Summit with LAMMPS workflow to demonstrate resilience to node failures.

Table 3: Initial configuration for *LAMMPS* workflow on Summit and Deepthought2 used for failure resilience

TASK	SETTING	Summit	Deepthought2
LAMMPS	PROCESSES	1500 (30 PER NODE)	100 (14 PER NODE)
LAMMPS	TOTAL ATOMS	65536000	8192000
LAMMPS	TOTAL STEPS	1000	1000
CNA_CALC	PROCESSES	200 (4 PER NODE)	20 (2 PER NODE)
RDF_CALC	PROCESSES	200 (4 PER NODE)	20 (2 PER NODE)
CS_CALC	PROCESSES	200 (4 PER NODE)	20 (2 PER NODE)
ALL ANALYSIS	TOTAL STEPS	100	50

low (0 to 3) in the following order; *Simulation*, *CS_Calc*, *CNA_Calc*, *RDF_Calc*.

Table 3 shows the initial configurations of LAMMPS on Summit and Deepthought2. **Summit:** Figure 11 shows the timestamps and durations of the dynamic events. 10 mins into the experiment one of the allocated nodes was taken out of service, causing the entire workflow to fail. Arbitration restarts all the tasks by excluding the failed node from the resource assignment and replace it using one of the free nodes in the allocation. In this experiment, we allocated 2 additional nodes. However, if free nodes were not available, DYFLOW will use the resources from low-priority tasks to restart the higher priority tasks. After restart, the simulation resumes from the last checkpoint(i.e., timestep 412), and all the tasks repeat several timesteps. The response time for Arbitration to finalize and wait for execution of the plan was ≈ 0.2 seconds. The additional time results from the delay imposed by frequency settings in evaluating the sensor output. **Deepthought2:** In a similar experiment on Deepthought2 (graph not shown), the response time for the reconfiguration was 0.4 seconds.

4.6 Cost analysis for using DYFLOW

On average the lag between an event and initiation of a response was less than 1 second based on results from both clusters. These times exclude the effects of the decision frequency set by a user. The lag time varies depending on the volume of data processed for metric computations. For instance, the average lag time is lower when a single variable is read from a file on disk (i.e., 0.2 seconds). When TAU-generated data is read that is actively streamed using ADIOS2 (as one of the values in a two-dimension variable), the average lag time is approx. 0.5 second. The total response time includes the time spent by Arbitration to finalize the plan and Actuation to apply the resultant changes to the workflow. The time taken by workflow tasks to terminate gracefully (i.e., the time to finish the current timestep after receiving a kill signal) dominates the response time. In our experiments, approximately 97% of the response time was spent waiting for tasks to terminate after receiving the signal, on both clusters. Overall, DYFLOW incurs a small cost to apply the workflow modifications at runtime, which can vary significantly

with the experiment setup and environment. However, the time spent formulating the plan is low.

5 Related work

To cope with the complexity of specifying and executing scientific workflows, many workflow management systems (WMSs) have emerged [2, 4, 7, 14, 19]. These systems focus on providing support via simplified scripting languages to ease workflow design on supercomputers and other systems. Systems like RADICAL [17] use a building blocks approach to enable interoperability across HPC machines supporting large-scale science. For workflow orchestration, cloud computing [3] has emerged as an alternative to dedicated clusters. Employing on-the-fly resource acquisition and release support they easily enable management systems to automate workflows. Various scheduling and provisioning strategies have been studied to optimize resource utilization or time or cost under various QoS constraints such as deadline or budget.Container-based solutions, such as Kubernetes[6], extend this automation by allowing users to define and customize performance metrics to adapt resources (scaling up and down) at runtime. However, all these solutions are limited to loosely-coupled workflows, where data exchanges are done through files and dependent workflow tasks run at different times.

Recent studies [5, 10, 20] have demonstrated the benefits of dynamic resource management on supercomputers. For instance, Dayal et al. [5] introduced queue monitoring policies that increase or decrease the number of task processes based on the work pending in the queue. Zheng et al. [20] and Goswami et al. [10] show improvements in overall runtime performance by running analysis tasks on CPUs and GPUs that have been allocated to simulations and are idle or waiting on other resources (e.g., large I/O operations). Several projects, for example Perarnau et al. [15] and Vallee et al. [18], introduce container-based solutions on supercomputers to enable dynamic resource management. Emerging workflow management systems, such as Flux [1], introduce nested hierarchical scheduling to address various workflow management challenges including high throughput, job monitoring, co-scheduling, and job portability on supercomputers. Our work addresses modern scientific workflow runtime challenges such as those imposed by *in situ* analysis, providing a flexible platform to take advantage of dynamic orchestration on the supercomputers.

