ABSTRACT

Title of Dissertation: A FLEXIBLE APPROACH FOR ORCHESTRATING ADAPTIVE SCIENTIFIC WORKFLOWS FOR SCALABLE COMPUTING

Swati Singhal,
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Alan Sussman
Department of Computer Science

Modern scientific workflows are becoming complex with the incorporation of non-traditional computation methods, and advances in technologies enabling on-the-fly analysis. These workflows exhibit unpredictable runtime behaviors and have dynamic requirements. For example, such workflows must maintain overall performance and throughput while dealing with undesired events, adapting to failures, and supporting data-driven adaptive analysis. A fixed, predetermined resource assignment common to HPC machines is inefficient for overall performance, throughput, and data-driven adaptive analysis. While solutions exist to enable elastic resource management, there is no support that can manage the workflows at runtime to determine when the resource assignment and/or the runtime state of tasks (i.e. stopping, starting, changing the task parameters for adapting analysis, or changing how data is sent/received by the workflow tasks) needs to be revised, and perform the feasible changes at runtime accordingly.

This dissertation provides a flexible and portable model, DYFLOW, with strategies to automate the management of scalable and adaptive workflows. The model gathers runtime statistics, tracks the occurrence of important events, and finalizes a plan of action to execute in response to events that occurred, by mediating between suggested actions with respect to the running state of the workflow tasks and resource availability. Further, the model supports a wide range of constructs and tunable parameters that allow users to express events of interest, select prospective responses, and set various preferences to set the service expectation, e.g., throughput, performance, resilience to failures, or quality of results. To showcase that the DYFLOW model supports adaptive functionality desired for emerging workflows, several examples of problematic behavior are demonstrated where DYFLOW accommodates the specific requirements and automates the runtime management process for scientists while delivering the quality of service desired.

A FLEXIBLE APPROACH FOR ORCHESTRATING ADAPTIVE SCIENTIFIC WORKFLOWS FOR SCALABLE COMPUTING

by

Swati Singhal

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Dr. Alan Sussman, Chair/Advisor
Dr. Matthew D. Wolf
Dr. Abhinav Bhatele
Dr. Howard Elman
Dr. Donald Yeung

Acknowledgments

First and foremost, I'd like to thank my advisor, Dr. Alan Sussman, without whose guidance this dream would have been impossible. I am grateful for his invaluable insights and strong support that helped me nurture my ideas and pave my path toward the completion of this goal. He never doubted me and my work, even when I developed self-doubts. A special thanks to Dr. Matthew D. Wolf for his irreplaceable support in finishing this thesis. His expertise in workflow orchestration helped me gain perspective and shape my ideas in the right direction. Despite his busy schedule, he has selflessly made himself available for help and advice. I feel extremely fortunate to be able to work with and learn from these extraordinary individuals.

Thanks are due to Dr. Abhinav Bhatele, Dr. Howard Elman, and Dr. Donald Yeung for agreeing to serve on my thesis committee and for sparing their time reviewing the manuscript.

Words cannot express my gratitude towards all my loving family members. Especially, my husband Anuj, who stood by my side every step of the way, encouraging me and cheering me during setbacks and hardships. And my son Vihaan, who has been a source of immense joy and happiness in my life. I want to dedicate my work to my parents, Dinesh Kumar Singhal and Manju Singhal, whose unconditional love and care always gave me the strength to keep moving forward against all odds.

I also want to extend my gratitude to Dr. Neelima Gupta, who mentored me during my master's. Her guidance has been instrumental in implanting the idea that crystallized into a

decision to aspire to this goal.

Lastly, I want to thank everyone and every situation that lead me here. It has been a memorable and unparalleled experience in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Large-scale scientific workflows are often built from a complex set of coupled tasks that can include simulations, data analysis, and visualization, among others. With advances in computing power on state-of-art HPC systems, the traditional file-based approach of coupling tasks and exchanging data is becoming a significant performance bottleneck – I/O performance is unable to keep up with the volume and velocity of the data generated. To bridge this gap, a new model of *in situ* analysis has emerged as a promising alternative for workflow coupling [47]. *In situ* analysis minimizes the number and size of disk I/O operations by replacing disk storage with in-memory staging (buffering), on-the-fly execution of analysis, on-node, and off-node data transfers. But, unlike loosely coupled workflows where the workflow tasks run discretely and exchange data through files, managing *in situ* workflows is challenging as the tasks run in parallel, share resources, interfere with each other's performance, have different rates of data flow, and sometimes fail [24] which may require revising resource assignment at runtime. On-the-fly execution approach further opens opportunities for adapting analysis methods and/or tasks based on the state of the simulation. As workflow tasks become more dynamic in their structure, behavior and requirements for the experiment, scientists can no longer rely on the static workflow management support offered by existing systems and tools.

Unfortunately, resource and job management support on most systems designed for HPC,

e.g. compute clusters, is usually static and doesn't support on demand resource acquisition and release. Thus, predetermining an efficient resource assignment becomes challenging for a workflow with changing runtime resource requirements, resulting in over-provisioning of resources or loss of workflow performance or failures due to under-provisioning. While there are attempts to support elastic resource management [1, 23], they are not sufficient to cater to the runtime challenges of dynamic workflows [24]. Orchestration services are required that could manage the workflows at runtime to determine when the resource assignment and run status of tasks (i.e. stopping, starting or changing the task parameters for adapting analysis) need to be revised, and perform the feasible changes at runtime accordingly. Management of adaptive workflows often results in a chicken or the egg dilemma, as the inflexibility to change resource assignments or manipulate job runtime state on HPC clusters prevents scientists from availing themselves of the benefits of on-the-fly analysis. On the other hand, lack of widespread use of *in situ* approach slows the research and development of technologies supporting adaptive workflows.

Therefore, this dissertation offers a flexible orchestration model, called DYFLOW, that autonomously manages adaptive workflows, supervising the workflows to monitor and respond to the important events and meet desired Quality of Service (QoS) goals (e.g., maintaining throughput, performance, the accuracy of the result, resilience to failures, etc.). DYFLOW provides scientists the means to transparently express and control the events that are relevant to their workflow. Managing *in situ* workflows is complex where requirements could be specific, so often proposed orchestration solutions rely on interactive commands from a human-in-the-loop. An orchestration model, such as DYFLOW, that can reliably automate the management of dynamic workflows based on criteria (policies) set by scientists can significantly minimize the effort required by scientists to improve undesirable runtime behavior, especially for long-running ex-

periments. For complex workflows, expressing policies can often involve a team of scientists and other experts, a key advantage that can manage the workflows at runtime so that team members can easily use, collaborate, and validate any changes performed in response to the dynamic events that are important for a computational experiment.

DYFLOW is a four-stage conceptual model that categorizes dynamic management as **Monitor**, **Decision**, **Arbitration**, and **Actuation** stages. These stages are responsible for monitoring the workflows and gathering meaningful data, identifying the occurrence of events and generating relevant responses, mediating amongst the suggested responses for these events, and applying suitable actions to change the workflow state. DYFLOW contains programmable constructs that are made available to the user through an easy-to-use interface. These constructs support rich features to enable users to set policies to define events and select appropriate responses to those events, and set preferences (optional) to assist in mediating between events and responses. For instance, these features include methods to collect, process, group and reduce runtime measurement data, methods to analyze processed data to observe patterns, and various runtime actions to manipulate resource assignments, change workflow tasks runtime states, or manage flow of data between workflow tasks.

DYFLOW is not a standalone workflow management system (WMS), but an additional service that dynamically manages workflow tasks at runtime and employs the basic services provided by an existing WMS. Extensive research in workflow management and task scheduling has resulted in numerous systems that focus on different aspects of managing workflows, and this research has designed and built DYFLOW leveraging these prior systems and their insights. DYFLOW uses these services 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 [51], for acquiring real-time monitoring information. As a demonstration, DYFLOW has been implemented using an existing workflow management system, Cheetah/Savanna [26]. Some of the work in this dissertation has been published [42, 53].

In addition to the scientific workflow community, this research draws 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 [61], 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.

To demonstrate the effectiveness of the DYFLOW model, several example cases of problematic workflow behaviors are presented where DYFLOW adapts to the specific requirements of the workflow and automates the runtime management process for scientists. From experiments performed on different HPC platforms, the dissertation illustrates that DYFLOW incurs minimal overhead.

1.1 Thesis statement

Is it possible to orchestrate adaptive scientific workflows at runtime based on user-specific requirements? By designing a flexible and portable model with runtime strategies, I have demonstrated that the orchestration of adaptive workflows can be automated based on user-specified requirements to meet the desired quality of service. Using examples from several in-

teresting runtime scenarios, I have shown that the model meets user expectations while incurring minimal overhead.

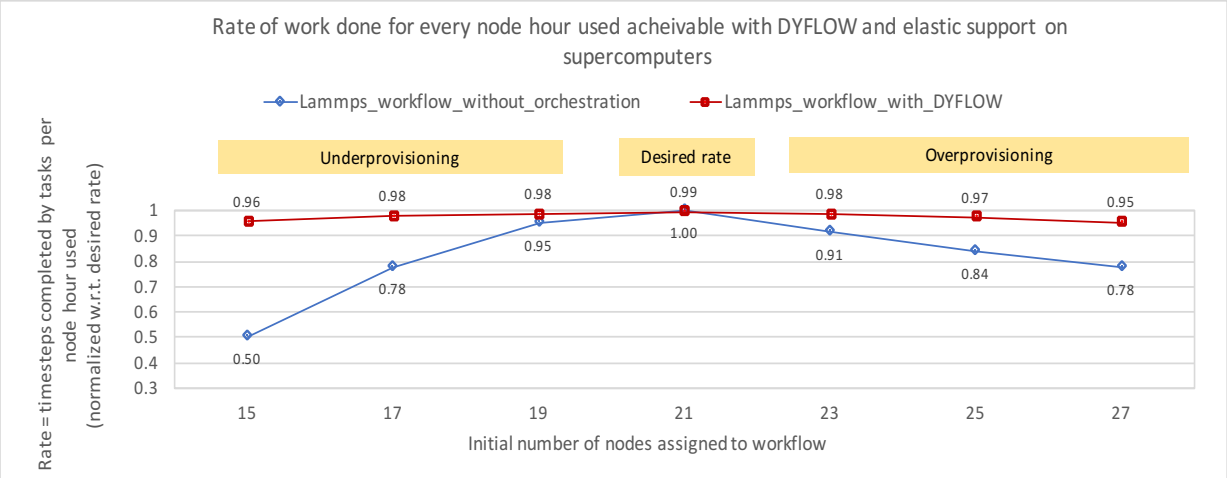
1.2 Motivating example

In many large-scale scientific simulations, analysis codes identify features that emerge over the course of the simulation, and the memory footprint, compute requirements, and total runtime for such analytics can be challenging to predict statically. Determining efficient resource assignments is a fundamental challenge on large-scale parallel systems, even if a workflow does not exhibit large runtime variability. A scientist must decide how to assign resources to each workflow task to match their data flow rates.

In practice, scientists usually either choose to over-provision based on a worst-case scenario or make a best-guess choice and hope for the best. Such a decision can result in premature termination of the workflow due to under-provisioning or wasting expensive compute resources due to over-provisioning. Over-provisioning is more frequently observed as (1) it avoids situations where tasks are vulnerable to interference from other co-located tasks, and (2) it enables meeting the potentially very high resource requirements of tasks at peak times during the workflow run. These undesirable situations can be avoided with on-demand resource assignment along with an orchestration service such as DYFLOW that can detect and respond to events and circumstances that can benefit from changes in resource assignment.

Figure 1.1 illustrates scenarios constructed from a molecular dynamics *i*n situ workflow (LAMMPS) where a simulation was performed with three analysis tasks. The scenarios represent three categories of experiments where the initial resource assignments (shown on the X-axis) lead

Figure 1.1: A comparison of under-provisioning, desired, and over-provisioning scenarios constructed using an *in* situ workflow on a standard Linux cluster (Deepthought2) with and without DYFLOW orchestration support.



to under-provisioned, correctly provisioned and over-provisioned outcomes. The figure demonstrate that a higher rate of work is achievable with DYFLOW when elastic resource support is available on a parallel system to add or release resources at runtime as necessary. The rate of work done is defined as the number of simulation timesteps completed by workflow tasks for every node hour consumed. The x-axis shows the initial number of nodes assigned to the experiment, while the y-axis shows the normalized rate, i.e., the timesteps completed per node hour used. In the under provisioning case, the rate of work done gradually increases as more CPUs are added. In the over provisioning case, the rate of work done decreases as more CPUs are added because the additional CPUs added are mostly idle and the work done every node hour consumed decreases. With DYFLOW a steady rate is achieved that is close to the user expectation of the desired pace of progress (expressed as policy settings) irrespective of the initial resource assignment as the resources are adjusted (i.e., assigned and released) on the fly. Each of the experiments ran for 30 minutes of wall-clock time. This scenario repeated the experiments using DYFLOW and compared the rate of work done, i.e., the number of timesteps completed by the workflow

tasks, for every node hour used by the experiment (shown on the Y-axis). For clarity, the graph has normalized the rate to the properly provisioned scenario without orchestration. DYFLOW continuously streams the monitoring data from the workflow tasks to determine the pace of the experiment and adds additional nodes or releases nodes (in increments of up to two nodes at a time) to increase, decrease or redistribute (co-locating the tasks) the number of processes assigned to the tasks to meet the user expectation set in the policy statements. As is evident from the graph, DYFLOW maintains a steady rate of work done per node hour used, which is close to the rate desired by the user, irrespective of the initial resource assignment.

DYFLOW uses one core on each node to monitor tasks. The impact of monitoring on workflow tasks is low. Since on-demand support for resource acquisition and release is not commonplace on large parallel systems, this experiment has emulated this feature by allocating spare nodes in advance. In practice, the overheads could be slightly higher, as there will be variability imposed in obtaining additional nodes due to delays in acquiring the nodes from the cluster's resource manager. To minimize the delay, DYFLOW can initiate the resource requests in advance by making predictions about future resource requirements.

1.3 Background

Scientific Workflows: A scientific workflow is defined as a set of tasks where tasks may be dependent on one another. A task in the workflow could be a parallel (i.e., data-parallel or communication intensive) or a serial program. A typical example of these tasks include simulation, analysis, or visualization. Simulation tasks are often large-scale parallel tasks that model a scientific phenomenon. Analysis and visualization tasks are often post-processing techniques

applied to simulation data to extract meaningful information or make observations.

The data dependencies or coupling of scientific workflow can be modeled as a directed graph, where vertices represent the tasks and edges represent the data from the parent and the dependent task(s). Scientific workflows are coupled in two ways; loose coupling where data exchanges are supported through files, or tight coupling where data is exchanged through in-memory staging areas (an *in-situ* approach). In loosely-coupled workflows, the tasks can run discretely, such that a task T_j that depends on the data from a running task T_i can be scheduled/run when the task T_i finishes. However, in tightly coupled workflows the data to be exchanged is available while the experiment runs, so all the dependent tasks must run in parallel to generate the end result. These workflows are susceptible to various runtime events and situations.

With advances in computing technology, modern workflows are adopting *in situ* techniques to overcome the I/O bottlenecks of file-based coupling. As analysis techniques derived from non-traditional HPC methods, such as machine learning and graph algorithms, are gaining popularity, the requirements of modern scientific workflows increasingly show the need for dynamic control.

In situ analysis: *in situ* analysis and technologies are emerging as a promising replacement for temporary disk writes that have been done traditionally for exchanging data between different applications. At its core, *in situ* technologies use in-memory buffering or node-to-node data transfers to enable consumption/exchange of data while it is being generated, without the need to be written to disk. This means the applications can be run in parallel by means of in-memory coupling, which can be of either a synchronous or asynchronous nature. On-the-fly coupling this way has several advantages. For instance, this approach not only improves performance by limiting the number of I/O operations, but decreases the end-to-end runtime and increases resource utilization by co-locating the workflow tasks Further, by supporting on-the-fly analysis,

this approach provides early insights into the simulation data which could be useful, for instance, to correct the course of the simulation.

There are currently two modes of data staging, (1) *off-node staging* - dedicated servers or compute nodes on the cluster are allocated to store and manage the generated data in memory, for instance using DATASPACES [21]; or (2) *on-node staging* - data is stored on the same nodes where it is generated in the cluster or external libraries are used to manage and distribute the data, for instance the in-memory staging options provided by the Adaptable I/O system (ADIOS v2) [28] or the FlexPath library from ADIOS v1 [17]. Such advances in *in situ* technologies enable running workflow pipelines that can involve multiple analysis workflow components running in parallel with the simulation, resulting in huge performance benefits.

The implementation of DYFLOW built for this dissertation relies on ADIOS2 for *in situ* interactions because (1) ADIOS2 does not use additional resources (or dedicated nodes), (2) it has a simple interface that makes it easy to incorporate into workflow applications, (3) it supports both synchronous and asynchronous coupling, and (4) it is open source, and its modular design is relatively flexible to modify for experimental purposes.

Job scheduling on HPC systems: Supercomputers are the most common and preferred systems for HPC applications. A supercomputer is a massively parallel shared resource, and the jobs are scheduled using a centralized batch system. There are generally two types of computing units/nodes on these machines; login and compute nodes. Login nodes are directly accessible to a user upon login which can be used to build and compile applications. The compute nodes are where the user programs are run. The batch system on these systems is responsible for deciding how to assign the compute nodes' resources amongst user applications and when to launch the user jobs. Some popular batch systems used on modern supercomputers are SLURM [54],

PBS [48] and IBM Spectra LSF [38]. The users submit their jobs by means of a job script that defines what jobs are to be run and how many resources these jobs would require. The submitted job scripts are initially queued by the batch system, and whenever the resources are available, the batch system selects a job script that can be scheduled. The decision about which job script to be scheduled at any time is based on predetermined strategies, for instance, a first come, first served strategy with job priorities or scheduling based on multiple job queues. Once the job is launched, a user cannot change or control how the jobs are run, except for cancelling the jobs.

Each cluster enforces policies which govern when, how and what jobs can be launched and under what conditions. These policies can restrict how many jobs from the same or different users can be assigned to a node, how long a job can run, which running jobs can be preempted and for what reasons, etc. These policies are decided and set up by cluster administrators.

Each batch system also has a resource manager agent that is responsible for determining the resource assignments, monitor the running jobs and collect the statistics of all the running jobs like the exit status and maximum/minimum resource usage. These statistics are saved to a database, which can be queried by users or system administrators anytime through scheduler commands.

Flux [1] is a next generation resource and job scheduler system that provides an abstraction for scientist to launch and manage their jobs across different clusters in a portable way. Unlike traditional job schedulers, Flux support many customizable features. To support efficient co-scheduling of jobs, Flux allows users to determine how to configure their jobs on compute resources at various levels of heterogeneity such as within (or across) nodes, cores, sockets, GPUs, etc. Flux also provides various an in-memory key-value store and control API support for users to store the job's metadata, output and status. Users can also choose the type of scheduling policy

that is efficient for their workflows. There are ongoing efforts to support an elastic model in Flux to dynamically grow or shrink resource assignment of a running workflow task. This feature is not yet widely available.

1.4 Related Work

Support for elastic computing: Many HPC applications rely on the MPI library for scaling and inter-process communication. MPI-based applications cannot easily grow and shrink without restart. However, there are increasing efforts to incorporate elastic resource management support into process management on HPC clusters [14] and into MPI [44] in the future. With limitations in current MPI support, projects such as Perarnau et al. [49] and Vallee et al. [59] introduce container-based solutions to support dynamic resource management. S. Iserte et al. [30] demonstrated the benefits of resource adaptation by building an API to support runtime flexibility for standalone MPI-based embarrassingly parallel applications to change the number of processes. Other solutions rely on checkpoint and restart to enable applications to adapt to a different number of resources [27]. The implementation for this dissertation also relies on the checkpoint-restart approach for elastic resource assignment.

Existing scientific workflow management systems on clusters: To cope with the increasing complexity of HPC workflows, many modern workflow management systems (WMSs) have emerged [2, 4, 6, 46, 60] that provide support via scripting languages to ease workflow design, and manage workflow execution and data access across multiple architectures. The user interfaces in these systems include command line-based scripting languages, XML-based, e.g. [46], or GUI-based drag and drop features, e.g. [6]. Some systems describe a workflow using directed

acyclic graphs (DAG), for instance Pegasus [19] and ParSec [11], and are capable of running large-scale, multi-stage workflows with support for monitoring workflows with some dynamism. However, many emerging scientific workflows require more than a centralized WMS to achieve high efficiency and support complex coordination requirements. RADICAL [58] is a recent effort that integrates components of different WMSs together. It is based on defining different building blocks supported by RADICAL [58], which does not require uniform interfaces so can be easily adapted to existing workflow management tools to support scalability and interoperability across various HPC platforms. RADICAL also supports resource management and execution of large scale ensemble runs, and generates various statistics related to pre-defined events on the building blocks.

Some legacy and modern workflow systems allow for dynamic user steering of workflows; that is, they provide the capability for a human-in-the-loop to alter a workflow at runtime. Systems [36,46,60] allow for programmatic workflow tuning must wait for a task in the workflow to complete before taking actions to modify the workflow execution. In these systems, monitoring, at most, is limited to tracking the start and end times of the workflow jobs, and they do not provide support for monitoring and adapting workflows based on dynamic conditions or events that arise from the workflow jobs at runtime.

Workflow management on the cloud: Cloud architecture is a collection of various computing resources, for instance, processing units, storage and servers that are owned and managed by private organisations for commercial purposes. Cloud computing enables its clients to access and utilize these resources by means of three types of services; Infrastructure as a service (Iaas), Software as a Service (SaaS) and Platform as a Service (PaaS). Since scientific workflows require infrastructure support, the rest of the discussion will be focused on IaaS. IaaS enables clients to

access storage, networking, servers and other computing resources on on-demand basis and pay only for the resources they utilize. Clients use these resources to run multiple virtual machines (VMs) on which they can lauch their jobs. Examples of IaaS resources include Amazon EC2 [3] and Microsoft Azure [43].

Cloud computing is emerging as a flexible alternative to on-premise cluster machines for running scientific workflows, as the resources can be acquired or released elastically and users pay for only the resources used at runtime. Many state-of-the-art workflow management systems (WMS), for instance Pegasus and Kepler, have incorporated support for scientists to utilize these features of cloud computing by managing and adapting cloud resources according to their workflow needs. To manage resources for the workflows, these WMS employs resource provisioning and resource scheduling methods. The resource provisioning methods determine the number of resources that the workflow jobs will require at runtime. Resource scheduling methods are responsible for managing how the resources are assigned to the workflow tasks at runtime. Resource provisioning and scheduling also support Quality of Service (QoS) constraints like deadlines or budgets set by a user. Depending on user requirements, the aim of these methods is to optimize resource utilization, time or cost. There is a large body of research to design effective resource scheduling strategies for cloud. Several studies [40, 61] present an overview and classification of these strategies. Scheduling strategies can be static, dynamic or hybrid (a combination of both static and dynamic strategies). Static scheduling methods schedule tasks based on the assumption that precise estimates of the timing and communication cost of the tasks and the resources are available before execution, while dynamic scheduling methods can be employed when no prior information is available and the cost can only be estimated at runtime. Hybrid methods are employed when the cost estimates are known, but the resources cannot be

assigned before execution. These strategies can be further classified as meta-heuristic methods and heuristic methods. Some examples of meta-heuristic approaches include Particle Swarm Optimization (PSO) [50], Genetic Algorithms [39], and Cat Swarm Optimization [9]. Examples of heuristic approaches include Deadline constraint scheduling [52], Heterogeneous Earliest Finish Time (HEFT)-based scheduling [5], Priority Impact scheduling [62], and Compromised-Time-Cost (CTC) scheduling [37].

Container-based solutions, such as Kubernetes[1], 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.

There has been a lot of emphasis on integration of HPC and cloud services to provide the benefits of elastic resource management. Cloud technology has also significantly advanced to achieve comparable performance to on-premise HPC clusters. However, cloud services are not cost-effective for running large-scale workflows compared to on-premise HPC clusters [7]. Moreover, support for *in situ* workflows is limited on clouds. There are some recent efforts [8, 15] to enable *in situ* processing of scientific workflows by integrating HPC clusters and cloud services. However, the dynamic orchestration support desired for *in situ* workflows is not available in these systems.

Dynamic management approaches on HPC systems: Recent studies [18, 29, 64] have demonstrated the benefits of dynamic resource management on HPC clusters. For instance, Dayal et al. [18] introduced queue monitoring policies that increased or decreased the number of component processes based on the work pending in the queue compared to the work queue

[1]Kubernetes website: `https://kubernetes.io`

for a component with which data is exchanged. GoldRush [64] and LandRush [29] attempt to improve overall runtime performance by utilizing otherwise wasted idle CPU and GPU resources available on nodes that have been allocated to simulations that are waiting on other resources (e.g., large I/O operations), so instead can run an analysis task by using careful task scheduling and switching methods. Another related system is ActiveSpaces [22], which attempts to bring computation to staging nodes and sends processed results instead of sending raw data to applications to minimize the amount of data that must be transmitted over the network. The work in this dissertation focuses on providing portable and extensible strategies that manage both tightly and loosely coupled workflows at runtime and are not tied to any specific workflow management system.

1.5 Outline

The rest of the document is organized as follows. Chapter 2 delves into the process of expressing the complex requirements of *in situ* workflows from the perspective of a team of scientists, discussing examples of challenging *in situ* workflow scenarios with dynamic needs. Following this, Chapter 3 discusses the details of the DYFLOW model and showcases how users can use the DYFLOW interface to express their workflow requirements. Chapter 4 probes deeper into the complexities of the arbitration protocol used to mediate the policy responses. The details of the prototype implementation of DYFLOW are presented in Chapter 5. Chapter 6 provides evidence of the benefits of applying the flexible orchestration approach to different scenarios with dynamic needs, and Chapter 7 discusses conclusions and future directions.

Chapter 2: Requirements for achieving desired orchestration

This chapter focuses on the various orchestration requirements for adaptive scientific workflows from the perspective of a team of scientists, by looking at the requirements of two real-life use cases with problematic behavior. Both scenarios are based on workflow containing tightly coupled tasks, i.e., all the tasks run simultaneously and depend on other running tasks for input data. The data sharing is enabled via in-situ services. In the first scenario, the tasks are co-located on the same compute nodes. Variability in the resource usage behavior of one or more tasks could affect the performance of other tasks as tasks share the resources of the compute node. The event of interest for this scenario is tracking significant memory bandwidth usage imposed by external interference from co-located workflow tasks or background tasks, and responding in a timely way to decrease the pressure on memory bandwidth. In the second scenario, the tasks are located on geographically distant machines. When the remote network connection used to transfer data (i.e, a timestep) to a task is congested, the overall workflow throughout could be affected. The event of interest for this scenario is tracking the slowdown in network transfer times and responding in a timely way to maintain throughput with an acceptable loss in accuracy.

2.1 Coupled tasks sharing compute nodes - simulation, analysis, and machine learning applications (MemContScenario):

In some instances, background tasks are run periodically to access and collect data. These tasks could be part of the workflow or service required for the experiment that shares resources with workflow tasks. For example, a workflow can have a machine learning task that is invoked periodically to train models to make better predictions, or these tasks could be the server instances of off-node *in situ* data management services like Dataspaces [21]. Co-locating workflow tasks on the same compute nodes is helpful in improving resource utilization and the overall workflow performance by improving data locality, i.e., consuming the data where it is generated. However, these background tasks can be highly unpredictable in their resource usage behavior and potentially affect other tasks that share resources on the same compute node, thereby degrading the performance of the workflow.

The specific workflow of interest contains a large-scale simulation (XGC), a particle subselector (ParSub), and a particle path analyzer (ParAnlz). XGC [35] is a gyrokinetic Particle-In-Cell code used for studying plasma turbulence in fusion devices such as Tokamaks and ITER reactors, where the reaction is confined through magnetic pressure. At each time step, the particle states are updated based on the underlying physics, and then the updated particles' locations are statistically determined and mapped to the grid cells. The output of each timestep is sent to the $ParSub$ component to filter the particles of interest in the simulation based on user-specified criteria. The selection criteria used in our workflow is uniformly distributed random selection, as the particles are equally likely to be in a plane of reference. The selected particles are then sent

to the last component $ParAnlz$, the main analysis component, which keeps track of the changes in the position of the particles using particle position vectors to compute particle trajectories. As a proxy to a set of background tasks that utilize shared resources, instances of the STREAM [41] benchmark were run with the fusion workflow tasks. The STREAM benchmark is run at a random time to create high memory bandwidth utilization.

2.2 Coupled tasks located on geographically distant machines (NetContScenario):

In some studies, a team of scientists collaborating from remote locations or different organizations may need to stream the output of an experiment running on a cluster to a set of local machines for visualization or analysis due to cluster access restrictions. Sometimes a scientist may need to upload experiment data to a cloud provider or a specialized data analysis machine for further analysis. Streaming data in real-time saves time and storage and avoids data replication. There could be variability in network speed between the cluster and the clients observed at different times. Slow network speed could become a bottleneck to the experiment, and significantly impact the overall throughput.

This workflow also has three main components; A large simulation (LAMMPS), a common neighbor analysis computation (CNA), and a visualization (Viz) task that generates images from the CNA output. LAMMPS [56] is a widely-used molecular dynamics simulation code, used for applications ranging from engineering nanomaterials to designing new alloys to exploring protein folding. The common neighbor analysis is one of the LAMMPS methods that is used

to study the structure of solids under pressure. On HPC clusters, typically only login nodes can perform data transfers to remote clients (the visualization task), hence the workflow uses an additional task (DMZ) that runs on the login node that streams data from the CNA task and sends it to the Viz task on the remote machine. The transfer is done in lock-step mode, where after receiving a simulation timestep the Viz task sends an acknowledgment to DMZ and requests the next timestep.

2.3 Gathering data to observe dynamic events

The first requirement relates to gathering data that will enable tracking the events of interest. This section presents insights into the process to identify data that could be collected at runtime to recognize the occurrence of an event.

Identifying the relevant data to gather requires an understanding of what data is available at runtime. Modern profilers provide useful runtime statistics about an application while it runs on a cluster through code instrumentation, such as resource usage, data transfer volume and speed, and more. The generated information could be periodically saved in a database and to be queried in real-time. Modern profilers like TAU also support direct streaming of these statistics in real time. However, the code instrumentation could slow down the application as it is constantly interrupted by profilers to record statistics. On Linux systems, the operating system creates temporary files that record a few useful application usage statistics such as the number of memory pages accessed, amount of physical memory allocated to an application, number of packets transferred on a network connection, and so on. This information can be retrieved periodically in real-time with minimal to no overhead from the running application. Libraries like PAPI [12] and LIK-

WIID [57] enable access to hardware counter data for capturing the performance of different components of a compute node at runtime, measuring per-core metrics (e.g., L1, L2, L3 cache misses and references for each core/thread) and so-called ”uncore” metrics [16] (e.g., read and write events received by a memory controller on a node, thermal power usage of a node). State-of-the-art profilers also integrate support from these libraries to access hardware counter data. For GPU usage statistics, NVIDIA Cupti [45] can be utilized. Finally, the application code could be modified to output any specific data required for event identification.

NetContScenario: First, the case of the molecular dynamics workflow is demonstrated. For this scenario, we wanted to measure the end-to-end transfer time of a timestep from the DMZ task to the Viz task to track slow transfer times. On Linux systems, the '/proc/net/dev' file provides the number of bytes that are transferred to and from the login node by a running process. These measures are available for transfers using the cluster's internal high-bandwidth interconnect or a WAN interconnect. Using these measures, an estimate of the time spent to transfer the amount of requested data can be performed indirectly. However, computing this separately for every timestep is not straightforward. The TAU tracing and profiling toolkit [51] supports code instrumentation to time methods, loops, and other blocks of code. We apply this feature to DMZ and instrument the method that transfers a timestep and returns when the Viz component sends an acknowledgment requesting the next timestep. This value is recorded by TAU after the instrumented method returns.

MemContScenario: Now, the complex use case of fusion workflow where we focus on memory usage patterns to recognize high memory bandwidth usage is demonstrated. Memory usage can be classified into two categories; (a) physical memory usage, which directly relates to a node's memory capacity, and (b) Memory bandwidth, which measures the rate of data exchanged

between memory and processors over some time.

Memory capacity is a straightforward measure of the total physical memory usage of the processes on a compute node (provided by many modern profilers) relative to the total physical memory available on a compute node. Resident Set Size (RSS) is a measurement statistic maintained by the operating system (i.e. Linux) to keep track of physical memory usage and is based on the page frames allocated to a given process at any given time. This information is available as a Linux temporary file $/proc/\langle process\_id \rangle$ for each running process.

On the other hand, memory bandwidth is complicated to measure. Even though modern processors provide some support via hardware counters that enables capturing the performance of different components, there are no direct measures for memory bandwidth. Further, there is a lot of variability and reliability in the support and performance of hardware counters across computer system architectures.

To evaluate the slowdown observed by a process due to memory transactions, we begin by measuring the cache miss rate (on most processors, this is for the last-level L3 cache since L3 misses must access the main memory). However, by itself, this measure has a lot of inherent variability during a run and more information is required to distinguish an acceptable momentary variation from a change due to memory congestion pressure. Based on extensive experimentation, we found two additional useful measures that helped to disambiguate normal and problematic cases; The memory stall percentage measures the percentage of CPU cycles that were stalled/idle waiting for load operations to complete, and the instructions per cycle give the total number of instructions completed per cycle. When using these three measures simultaneously, we have been able to correctly detect changes in memory pressure (while avoiding false positives).

This set represents a concise measurement that could be collected together (with high pre-

Table 2.1: Hardware counter for memory bandwidth computations on Deepthought2

Hardware counter	Description
LLC_REFERENCES	#cache requests to last level cache.
LLC_MISSES	#cache misses in last level cache.
CLK_UNHALTED_CYCLES	#cycles used for execution.
CYCLE_ACTIVITY: STALLS_LDM_PENDING	#cycles stalled waiting for memory loads.
INSTRUCTION_RETIRED	#instructions completed.

cision) from the cluster used in the experiments. When a workflow task experiences high memory bandwidth usage, the load stall percentage and cache miss percentage increase, while the instructions per cycle decrease. These three measures are not directly available through profilers or tracing libraries, but can be computed using hardware counters available on many modern architectures, as I now describe.

$$\text{load stall percentage} = \frac{\text{\#cycles stalled waiting for memory loads}}{\text{\#cycles used for execution}} * 100$$

$$\text{cache miss percentage} = \frac{\text{\#cache misses in last level cache}}{\text{\#cache requests to last level cache}} * 100$$

$$\text{instructions per cycle} = \frac{\text{\#instructions completed}}{\text{\#cache requests to last level cache}}$$

Table 2.1 describes several counters that can be collected on a standard Linux cluster, Deepthought2 that relate to last level cache accesses and misses, total CPU cycles utilized, total memory cycles when execution was stalled for memory load operations to finish, and the number of instructions completed.

This data can be collected using the PAPI [12] library support integrated into the TAU profiler [51] that provides the difference between the current and last values read from the hardware

counters. PAPI provides cross-platform support infrastructure for accessing the same logical categories of counters on different chip architectures. The counter values were read every second for this workflow to observe the events in a timely way without imposing large overheads on the running task.

Requirement 1. *While analysts (e.g., application developers, system experts, performance engineers, or data analysts for data-driven events) determine what information is useful to identify runtime events, a separate program needs to be developed that can collect the desired data at a given frequency, for example querying a database, real-time streaming or reading a file. Additionally, for a multithreaded or multiprocess task, the raw data must be further processed, i.e., filtered and aggregated based on all the processes of a task running on a compute node, and summarized into a single meaningful metric. For instance, to estimate the memory bandwidth utilization per node, the values read from every process of a task running on a compute node are aggregated.*

2.4 Defining events of interest from the metric values

Events can be described as extreme values, special values, or changes in the values from the computed metric. For some events, a straightforward condition to identify events from the metric values is to compare the measurement collected at any instance against a threshold. For instance, a condition comparing the resident set size measurement at any time against the total memory available on the compute node can detect high physical memory usage.

However, event identification based on measurements at a single point in time is not always reliable for defining events. The computed measurements can often be noisy and show

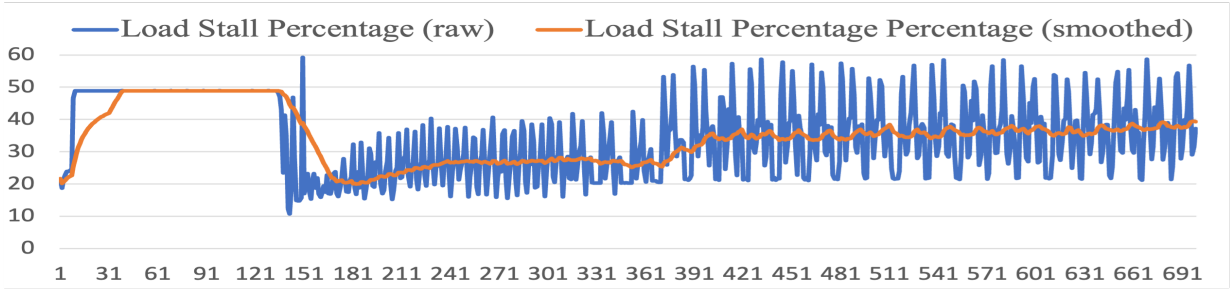


Figure 2.1: The graph shows raw load stall percentage data obtained by monitoring XGC in comparison to when it was smoothed by computing a rolling average over 30 samples. The X-axis represents time elapsed in minutes, and the Y-axis represents the load stall percentage. The metric is computed based on the raw counter data collected at a frequency of 1 second.

significant variation over short time periods. For example, Figure 2.1 shows that raw load stall percentage measurements have high jitter, i.e., the frequent presence of both low and high values over short time intervals. The presence of such a jitter makes it unreliable to differentiate between low-pressure and high-pressure situations for the memory bandwidth metric. Based on this observation, I now discuss the process used to identify if events of interest have occurred for the two use cases.

MemContScenario: Post-processing of the metric values, such as a smoothing function applied over a time interval, is often performed to mitigate the effects of measurement noise and enable better analysis of changes in behavior as workflow tasks to run, assuming the workflow runs for a long time compared to the measurement frequency. Hence, to deal with the unreliability of single point-in-time measurements, I evaluated a range of statistical functions for smoothing. From experimentation, I have found that an average computed for the most recent N measurements to be effective in capturing the events of interest. Figure 2.1 shows the effect of smoothing by using a rolling average computed for 30 values. I realized that choosing the value of the interval size M is important. A short window size may not eliminate the jitter effect, and a large window size may over-smooth the data, making it difficult to recognize changes in the metric.

For the memory bandwidth measure, I further observed that different tasks have different patterns of resource usage and thus have a relative threshold that defines high or low-pressure situations. For instance, it is difficult to distinguish between a task that experiences a maximum 30% cache miss percentage, 40% load stalls, and 0.7 IPC value in isolation, as compared with another task that may show similar metrics when run at the same time on a node with other memory bandwidth-intensive applications.

Since comparing absolute values against the metric value (smoothed) was not a reliable predictor of resource pressure, I looked for changes in these measurements that indicate changes in usage patterns. For this, I compute the percentage change in the measurements. The percentage change measure captures the amount of change in the measurements for the maximum variation observed in the measurements. Let, SV_{t_i} represents the smoothed metric value computed at time t_i and $SV_{t_{i-r}}$ represents the smoothed metric value that was computed in the past, i.e., at time t_{i-r}. The value r represents the size of the interval used in the computation. RV_{max} and RV_{min} represent the global maximum and minimum observed over the individual point in time measurements, respectively. Based on these values, the percentage change at any time t_i ($change\_percentage_{t_i}$) can be computed as follows.

$$change\_percentage_{t_i} = \frac{SV_{t_i} - SV_{t_{i-r}}}{RV_{max} - RV_{min}} * 100$$

The term $SV_{t_i} - SV_{t_{i-r}}$ gives the current change in the pattern with respect to the past, and the term ($RV_{max} - RV_{min}$) gives the maximum variation in the values observed up to time t_i. This metric can be compared against a threshold value (δ) to determine a significant increase or decrease in usage compared to the recent past. In the case of load stall percentage and cache

miss percentage, $change\_percentage_{t_i} > \delta$ is used to indicate a substantial increase in resource usage, and $change\_percentage_{t_i} < -\delta$ is used to indicate a substantial decrease in usage. The appropriate δ value can be set by a performance analyst based on experimentation. The δ value can vary depending on the underlying system and sensitivity (i.e., performance degradation) of the workflow tasks to external interferences.

Figure 2.2 showcases how the behavior of the load stall percentage, cache miss percentage, and instructions per cycle measures change due to external interference related to memory bandwidth usage. Since XGC is the most sensitive task to bandwidth pressure, in the rest of the discussion I will focus on XGC. The graph shows the percentage difference in the values of the three measurements when memory bandwidth was stressed by external interference after 360 seconds, with respect to the situation when memory bandwidth usage was low. To elaborate, the comparison is performed on two types of experiment arrangements; (1) tasks share the compute nodes and memory bandwidth usage is low (**LMB**), and (2) tasks share the compute nodes and memory bandwidth becomes high after instances of the STREAM benchmark start around 6 minutes into the experiment (**HMB**). The load stall percentage and cache miss percentage measurements show an increase, while instructions per cycle show a decrease, as the STREAM benchmark starts and stresses available memory bandwidth.

I evaluated the effect of smoothing with varying window sizes, ranging from 5 to 120 samples, where a sample is collected every second from different tasks. A window size of 30 (i.e., $N = 30$) smoothed the data without losing the pattern. Next, I examined different values for r by varying the interval size from 30 to 120. While large interval sizes were able to detect the change in memory bandwidth usage more often, they also take longer to begin detecting the change. An interval size of 30 (i.e., $r = 30$) provided a reasonable tradeoff between the

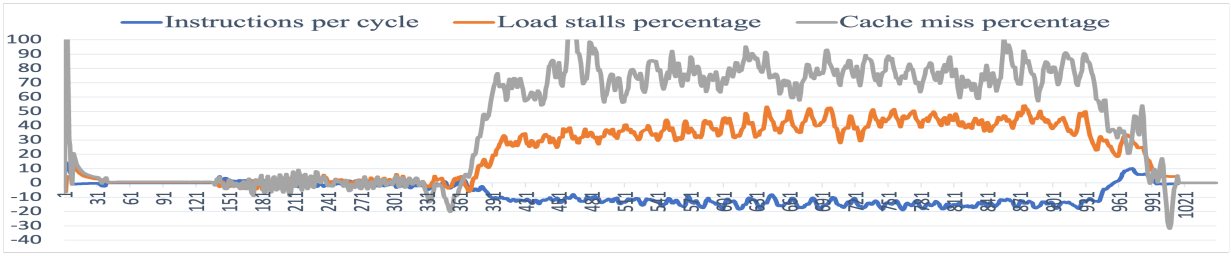


Figure 2.2: A time series graph that shows the percentage difference in memory bandwidth pressure measurements for XGC on compute node 0. The X-axis shows the elapsed time in seconds since the experiment start. The Y-axis shows the percentage difference as $\frac{M_{HMB}(t)-M_{LMB}(t)}{M_{LMB}(t)} * 100$. Here, $M_{LMB}(t)$ and $M_{HMB}(t)$ represents measurement M observed at time t when memory bandwidth usage was low (LowMem) vs when memory bandwidth usage was high (HighMem), respectively.The measurement values were smoothed using a rolling average of 30 values before computing the percentage difference.

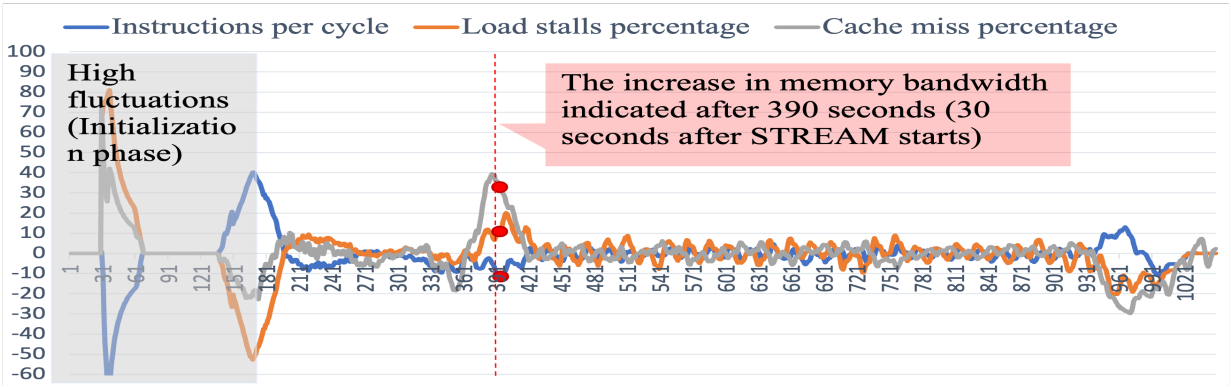


Figure 2.3: A time series graph that shows the $change\_percentage$ computations for the three memory bandwidth pressure measurements. The measurement data is shown for XGC on compute node 0 when STREAM was started after 360 seconds. The X-axis shows the elapsed time in seconds since experiment start. The Y-axis shows the percentage change.

delay in detection and the number of times the event is notified. This particular function and tuning process is representative of what I would expect a performance engineer to do to provide a decision module for a particular application, and the specific parameter values are not expected to be universal.

Figure 2.3 shows the change_percentage metric values when the memory bandwidth usage was high, i.e., arrangement HMB. The dynamic actions were disabled for this run. All three measurements show at least a 10% change during high memory bandwidth usage. However, cache

miss percentage should also have greater variability (more than 20%) to detect high memory bandwidth usage. The thresholds (δ) are chosen in the experiments based on these observations. As the graph shows, the metric correctly indicates the increase and decrease in usage as positive or negative values depending on the measurement at various times during execution. Values close to 0 indicate no change. XGC spends its initial three minutes (about 180 seconds) in a setup phase. This time shows a sharp increase followed by a decrease in memory bandwidth usage. These initial events for the tasks are ignored by the Arbitration stage of DYFLOW to allow the workflow state to stabilize after startup. After this, an increase in memory bandwidth usage is observed at about 392 seconds, approximately 32 seconds after the STREAM benchmark starts. The opposite change indicating a decrease in memory bandwidth usage is observed around 960 seconds, approximately 60 seconds after STREAM completes.

NetContScenario: Similar to the MemContScenario, the end-to-end transfer time data also requires smoothing, as this data is also susceptible to jitter (e.g., a suddenly slow remote transfer). To avoid bottlenecks as a consequence of a congested connection, a scientist may prefer to suspend the transfer. With the raw observation, the policies may react to one slow transfer and turn off the transfer, where future transfers could have been performed fast enough. After analyzing standard smoothing techniques, the most effective technique for our experiments was achieved by computing a rolling average where a higher weight is given to the new observations. For this technique, the smoothed values represent the most recent transfer times while minimizing the effects of anomalies.

To calculate weights, I assign a score to the values out of M, such that the latest value gets a score of M, the next value gets a score $M-1$, and so on. The weights are calculated by dividing the score by M. In our experiments, a maximum window size of $M = 5$ was reasonable to get

the desired estimate of the transfer times.

Requirement 2. *The event expression can be complex and may depend on various parameters that need to be determined by the appropriate expert (e.g., a performance engineer or data analyst) based on the QoS requirements of the workflow and underlying platform. For example, the number of metric values to use for processing the metric values, or the thresholds to identify the events of interest, may vary depending on the size of the input instance or the HPC system employed for running the workflow. Thus, flexible approaches are required that can ease the expression of events for the experts and support tunable parameters for setting or adjusting the settings as necessary.*

2.5 Determining relevant responses for the events

Once the conditions that define events of interest are known, a response needs to be determined. Response refers to the action to perform to change the workflow configuration when an event of interest occurs. The following are runtime control categories that can be used as responses to an event of interest.

Dataflow level: This category refers to actions that manage the data distribution between writer (publisher) and reader (subscriber) tasks. For example, when a reader task is relatively slow, it may be useful to change the frequency at which a writer publishes data.

Task level: This category refers to actions that a task developer can provide to control the running parameters of the task while it is running. For example, using a faster but lower-precision computational method over a higher precision, but slower computation method may be a

good strategy when performance is lower than expected or desired for a specific task (e.g., a physical simulation).

System level: This category refers to actions that change the resource assignment of running workflow tasks at runtime. As an example, the number of CPUs or GPUs assigned to a task can be changed. Depending on changes in resource usage patterns, a task can be assigned more resources or required to shed resources to give the resources to other tasks.