6 Conclusion and future work

In this paper, we have shown a flexible framework that enables complex scientific workflows to employ various benefits of dynamic orchestration on supercomputers. Our framework compartmentalizes dynamic management into four stages; Monitor, Decision, Arbitration, and Actuation. These stages are exposed as sensors, policies, and rules supporting fine constructs that enable workflow users to easily and inclusively express different dynamic requirements. The framework supports an in-build arbitration protocol that reserves the final authority over the runtime changes ensuring that the workflow runtime state is always valid and consistent.

Via experiments on three different scientific workflows, $XGC1 - XGCa$, $Gray - Scott$ and $LAMMPS$ on a standard Linux cluster and a state-of-the-art supercomputing cluster, we have illustrated that a dynamic management system based on our framework enables users to easily employ runtime orchestration capabilities for both loosely- and tightly- coupled workflows, incurring low cost to carry-out runtime changes. These scenarios show that the framework enables workflow tasks to respond to science-driven events, reassign resources in response to performance assessment, or recover from failure events.

We see great potential in investigating additional complex runtime scenarios where the role of Arbitration can be extended from a reactive to be a pro-active or predictive stage that can also assist in decision making. We also foresee opportunities in exploring instrumentation for *in situ* applications so that finer-grained control operations, beyond just stopping and relaunching, can be used to reconfigure a workflow.

[6]Kubernetes website: https://kubernetes.io

References

[1] D. H. Ahn, et al. 2018. Flux: Overcoming Scheduling Challenges for Exascale Workflows. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 10–19.

[2] M. Albrecht, et al. 2012. 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*.

[3] E. N. Alkhanak, et al. 2016. Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. *Journal of Systems and Software* 113 (2016), 1–26.

[4] D. Barseghian, et al. 2010. Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics* 5, 1 (1 Jan 2010), 42–50.

[5] J. Dayal, et al. 2015. SODA: Science-Driven Orchestration of Data Analytics. In *2015 IEEE 11th International Conference on e-Science*. 475–484.

[6] E. Deelman, et al. 2018. The Future of Scientific Workflows. *Int. J. High Perform. Comput. Appl.* 32, 1 (Jan. 2018), 159–175.

[7] E. Deelman, et al. 2016. Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing* 20, 1 (2016), 70–76.

[8] M. Dorier, et al. 2019. The Challenges of Elastic in Situ Analysis and Visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '19)*. ACM, 23–28.

[9] I. Foster, et al. 2017. Computing Just What You Need: Online Data Analysis and Reduction at Extreme Scales. In *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, et al. (Eds.). Springer International Publishing, Cham, 3–19.

[10] A. Goswami, et al. 2016. Landrush: Rethinking In-Situ Analysis for GPGPU Workflows. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 32–41.

[11] S. Janhunen, et al. 2015. Integrated multi-scale simulations of drift-wave turbulence: coupling of two kinetic codes XGC1 and XGCa. In *APS Meeting Abstracts*.

[12] S. Ku, et al. 2009. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion* 49, 11 (2009), 115021.

[13] K. Mehta, et al. 2019. A Codesign Framework for Online Data Analysis and Reduction. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 11–20.

[14] E. Ogasawara, et al. 2013. Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience* 25, 16 (2013), 2327–2341.

[15] S. Perarnau, et al. 2017. Argo NodeOS: Toward Unified Resource Management for Exascale. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 153–162.

[16] S. S. Shende et al. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311.

[17] M. Turilli, et al. 2018. Building blocks for workflow system middleware. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 348–349.

[18] G. Vallee, et al. 2019. On-node Resource Manager for Containerized HPC Workloads. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 43–48.

[19] M. Wilde, et al. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37 (2011), 633–652.

[20] F. Zheng, et al. 2013. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)*. ACM, Article 78.