Workflow level: This category refers to actions that change which tasks will run at runtime. For instance, scientists may want to start an analysis task only when interesting features emerge in the data produced by a simulation task (i.e. as determined by another analysis task).

There could be another action category that can deal with ensemble workflows (described in Section 7.3). This category requires further exploration and is beyond the scope of this dissertation.

MemContScenario: The potential responses to deal with high memory bandwidth usage include reducing memory bandwidth congestion by reassigning a task to other resources (i.e. a different cluster node) or enabling the simulation to publish fewer timesteps for analysis (decimation in time). The latter response is not feasible for this workflow, as the accuracy of the positions and trajectories of particles computed by the $PAnlz$ task can be affected by missing intermediate timesteps.

NetContScenario: Multiple options may be applicable as a response to an event. Based on the QoS requirements of the workflow, the best response could be a trivial choice or may depend on information only available at runtime. For instance, for network slowdown events, a scientist may want to choose between compressing data with different methods, publishing fewer

timesteps, or forgoing the transfer completely. For this example, some loss in accuracy may be acceptable for visualization. However, for a scientist who wants to minimize data loss, the best response may depend on the amount of time it takes to transfer the data for a timestep. For instance, the scientist may prefer to use lossless compression instead of other responses, hoping to get the desired transfer time when the connection is slow.

Requirement 3. *To respond to events, solutions are required that support various controls that can manipulate running workflow tasks by changing resource assignments, managing dataflow, changing task parameters, or managing the tasks to be executed at any time.*

2.6 Validating a response at runtime

Large-scale workflows that couple multiple tasks are vulnerable to various dynamic uncertainties related to performance-driven, science-driven, or system-driven events such as failures. For such workflows, a team of research scientists may manage different components of the experiments and determines the event and responses. For instance, physicists and data scientists will best understand the behavior of analysis and visualization tasks and be able to identify the events relevant for science-driven functionality. The task developers who program and design tasks can identify events that lead to performance bottlenecks at runtime. The system or performance engineer can identify the common triggers for all the experiments performed on the system, such as failure events.

While experts can describe different runtime events and responses to these events, that may be insufficient to ensure a consistent workflow state at runtime. For example, one or more events can occur near in time, and responses to events may involve incompatible actions, i.e., actions

can contradict when applied to the same task, actions can conflict over resources, or can result in an undesired state when performed simultaneously on different tasks. Therefore, there is a need for screening that accounts for such situations to formulate a plan validating that the set of actions applied to the running workflow tasks is always valid, compatible, and feasible with the available resources.

MemContScenario: The three metrics, Instructions Per Second, Load Stall Percentage, and Last Level Cache Miss Percentage, together indicate changes in memory bandwidth pressure. However, analysis of any one of these metrics in isolation may incorrectly signify memory bandwidth pressure changes. A scientist may want to set a runtime dependency of these metrics on one another to respond to changes in memory bandwidth pressure.

NetContScenario: As an example, I posit a scientist that has a set of preferences for data quality based on satisfying a key quality-of-service measure. For instance, when lossless compression does not provide the desired compression ratio, the scientist may want to achieve the desired transfer time by incrementally compromising on accuracy using lossy compression until a tolerable error limit is reached. Other actions, such as publishing fewer timesteps, may be taken when compression is not effective in meeting time constraints. To achieve this, the current choice of compression algorithm or algorithm parameters needs to be updated based on the outcome of the previous choice.

Requirement 4. *When multiple events occur near the same time, conflicts could emerge between the different event responses. While some of these conflict situations can be handled by pre-programmed conditions or a decision tree that mediates between events and responses, there could be conflicts that arise based on the runtime state of workflow tasks and the resources. For*

these situations, the pre-selected response may not be an appropriate or effective choice. Moreover, modifying a decision tree to integrate new events or update the event parameters could be susceptible to inconsistencies and undesired outcomes at runtime. Thus, a runtime service is desired that can dynamically validate and select the most suitable set of actions from the proposed responses. Such a dynamic service will also enable different team members to collaborate efficiently.

Chapter 3: Flexible orchestration service: DYFLOW

DYFLOW is a four-stage conceptual model that categorizes dynamic management as **Monitor**, **Decision**, **Arbitration**, and **Actuation** stages. Each stage independently manages one aspect of dynamic management and exposes features that enable users to express events that are important for the workflow and determine the actions to perform in response to those events. The stages exist simultaneously and function continuously on the input received from the previous stage. The Monitor is the first stage and does the bulk of the processing. It is responsible for gathering runtime data from the running workflow tasks, necessary to identify dynamic events. Since collected data is usually large and raw, the Monitor stage further simplifies and reduces the data into meaningful metric values that get transferred to the next stage. The second stage is the Decision. This stage processes the incoming values to identify if an event of interest has occurred, and then determine the actions needed in response to the event. The selected responses get passed on to the third stage, Arbitration. The Arbitration stage constructs a plan of action that is feasible and consistent with the workflow specifications on receiving the input from the Decision stage. The last stage is Actuation, which executes the plan of action sent by the Arbitration stage. To perform the requested actions, it sends signals to workflow tasks or invokes services of the underlying workflow management system. The stages provide users with features to express how the workflow tasks will be monitored, what defines interesting events, and what

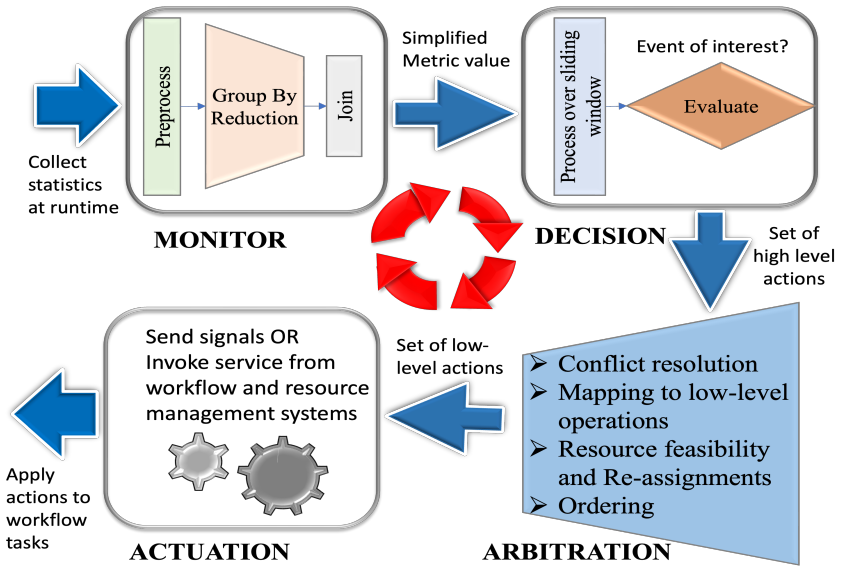


Figure 3.1: Diagram illustrating the DYFLOW model

are the preferences to orchestrate the workflow in response to the events. In chapter 5, I will describe an implementation of this model along with the user interface (section 5.3) that enables users to set these features. The detailed description of the DYFLOW stages is described next.

3.1 Monitor

The Monitor stage enables users to define monitoring requirements. Fulfilling Requirement 1 (as described in Chapter 2), the Monitor stage allows users to define sensors. Each sensor 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 raw data into metrics for identifying events of interest.

At runtime, this stage continuously gathers the user-desired statistics from a variety of sources. The data, if collected frequently (in time), can be huge, for instance, a value could be generated per process/thread assigned to any task. This stage further reduces the collected

data using pre-defined or custom operations and translates the data into meaningful metrics. The resultant metrics are accumulated across the distributed set of resources and delivered to the Decision stage. The Monitor stage also manages the background activities of the user-defined settings to oversee 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 the monitoring settings based on the changes performed on the workflow at runtime.

Sensors support abstract features that provide a set of commands for users to express wide-ranging needs that vary from simple metrics like the maximum memory consumed by a task to complex metrics computed from workflow measurements. These features can accommodate the needs of scientists to express various monitoring, such as those described in requirements discussed in Chapter 2. I describe these features next.

Source: Depending on the workflow, the required statistics could be organized in a specific format and available through a given medium. This feature enables users to determine how data of interest get generated and exchanged at runtime for a sensor. For instance, the desired data can be generated by an online profiler, generated by a running task, or generated by the operating system, and is available through a database service, a streaming service, or files. Based on this information, the model will collect the data in a distributed way. For example, for the MemContScenario (Section 2.1) and the NetContScenario(Section 2.2), the source will include streaming data using the TAU profiler and ADIOS2 support. The design of the Monitor stage is extensible to support a variety of collection methods.

Preprocessing: Preprocessing operations distill the collected data into a single value per process

before it can be processed into the desired metric. This feature can be useful when the input read from each process/thread is sizeable, for instance, a vector or multidimensional array. Examples of preprocessing operations include sum, average, standard deviation, variance, minimum, or maximum. Custom operations can also be provided as required.

Group-by and reduction: These operations dictate metric formulation. Group-by takes the data from the monitored tasks (i.e. a single value for every process of the monitored tasks) and organizes it based on the granularity, while a reduction operation summarizes the grouped information into a metric. 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 task that shares the compute node. The node-workflow granularity groups the information from all the processes belonging to the same workflow and sharing a compute node. 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. The design of the Monitor stage is extensible to support other granularity levels.

A granularity-based grouping enables expressing metrics from the collected data that can capture events in different scopes. The grouped values can be reduced and summarized into simplified values using operations such as sum, average, standard deviation, variance, minimum, maximum, or user-provided. For instance, in the MemContScenario, the data can be grouped at node-task level granularity (i.e., grouping the data collected from every process of a task assigned to a compute node). The grouped values can be reduced to a metric using the aggregation operation, such that there would be one metric value for each node assigned

to that task. Similarly, for the NetContScenario, the metric can be generated by grouping at task level granularity (i.e., grouping data from every process of a task) and reducing the grouped values by choosing the data from the first process of the task (i.e., MPI rank 0). The resulting metric would be a single value for the task of interest.

Join: A sensor can join its output with another sensor to compute a complex metric that relies on multiple data inputs. The join operation can be an operation such as sum, multiplication, division, or one that is user-provided. For instance, in the MemContScenario(section 2.1), 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 may have separate paths to the inputs from which the data is being read, the variable name, the variable type, and so on. Similarly, the hardware counter information used for defining metrics can differ across architectures.

3.2 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.

As per Requirement 2 and Requirement 3, the Decision stage in the model allows users to define policies that provide abstract features that simplify setting these guidelines and supporting a broad range of policies. The Decision stage 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. Examples of analysis functions include average, sum, minimum, maximum, mean, median, mode, standard deviation, kurtosis, skewness, exponential average, and variation. For instance, in the MemContScenario, a user could identify the events based on the running average rather than the last observed value of the IPC metric. Similarly, in the NetContScenario, a user could identify the events based on the exponentially weighted average rather than the last observed value of the time to send the timestep.

Evaluation condition: The evaluation condition compares the input against a threshold (i.e., a numeric value, a boolean value, or an interval, meaning a pair of values) and the result determines if an event of interest has occurred. The comparison condition could be less than, greater than, equal, not equal, greater than equal, less than equal, is in the interval, or is not in the interval. 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 should be concise and easy to understand, as they encapsulate different low-level operations required to perform the desired action. The high-level actions correspond to various dataflow, workflow, task, and system-level actions. For example, a SWITCH operation could represent 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 possible high-level actions include ADD-

CPU, 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 in time to decide when to trigger the evaluation condition. The evaluation condition is executed if new inputs were received since the last trigger. 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.

3.3 Arbitration

As per Requirement 4, the Arbitration stage works by collating the suggestions sent by different policies within a short window of time and then determining which of these suggestions, if any, will be applied to modify the current state of the workflow. Formulating a plan of action involves the following. The first responsibility of the Arbitrator stage is to eliminate invalid or futile suggestions. The suggestions that conflict with the past actions applied to the workflow tasks or data streams are considered invalid. Futile suggestions refer to the repeating application of identical actions on workflow tasks or data streams.

The second responsibility of the Arbitration stage is to determine the dependent actions for the suggested actions. To give an example of dependencies, when a task is stopped or restarted all the (tightly) coupled tasks dependent on it need to be signaled, stopped, or restarted.

The third responsibility of the Arbitration stage is to screen the high-level actions suggested by the Decision stage to resolve conflicts. Conflicts can be direct or indirect. A direct conflict results when high-level actions apply to the same task and lead to contradictory or inconsistent behavior. For example, one action may request to add more CPUs to a task, while other requests to stop the same task or release some of its CPUs. An indirect conflict results when the suggested actions do not conflict directly but are inconsistent with one another and lead to an undesirable workflow state at runtime when applied simultaneously. For instance, the user may set multiple policies in response to performance events. One policy may suggest signaling the simulation task to output fewer timesteps to memory buffers when memory usage is high, while another policy may suggest switching to a faster and lower precision analysis when the rate of progress is slower than expected. If both policies are triggered simultaneously, then the overall experimental output could lose significant accuracy. Resolution of conflicts results in selecting a set of high-level actions.

The fourth responsibility of the Arbitration stage is to map the high-level actions to low-level operations. These low-level operations represent the API calls to a resource manager or underlying workflow management service or operating system.

The fifth responsibility of the Arbitration stage is to determine the feasibility of the operations and create the final plan of action. The Arbitration stage is also in charge of resource management, as a feasible final plan is dependent on the available resources. This stage maintains information about the total allocated resources, resource health, and the current resource

assignment to workflow tasks. The stage 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 the requests 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 number of processes of a task releasing resources, it should precede others that will use those resources.

The Arbitration stage can function automatically to formulate a plan of action while handling conflict resolution. However, users have the flexibility to define rules to guide the plan of action suitable for their workflows by dictating the conflict resolution parameters. The rules allow users to set priorities, constraints, and dependencies.

Policy priorities: Users can set explicit priorities to deal with situations where different policy responses have high-level conflicts. In the absence of user priorities, the Arbitration stage automatically determines the policy priorities based on default criteria described in Chapter 4. To illustrate how explicit policy priorities are useful to meet specific requirements of the workflow. Suppose, two events; one related to performance, and the other related to data accuracy, occur at nearly the same time. One event suggests adding more resources to Task1, while another suggests switching Task1 with Task2. If data accuracy is an important requirement for the workflow, then users can make this requirement explicit using policy priorities.

Policy constraints: Users can set constraints to establish relationships between policies. The

relationships refer to incompatibilities where suggested actions do not explicitly conflict, or where there are interdependencies across different policies. The constraint P_X "depends-on" P_Y establishes a dependency of P_X on P_Y and instructs the Arbitration stage to accept policy P_X's suggestions only if policy P_Y provides suggestions at the same time. On the other hand, the constraint P_X "incompatible-with" P_Y instructs the Arbitration stage to reject suggestions of the policy with the least priority of the two policies if both provide suggestions at the same time.

Task priorities: Like policies, users can set explicit priorities of tasks based on their relevance. Task priorities are used by the Arbitration stage to decide which task gets preference over others when low-level operations compete for resources. In the absence of user priorities, the Arbitration stage automatically determines the task priorities based on default criteria described in Chapter 4. Explicit task priorities are useful, for instance, in large-scale experiments where users may prefer a task, such as a simulation, to run without interruption and never release resources.

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 task it depends on and gets data via an *in situ* medium) or loose (i.e., the dependent task runs uncoupled from the parent and gets data from files on disk). This information helps in identifying dependent operations. Task interdependencies are required when the underlying workflow management system does not compute a dependency graph.

Workflow setup time: This allows users to override the default time used by the Arbitration stage to disable accepting suggestions from the Decision stage to allow the workflow to settle

during the initial launch and after any dynamic changes are applied.

3.4 Actuation

The plan of action consists of low-level operations that invoke the services of an underlying workflow management service, send signals to running tasks, or interact directly with the cluster resource manager. The Actuation stage in our model serves as an abstract implementation for all the low-level operations invoked by the Arbitration stage in the final plan of action. Some examples of such low-level abstract operations include starting a task with a resource assignment, sending signals to a task, terminating a task, requesting or releasing resources via the cluster resource manager, inquiring about resources' health, or inquiring about the task runtime status.

3.5 Limitations of the DYFLOW model

There are several limitations of this model. First, the model assumes that a workflow consists of simulation, analysis, and visualization tasks. The model does not support ensemble experiments. In ensemble experiments, a set of workflow instances are executed in parallel. Each instance has variation in input parameters, the configuration of the workflow tasks, the boundary conditions, and so on. For such experiments, further exploration of the DYDLOW model is required to understand what features are required for metric formulation, what dynamic actions are required for runtime modification, and what methods are required for arbitration.

Second, the quality of service delivered by DYFLOW at runtime depends on the user settings. Hence, it is the user's responsibility to ensure that the expression of requirements is in accordance with user expectations. Users also need to decide on an initial resource assignment.

Third, the model is limited by the support provided by resource managers (i.e. cluster scheduleres) for on-demand resource acquisition and release. While cloud services allow a running task to expand and shrink the number of processors, amount of memory, etc., on-premise cluster machines do not generally enable such functionality. Besides the resource managers, elastic support from parallel communication libraries, such as MPI, is also not available in practice. Hence, the model by default relies on a checkpoint-restart mechanism to adapt the resource assignments based on the resource requirements of tasks at runtime.

Last, the Arbitration stage provides default prioritization criteria for ranking policies and tasks. These criteria rank the policies and tasks on a reasonable but limited set of parameters. Further criteria need to be explored that can be turned on and off by users based on their requirements. The Arbitration stage can also be trained to automatically enable and disable various criteria based on the types of workflows encountered, using prior experience to learn from earlier runs.

3.6 Meeting requirements of MemContScenario and NetContScenario

In this section, I give an overview of how DYFLOW model can be used to describe the orchestration for the two scenarios described in Chapter 2.

3.6.1 Expressing dynamic requirements of MemContScenario

Using the features of the DYFLOW model, the requirements of MemContScenario can be expressed as sensors, policies, and arbitration rules corresponding to observing and responding to changes in memory bandwidth that indicate high usage.

A sensor can be defined to collect each of the five measurements: the total last level cache references, the total last level cache misses, instructions completed, CPU cycles used, and cycles stalled waiting for load operations. The metric computations can be performed by grouping and summing the raw data from every process (of the monitored task) per compute node for each of these computations. Next, the three metric values can be obtained by using join sensor features with a percentage operation. For example, the sensor corresponding to the total last-level cache misses can be joined with the sensor corresponding to the total last-level cache references.

Three policies can be set to evaluate the events representing changes in cache miss percentage, instructions per cycle, and load stall percentage that indicates high memory usage. These policies can evaluate the metric values collected from the three of the five sensors (joined) that return the cache miss percentage, instructions per cycle, and load stall percentage, respectively. Each of these policies can be set to maintain a history of the last 30 measurement values for computing the change percentage operation (as a user-provided operation, as discussed in Chapter 2), at node-level granularity. The evaluation conditions and thresholds for indicating high usage event could be checking if percentage change is greater than the threshold value $Th1$ (i.e., $Th1 = 20$) for cache miss percentage, greater than the threshold value $Th2$ (i.e., $Th2 = 10$) for load stall percentage, and less than the threshold value $Th3$ (i.e., $Th3 = -10$) for instructions per cycle. When the events evaluate to be true, the possible response could be unpacking (moving) the lowest priority task from the compute nodes shared by the workflow tasks.

To be certain that the response is taken when all three events occur together, a dependency constraint can be set on each of these policies such that a policy suggestion will be acceptable only if the other two policies are also evaluated to be true at the same time. The policies are applied to XGC and $PAnalz$ tasks as they are sensitive to external interference. XGC tasks have the

highest preference, while $PAnalz$ tasks have the lowest preference of the three workflow tasks. $STREAM$, the proxy for a memory-bandwidth intensive background task, is assigned the lowest priority.

3.6.2 Expressing dynamic requirements of NetContScenario

Similar to the MemContScenario, I now describe how to configure sensors, policies, and arbitration rules to manage the flow of data from the DMZ task to the Viz task. A sensor can be set to track the transfer times for data streamed from the TAU profiler. compute the weighted average of

Four policies can be set to act on this sensor's output. Each of these policies can maintain a history of the last 5 inputs collected from a sensor that measures the transfer time (not shown) and computes the exponential average operation discussed in Chapter 2. The first policy, say, "LossLessComp", can suggest applying BLOSC lossless compression [10] to the data for each timestep when the evaluation condition, i.e., transfer time metric value is greater than the threshold representing $T1$ seconds. The next two policies, say, "LossyComp1" and "LossyComp2" could suggest applying the SZ [20] lossy compression scheme with different accuracy parameters to the data for each timestep when the transfer time metric value is greater than $T2$ seconds and $T3$, respectively. The last policy, say, "StopTransfer" can suggest stopping streaming and writing the timesteps to disk when the transfer time metric value is greater than $T4$ seconds. The actions of the three policies, "LossLessComp", "LossyComp1", and "LossyComp2" could result in direct conflicts which could be detected by the framework. Incompatible constraints can be set for these policies with the "StopTransfer" policy to express indirect conflict.

These threshold values can be set based on the quality of service desired, e.g., quality of results or runtime performance. For instance, two configurations can be defined. The first configuration, say configA, is set from the perspective of a performance analyst who prefers performance to the accuracy of results. The second configuration, say configB, is set from the perspective of a data analyst who prefers accuracy to throughput. In configA, the policy thresholds are set as $T1 = 60$, $T2 = 90$, $T3 = 120$ and $T4 = 250$. As these policies can be triggered simultaneously, priorities can be set explicitly to define the preferred order that is focused on performance. Hence, the priorities can be assigned from lowest to highest in the order, "LossLessComp", "LossComp1", "LossyComp2" and "StopTransfer". In configB, the policy thresholds are set as $T1 = T2 = T3 = T4 = 60$. In this configuration, the priorities are assigned from highest to lowest in the order, "LossLessComp", "LossComp1", "LossyComp2" and "StopTransfer", respectively, as preference is given to the quality of results.

3.7 Experimental demonstration

In this section, I demonstrate how the DYFLOW model can meet the requirements of the MemContScenario and the NetContScenario based on the setting discussed above. A subset of the user configuration file for these scenarios is provided in Appendix B. The implementation of DYFLOW used in this demonstration is described later in Chapter 5.

3.7.1 Showing that DYFLOW meets the dynamic requirements of MemContScenario

I compared three types of experiment arrangements; (1) none of the tasks share a compute node with other tasks and resources are over-provisioned (**arrangement one**), (2) tasks share the compute nodes and memory bandwidth usage is low (**arrangement two**), (3) tasks share the compute nodes and memory bandwidth becomes high after instances of the STREAM benchmark, the proxy for a memory-bandwidth intensive background task, start around 6 minutes into the experiment (**arrangement three**). Table 3.1 shows the experiment settings on a standard Linux cluster(Deepthought2). For arrangement three, I ran the STREAM benchmark with different settings to create memory bandwidth contention. The impact of running the STREAM benchmark was noticeable when 20 GB was allocated for every STREAM process. Beyond this limit, the physical memory usage of the nodes is reached, which causes tasks to fail.

In this experiment, the evaluation of the policies is performed every 2 second. The policies trigger a response when the evaluation is true for at least 50% of the nodes used. The policies were applied to $theXGC$ and $PAnalz$ tasks as they are sensitive to external interference. XGC tasks have the highest preference, while $PAnalz$ tasks have the lowest preference of the three workflow tasks. $STREAM$ is assigned the lowest priority.

Table 3.2 shows the end-to-end overhead incurred in the three arrangements. the over-provisioning arrangement is the most expensive, both in terms of the overall experiment runtime and resource usage. By co-locating reader tasks on the same nodes as writer tasks, the overall runtime was approximately 35% faster and uses 20% fewer resources when there was no resource (memory bandwidth) pressure. The overhead of actively monitoring the workflow tasks (at a 1

Table 3.1: Runtime settings of the fusion workflow on Deepthought2

Task	Setting	Over-provisioned (arrangement one)	Shared (arrangement two)	Shared (arrangement three)
XGC	Processes	96 (12 per compute node)	96 (12 per compute node)	96 (12 per compute node)
PSelect	Processes	16 (16 per compute node)	16 (2 per compute node)	16 (2 per compute node)
PAnal	Processes	16 (16 per compute node)	16 (2 per compute node)	16 (2 per compute node)
Stream	Processes	—	—	32 (4 per compute node)
—	Compute Nodes	10	8	8
—	Total Timesteps	80	80	80

Table 3.2: End-to-end overheads for MemContScenario

Arrangement	Description	Total runtime
Over-provisioned (No Stream)	No orchestration	41 mins
Shared resources (No Stream)	No orchestration	26.05 mins
	Using Orchestration	26.07 mins
Shared resources (20GB per Stream process)	Using Orchestration (Dynamic actions disabled)	28.48 mins
	Using Orchestration	26.63 mins
Shared resources (Beyond 20GB per Stream process)	Using Orchestration (Dynamic actions disabled)	Failure
	Using Orchestration	26.72 mins

second frequency) and coordinating data streams was negligible compared to the cost of the running tasks. When extensive memory bandwidth pressure was induced (arrangement three), the policies respond to this event at around 390 seconds and a suggestion is made to reassign a task to other resources. Since additional resources were not available for reassignment, the STREAM task (with the lowest priority) was killed. The response time, i.e., from the time the event was detected to when the action was applied, was about 12 microseconds. By facilitating dynamic readjustment of the resource assignments in response to undesirable events, the policy-driven orchestration allows scientists to reliably co-locate tasks and reap benefits in both decreased execution time and lower resource usage from data locality.

Table 3.3: Run setting of LAMMPS workflow

TASK	Proceses	Compute nodes	Timesteps
Lammps	100 (20 per compute node)	5	30
CNA	100 (20 per compute node)	5	30
DMZ	1 (login node)	0	30
Viz	1 (remote machine)	0	30

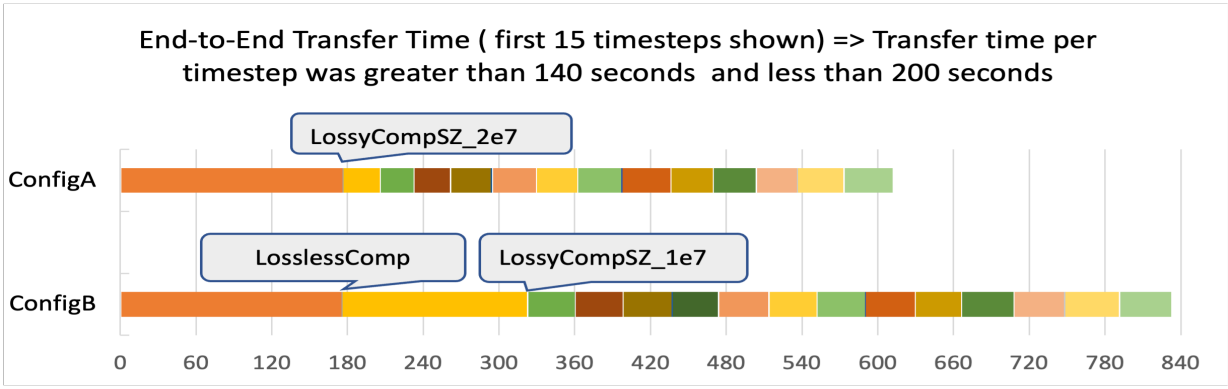


Figure 3.2: The Gantt chart compares the transfer times of the first fifteen timesteps observed by DMZ for two configurations when orchestration was enabled. For each configuration, the chart shows the different action(s) that were performed to achieve the user-desired transfer time. The X-axis shows the time elapsed in seconds since the experiment start.

Table 3.4: End-to-end overheads for molecular dynamics use case

Description	Total runtime
No orchestration	70 mins
Orchestration with ConfigA	18.38 mins
Orchestration with ConfigB	24 mins

3.7.2 Showing that DYFLOW meets the dynamic requirements of NetContScenario

Table 3.3 shows the experiment settings for the use case. Based on this setting, I set the maximum time to transfer a timestep to be 60 seconds. The experiment was run for a maximum of 30 minutes, where the tasks generate and transfer 30 timesteps. For simplicity, I show the results when all these dataflow controls operate on a memory buffer that holds one timestep on each end of the reader-writer communication.

Figure 3.2 shows the total time taken by the DMZ task, i.e., to read a new timestep, transfer the timestep (and receive the acknowledgment) to the Viz task, for the first few timesteps (shown as different colors) that were observed in the two configurations as a result of applying the policy actions. The initial transfer time per timestep was greater than 60 seconds.

For configA, "LossLessComp", "LossyComp1" and "LossyComp2" trigger. Based on user preference (a performance engineer that wants to minimize delay), the Arbitration stage selects the suggestion to compress the data using SZ compression, with accuracy $2e-7$, which results in a greater loss in accuracy but the fastest transfer time possible with the suggested responses. After this, the desired transfer time was obtained and no further actions were performed during this experiment.

For configB, all the policies are triggered at the same time. First, the policy "LossyComp" is accepted based on preference. After applying the action, the Arbitration stage stops accepting further suggestions for 2 minutes, allowing the workflow tasks to stabilize. When this period is over, the transfer time estimate was still slower than desired and all the policies respond to this event. This time, the Arbitration stage rejects the "LossLessComp" suggestion (identifying it as an ineffective response this time) and accepts the suggestion of the next preferred policy, "LossyComp1", which eventually obtains the desired transfer time. For this configuration, the policy suggestions are tried in decreasing order of the preference (as could be set by a data scientist to minimize the loss in accuracy) until the desired transfer time is achieved.

The average response time, i.e., between the time the event was generated, and the action was applied, was 10 microseconds. Table 3.4 summarizes the overall workflow execution times observed when no orchestration was performed and when orchestration was performed using the two policy configurations. The experiment shows that policy-driven orchestration can be easily

controlled to achieve the trade-offs needed to achieve the user's QoS goals.

Chapter 4: Arbitration Protocol

The Arbitration stage is the third stage of the DYFLOW model that determines how the suggestions from the Decision stage will be translated into a plan of action that would be executed in response to runtime events, i.e., the operations that will be invoked to change the resource assignment or the execution status of the workflow tasks. This chapter discusses in-depth the functionality of this stage.

An arbitration protocol governs how the plan of action is formulated at any instance from the collected suggestions. The detailed Arbitration protocol is described in Algorithm 1. The algorithm takes the following inputs; incoming suggestions indexed by the policy IDs (A_{sugg}) collected from the decision stage for determining a plan of action, the free (healthy) resources (R_{free}), the resources assigned to tasks(R_{asgn}), task priorities (T_{pri}), task dependencies (T_{dep}), policy priorities (P_{pri}), task dependencies(T_{dep}, policy constraints (P_{const}) and the tasks waiting to acquire resources ($T_{waiting}$).

The protocol begins with validation of the input suggestions. This step is performed to eliminate futile or invalid suggestions. Such suggestions could refer to reapplying an identical action to any task or data stream consecutively or applying an action that conflicts with the last action applied to any task or data stream. The validation is based on two assumptions. First, the policies that need to change the runtime state by repeating the identical actions from the past

that are ineffective, and stable. For instance, repeatedly requesting to compress the data with SZ algorithm with an accuracy $1e-7$ is futile. Second, when the suggested action conflicts with the latest state of any task or data stream, the last applied action should not be overridden by a lower priority policy. For instance, a lower priority policy requesting to increase CPUs should not be allowed when CPUs were removed recently by a higher priority policy.

Algorithm 1 Arbitration Protocol

Input:
$A_{sugg} \leftarrow$ Incoming suggestions indexed by policy IDs
$R_{free} \leftarrow$ list of healthy and unassigned resources
$R_{asgn} \leftarrow$ list of resources assigned to any task
$T_{pri}, P_{pri} \leftarrow$ Task and policy priorities set by user
$T_{dep} \leftarrow$ Set of all tasks that dependent on a task for input
$P_{const} \leftarrow$ list of constraints for any policy IDs
$T_{waiting} \leftarrow$ Priority queue of waiting tasks
Output:

function ARBITRATION( )
    $A_{valid}, P_{pri} \leftarrow$ validate_suggestions(A_{sugg}, P_{pri})
    $A_{total} \leftarrow$ dependent_actions(A_{valid}, T_{dep})
    $A_{filter} \leftarrow$ resolve_conflicts(A_{total}, A_{sugg}, P_{pri}, P_{const})
    $S_{op} \leftarrow$ low_level_operations(A_{total})
    $S_{op} \leftarrow$ resource_distribution(S_{op}, P_{pri}, T_{pri}, R_{free}, R_{asgn}, $T_{waiting}$)
    $op_{final} \leftarrow$ order operations in the set S_{op} and assign resources. Save the selected policies and their suggestions in history.
end function

After validation, the Arbitration module identifies the dependent actions with respect to the selected suggestions to determine a complete list of suggestions. Next, the total suggestions are filtered for conflicts. This step selects a subset of all the selected suggested actions from different policies so that none of the actions in the subset generate any conflict (direct or indirect) with one another. Conflict resolution is performed using policy priorities.

After conflict resolution, the high-level actions are translated into low-level operations. These operations are function calls in the Actuation module that are wrapper methods enclosing the underlying workflow management system methods or signals to send to running tasks.

Though the translated operations represent a tentative plan of action, the feasibility of this plan with respect to the available resources needs to be determined. The additional resources (e.g., CPU cores) required to execute the plan are estimated based on resources requested or

freed by mapped operations.

When the resources requested for the mapped operations cannot be fulfilled by available (free) resources from the allocated set, it is desirable to initiate a request to allocate more resources. However, resource manager support for on-demand resource allocation and deallocation is not commonplace on large clusters or supercomputers. Hence, when the available resources are insufficient to perform all the operations, the Arbitration module revises the resource assignments giving preference to high-priority operations to acquire the requested resources. This provides the higher priority tasks a chance to maintain their desired performance over the lower priority tasks in under-provisioned scenarios. To produce the executable plan, the selected operations are ordered, and the revised resource assignment is determined. Ordering is important to enforce applying a correct sequence of operations to the workflow tasks.

In the following subsections, I discuss each of these steps in more detail.

4.1 Policy and task priority computation

At various stages in the Arbitration protocol, the policy and task priorities are utilized to determine the preferences when the actions suggested by multiple policies conflict or when multiple tasks compete for resources. Users can set these preferences explicitly. However, in the absence of explicit policy priorities expressed by the user, the Arbitration stage will automatically assign priorities to policies and tasks.

4.1.1 Default policy prioritization criteria

If a policy is not marked (invalidated), the following guidelines are used to automatically assign priorities to policies.

Granularity: Every policy depends on the inputs from sensors. The Monitor module service provides these inputs at various granularity. Granularities are ordered from low to high as node-task, task-level, node-level, and workflow-level. A policy whose sensor output(s) are evaluated at a higher granularity level is preferred over ones with lower granularity levels. For instance, a policy based on workflow-level granularity is preferred over the task-level granularity of the sensor output(s). This is based on the assumption that higher granularity-based metrics are usually required for decisions that involve inputs from multiple workflow tasks.

Estimated cost of responses (cost): The measure we use to estimate the cost of the suggestion is the number of tasks that actions apply to and the number of dependent actions that are needed to apply these suggestions. This measure reflects how many of the running tasks need to be acted on for this suggestion. Higher preference is given to a policy that will affect fewer tasks. This criterion is used when priorities cannot be resolved by granularity. Also, the cost estimate changes depending on the number of tasks running at any time.

Number of resources requested (resources): This criterion is used when the above two criteria cannot resolve conflicts. A policy suggestion may request resources such as additional CPUs. A policy that requires fewer resources or sheds resources is given preference over other policies.

Number of sensors used for evaluation (sensors): This criterion prefers a policy that operates on numerous sensor outputs, under the assumption that a policy that relies on multiple sensors is more reliable than one that relies on fewer sensors.

Pick according to the default order: If none of the above criteria resolves the conflicts amongst policies, the order in the user specification file is used to determine priorities from high to low (i.e., the first policy listed gets the highest priority, while the last listed policy gets the lowest priority).

4.1.2 Default task scoring criteria

The following guidelines are used to automatically assign priorities (or scores) to tasks.

Number of dependent tasks: A task has higher priority over another if it has a larger number of dependent tasks (direct descendants) running at a given time, as any operation on this task affects more running tasks. For instance, a simulation task will likely be given higher priority over other tasks when competing for resources and would be selected last to shed resources.

Number of ancestor tasks: When tasks have the same number of dependent tasks, then priority is determined based on the number of ancestor tasks running at that time. Higher priority is given to a task that has fewer ancestors. This criterion assumes that tasks near the beginning of the workflow pipeline (closer to the source data) do most of the processing and consume a larger volume of data than the ones near the end of the workflow, and so they are more important.

Pick according to the default order: When the above two criteria cannot be applied, the tasks are

assigned priority (high to low) based on the order in the specification file.

4.2 Validating Suggestions

To determine which suggestions are valid concerning the past, the Arbitration stage maintains a short history of all the changes that were applied to any task and associated data streams. The changes include the runtime status and all actions and action parameters along with the policies that suggested those actions. The implementation discussed in Chapter 5 maintains a history of the past five actions. This was sufficient for the experiments performed in this dissertation.

Algorithm 2 describes the $validate\_suggestions$ method that takes as input the suggestions and policy priorities.

Initially, all the suggestions are accepted as valid. Then, for each policy whose suggestions were received by the Arbitration protocol, the suggested actions are evaluated sequentially in the order received. For any suggested action, the task and data stream (if any) it applies to and the action parameters defined are compared with the last applied action history for the same task or data stream (i.e., the current state). If the identical action is repeated for a task (or data stream), all the suggestions of the policy under consideration are removed from the output and the policy is marked as invalid for the future and the priority level set to the lowest. Reapplying the identical action incurs a cost without any benefit. Changing the priority of the policy with the repeated suggestion to the lowest is performed to avoid the rejection of other policies with conflicting suggestions to override or change the state of the same task (or data stream) going forward. There can be exceptions where repeated actions are necessary. For instance, resources may be added or removed multiple times to achieve the desired performance or other QoS metrics. For

such actions, the validation is skipped in this step by the Arbitration stage. Users can also enable or disable the validation for certain actions in the specification file.

If the action under consideration for a task or data stream is inconsistent with the last applied action to that task/data stream, then the action is only accepted if the priority of the current policy is higher than or equal to the previously performed action. Example of inconsistent actions include *STOP-START*, *STOP-RESTART*, *RMCPU- ADDCPU*. To resolve conflicts, the priorities of the current policy and the policy for which the last action was applied are compared. If the current policy has a lower priority, then the suggestion is rejected. To give an example, assume there are two policies with conflicting actions; PolicyA (with high preference) suggested starting task1 with new parameters, while PolicyB suggests stopping task1 in response to events based on the quality of results and performance, respectively. Due to preference, the PolicyA suggestion was accepted in the past. PolicyB should not be allowed to override this change in the future, unless the priority of PolicyA gets lowered by the Arbitration stage (during validation for repeated suggestions in the previous step). This validation is performed after the elimination of futile suggestions to make a decision based on the updated priorities. The valid suggestions and updated priorities are output for further steps in the Arbitration protocol.

4.3 Identifying dependent actions

In the case of coupled workflows, tasks depend on one another for input. A parent task is the producer of data, while the dependent tasks are the consumers of data. In *in situ* workflows, the workflow tasks run in parallel and the data may be streamed employing direct (e.g. ADIOS2 [28]) or indirect (e.g. Dataspaces [21]) runtime connections between tasks. If a running parent task

Algorithm 2 Validate suggestions

Input:
$A_{sugg} \leftarrow$ list of incoming suggestions indexed by policy IDs
$P_{pri} \leftarrow$ Policy priorities set by user or determined by Arbitration stage
Output:
$A_{valid} \leftarrow$ list of validated suggestions indexed by policy IDs
$P_{pri} \leftarrow$ Updated policy priorities

function $validate\_suggestions(A_{sugg}, P_{pri})$
  Add all actions as valid, $A_{valid} \leftarrow A_{sugg}$
  for Policy p in keys of A_{valid} **do**
    if p is marked invalid **then**
      Remove suggestion by policy p in A_{valid}
    else if
      then$A_p \leftarrow$ set of all actions suggested by policy p in A_{valid}
      for tuple $(action, task, stream, params)$ in A_p **do**
        if $(action, params)$ are identical to the last applied action and parameters for task and/or stream (with exception to actions disabled for this step) **then**
          Set policy p to the lowest priority in P_{pri} and mark invalid
          Remove all actions suggested by policy p in A_{valid}
        end if
      end for
    end if
  end for
  for Policy p in keys of A_{valid} **do**
    $A_p \leftarrow$ set of all actions suggested by policy p in A_{valid}
    for tuple $(action, task, stream, params)$ in A_p **do**
      if $(action, params)$ directly conflict with last applied action and parameters for task and/or stream such that the priority of p is greater than priority of the policy associated with last action **then**
        Remove all actions suggested by policy p in A_{valid}
      end if
    end for
  end for
  return A_{valid}, P_{pri}
end function

stops or restarts with new parameters or a revised resource assignment, actions need to be taken on dependent task(s) to avoid the dependent task failing or wasting resources waiting for data. When a parent task is stopped prematurely (i.e., before completing all the timesteps it was configured for), the dependent task may also need to terminate.

When data is directly streamed through the tasks, and a parent task restarts (i.e., stops and restarts), the connections between tasks in the workflow should be closed and re-established. Otherwise, the dependent task may terminate assuming that no further input is available on the streaming connection. To avoid these situations, DYFLOW will also stop or restart the dependent tasks accordingly.

Similarly, when a dependent task is started or restarted, the parent task needs to be notified

to avoid overwriting buffers while the dependent task restarts. DYFLOW offers data flow controls that can be utilized to manage the data streams between the parent and dependent tasks. For this scenario, DYFLOW will force the parent task to keep the buffers while the dependent task starts/restarts. In the DYFLOW implementation (discussed in Chapter 5), this is enabled through control messages sent to the stream actuator service that intercepts the streaming service calls and controls their functionality according to the message instructions.

Algorithm 3 describes the $dependent\_actions$ method which takes the map of validated policies and suggestions, and the policy priorities. Initially, all the suggestions are added to the output suggestions. For each policy suggestion, if an action stops or restarts a task, say Task1, then for all the output dependencies of Task1 (i.e. all the tasks that directly consume the output of the task1 or are dependent on the children of Task1), the same action is added to the output suggestions. Similarly, when a task is started or restarted, all the parent tasks for Task1 (i.e., the task whose output is consumed by Task1) a data control action to hold buffers in the parent task is added to the output suggestions. The newly added actions are associated with the same policy ID that was responsible for this addition.

4.4 Resolving high-level conflicts

High-level conflicts are resolved based on policy constraints and priorities. Algorithm 4 describes the $resolve\_conflict$ method that takes as input the validated suggestions (including dependent actions), the original list of suggestions received, policy priorities and policy constraints.

All the valid suggestions are initially added to the resultant filtered suggestions. First,

Algorithm 3 Dependent actions

$A_{valid} \leftarrow$ list of validated suggestions indexed by policy IDs
$P_{pri} \leftarrow$ Updated policy priorities
$T_{dep} \leftarrow$ Task dependencies
Output:
$A_{total} \leftarrow$ Updated list of validated suggestions indexed by policy IDs.

function $dependent\_actions(A_{valid}, P_{pri}, action\_set)$
    $A_{total} \leftarrow A_{filter}$
    for Policy p in keys of A_{filter} **do**
        $A_p \leftarrow$ set of all suggestions of policy p in A_{valid}
        for tuple $(action1, task1, stream, params)$ in A_p **do**
            for each (tightly-coupled) dependency d of task1 in T_{dep} **do**
                if a **then**ction1 restarts task1
                    add action, restart task d, to A_{total} and associate this action with policy p
                end if
                if a **then**ction1 stops task1
                    add action, stop task d, to A_{total} and associate this action with policy p
                end if
            end for
            if a **then**ction1 starts or restarts task1
                for each (tightly-coupled) parent d of task1 in T_{dep} **do**
                    add data control action to hold buffers of input from d to A_{total} and associate this action with policy p.
                    Update A_{total} with added actions
                end for
            end if
        end for
    end for
    return A_{total}
end function

the indirect conflicts are resolved based on the policy constraints established by the user. The indirect conflicts refer to incompatibilities and dependencies constraints amongst policies that were established by the user. For each policy in the valid suggestions, the policy constraints are verified, and all the suggested actions of the policy are rejected when any of the constraints are not met. A dependency constraint is satisfied if all the policies that a given policy depends on have made suggestions at the same time, i.e, it belongs to the original suggestions. Contrarily, an incompatibility constraint is satisfied if none of the policies that a given policy is incompatible with are present in the original suggestions. When the dependency constraint is violated, the suggested actions of the policy under consideration are removed from the filtered suggestions. However, when an incompatibility constraint is violated, the suggested actions of the policy with lower preference are removed from the filtered suggestions.

The filtered suggestions in the previous step are then evaluated for direct conflicts. Direct

conflicts refer to those actions that, when applied to a particular task or data stream, have contradictory effects. Examples include *STOP-START*, *STOP- RESTART*, or *RMCPU– ADDCPU*. Conflict resolution is performed based on policy priorities. First, the suggestions (policies) are sorted based on policy priorities, such that the higher priority policies are given preference. For each policy suggestion, the actions are registered in a temporary $action\_set$ that records the set of actions suggested for each task and/or data stream along with the policy ID that suggested the action. If a given action is identical in $action\_set$, then the action is removed from the suggestions of the current policy (as the existing action in the $action\_set$ was suggested by a higher priority policy). If a given action conflicts with any action present in the action_set for a particular task and/or data stream, then the actions which have higher policy priority are accepted, and the conflicting policy's suggestions (with lower precedence) are discarded from the action_set and the filtered suggestions. The filtered suggestions are output for further steps in the Arbitration protocol.

4.5 Translating to low level actions

The filtered suggestions represent the high-level actions that are an abstraction of a set of low-level operations that represent the services used to invoke function calls to a resource manager or underlying workflow management system. As an example, to restart a task, the task needs to be signaled, stopped, and start again with new parameters or a changed resource assignment. Similarly, MPI-based tasks that depend on inter-and intra- task communication cannot grow and shrink without a restart. So, an $ADDCPU$ TaskA action is translated as signal TaskA, stop TaskA, and start TaskA with a new resource assignment.

Algorithm 4 Conflict resolution

Input:
$A_{total} \leftarrow$ list of validated suggestions (including dependent actions) indexed by policy IDs
$A_{sugg} \leftarrow$ list of original (incoming) suggestions received by Arbitration stage indexed by policy IDs
$P_{pri} \leftarrow$ Updated policy priorities
$P_{const} \leftarrow$ Policy constraints
Output:
$A_{filter} \leftarrow$ list of filtered suggestions indexed by policy IDs

function $resolve\_conflict(A_{total}, A_{sugg} P_{pri}, P_{const})$
    $A_{filter} \leftarrow A_{valid}$
    for Policy p in keys of A_{total} **do**
        $Incom_p \leftarrow$ Set of IDs of policies incompatible with p in P_{const}
        $Dep_p \leftarrow$ Set of policy IDs of all the policies, the policy p depends on in P_{const}
        for $D_p \in Dep_p$ **do**
            if D_p not in keys of A_{sugg} **then**
                Remove suggestion by policy p in A_{filter}
            end if
        end for
        for $I_p \in Incom_p$ **do**
            if I_p in keys of A_{sugg} **then**
                if priority of I_p is less than priority of p **then**
                    Remove suggestion by policy p in A_{filter}
                else
                    Remove suggestion by policy I_p in A_{filter}
                end if
            end if
        end for
    end for
    action_set $\leftarrow \phi$
    $Sorted_p \leftarrow$ sorted set of keys of A_{filter} based on priorities (high to low).
    for Policy p in keys of $Sorted_p$ **do**
        $A_p \leftarrow$ set of all suggested action of policy p in A_{total}
        for tuple $(action1, task1, stream1, params1)$ in A_p **do**
            if (action1, params1) **then** is identical to any action for task1 and stream1 in action_set
                Remove the action1 from A_p and update A_{filter}
            else if (action1, params1) doesn't conflict with any action for task1 and stream1 in action_set **then**
                Add action1 (with params1) and policy p on task1 and stream1 in action_set
            else if priority of p is greater than the policy of the conflicting action **then**
                remove all the suggested action of conflicting policy from A_{total} and A_{filter}
                Add action1 (with params1) and policy p on task1 and stream1 in action_set
            else
                Remove suggestions of p from A_{filter}
            end if
        end for
    end for
    return A_{filter}
end function

Algorithm 5 describes the $low\_level\_operations$ method that takes as input the filtered suggestions and outputs low-level operations. For each policy suggestion, the low-level services required are added to the output.

Algorithm 5 Low level operations

Input:
$A_{filter} \leftarrow$ list of filtered suggestions indexed by policy IDs.
Output:
$S_{op} \leftarrow$ set of low level actions.

$S_{op} \leftarrow \phi$
function $low\_level\_operations(A_{filter})$
  for Policy p in keys of A_{filter} **do**
    $A_p \leftarrow$ Set of all suggestions of policy p from A_{filter}
    for $(action1, task1, stream1, params1)$ in A_p **do**
      add the all the low-level operations corresponding to $action1$ in S_{op}. Each added operation associates with policy ID, p
    end for
  end for
  return S_{op}
end function

4.6 Determining resource distribution

After the low-level operations are determined, the operations must be accepted based on the available resources. If too few resources are available than required for conducting the low-level operations, then extra resources must be allocated. Since support for on-demand resource allocation is not supported on most HPC machines, the resources need to be redistributed amongst the workflow tasks such that operations demanding resources for higher priority tasks are given preference. This step may result in stopping the lowest priority tasks and putting them in the wait queue.

Algorithm 5 describes the $resource\_distribution$ method that takes the low-level operations, the queue of waiting tasks, and outputs low-level operations acceptable based on resources available. For any operation, the resources required are calculated. If an operation frees resources, the resources are added tentatively to a list of free resources. Similarly, if an operation requires additional resources, the request is added to the required resources. The calculated resources required are then compared to the tentative list of free resources. If the free resources available are less than required, then a request to allocate more resources is initiated (which is

not possible for most HPC systems). If the request cannot be satisfied, then the current resource assignment will be readjusted by forcing the lower priority tasks to shed resources.

First, the operations are sorted based on policy priorities. For any operation applied to any task, say taskA, that requests additional resources, one or more running tasks with the lowest priorities are selected as victims that can relinquish the resources they currently hold to enable the high-priority operation to be performed.

Victim selection and operation priorities are derived from the task priorities. A running task becomes a victim if its priority is lower than the desired operation and has more resources to shed than other candidate tasks. By always using the lower priority task with more resources as the victim, we avoid scenarios where the resources are repeatedly handed back and forth between tasks (higher priority and a task gets killed repeatedly) selected as victims. If victim task(s) are not available, the operation is not executed.

Whenever an operation is removed from the plan, all the other operations for the same high-level actions are also removed. The victim tasks are put on a waiting list to be assigned resources when available. This process repeats until the available resources can be reassigned to meet the requirements of the revised plan. On the other hand, if resources are freed by the plan, the waiting tasks are provided the opportunity to start with preference given to higher priority tasks.

The selected operations represent the final actions that will be performed at runtime (by the Actuation stage). These operations along with the list of free resources and the queue of waiting tasks are returned as the output.

Algorithm 6 Determining resources distribution

Input:
$S_{op} \leftarrow$ list of low-level operations indexed by policy IDs
$T_{pri} \leftarrow$ Task priorities
$P_{pri} \leftarrow$ Policy priorities
$R_{free} \leftarrow$ list of healthy and unassigned resources
$R_{asgn} \leftarrow$ list of resources assigned to any task
$T_{waiting} \leftarrow$ Priority queue of waiting tasks

Output:
$S_{op} \leftarrow$ Tentative action plan

function $resource\_distribution(S_{op}, T_{pri}, P_{pri}, R_{free}, R_{asgn}, T_{waiting})$
  $temp_{free} \leftarrow R_{free}$
  $N_{des} \leftarrow \phi$
  for each operation o in S_{op} **do**
    if o defines an operation that releases resource **then**
      Add the set of released resources (e.g., CPUs, GPUs) to $temp_{free}$
    else if o defines an operation that requires resources **then**
      add the number of resources required (e.g., CPUs, GPUs) to N_{des}
    end if
  end for
  if $N_{des} > Count(temp_{free})$ **then**
    allocate additional resources required and update R_{free}
  end if
  sort operations in S_{op} based on policy priorities
  while $N_{des} > Count(temp_{free})$ **do**
    $V \leftarrow$ find a victim task that can shed resources for the next operation O_i (in S_{op}) that request resources on priority basis (i.e., priority of V is less than the priority of task associated with O_i)
    if V exists **then**
      $R_{rel} leftarrow$ resources assigned to V from R_{asgn}
      $S_{op} \leftarrow$ add operation to stop V (and dependents) in S_{op}
      add V(and dependents) to $T_{waiting}$
      $temp_{free} \leftarrow temp_{free} \cup R_{rel}$
    else
      drop operations with same policy ID as O_i from S_{op}. Update N_{des} and $temp_{free}$.
    end if
  end while
  while $N_{des} < Count(temp_{free})$ and a task (with highest priority) from $T_{waiting}$ can be started **do**
    update $S_{op}, T_{waiting}$ and $temp_{free}$.
  end while
  Return $S_{op}, T_{waiting}$
end function

4.7 Ordering operations to generate a final plan of action

Before the final operations can be forwarded to the Actuation stage, the operations must be ordered, so they are applied in a valid sequence. For instance, tasks need to be signaled before they can be stopped. Ordering is done by first performing all the operations that signal the tasks, The operations are ordered as follows; first, the operations that control data flow between the tasks are performed, then the operations that signal a task are performed, followed by operations that stop the task. Last, the operations that start a task are added.

The operation that controls data flow are messages to standalone data stream manager processes that govern how data will flow between readers and writers. They are performed first to make sure the data-flow control is effective before a reader/writer task starts or restarts. Similarly, a task must be signaled before it is stopped, so the task has an opportunity to save its state and terminate gracefully. All the operations to stop the tasks are performed before the operations that start the tasks so that the released resources can be reassigned.

For all the selected policies in the final plan of action, the high-level actions are stored in history. This history is later utilized for validation.

4.8 Recording Arbitration choices for users

The Arbitration protocol maintains records about the input suggestions (and policies), the choices made at various steps, and the total time to formulate the plan. The Arbitration stage further records the time that was taken to apply the changes at the runtime. These statistics can be accessed by the user at the end of the experiment for evaluation.

Chapter 5: DYFLOW Implementation

DYFLOW's functionality is accessible as a Python library that implements the different dynamic management stages. Figure 5.1 shows an overview of the implementation architecture. The DYFLOW library is designed as an independent module so that it is not limited by the services of the underlying workflow management system. This allows scientists to use the services of DYFLOW without having to employ a specific workflow management system.

5.1 Reusing existing support

Underlying Management system DYFLOW utilizes a base workflow management system that communicates with the cluster resource manager and performs the actions on the workflow tasks. The current implementation utilizes the functionality of an existing workflow management service, Cheetah/Savanna. The Cheetah and Savanna tools support the investigation of various resource allocation trade-offs as part of broader co-design studies and have been developed under the Department of Energy CODAR (Co-designing of Online Data Analysis and Reduction) project [26]. 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 is a Python code that is easy to extend, and offers the necessary services required

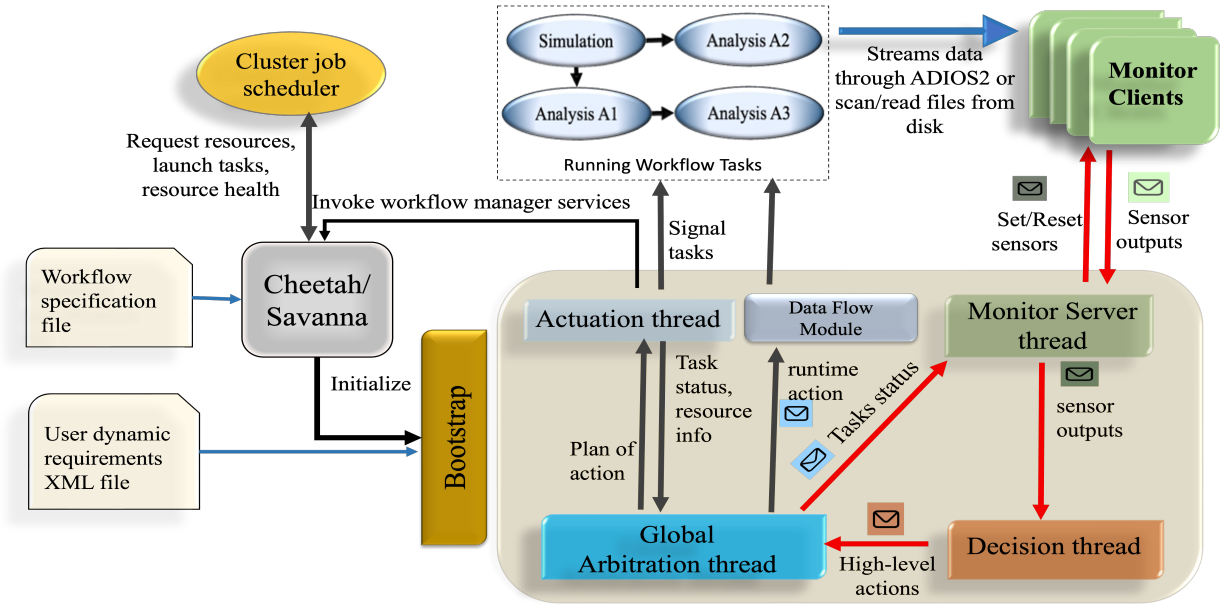


Figure 5.1: Overview of the DYFLOW implementation built on top of the Cheetah/Savanna workflow system. Arrows represent the exchange of data using JSON messages (in red), function calls (in black), or file/stream reads (in blue)

by DYFLOW. However, the DYFLOW implementation can be modified to utilize the services of other workflow management systems if desired.

Middleware service for in-situ workflows In *in situ* workflows, the workflow tasks run in parallel, communicate, and depend on one another for data. The data is often communicated between the reader and writer task through middleware services that support data sharing, utilizing on-node and off-node memory buffers (details are provided in Section 1.3). For *in situ* coupling of workflows, DYFLOW supports operations from the Adaptable Input Output System (ADIOS) middleware service[1]. ADIOS2 is a state-of-art unified I/O framework that encompasses a variety of transformations (e.g., compression methods) and data transport methods. It has been developed as part of the United States Department of Energy's Exascale Computing Program for efficiently managing scientific data. ADIOS2 uses a publish/subscribe methodology to make

[1]ADIOS2 website: `https://csmd.ornl.gov/software/adios2`

data exchanges and supports both disk-based I/O and *in situ* methods for application coupling. ADIOS2 supports in-memory coupling (asynchronous or synchronous) using various transport engines. For example, Sustainable Staging Transport (SST) enables dynamic connections between workflow tasks and provides high-performance data movement. SST allows readers and writers to have a different number of processes. The Strong Staging Coupler (SSC) engine is based on MPI methods to enable data movement. The data is read and written in lock step, and SSC requires both readers and writers to have the same number of processes. The Dataman engine is designed for wide-area network transfers, where a writer task sends data to a remote reader task. The implementation can be extended in the future to support other services.

Profilers DYFLOW utilizes existing support to gather monitoring data. To measure runtime performance, modern profilers, such as TAU [51], HPCToolkit [55], SCOREP [32] or DARSHAN [13], enable users to access runtime statistics that can be collected indirectly via system support or directly through code instrumentation. The system-generated information includes memory footprint, CPU utilization, and network bandwidth, while code instrumentation can measure time spent in various code sections. Our implementation employs TAU, an online profiler that collects performance data via code instrumentation and event-based sampling.

5.2 Implementation details

Users provide two inputs; the first input is a specification file for Cheetah/Savanna that describes the workflow and experiment setup with initial resource requirements; the second input is the DYFLOW specification file (in XML) that describes the orchestration requirements. Cheetah/Savanna incorporates the orchestration functionality by invoking the bootstrap module in the

DYFLOW library. An Actuation module in DFYLOW calls the services of Cheetah/Savanna whenever necessary.

Bootstrap The Bootstrap module parses an XML file with the user orchestration specification of the workflow and initiates threads corresponding to Monitor, Decision, and Arbitration modules, providing them with user-specified configuration information. For instance, the Monitor module receives the sensor information while the Decision module gets the policy details described in the specification file. All communication between the service threads occurs through shared queues and JSON [2] formatted messages. The Actuation module is a wrapper for the plugin inside Savanna that executes all the low-level services required by DYFLOW.

Monitor The Monitor 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 a launch node of a cluster. The server runs on the launch node within the DYFLOW library and connects to the client(s) using PyZMQ [3]. The server manages the client(s), and its activities include:

- start (or restart) client(s) with the sensors along with the tasks to monitor,
- update the client(s) whenever the runtime status of monitored tasks is changed,
- filter the out-of-order messages from the client(s), and
- send 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. The flexibility to launch multi-

[2] JavaScript Object Notation(JSON):https://www.json.org/json-en.html
[3] Python ZeroMQ website: https://zeromq.org/languages/python

ple clients on the compute or the launch nodes benefits the Monitor module to address requisite scaling needs. Running the server on the launch node maintains its availability in the event of computing resource failures.

The Monitor supports sensors that can stream user data through ADIOS2, stream data generated by the TAU [51] profiler using ADIOS2, scan disks for files, and read error status. Additional sensor types, such as querying a database or reading additional file formats (e.g., HDPF5, NetCDF), can be easily supported in the future.

Decision The Decision thread collects the incoming sensor messages from the queue, while discarding the out-of-order updates. It then routes the messages to appropriate policies. After this, the updates are post-processed for each policy and/or stored to maintain history as per the user specification. The evaluation of these updates is triggered according to its defined frequency interval. Policy responses (if any) are collected and sent as a single JSON message to the Arbitration module, and the Decision thread resumes processing the incoming messages. Each response from the Decision module includes a policy identifier and associated high-level action(s).

Arbitration When the Arbitration thread is nearly ready to process the suggestions from the Decision module, it starts accumulating incoming requests for a small interval of time. This interval attempts to accommodate the delay in communication of monitoring information or policy responses and enables the Arbitration module to gather responses for all the events that occurred over a given interval. An interval of one second was chosen to meet this requirement, which was derived based on experimentation and works for the scenarios used in this dissertation. After this interval is done, the Arbitration module formulates an action plan that will be executed at

runtime.

Once the plan is finalized, the Arbitration module waits for the Actuation module 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. In the experiments, a duration of two minutes is used to allow the workflow state to adjust to the new resource assignment. This duration was derived based on experimentation, and it works for the scenarios in this dissertation. The duration may depend on the workflow, middleware technology for task coupling, the types of actions used as responses, or the underlying system. DYFLOW allows users to override this setting as necessary in the specification. This temporary suspension of listening to incoming messages from the Decision module allows for the latest suggestions received by the Arbitration module to reflect the events that occurred on the revised runtime state of the workflow. The suspension time depends on the actions performed at runtime. For instance, when tasks undergo resource reassignment (i.e., by being gracefully stopped and restarted with new resource assignments), the state of the workflow is considered stable when restarted tasks reconnect to the other running tasks and complete at least one timestep. If the Arbitration module responds earlier than this settlement period, then the state may not be a useful representation of the last changes applied.

The Arbitration thread constantly collects updates on the runtime state of the workflow tasks and the resource health from the Actuation module. Any changes in the runtime state of the workflow tasks are informed to the Monitor server thread to set or reset sensor settings.

DataFlow Module For *in situ* workflows, controlling dataflow between workflow tasks is an important feature to deal with undesirable events. We will refer to the flow of data as send-

ing/receiving of snapshots. A snapshot is a view of the output state of a workflow task, which is regularly communicated to other tasks that depend on this input. For instance, in many physical simulations a snapshot is the simulation state which is output at every simulation time step.

Libraries such as ADIOS2 enable direct communication between reader and writer tasks. The library also supports in-memory buffers to temporarily store unconsumed timesteps. However, these libraries cannot adapt to changes at runtime. For instance, if a reader gets disconnected at runtime (e.g., restarted with more resources), the middleware libraries, such as ADIOS2, do not know when a reader task will reconnect, so the writing task can overwrite buffers. Another example would be employing work shedding to mitigate back pressure between the reader and writer tasks. to avoid throughput or performance loss. DYFLOW caters to these requirements by supporting an additional module focused on meeting the needs of *in situ* scientific workflows. This additional functionality enables runtime actions to indirectly manage a set of memory buffers of a given size on both ends of the reader/writer communication. For instance, the actions allow users to deal with back pressure development due to a slow reader task or a slow network connection, reliably start and stop tasks, deal with high/low resource usage events, or deal with failures.

The following runtime actions are available for users to control data flow:

- Enabling work shredding for slow readers: Round Robin distribution of snapshots to several instances of the reader tasks;
- Changing write frequency: Writing at a given frequency, every nth snapshot in the writer's buffers is sent to receivers, and others are discarded (not buffered);
- Changing read frequency: Reading at a given frequency, a given reader can consume every nth snapshot in its receiving buffer discarding others;

- Enabling reader restart: Preventing snapshots to be overwritten if buffers are full waiting for readers to consume the snapshot
- Checkpointing writer data: checkpointing every nth snapshot in the writer buffer to disk;
- Stopping streaming: forcing the writer to save all the snapshots in the buffer to disk and forgo streaming;
- Compressing writer data: compressing snapshots with a compression algorithm before buffering in the writer.

Figure 5.2 shows the design of the extensions that provide runtime knobs to control the data flow between tasks. A standalone StreamArbitrator process starts on the workflow launch nodes (e.g., cluster login nodes) for every data stream and informs the writers and readers on how to send and receive the data. On Linux systems, an environment variable LD_PRELOAD lets users load a specific library before any other shared library, for instance, the standard C library. Using the LD_PRELOAD functionality, we have developed a StreamActuator library stub that intercepts the streaming service call to request or send data from/to reader and writer tasks. The StreamActuator functionality is adapted for each of the ADIOS engines enabling *in situ* analysis (i.e., SST, SSC, and DataMan). For every method invocation, the StreamActuator (e.g., only the process with rank 0 for MPI programs) communicates with the StreamArbitrator through message passing enabled through the ZMQ network library [63]. A StreamArbitrator thread runs on the cluster launch node for every data stream.

The StreamArbitrator threads are started at the start of the experiment. StreamArbitrator receives the high-level actions to coordinate the data flow between readers and writers from the global Arbitration module (in the DYFLOW library). As an example, the StreamArbitrator may

request the writer to checkpoint every 10 timesteps, or compress the data stream. Internally, the StreamArbitrator maintains state information about active writers and readers, such as the program name, the process identifier, number of time steps completed, current time step number, flags, and parameters to apply the global Arbitrator instructions for the writer and reader. All the interactions between StreamActuator, StreamArbitrator, and the global Arbitration module are performed using ZMQ messages. The current implementation intercepts the ADIOS2 library calls.

StreamActuator requests instructions on how to perform the requested streaming operation. Intercepting method calls this way eliminates the need to modify the application software (only LD_PRELOAD, the Linux environment variable described previously, needs to be set). When an active StreamArbitrator is available, the StreamActuator broadcasts a response (an integer) to all the other processes (e.g., using MPI calls) and then applies the received instructions and invokes the original data call accordingly. Otherwise, all the processes proceed with the original streaming service call immediately.

For the set of evaluations in this dissertation, I implemented StreamActuator using MPI, but it is designed so that the transport has a well-defined interface. For workflows that are not MPI dependent, the StreamActuator could potentially support ZMQ [63], RabbitMQ [25], Kafka [33], or other similar publish/subscribe technologies.

Actuation The Actuation stage serves as an abstract implementation for all the low-level operations invoked by Arbitration in the final plan of action. This abstraction acts as a plugin that ports DYFLOW across cluster architectures and builds on the services supported by the underlying workflow management systems.

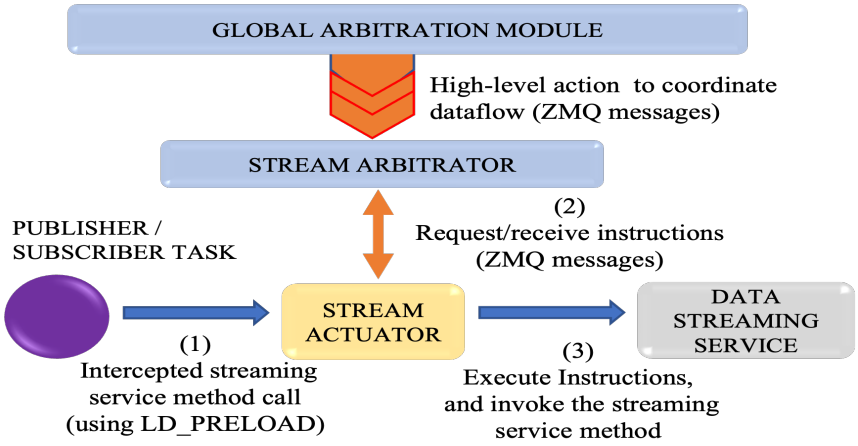


Figure 5.2: High-level design of Data flow module that arbitrates the flow of data between workflow tasks

5.3 User interface

The user interface contains three sections corresponding to the Monitor, the Decision, and the Arbitration stages. The user interface is exposed in XML so should have a small learning curve, since XML is straightforward to use and has a simple syntax. The detailed XML schema is provided in Appendix A.

Figure 5.3 shows an example XML demonstrating the settings for sensors, policies, and arbitration rules. The monitor section begins with a ”⟨ monitor⟩” tag that defines all the sensor functions and the workflow tasks to monitor using sensors. A ”⟨sensor⟩” tag defines a new sensor, where its ”id” attribute sets the sensor name, and the ”type” attribute sets the source of the sensor. A ”⟨group-by⟩” tag contains the set of metrics to compute for this sensor based on various grouping criteria and reduction operations. The ”⟨monitor-task⟩” tag applies a set of sensors (indicated by the ”⟨use-sensor⟩” tag) to a workflow task (set in the ”name” attribute). The ”var-source” tag sets the file or stream to use for monitoring. The ”distributed” attribute is set to true when the task is multi-process. The variable to be read is set in the ”var” attribute of the

Figure 5.3: An example XML specification illustrating the setting of sensors, policies, and rules for changing the frequency of output generated by the writer task when the physical memory usage of the reader task is high

```
<DYFLOW>
<monitor>
        <sensors>
            <sensor id="Sensor1" type="TAUADIOS2">
                <group-by> <group granularity="TASK" reduction-operation="MAX" /> </group-by>
            </sensor>
        </sensors>
        <monitor-tasks>
            <monitor-task name="Task2" workflowId="W1" var-source="tau-t2.bp" distributed="true">
            <apply-sensor sensor-id="Sensor1" var="RSS">
                <parameter key="var-type" value="long int" />
            </apply-sensor>
            </monitor-task>
        </monitor-tasks>
</monitor>

<decision>
    <policies>
        <policy id="HIGH-MEM" >
            <eval operation="GT" threshold="190" />
            <sensors-to-use>
                <use-sensor sensor-id="Sensor1" granularity="TASK" />
            </sensors-to-use>

            <actions> READ-FREQ </actions>
            <action-params action="READ-FREQ">
               <param key="N" value="5"/>
            </action-params>
            <frequency seconds="5"/>
        </policy>
    </policies>
    <apply-policies>
    <apply-on workflowId="W1" >
        <apply-policy policy-id="HIGH-MEM" eval-task="Task2">
            <act-on-task task-name="Task2" action="READ-FREQ" data-stream="tau-t1.bp"/>
        </apply-policy>
    </apply-on>
    </apply-policies>
</decision>

<arbitration>
    <rules>
        <rules-for workflowId="W1">
            <workflow-setup seconds="120" />
            <task-dependencies>
                <task-dep task-name="Task2" depends-on="Task1" type="INSITU" data-stream="tau-t1.
    bp" />
            </task-dependencies>
        </rules-for>
    </rules>
</arbitration>
</DYFLOW>
```

"⟨use-sensor⟩" tag and other details such as the type of the variable are set using the "⟨param⟩" tag. For example, in fig. 5.3, a sensor, "Sensor1", is set that will stream data from TAU using ADIOS2 middleware service. To compute the metric, the sensor will take the maximum over all the values collected from the processes of a task (i.e., TASK granularity). The sensor is applied to monitor task "Task2" at runtime and tracks the Resident Set Size (RSS) variable, which gives the physical memory usage of any process (in GBs).

The decision section (beginning with a "⟨decision⟩" tag) sets various policies and describes the workflow tasks to which the policies will be applied. Every "⟨policy⟩" tag defines a new policy, with its name set in the "id" attribute. The sub-tag "⟨eval⟩" sets the evaluation condition, and "⟨sensors-to-use⟩" is used to specify the sensor input(s) and the metric granularity of the sensor to use. The "⟨action⟩", "⟨history⟩" and "⟨frequency⟩" set the response (i.e. the action(s) to perform), the sliding window on the incoming values, and the frequency at which the evaluation will be performed. The "⟨apply-on⟩" contains the settings for applying policies to workflow tasks. The "⟨apply-policy⟩" tag is used to specify the policy (specified in the "policy-id" attribute) to be applied to a task (specified by the "eval-task" attribute). For any task under evaluation, "⟨act-on-task⟩" tag can be used to set the target task (or data stream) on which the response will be applied. Figure 5.3 shows an example policy that acts on the "Sensor1" data and every 5 second checks if the sensor output(RSS metric) is greater than 190 (or 190 GBs). If memory usage is high, then the action is applied to read every fifth timestep and discard others. This policy is set to evaluate "Task2" and the target for the response is set as "Task2" itself (and Task2's input "tau-t1.bp").

The arbitration section (beginning with an "⟨arbitration⟩" tag), along with the "⟨rules⟩" tag, is used to set the rules for the workflow that corresponds to setting priorities and constraints. The "⟨task-priorities⟩" and "⟨policy-priorities⟩" tags can be used to set explicit task and policy

priorities or unset the default prioritization criteria used by the Arbitration stage. The "⟨policy-constraints⟩" tag can be used to establish constraints between policies. The arbitration section, in Figure 5.3, shows the settings to express the *in situ* dependency of "Task2" and "Task1".

5.4 Limitations of the DYFLOW Implementation

The current DYFLOW implementation has several limitations. First, DYFLOW depends on the underlying workflow management systems for various services. For instance, to begin the experiment, DYFLOW requires services from the workflow management system to set up the experiment and launch tasks with an initial resource assignment. At runtime, DYFLOW requires services that can stop a task, start a task, send signals to a running task and provide task runtime status. The support provided by the underlying workflow management systems limits the capabilities achievable by DYFLOW. For instance, DYFLOW needs services that can take action on a remote workflow task. The current implementation integrates with a specific workflow management system, Cheetah/Savanna. To use the DYFLOW service, users either have to run their workflow with the Cheetah/Savanna system or invest in development efforts to extend DYFLOW with other workflow management systems.

Second, for tightly coupled workflows, DYFLOW depends on the ADIOS2 middleware service for *in situ* data communication. Thus, the DataFlow module only supports managing ADIOS2 library calls in the current implementation. The implementation is designed based on the assumption that the middleware streaming services cannot require support for adapting to runtime changes. For instance, ADIOS2 assumes that the connection between the reader and writer tasks will be establish when the tasks start, and that tasks will remain connected throughout

the experiment. Similar to the limitations of MPI, ADIOS2 does not support the ability to shrink and expand the connections between reader and writer processes to adapt the communication pattern with changes in resource requirements of the tasks.

Third, DYFLOW supports limited data collection methods. For instance, it supports sensors that can scan files on disk to track the output files generated, sensors that rely on the TAU online profiler to collect various runtime statistics, and sensors that can directly stream the results from a running task. To support other kinds of sensors, additional data collection methods need to be built in DYFLOW.

Last, the DYFLOW implementation assumes that the workflow tasks can handle signals so that they terminate gracefully by checkpointing the simulation state and restarting without generating failures. A user is required to provide this support.

Chapter 6: DYFLOW Evaluation

This chapter demonstrates examples from scientific workflows, highlighting some dynamic capabilities achievable through DYFLOW. In these examples, DYFLOW flexibly enables modification of the workflow state in response to science-driven events, reassignment of computation resources in response to performance-driven events or resource usage changes, and resilience from failure. The experiments show the capabilities of DYFLOW based on the implementation discussed in Chapter 5. 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.

6.1 Clusters

To show the costs incurred by DYFLOW, I conducted these experiments utilizing a standard Linux cluster and a state-of-the-art high-end supercomputer. The specifications of these clusters are described below.

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

6.2 Responding to science-driven events

When data is processed on-the-fly, there could be events triggered by the changes that occur in the data and underlying science. For instance, when interesting features emerge in data, a new analysis may be required or when error accumulation is high, an alternate algorithm may be initiated. Handling such events could be crucial for the quality and correctness of the information derived from the experimental results. This section demonstrates the utilization of DYFLOW to orchestrate science-driven events by employing a loosely coupled workflow with two tasks, XGC1 and XGCa, to simulate nuclear fusion reactions. XGC1 [34] and XGCa are gyro-kinetic particle-in-cell codes developed to study complex multiscale 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 sim-

ulation is alternating the simulation between XGC1 and XGCa such that XGCa pushes the simulation forward at a faster rate [31]. The scientists choose the alternation frequency to enable the experiment to move 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.

The scientists start the simulation with XGC1 and switch to XGCa to speed up the computation. To check the correctness of this setup, the scientist also wants to rely on the error analysis of XGCa so that XGC1 takes over the simulation whenever the error accumulation is too high. The properties of the fusion simulation output that could define an error estimation function are ongoing research by fusion scientists. Hence, for this simulation scenario, the simulation tasks run alternately, each for a fixed number of timesteps, until they jointly complete the desired total number of timesteps. We also employ a proxy error function to show the capabilities of DYFLOW that enable the switching of tasks at runtime.

Monitoring requirements: To accommodate these requirements, two sensors were defined in the DYFLOW specification, as shown in the sample XML in fig. 6.1. The first sensor, *NSTEPS*, is set to track progress, i.e., the number of global timesteps completed during the simulation to determine when to alternate between the two tasks. Both XGC1 and XGCa write an output file once a fixed number of global simulation timesteps are complete. The source type, *'DISKSCAN'*, corresponds to this collection method. It scans the disk to count how many output files are generated and returns the count multiplied by a fixed number (representing the frequency of output). The metric for this sensor computes the total 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'*. In absence of the appropriate definition of this ERROR sensor, I use the *NSTEPS* sensor output

Figure 6.1: XML example illustrating the sensor for switching on error and restarting the experiment for the desired number of timesteps

```
<monitor>
 <sensors>
    <sensor id="NSTEPS" type="DISKSCAN">
        <group-by>
            <group granularity="WORKFLOW" reduction-operation="SUM" />
        </group-by>
    </sensor>

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

 <monitor-tasks>
     <monitor-task name="XGC1" workflowId="Fusion1" var-source="xgc-particle*.bp" distributed="
    false">
        <apply-sensor sensor-id="NSTEPS" >
          <parameter key="frequency" value="3" />
        </apply-sensor>
     </monitor-task>
     ...
 </monitor-tasks>
```

Figure 6.2: XML example illustrating the user policies for switching on error and restarting the experiment for the desired number of timesteps

```
<decision>
  <policies>
    <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>

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

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

  <apply-policies>
    <apply-policy workflowId="Fusion1" policyId="RESTART_UNTIL_COND" eval-task="XGCA">
       <act-on-tasks action="RESTART" task="XGC1">
             <param key="restart-script" value="restart-xgc1.sh"/>
          </act-on-tasks>
    </apply-policy>

    <apply-policy workflowId="Fusion1" policyId="SWITCH_ON_COND" eval-task="XGCA">
       <act-on-task action="SWITCH" task="XGC1">
          <param key="start-script" value="restart-xgc1.sh"/>
       </act-on-task>
    </apply-policy>
    <apply-policy workflowId="Fusion1" policyId="STOP_ON_COND" eval-task="XGCA">
        <act-on-task action="STOP" task="XGCA">
       </act-on-task>
    </apply-policy>
    ...
   </apply-policies>
</decision>
```

Figure 6.3: XML example illustrating the user specification (arbitration preferences) for switching on error and restarting the experiment for the desired number of timesteps

```
<arbitration>
  <rules>
    <rule-for workflowId="Fusion1" >
      <workflow-setup seconds="120" />
      <task-priorities>
        <task-prio task="XGC1" priority="0" />
        <task-prio task="XGCA" priority="1" />
      </task-priorities>
      <policy-priorities>
        <policy-prio policy-id="STOP_ON_COND" priority="0" />
        <policy-prio policy-id="RESTART_UNTIL_COND" priority="1" />
        <policy-prio policy-id="SWTICH_ON_ERROR" priority="2" />
      </policy-priorities>
    </rule-for>
  </rules>
</arbitration>
```

at workflow granularity to generate a proxy error condition. The proxy error condition causes termination of XGCa after the 374th timestep of the simulation completes, as a proxy for error estimation to show the functionality of DYFLOW for this scenario.

Policy settings: The following dynamic events can occur at runtime; (1) start XGC1 or XGCa alternately if the desired number of timesteps are not completed, (2) stop the experiment whenever the desired number of timesteps are done to avoid computing more than the desired number of steps, (3) switch to XGC1 when error accumulation is high. To register these dynamic events, I set three policies (settings shown in Figure 6.2) as follows. *RESTART_UNTIL_COND* enables the start of XGC1 when XGCa stops running (and vice versa) when the output *NSTEPS* from the workflow is less than 500. *STOP_ON_COND* enables stopping XGC1 (or XGCa) if the output of *NSTEPS* is greater than 500. *SWITCH_ON_ERROR* stops XGCa and starts XGC1 when a high error accumulates in the simulation output generated by XGCa. I applied *RESTART_UNTIL_COND* and *STOP_ON_COND* to both XGC1 and XGCa. The policy,

Table 6.1: A single run configuration of XGC1 and XGCa

TASK(S)		SETTING	Summit	Deepthought2
XGC1	*XGCa*	PROCESSES	192 (14 per compute node)	192 (4 per compute node)
XGC1	*XGCa*	THREADS PER PROCESS	10	10
XGC1	*XGCa*	TIMESTEPS PER RUN	100	100
XGC1	*XGCa*	PARTICLES PER PROCESS	250K	250K
XGC1	*XGCa*	TOTAL COMPUTE NODES	13	48

SWITCH_ON_ERROR, was only applied to XGCa. Before starting XGC1, a user script, restart-xgc.sh, runs to set XGC1 inputs to restart from the last saved output of XGCa (XGCa doesn't require a restart file as it always starts based on XGC1 output).

Note: Another way to restart the task, alternatively, is to set a sensor to track the error status of the tasks. Based on this sensor, a policy can be set to start another task when the error status is "0" for a running task, indicating that the running task terminated normally. This sensor was supported in the later version of the implementation than used in this experiment.

Arbitration settings: Figure 6.3 show the arbitration preferences. Both tasks have priority 0 (the highest priority) since they run alternately. Keeping the task priorities equal further prevents the Arbitration stage from starting a task using resources of the running task on "RESTART_UNTIL_COND"'s suggestion. I prioritize the policies so that *STOP_ON_COND* has the highest priority as it signifies experiment completion to avoid situations where conflicts arise between restarting a task or stopping a task. *SWITCH_ON_COND* is assigned higher priority over *RESTART_UNTIL_COND* to resolve conflicts in determining which task to launch.

Table 6.1 shows the initial setup for the experiments on both clusters. The setup represents a single-run configuration of the workflow tasks. DYFLOW will rerun the tasks until 500 overall simulation timesteps are completed.

Summit: Figure 6.4 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

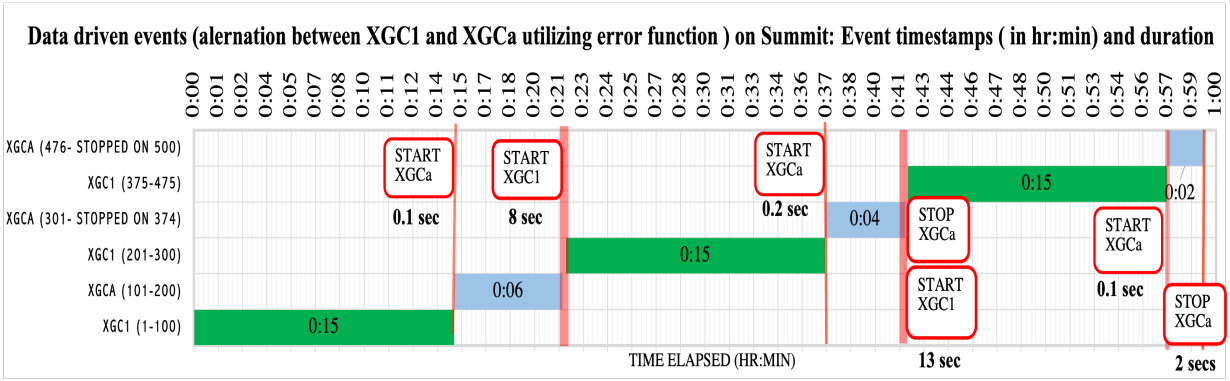


Figure 6.4: Gantt-chart showing the experiment performed on Summit for the XGC1-XGCa workflow to demonstrate running iterative experiments and terminating tasks based on runtime events.

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 from the occurrence of the event to finalizing the plan and executing it for these events is $\approx$ 0.1-0.2 seconds. Similarly, XGC1 starts when XGCa finishes (when 200 global simulation steps are complete). The response time, in this case, is 8 seconds - 4 seconds of this time is due to the delay enforced by the 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 restart 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 (as shown in Figure 6.5), the

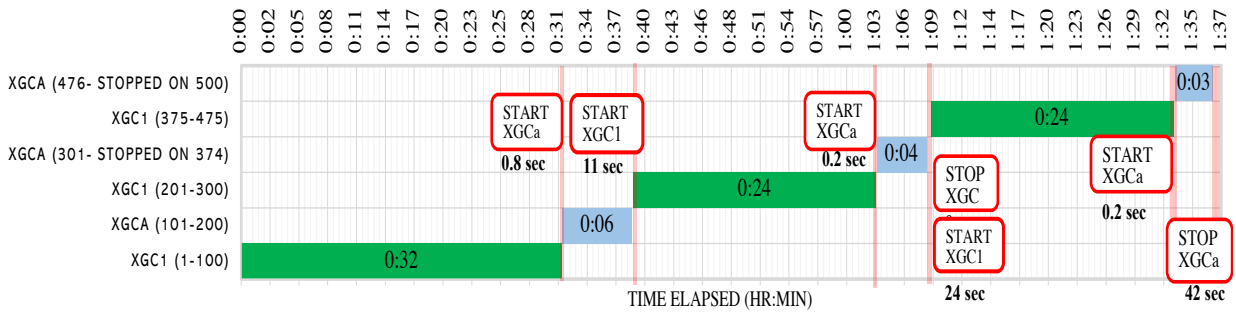


Figure 6.5: Gantt-chart showing the experiment performed on Deepthought2 for the XGC1-XGCa workflow to demonstrate running iterative experiments and terminating tasks based on runtime events.

response times are as follows: $0.8 - 0.2$ seconds for starting XGCa, 11 seconds for starting XGC1, 24 seconds for switching to XGC1 from XGCa, and 42 seconds to stop XGCa. The time to start XGC1 is affected by running the user restart script when XGCa was not stopped by DYFLOW. As XGCa was stopped gracefully, the response time varies depending on how far XGCa was in processing the current timestep when it received the stop signal.

Without DYFLOW, a scientist could list the order in which the jobs will run to complete the desired number of steps in the bash script. For input settings where error accumulation is known to be low, such a script is feasible. In general, error accumulation should be monitored via an additional task, and additional mechanisms need to be created to reconfigure the script revising the order in which the tasks will complete the remaining simulation timesteps after the error accumulation was found to be high. In practice, scientists run the entire simulation using XGC1. With the input used in our experiments this takes approximately 25% more time on each cluster compared to the run using DYFLOW.

6.3 Managing resource assignments to improve throughput/performance

Two runtime scenarios are commonly encountered by scientists due to changes in the resource requirements of the workflow at runtime; (a) Under-provisioning: one or more tasks are assigned fewer resources than required to attain the desired performance, 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: one or more tasks are assigned more resources than required to attain desired performance, so there are times when resources are under-utilized.

For the experiments, I utilized a workflow based on a Gray-Scott simulation. 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 the study. These experiments are based on an earlier implementation of DYFLOW that did not support dataflow control actions.

For this study, I 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 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 vary in computation cost in a data-dependent way (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

Figure 6.6: XML illustrating the setting of sensors for changing the number of CPUs when the pace of the task is outside a desired interval.

```
<monitor>
        <sensors>
            <sensor id="PACE" type="TAUADIOS2">
                <group-by> <group granularity="TASK" reduction-operation="MAX"/> </group-by>
            </sensor>
        </sensors>
        <monitor-tasks>
            <monitor-task name="Isosurface" workflowId="GS-WORKFLOW" var-source="tau-iso.bp"
    distributed="true">
            <apply-sensor sensor-id="PACE" var="looptime">
                <parameter key="var-type" value="double" />
            </apply-sensor>
            </monitor-task>
            ...
        </monitor-tasks>
</monitor>
```

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.

The time taken to complete an iteration of the Gray-Scott simulation (a single iteration of the outermost loop in the application) represents the pace at which the tasks are progressing. The pace is used as a performance metric. Representative settings in DYFLOW for managing the under- and over-provisioning scenarios are shown in Figures 6.6 to 6.8.

Monitoring requirements: I defined a sensor, PACE, that expresses the monitoring metric to use (shown in fig. 6.6). 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 to obtain the maximum time spent in completing a timestep.

Policy setup: I set two policies, *INC_ON_PACE* and *DEC_ON_PACE*, to identify the under-

Figure 6.7: XML illustrating the setting of policies for changing the number of CPUs when the pace of the task is outside a desired interval.

```
<decision>
    <policies>
        <policy id="INC_ON_PACE" >
            <eval operation="GT" threshold="36" />
            <history window="10" operation="AVG "/>
            <sensors-to-use>
                <use-sensor sensor-id="PACE" granularity="TASK" />
            </sensors-to-use>
            <actions> ADDCPU </actions>
            <frequency seconds="5"/>
         </policy>

        <policy id="DEC_ON_PACE" >
            <eval operation="LT" threshold="22" />
            <history window="10" operation="AVG "/>
            <sensors-to-use>
                <use-sensor sensor-id="PACE" granularity="TASK"/>
            </sensors-to-use>
            <actions> RMCPU </actions>
            <frequency seconds="5"/>
        </policy>
    </policies>

   <apply-policies>
    <apply-on workflowId="GS-WORKFLOW">
        <apply-policy policy-id="INC_ON_PACE" eval-task="Isosurface">
            <act-on-task task-name="Isosurface" action="ADDCPU" >
                <param key="N" value="20"/>
            </act-on-task>
        </apply-policy>
    </apply-on>
    ...
   </apply-policies>
</decision>
```

Figure 6.8: XML illustrating the setting of arbitration preferences for changing the number of CPUs when the pace of the task is outside a desired interval.

```
<arbitration>
    <rules>
        <rules-for workflowId="GS-WORKFLOW">
            <workflow-setup seconds="120" />
            <task-priorities>
                <task-prio task-name="GrayScott" priority="0" />
                <task-prio task-name="Isosurface" priority="1" />
                <task-prio task-name="Rendering" priority="2" />
                <task-prio task-name="FFT" priority="3" />
                <task-prio task-name="PDF_Calc" priority="4" />
            </task-priorities>
            <task-dependencies>
                <task-dep task-name="Isosurface" depends-on="GrayScott" type="INSITU" data-stream
    ="gs.bp"/>
                <task-dep task-name="FFT" depends-on="GrayScott" type="INSITU" data-stream="gs.
                    bp"/>
                <task-dep task-name="PDF_Calc" depends-on="GrayScott" type="INSITU" data-stream=
                    "gs.bp" />
                <task-dep task-name="Rendering" depends-on="Isosurface" type="INSITU" data-stream
                    ="iso.bp"/>
            </task-dependencies>
        </rules-for>
    </rules>
</arbitration>
```

and over-provisioning events (shown in fig. 6.7). These policies increase or decrease the number of CPUs assigned to any monitored task (i.e., by 20, the minimum number of processes assigned to any task in these experiments) if the average pace is slower or faster than the desired values (i.e., thresholds) respectively. Policy evaluations are performed based on the average computed over a sliding window of 10 values. This avoids making decisions based on outlier instances of high or low timestep timings. The policies evaluate the sensor output every 5 seconds which was derived based on experimentation. The interval between evaluations impacts the response time of DYFLOW. A large interval can lead to missing or delaying the response to important events. On the other hand, a small interval may lead to responding to short-lived behaviors. For rectifying the effects of runtime performance of the analysis tasks on the simulation to maximize the simulation performance, these policies are only applied to the analysis tasks.

Arbitration settings: Figure 6.8 shows the arbitration preferences. Since these policies will not result in high-level conflicts, there was no need to set policy priories or constraints in the XML. In our experiments, the task priorities are explicitly assigned high to low (0 to 4) based on the order; *Gray-Scott*, *Isosurface*, *Rendering*, *FFT*, *PDF_Calc*.

6.3.1 Responding to under-provisioning to achieve desired performance

I first show an experimental setting in which the resources were under-provisioned. Table 6.2 provides the initial configuration for the under-provisioning experiment on Summit and Deepthought2 that runs for up to 30 minutes.

The XML settings in figs. 6.6 to 6.8 are applied to this scenario. For the threshold for *INC_ON_PACE*, I used a value of 36 seconds based on the desire that the workflow complete

Table 6.2: Initial configuration for *Gray-Scott* workflow that results in resource under- provisioning

TASK(S)	SETTING	SUMMIT	DEEPTHOUGHT2
GRAY-SCOTT	PROCESSES GRID/PROCESS	340 (34 per compute node) $42 \times 140 \times 175$	320 (16 per compute node) $88 \times 88 \times 140$
ISOSURFACE, *RENDERING*, *FFT*, *PDF_CALC*	PROCESSES	20 (2 per compute node)	20 (1 per compute node)
ALL WORKFLOW TASKS	TOTAL STEPS TOTAL COMPUTE NODES TIME LIMIT	50 10 30 MINS	50 20 30 MINS

50 timesteps in the total allotted experiment time of 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.

Summit: Figure 6.9 shows all the dynamic events, giving the timestamps and durations. Figure 6.10 shows the average time per timestep as the Decision module receives them. Gray-Scott was started along with all the analysis tasks; *Isosurface*, *Rendering*, *PDF_Calc*, and *FFT*. 2 mins into the experiment, the Arbitration module considers the suggestion from policy INC_ON_PACE to increase the number of processes for all the analysis processes because the average time per timestep was above the threshold of 36 seconds. The Arbitration module only enables the action to increase the number of processes of the highest priority task, *Isosurface*, from 20 to 40 by acquiring the extra resources from *PDF_Calc*, as it is the lowest priority task. Due to the runtime dependency on *Isosurface*, *Rendering* was also restarted. The Arbitration module took 107 seconds to finalize the plan and wait for the Actuation to finish executing it.

After waiting for 2 mins, the Arbitration module again considered the actions suggested. At

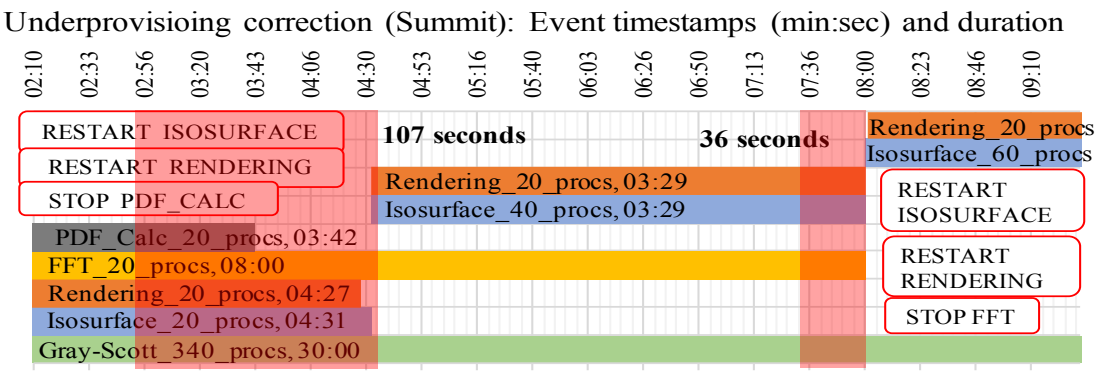


Figure 6.9: Gantt-chart showing the experiment performed on Summit with the **Gray-Scott** workflow to demonstrate correcting under-provisioning of resources along with the response times of DYFLOW

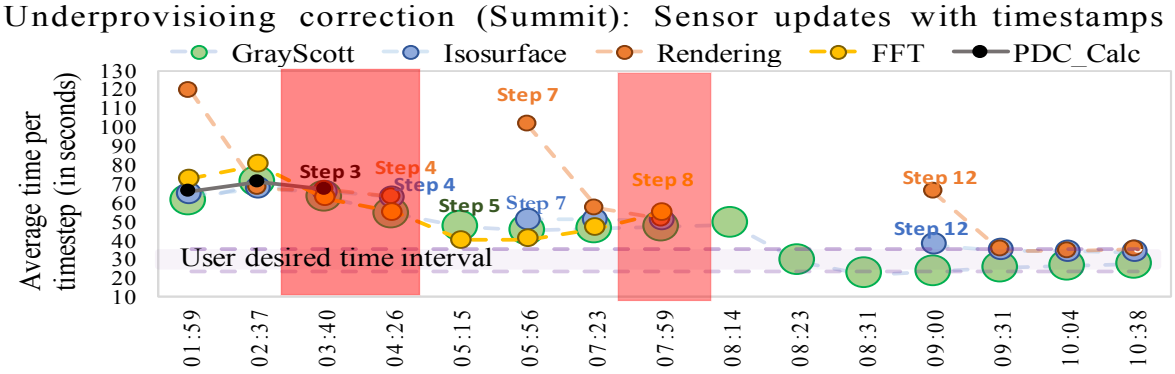


Figure 6.10: Average time per timestep information obtained from the **Gray-Scott** workflow tasks used by DYFLOW to improve performance on Summit.

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*. The Arbitration module took 36 seconds to finalize the plan and wait for the Actuation module to apply the plan. The longer response time was due to waiting for tasks to stop gracefully. After these changes, the average time per timestep for all the tasks was within the desired interval.

Deepthought2: In a similar experimental on Deepthought2 (shown in Figure 6.11), when the average time per timestep was above the threshold of 36 seconds for *Isosurface*, *Isosurface* was restarted by acquiring resources from *PDF_Calc* and *FFT_Calc* while *Rendering* was restarted due to its runtime dependency. After this action, the average time was under the desired threshold for all the tasks. The time to finalize the action plan and execute the same was 87 seconds. The longer response time was due to waiting for tasks to stop gracefully.

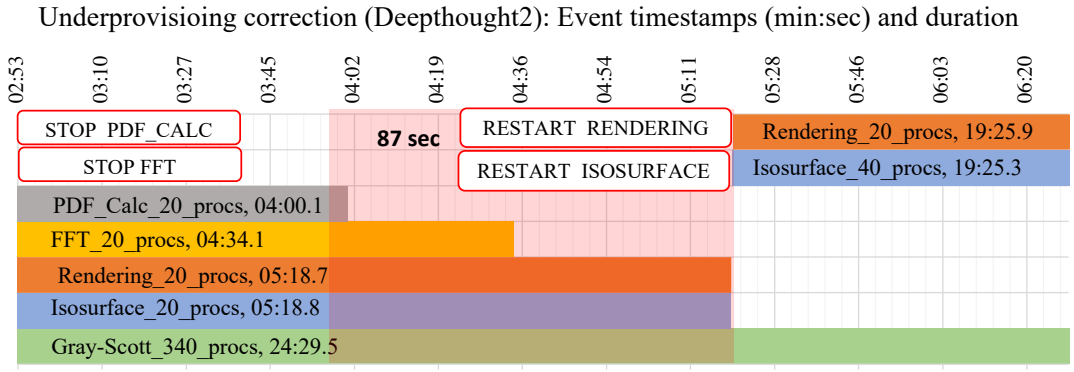


Figure 6.11: Gantt-chart showing the experiment performed on Deepthought2 with the **Gray-Scott** workflow to demonstrate correcting under-provisioning of resources along with the response times of DYFLOW

Without using DYFLOW, the experiment exceeds the allocation time limit, and the workflow tasks terminate prematurely due to timeout (requiring approx. 10-12% additional time to finish on both clusters).

6.3.2 Responding to over-provisioning to achieve desired throughput

I now show an experimental setting where the resources were over-provisioned. Table 6.3 shows the details of the initial configuration for these experiments on Summit and Deepthought2 that simulate a small input instance with an allotted time of 15 minutes.

I use the same XML settings shown in Figures 6.6 to 6.8 for this scenario while adjusting

Table 6.3: Initial configuration for *Gray-Scott* workflow that results in over-provisioning

TASKS	SETTING	SUMMIT	DEEPTHOUGHT2
GRAY-SCOTT	PROCESSES GRID/PROCESS	320 (16 per compute node) $88 \times 88 \times 140$	340 (34 per compute node) $42 \times 140 \times 175$
ISOSURFACE	PROCESSES	60 (6 per compute node)	60 (3 per compute node)
RENDERING	PROCESSES	20 (2 per compute node)	20 (1 per compute node)
FFT, PDF_CALC	PROCESSES	0	0
All tasks	TOTAL STEPS TOTAL COMPUTE NODES TIME LIMIT	50 10 15 MINS	50 20 15 MINS

Overprovisioning correctionon (Summit) : Event timestamps and duration (in min:sec)

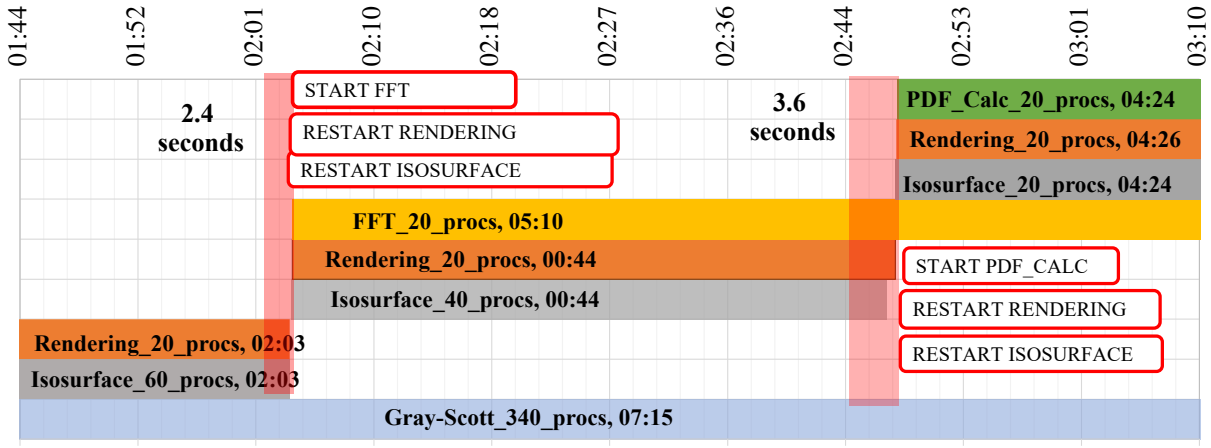


Figure 6.12: Gantt-chart showing the experiment performed on Summit with **Gray-Scott** workflow to demonstrate correcting over-provisioning of resources with low overhead

the threshold based on input settings. For the threshold for *INC_ON_PACE*, I used a value of 18 seconds based on the desire that the workflow complete 50 timesteps in the total allotted experiment time of 15 minutes. For *DEC_ON_PACE*, the threshold value is 12 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 18, or 112), then it can use fewer resources.

Summit: Figure 6.12 shows the dynamic events with their timestamps and duration. Figure 6.13 shows the average time per timestep for updates as they were received by the decision policies on Summit. Gray-Scott was started with two analysis applications; *Isosurface* and *Ren-*

dering. *PDF_Calc* and *FFT* are put in a waitlist. 2 mins into the experiment, the Arbitration starts receiving suggestions from the Decision module. At this point, the Decision module observes that the average time per timestep for the monitored application, i.e. *Isosurface*, is below 12 seconds based on policy DEC_ON_PACE. Therefore, the Decision module suggests that the Arbitration module reduce the number of processes of *Isosurface* by 20 (as suggested by the policy settings). The Arbitration module then selects *FFT* from the waitlist and starts *FFT* with the freed resources. *FFT* was selected over *PDF_Calc* based on priority. Once the workflow settles, the workflow state is re-examined and Decision observes that *Isosurface* and *FFT* are running faster than necessary. The Decision module again suggests the Arbitration module reduce by 20 the number of processes for both tasks. The Arbitration module assigns 20 processes of *Isosurface* to the waiting process *PDF_Calc*. Since *FFT* only had 20 processes, the suggestion from Decision was rejected. Once these changes were applied by the Actuation module, the average times per timestep were in the desired range for all the tasks, and no further actions were suggested throughout the experiment. Due to the runtime reassignment, we see an increase in throughput while the experiment still ran in under 15 minutes (7 minutes 14 seconds on Summit). The response time on Summit was 2.4 and 3.6 seconds for the two dynamic events (less than 2% of the execution time). In this scenario, the timesteps were completed in a small amount of time by each task, so the wait time that allows tasks to stop gracefully was low. Due to the runtime reassignment, we see an increase in throughput while the experiment still ran in under 15 minutes (7 minutes 14 seconds on Summit).

Deepthought2 Figure 6.14 shows the Gantt chart of the experiment on Deepthought2. Only one dynamic change was performed, where *FFT* was started with 20 processes shed by *Isosurface* as the average time per timestep for the monitored application, i.e. *Isosurface*, is

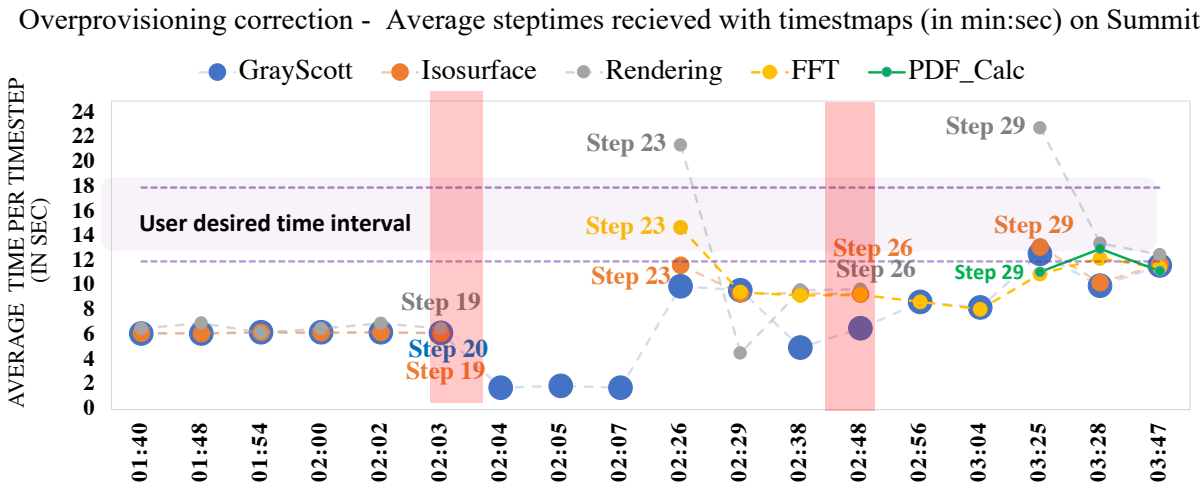


Figure 6.13: Average time per timestep information obtained from the **Gray-Scott** workflow applications used by the dynamic strategies to improve throughput on Summit.

below 12 seconds based on policy DEC_ON_PACE. The response time, in this case, was 8.3 seconds. The entire experiment finished in 9 minutes 2 seconds (under 15 minutes) with more analysis tasks.

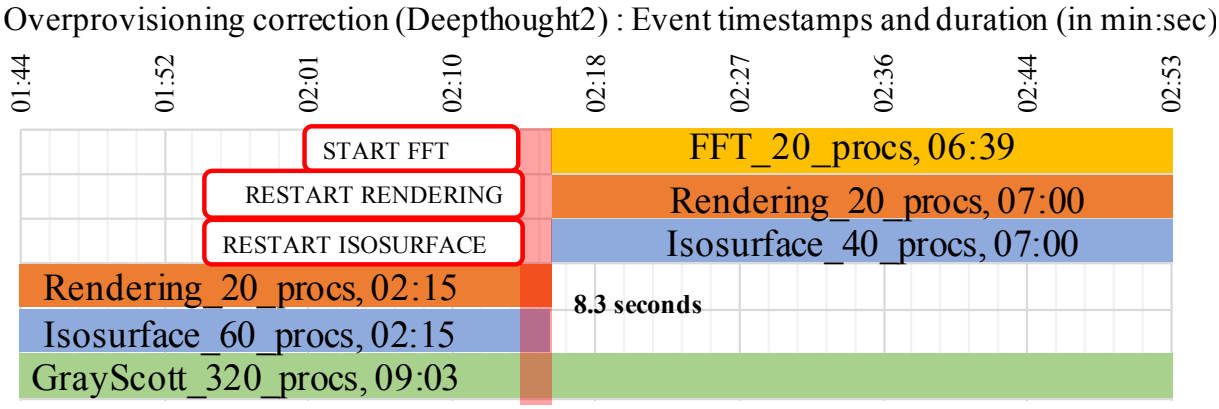


Figure 6.14: Gantt-chart showing the experiment performed on Deepthought2 with **Gray-Scott** workflow to demonstrate correcting over-provisioning of resources with low overhead

6.4 Adapting in response to failures

Handling failures gracefully is desirable for large-scale scientific workflows. Workflow tasks can be exposed to different runtime failures. For instance, a node or network hardware

failure can occur, or memory corruption due to software errors can result in software failure. For *in situ* workflows, loss of data is often another type of failure. For example, the workflow can lose timestep information when tasks reset, i.e., stopping and restarting at runtime (as occurred in Figure 6.10), or buffer overwrites can occur when buffer capacity is exceeded. Identifying the cause of failure in these scenarios usually requires deep analysis. More specifically, there are cases when the failure events are generated by a task or by the flow of data between tasks. This makes failure diagnosis challenging. Therefore, the focus of this section is limited to demonstrating the capability of DYFLOW to help workflows achieve resilience to some types of hardware failures. During resource reassignment, the Arbitration module ensures the exclusion of problematic resources. For this, the Arbitration module continually collects the status of allocated resources from the Actuation module, which relies on either the underlying job scheduler or the workflow management system to provide this information.

The experiment in this section shows a form of resilience to node failure(s) in the cluster using the LAMMPS-based molecular dynamics *in situ* workflow, where the simulation is tightly coupled and co-located with three analysis tasks at runtime. Here, we focused 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*).

A representative XML input for DYFLOW that enables the workflow tasks to restart after failure is shown in Figure 6.15.

Monitor settings: To become aware of failures, I define a sensor, STATUS, that reads the error files generated by job schedulers when tasks fail to get the error number returned. The

Figure 6.15: XML example illustrating the user specification for restarting tasks on failure

```
<DYFLOW>
 <monitor>
   <sensors>
     <sensor id="STATUS" type="ERRORSTATUS">
        <group-by> <group granularity="TASK" reduction-operation="FIRST"/> </group-by>
     </sensor>
   </sensors>
   ...
 </monitor>

 <decision>
  <policies>
    <policy id="RESTART_ON_FAILURE" >
    <eval operation="GT" threshold="128" />
    <sensors-to-use> <use-sensor sensor-id="STATUS" granularity="TASK"/> </sensors-to-use>
    <actions> RESTART </actions>
    <frequency seconds="30"/>
    </policy>
  </policies>
  ...
 </decision>

 <arbitration>
  <rules>
   <rules-for workflowId="Lammps">
    <workflow-setup seconds="120" />
    <task-dependencies>
        <task-dep task-name="RDF_Calc" depends-on="LAMMPS" type="INSITU" data-stream="
     lammps-out.bp"/>
        <task-dep task-name="CS_Calc" depends-on="LAMMPS" type="INSITU" data-stream="lammps
            -out.bp"/>
        <task-dep task-name="CNA_Calc" depends-on="LAMMPS" type="INSITU" data-stream="
            lammps-out.bp"/>
    </task-dependencies>
   </rules-for>
  </rules>
 </arbitration>
</DYFLOW>
```

Table 6.4: Initial configuration for *LAMMPS* workflow on Summit used for failure resilience using 50 compute nodes

SETTING	*LAMMPS*	*CNA_CALC*	*CS_CALC*	*RDF_CALC*
PROCESSES	1500 (30 per compute node)	200 (4 per compute node)	200 (4 per compute node)	200 (4 per compute node)
TOTAL ATOMS	65536000	65536000	65536000	65536000
TOTAL STEPS	1000	100	100	100

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, I read the exit status saved by Savanna after the workflow task completes or fails.

Decision settings: I define a policy $RESTART\_ON\_FAILURE$ to detect failures that restarts the workflow tasks when the error number is greater than 128 (the standard exit codes for system signals). A user can provide a bash script to run before the restart, which modifies the input settings of the tasks to enable them to resume the processing from the last checkpointed output instead of starting from the beginning.

Arbitration settings: The task priorities are not set in this case and will be automatically assigned by the Arbitration module. The task priorities are determined based on the criteria defined in Section 4.1.2. The LAMMPS task will be given the highest priority because of dependencies. All the analysis tasks have the same number of ancestors and dependents so they will be assigned priorities based on the specification order in the workflow configuration file.

Summit: Table 6.4 shows the initial configurations of LAMMPS on Summit utilizing 50 compute nodes for 30 mins. The LAMMPS simulation provides the output after processing 10 timesteps for the analyses. Figure 6.16 shows the timestamps and durations of the dynamic events. 10 mins into the experiment, one of the allocated nodes were taken out of service, causing the entire workflow to fail. The Arbitration module restarts all the tasks by excluding the failed node from the resource assignment and replacing it using one of the free nodes in the allocation.

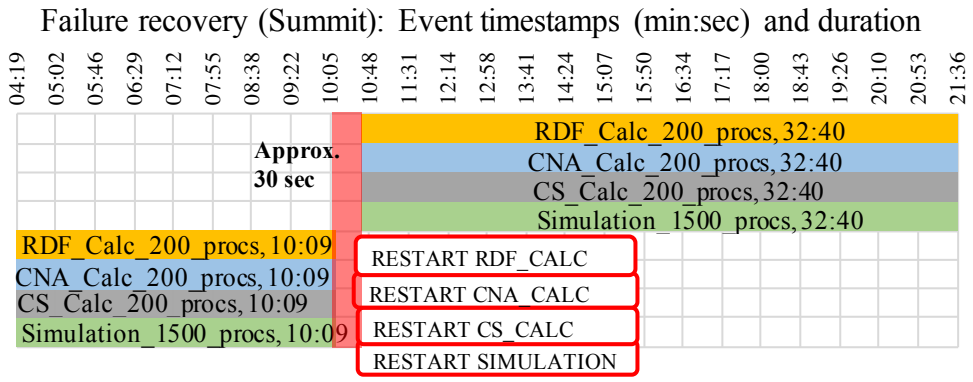


Figure 6.16: Gantt-chart showing the experiment performed on Summit with **LAMMPS** workflow to demonstrate resilience to node failures.

In this experiment, I 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 average time lag from event generation and for the Monitor module to collect and process data was ≈ 0.2 seconds. The time for the Arbitration module to finalize and wait for the execution of the plan was ≈ 0.2 seconds. The additional time results from the delay imposed by frequency settings in evaluating the sensor output.

6.5 Cost summary

I now summarize the runtime costs for the above experiments.

Response time: Cost of generating actions: On average, the interval between an event and initiation of response was less than one second, based on results from both clusters. For LAMMPS, the average interval was 0.2 seconds when a single variable was read. For the XGC experiments, the number of output files generated, which corresponds to the timesteps completed by a task, was used to switch between tasks. The average interval was 0.1 seconds when the

disk was scanned for output files. For the Gray-Scott experiments, when a TAU-generated two-dimensional variable was read to extract desired statistics, the average interval was approximately 0.5 seconds. These times include the time to generate metric values, the time to output suggestions, and the time to formulate the plan. For these times, the time delays imposed by the decision frequency and workflow setup time were excluded (by disabling these features temporarily) The time interval between the event occurrence and generation of an action plan was observed to be affected by the metric computation time (i.e., the volume of measurement data processed).

Cost of performing actions: 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. For the XGC1-XGCa and Gray-Scott experiments, up to 97% of the response time was observed to be spent waiting for tasks to terminate after receiving the signal, on both clusters for most of the experiments. Overall, DYFLOW incurs a small cost to apply the workflow modifications at runtime, but the cost can vary significantly with the experiment setup and environment.

Monitoring overheads: The monitoring overheads on the workflow tasks for the XGC1-XGCa use case were negligible, since the monitoring client ran on the launch node, scanned the disk for file generation, and did not communicate with the workflow tasks. Similarly, no overheads occurred in the case of the LAMMPS workflow, which was configured to handle node failures. In scenarios where TAU intercepts the runtime tasks and produces output statistics on a stream at a given frequency (1 second for the Gray-Scott experiments), the maximum overhead was below 0.1% of the execution time without DYFLOW orchestration. The generated data were continuously written to the stream by TAU, overwriting the unconsumed data in buffers, to keep the memory footprint low and avoid delay imposed by slow consumption of data.

Chapter 7: Conclusion and Future Work

7.1 Conclusions

Contemporary scientific workflows are complex, with unpredictable runtime requirements that require adapting resource assignments, adapting the runtime state of the workflow tasks, and/or managing the data flow. Though existing support is provided to enable elastic resource management to grow and shrink the assigned resources to cater to the changing resource requirements of running tasks, an orchestration service is required that can observe different runtime events and autonomously trigger appropriate actions to adapt the runtime state of the workflow accordingly. This dissertation addresses this concern through a flexible and portable model that enables scientific workflows to employ the many benefits of dynamic orchestration on large-scale parallel systems. The model compartmentalizes dynamic management into four stages; Monitor, Decision, Arbitration, and Actuation. These stages can be individually configured by users to express the events, responses to events, and preferences to establish the workflow requirements and the quality of service desired. The model supports an arbitration protocol that finalizes the actions to perform at runtime by mediating between the responses of multiple policies and ensuring that the workflow runtime state is always valid and consistent with user specifications.

To demonstrate the effectiveness of the model, the dissertation discusses several examples of real-life workflow scenarios with problematic runtime requirements. The proposed model was

applied to orchestrate these scenarios on a standard Linux cluster and a state-of-the-art supercomputing cluster. The results illustrated that the dynamic orchestration service provided by the DYFLOW model enables users to easily employ runtime orchestration capabilities for scientific workflows, incurring a low cost to carry-out runtime changes.

7.2 Extensions and improvements

This section describes several extensions to this work that could be performed to further strengthen the generality and reliability of the proposed model.

Empirical evaluations The experiments in this dissertation cover interesting dynamic scenarios. For instance, I demonstrated that the model can efficiently handle scenarios that require support for adaptive analysis to improve the quality of results, require maintaining desired performance or throughput by handling events based on resource-usage variations or inefficient resource assignments, or require dealing with failures. However, further evaluation could be performed to explore more use cases to showcase the generality of the approach.

The effect of policy constraints and preferences was explored using a scenario where slow network connections could become a performance bottleneck, to achieve higher accuracy of results over performance and vice-versa. More elaborate demonstrations could be performed that show how policy constraints and preferences can be modified to achieve the desired effects.

Implementation support The implementation used in this dissertation relies on several technologies to collect monitoring data at runtime. For instance, the implementation heavily relies on ADIOS2 and the TAU profiling tools. Though these technologies were sufficient for the experi-

ments performed for the dissertation, for more general applicability additional tools and methods to collect the runtime data should be supported. For instance, supporting sensors based on reading data from a DataSpaces server or querying from a database would be useful for some workflow scenarios.

The implementation extends the Cheetah/Savanna workflow tools, which are relatively new for workflow management. Users often choose a workflow management system based on the specific services offered by the system, for instance, the interface. Thus, the implementation of DYFLOW could also extend popular workflow management systems, for example, Pegasus [19], to make DYFLOW widely adopted in practice.

7.3 Future directions

This dissertation performed foundational research towards an orchestration service to flexibly accommodate the various runtime needs of emerging scientific workflows. This section discusses certain areas that may require deeper exploration.

Support for ensemble workflows Scientists sometimes require conducting experiments where several instances of a workflow (i.e. workflow ensembles) are run simultaneously with different coupling complexities, different parameters, or different input or boundary conditions. Parameter sweep experiments are a form of ensemble workflows where instances of workflows are run with variations in the input parameters for scientific analysis. For large ensembles, sophisticated resource management strategies are required to manage ensemble tasks at runtime. The DYFLOW model is currently limited to managing a single instance of a workflow run. Further study is required to explore the features that need to be supported in each of the DYFLOW stages to meet

the requirements of ensemble runs. For instance, work is needed to support features for the expression of the events and policies, and to explore additional actions and preferences that might be useful to manage ensembles.

Improving Arbitration decisions There are opportunities to further improve the Arbitration stage.

Policy prioritization criteria: The default policy prioritization criteria used in the Arbitration stage are based on granularity and cost in terms number of dependent actions, resource requests, and sensors used. Further criteria need to be explored that incorporate both the costs and benefits of the policy responses. Machine learning methods can be utilized so that the Arbitration can learn the criteria where users prefer certain policies to others from similar experiments performed by other users. This can be enabled by maintaining a database that records the user configurations.

Policy validation: The validation step in the Arbitration stage is designed to proactively rule out responses that are futile or invalid based on the latest changes applied. Additional validation methods can be explored to rule out poor policies, such as methods based on a cost/benefit analysis over the course of the experiment.

Resource assignment: The assignment of free resources to a task at runtime is done by first selecting the available free resources (e.g., CPUs) on any node assigned to that task, and then selecting the first available resources in the list of free resources. Topographical-aware placement methods could be employed to provide performance benefits, especially for communication-heavy tasks. Further, the Arbitration stage should be expanded to manage other resources available on

modern compute nodes, such as GPUs and FPGAs.

Supporting adaptable policies When configuring policies, sometimes it is useful to have a set of potential responses instead of only one. One of these responses can be chosen based on the availability of resources or feasibility with respect to other suggestions received by the Arbitration stage To give an example, suppose a policy suggests balancing the overall performance of a workflow by assigning more CPUs to a slow analysis task, but the additional CPUs are not available at runtime to perform this action. In this scenario, instead of taking no action, it might be useful to consider an alternative action, such as replacing the slow analysis task with a lower precision but faster analysis task or terminating the slower analysis task and writing the data to the disk for offline analysis. Thus, for greater flexibility, the policy specifications could allow users to provide multiple actions in response to an event of interest.

When choosing from multiple response choices from a policy, the Arbitration stage could be trained via machine learning models to select the most efficient response based on a cost/benefit analysis from the set of feasible responses. Significant data must be collected from experimentation to enable such training. The Arbitration stage can also adapt the response parameters based on how the response affects the runtime state. For instance, one parameter to change could be how many resources to request for add or remove task operations.

Assisting the user in creating a specification The user specification defines how the workflow will be orchestrated at runtime. Expressing the workflow requirements may not be straightforward, and any poor choice in policy parameters, constraints, or preferences can have huge system cost (resource usage) implications. Hence, a simulator system should be designed that enables

the user to validate the settings in a test environment. Based on the results, users can rule out choices that may lead to undesired decisions.

Further, helper utilities can be supported in DYFLOW that could assist users in determining policy parameters (e.g., thresholds) and rules by providing recommendations. These suggestions could be learned based on data stored in a database that maintains statistics from similar experiments by DYFLOW users.

Appendix A: XML Examples

A.1 Subset of settings for MemContScenario

```
<DYFLOW>
  <monitor>
    <sensors>
      <sensor id="cache-references" type="TAUADIOS2">
        <group-by>
          <group granularity="NODE-TASK" reduction-operation="SUM"/>
        </group-by>
      </sensor>
      <sensor id="cache-miss-percentage" type="TAUADIOS2">
        <group-by>
          <group granularity="NODE-TASK" reduction-operation="SUM"/>
        </group-by>
        <join-sensor sensor-id="cache-references" join-operation="PERCENTAGE"/>
      </sensor>
                    ...
    </sensors>
    <monitor-tasks>
      <monitor-task name="XGC" workflowId="Fusion" var-source="tau-xgc.bp" distributed="true">
        <apply-sensor sensor-id="cache-references" var="LLC-REFERENCES">
```

```
            <parameter key="var-type" value="long"/>
            <parameter key="var-dim" value="none"/>
        </apply-sensor>
    </monitor-task>
    <monitor-task name="XGC" workflowId="Fusion" var-source="tau-xgc.bp" distributed="true">
        <apply-sensor sensor-id="cache-miss-percentage" var="LLC-MISSES">
            <parameter key="var-type" value="long"/>
            <parameter key="var-dim" value="none"/>
        </apply-sensor>
    </monitor-task>
  </monitor-tasks>
</monitor>
<decision>
  <policies>
    <policy id="high-cache-miss-per">
      <eval operation="GT" threshold="20"/>
      <history window="30" operation="CUSTOM" custom-op-name="ChangePercentage" custom-
          op-path="ChangePercentage.py"/>
      <sensors-to-use>
        <use-sensor sensor-id="cache-miss-percentage" granularity="NODE-TASK" true-for="0.5
            "/>
      </sensors-to-use>
      <actions> UNPACK-TASKS </actions>
      <frequency seconds="1"/>
    </policy>
    <policy id="low-ipc">
      <eval operation="LT" threshold="-10"/>
```

```
        <history window="30" operation="CUSTOM" custom-op-name="ChangePercentage" custom-
            op-path="ChangePercentage.py"/>
        <sensors-to-use>
            <use-sensor sensor-id="instruction-per-cycle" granularity="NODE-TASK" true-for="0.5"
                />
        </sensors-to-use>
        <actions> UNPACK-TASKS </actions>
        <frequency seconds="1"/>
    </policy>
    <policy id="high-load-stall-per">
        <eval operation="GT" threshold="10"/>
        <history window="30" operation="CUSTOM" custom-op-name="ChangePercentage" custom-
            op-path="ChangePercentage.py"/>
        <sensors-to-use>
            <use-sensor sensor-id="load-stall-percentage" granularity="NODE-TASK" true-for="0.5"
                />
        </sensors-to-use>
        <actions> UNPACK-TASKS </actions>
        <frequency seconds="1"/>
    </policy>
</policies>
<apply-policies>
    <apply-on workflowId="Fusion">
        <apply-policy eval-task="XGC" policy-id="high-cache-miss-per">
            <act-on-task action="UNPACK-TASKS" task-name="STREAM"/>
        </apply-policy>
        <apply-policy eval-task="PAnalz" policy-id="high-cache-miss-per">
```

```
            <act-on-task action="UNPACK-TASKS" task-name="STREAM"/>
        </apply-policy>
                                ...
    </apply-on>
  </apply-policies>
 </decision>
 <arbitration>
  <rules>
   <rules-for workflowId="Fusion">
    <workflow-setup seconds="120"/>
    <policy-constraint name="high-cache-miss-per" depends-on="low-ipc high-load-stalls-per
      "/>
    <policy-constraint name="low-ipc" depends-on="high-cache-miss-per high-load-stalls-per
      "/>
    <policy-constraint name="high-load-stalls-per" depends-on="high-cache-miss-per low-ipc
      "/>
   </rules-for>
  </rules>
 </arbitration>
</DYFLOW>
```

A.2 Subset of settings for NetContScenario

```
<?xml version="1.0" encoding="UTF-8"?>
<DYFLOW>
 <monitor>
  <sensors>
```

```
        <sensor id="transfer-time" type="TAUADIOS2">
            <group-by>
                <group granularity="TASK" reduction-operation="FIRST"/>
            </group-by>
        </sensor>
    </sensors>
    <monitor-tasks>
        <monitor-task name="DMZ" workflowId="LAMMPS" var-source="tau-DMZ.bp" distributed="
            false">
            <apply-sensor sensor-id="transfer-time" var="transfer-time">
                <parameter key="var-type" value="long"/>
                <parameter key="var-dim" value="none"/>
            </apply-sensor>
        </monitor-task>
    </monitor-tasks>
</monitor>
<decision>
    <policies>
        <policy id="LosslessComp">
            <eval operation="GT" threshold="T1"/>
            <history window="5" operation="EXP-AVG"/>
            <sensors-to-use>
                <use-sensor sensor-id="transfer-time" granularity="NODE-TASK"/>
            </sensors-to-use>
            <actions> COMPRESS-BLOSC </actions>
            <frequency seconds="2"/>
        </policy>
```

```
<policy id="LossyCompSZ-1e7">
    <eval operation="GT" threshold="T2"/>
    <history window="5" operation="EXP-AVG"/>
    <sensors-to-use>
        <use-sensor sensor-id="transfer-time" granularity="NODE-TASK"/>
    </sensors-to-use>
    <actions> COMPRESS-SZ </actions>
    <action-params action="COMPRESS-SZ">
        <param key="absErrBound" value="1e7"/>
        <param key="errorBoundMode" value="ABS"/>
    </action-params>
    <frequency seconds="2"/>
</policy>
<policy id="LossyCompSZ-2e7">
    <eval operation="GT" threshold="T3"/>
    <history window="5" operation="EXP-AVG"/>
    <sensors-to-use>
        <use-sensor sensor-id="transfer-time" granularity="NODE-TASK"/>
    </sensors-to-use>
    <actions> COMPRESS-SZ </actions>
    <action-params action="COMPRESS-SZ">
        <param key="absErrBound" value="2e7"/>
        <param key="errorBoundMode" value="ABS"/>
    </action-params>
    <frequency seconds="2"/>
</policy>
<policy id="StopTransfer">
```

```
            <eval operation="GT" threshold="T4"/>
            <history window="5" operation="EXP-AVG"/>
            <sensors-to-use>
                <use-sensor sensor-id="transfer-time" granularity="NODE-TASK"/>
            </sensors-to-use>
            <actions> SAVE-STEPS </actions>
            <frequency seconds="2"/>
        </policy>
    </policies>
    <apply-policies>
        <apply-on workflowId="LAMMPS">
                        <apply-policy eval-task="DMZ" policy-id="LosslessComp">
            <act-on-task task-name="DMZ" action="COMPRESS-BLOSC" data-stream="tau-DMZ.bp"
                />
             </apply-policy>
             <apply-policy eval-task="DMZ" policy-id="LosslessComp1">
            <act-on-task task-name="DMZ" action="COMPRESS-SZ" data-stream="tau-DMZ.bp"/>
             </apply-policy>
             <apply-policy eval-task="DMZ" policy-id="LosslessComp2">
            <act-on-task task-name="DMZ" action="COMPRESS-SZ" data-stream="tau-DMZ.bp"/>
                        </apply-policy>
                        ...
        </apply-on>
    </apply-policies>
</decision>
<arbitration>
    <rules>
```

```
        <rules-for workflowId="LAMMPS">
            <workflow-setup seconds="120"/>
            <policy-constraint name="StopTransfer" incompatible-with="LossyComp1 LossyComp2
                LossLessComp"/>
            <policy-priorities>
                <policy-prio policy-id="LosslessComp" priority="P1" />
                <policy-prio policy-id="LossyComp" priority="P2" />
                <policy-prio policy-id="LossyComp" priority="P3" />
                <policy-prio policy-id="StopTransfer" priority="P4" />
            </policy-priorities>
        </rules-for>
    </rules>
  </arbitration>
</DYFLOW>
```

Appendix B: XML schema for user specification

B.1 Main DYFLOW schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:include schemaLocation="./Monitor.xsd"/>
    <xs:include schemaLocation="./Decision.xsd"/>
    <xs:include schemaLocation="./Arbitration.xsd"/>
    <xs:element name="DYFLOW">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="monitor"/>
                <xs:element ref="decision"/>
                <xs:element ref="arbitration"/>
            </xs:sequence>
        </xs:complexType>
        <xs:key name="sensor-key">
            <xs:selector xpath=".//sensor"/>
            <xs:field xpath="@id"/>
        </xs:key>
        <xs:key name="policy-key">
            <xs:selector xpath=".//policy"/>
```

```
        <xs:field xpath="@id"/>
      </xs:key>
      <xs:keyref name="sjoin-ref" refer="sensor-key">
        <xs:selector xpath=".//join-sensor"/>
        <xs:field xpath="@sensor-id"/>
      </xs:keyref>
      <xs:keyref name="smap-ref" refer="sensor-key">
        <xs:selector xpath=".//apply-sensor"/>
        <xs:field xpath="@sensor-id"/>
      </xs:keyref>
      <xs:keyref name="psmap-ref" refer="sensor-key">
        <xs:selector xpath=".//use-sensor"/>
        <xs:field xpath="@sensor-id"/>
      </xs:keyref>
      <xs:keyref name="pmap-ref" refer="policy-key">
        <xs:selector xpath=".//apply-policy"/>
        <xs:field xpath="@policy-id"/>
      </xs:keyref>
      <xs:keyref name="pprio-ref" refer="policy-key">
        <xs:selector xpath=".//policy-prio"/>
        <xs:field xpath="@policy-id"/>
      </xs:keyref>
    </xs:element>
</xs:schema>
```

B.2 Monitor settings: configuring sensors

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:cmn="Common" xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault=
    "qualified">
  <xs:import schemaLocation="./CommonTypes.xsd" namespace="Common"/>
  <xs:element name="monitor">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sensors" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="monitor-tasks" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="sensors">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sensor" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="sensor">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="preprocess" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="group-by" minOccurs="1" maxOccurs="4"/>
        <xs:element ref="join-sensor" minOccurs="0" maxOccurs="1"/>
```

```
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="id" use="required"/>
        <xs:attribute type="cmn:sources" name="type" use="required"/>
    </xs:complexType>
</xs:element>
    <xs:element name="preprocess">
        <xs:complexType>
            <xs:attribute type="cmn:prepop" name="operation" use="required"/>
            <xs:attribute type="xs:string" name="custom-op-name" use="optional"/>
            <xs:attribute type="xs:string" name="custom-op-path" use="optional"/>
        </xs:complexType>
    </xs:element>
<xs:element name="group-by">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="group" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
    <xs:key name="group-key">
        <xs:selector xpath=".//group"/>
        <xs:field xpath="@granularity"/>
    </xs:key>
</xs:element>
<xs:element name="group">
    <xs:complexType>
        <xs:attribute type="cmn:granularitylevels" name="granularity" use="required"/>
        <xs:attribute type="cmn:reductionop" name="reduction-operation" use="required"/>
```

```
        <xs:attribute type="xs:NCName" name="custom-op-name" use="optional"/>
        <xs:attribute type="xs:string" name="custom-op-path" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="join-sensor">
    <xs:complexType>
        <xs:attribute type="xs:NCName" name="sensor-id" use="required"/>
        <xs:attribute type="cmn:joinop" name="join-operation" use="required"/>
        <xs:attribute type="xs:string" name="custom-op-name" use="optional"/>
        <xs:attribute type="xs:string" name="custom-op-path" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="monitor-tasks">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="monitor-task" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="monitor-task">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="apply-sensor" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="name" use="required"/>
        <xs:attribute type="xs:NCName" name="workflowId" use="required"/>
        <xs:attribute type="xs:string" name="var-source" use="required"/>
```

```
        <xs:attribute type="xs:boolean" name="distributed" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="apply-sensor">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="sensor-id" use="required"/>
        <xs:attribute type="xs:token" name="var" use="optional"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="parameter">
      <xs:complexType>
        <xs:attribute type="xs:token" name="key" use="required"/>
        <xs:attribute type="xs:token" name="value" use="required"/>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

B.3 Decision settings: configuring policies

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:cmn="Common" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:import schemaLocation="./CommonTypes.xsd" namespace="Common"/>
    <xs:element name="decision">
      <xs:complexType>
```

```
        <xs:sequence>
            <xs:element ref="policies" minOccurs="1" maxOccurs="1"/>
            <xs:element ref="apply-policies" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="policies">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="policy" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="policy">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="eval" minOccurs="1" maxOccurs="1"/>
            <xs:element ref="history" minOccurs="0" maxOccurs="1"/>
            <xs:element ref="sensors-to-use" minOccurs="1" maxOccurs="1"/>
            <xs:element ref="actions" minOccurs="1" maxOccurs="1"/>
            <xs:element ref="action-params" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element type="cmn:time" name="frequency" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="id" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="eval">
```

```
    <xs:complexType>
        <xs:attribute type="xs:string" name="threshold" use="required"/>
        <xs:attribute type="cmn:comparison" name="operation" use="required"/>
        <!--<xs:attribute type="xs:NCName" name="operation" use="required"/> -->
    </xs:complexType>
</xs:element>
<xs:element name="history">
    <xs:complexType>
        <xs:attribute type="xs:decimal" name="window" use="required"/>
        <xs:attribute type="cmn:postop" name="operation" use="required"/>
        <xs:attribute type="xs:string" name="custom-op-path" use="optional"/>
        <xs:attribute type="xs:string" name="custom-op-name" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="sensors-to-use">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="use-sensor" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="use-sensor">
    <xs:complexType>
        <xs:attribute type="xs:NCName" name="sensor-id" use="required"/>
        <xs:attribute type="cmn:granularitylevels" name="granularity" use="required"/>
        <xs:attribute type="xs:decimal" name="true-for" use="optional"/>
    </xs:complexType>
```

```
</xs:element>
<xs:element name="actions">
    <xs:simpleType>
        <xs:list itemType="cmn:action-supported"/>
    </xs:simpleType>
</xs:element>
<xs:element name="action-params">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="param" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="cmn:action-supported" name="action" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="param">
    <xs:complexType>
        <xs:attribute type="xs:token" name="key" use="required"/>
        <xs:attribute type="xs:token" name="value" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="apply-policies">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="apply-on" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
<xs:element name="apply-on">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="apply-policy" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="workflowId" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="apply-policy">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="act-on-task" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="policy-id" use="required"/>
        <xs:attribute type="xs:NCName" name="eval-task" use="required"/>
        <xs:attribute type="xs:string" name="threshold" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="act-on-task">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="param" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="task-name" use="required"/>
        <xs:attribute type="xs:NCName" name="data-stream" use="optional"/>
        <xs:attribute type="cmn:action-supported" name="action" use="required"/>
    </xs:complexType>
```

```
    </xs:element>
</xs:schema>
```

B.4 Arbitration settings: establishing constraints and preferences

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:cmn="Common">
    <xs:import schemaLocation="CommonTypes.xsd" namespace="Common"/>
    <xs:element name="arbitration">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="rules" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="rules">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="rules-for" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="rules-for">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="workflow-setup" minOccurs="1" maxOccurs="1"/>
```

```
                    <XS:element ref="" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="policy-constraint" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="policy-priorities" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="task-priorities" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="task-dependencies" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:NCName" name="workflowId" use="required"/>
    </xs:complexType>
</xs:element>
        <xs:element name="workflow-setup">
    <xs:complexType>
        <xs:attribute type="xs:nonNegativeInteger" name="seconds" use="required"/>
        <xs:attribute type="xs:nonNegativeInteger" name="mseconds" use="optional"/>
    </xs:complexType>
</xs:element>
      <xs:element name="validation-setup">
    <xs:complexType>
        <xs:attribute type="cmn:valid-criteria" name="type" use="required"/>
        <xs:attribute type="cmn:action-supported" name="action" use="optional"/>
                    <xs:attribute type="xs:boolean" name="disable" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="policy-constraint">
    <xs:complexType>
        <xs:attribute type="xs:NCName" name="name" use="required"/>
        <xs:attribute type="cmn:listtype" name="incompatible-with"/>
        <xs:attribute type="cmn:listtype" name="depends-on"/>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="policy-priorities">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="unset-criteria" type="cmn:policy-criteria" minOccurs="0" maxOccurs="4
          "/>
        <xs:element ref="policy-prio" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="policy-prio">
    <xs:complexType>
      <xs:attribute type="xs:NCName" name="policy-id" use="required"/>
      <xs:attribute type="xs:nonNegativeInteger" name="priority" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="task-priorities">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="unset-criteria" type="cmn:task-criteria" minOccurs="0" maxOccurs="2"
          />
        <xs:element ref="task-prio" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="task-prio">
```

```
    <xs:complexType>
      <xs:attribute type="xs:NCName" name="task-name" use="required"/>
      <xs:attribute type="xs:nonNegativeInteger" name="priority" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="task-dependencies">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="task-dep" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="task-dep">
    <xs:complexType>
      <xs:attribute type="xs:NCName" name="task-name" use="required"/>
      <xs:attribute type="cmn:listtype" name="depends-on" use="required"/>
      <xs:attribute type="cmn:deptype" name="type" use="required"/>
      <xs:attribute type="xs:string" name="data-stream" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.5 Declarations: pre-defined operations, actions and supported values

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="Common">
  <xs:simpleType name="granularitylevels">
```

```
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="TASK"/>
        <xs:enumeration value="NODE-TASK"/>
        <xs:enumeration value="NODE"/>
        <xs:enumeration value="WORKFLOW"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="sources">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="TAUADIOS2"/>
        <xs:enumeration value="ADIOS2"/>
        <xs:enumeration value="ERRORSTATUS"/>
        <xs:enumeration value="DISKSCAN"/>
        <xs:enumeration value="TEXTFILE"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="action-supported">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="ADDCPU"/>
        <xs:enumeration value="RMCPU"/>
        <xs:enumeration value="STOP"/>
        <xs:enumeration value="START"/>
        <xs:enumeration value="RESTART"/>
        <xs:enumeration value="STOP"/>
        <xs:enumeration value="SWITCH"/>
        <xs:enumeration value="UNPACK-TASKS"/>
        <xs:enumeration value="COMPRESS-BLOSC"/>
```

```
        <xs:enumeration value="COMPRESS-BZIP2"/>
        <xs:enumeration value="COMPRESS-SZ"/>
        <xs:enumeration value="COMPRESS-ZFP"/>
        <xs:enumeration value="COMPRESS-MGARD"/>
        <xs:enumeration value="WRITE-FREQ"/>
        <xs:enumeration value="READ-FREQ"/>
        <xs:enumeration value="HOLD-STEPS"/>
        <xs:enumeration value="SAVE-STEPS"/>
        <xs:enumeration value="CHECKPOINT-FREQ"/>
        <xs:enumeration value="RR-DIST"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="prepop">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="MIN"/>
        <xs:enumeration value="MAX"/>
        <xs:enumeration value="SUM"/>
        <xs:enumeration value="AVG"/>
        <xs:enumeration value="STD"/>
        <xs:enumeration value="VAR"/>
        <xs:enumeration value="FIRST"/>
        <xs:enumeration value="LAST"/>
        <xs:enumeration value="ANY"/>
        <xs:enumeration value="CUSTOM"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="reductionop">
```

```
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="MIN"/>
        <xs:enumeration value="MAX"/>
        <xs:enumeration value="SUM"/>
        <xs:enumeration value="AVG"/>
        <xs:enumeration value="STD"/>
        <xs:enumeration value="VAR"/>
        <xs:enumeration value="FIRST"/>
        <xs:enumeration value="LAST"/>
        <xs:enumeration value="ANY"/>
        <xs:enumeration value="CUSTOM"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="joinop">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="MIN"/>
        <xs:enumeration value="MAX"/>
        <xs:enumeration value="SUM"/>
        <xs:enumeration value="DIV"/>
        <xs:enumeration value="MUL"/>
        <xs:enumeration value="PERCENTAGE"/>
        <xs:enumeration value="CUSTOM"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="postop">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="MIN"/>
```

```
        <xs:enumeration value="MAX"/>
        <xs:enumeration value="SUM"/>
        <xs:enumeration value="AVG"/>
        <xs:enumeration value="STD"/>
        <xs:enumeration value="VAR"/>
        <xs:enumeration value="EXP-AVG"/>
        <xs:enumeration value="SKEW"/>
        <xs:enumeration value="KURT"/>
        <xs:enumeration value="CUSTOM"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="comparison">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="GT"/>
        <xs:enumeration value="LT"/>
        <xs:enumeration value="EQ"/>
        <xs:enumeration value="GE"/>
        <xs:enumeration value="LE"/>
        <xs:enumeration value="NEQ"/>
        <xs:enumeration value="IN"/>
        <xs:enumeration value="NOTIN"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="policy-criteria">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="GRANULARITY"/>
        <xs:enumeration value="COST"/>
```

```
        <xs:enumeration value="RESOURCES"/>
        <xs:enumeration value="SENSORS"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="task-criteria">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="DEPENDENTS"/>
        <xs:enumeration value="ANCESTORS"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="valid-criteria">
     <xs:restriction base="xs:NCName">
        <xs:enumeration value="REPEATITION"/>
        <xs:enumeration value="CONFLICT"/>
     </xs:restriction>
</xs:simpleType>
<xs:simpleType name="listtype">
    <xs:list itemType="xs:NCName"/>
</xs:simpleType>
<xs:simpleType name="deptype">
    <xs:restriction base="xs:NCName">
        <xs:enumeration value="INSITU"/>
        <xs:enumeration value="LOOSE"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="time">
    <xs:attribute name="seconds" type="xs:nonNegativeInteger"/>
```

```
        <xs:attribute name="minutes" type="xs:nonNegativeInteger"/>
        <xs:attribute name="mseconds" type="xs:nonNegativeInteger"/>
    </xs:complexType>
</xs:schema>
```

Bibliography

[1] Dong H Ahn, Ned Bass, Albert Chu, Jim Garlick, et al. Flux: Overcoming scheduling challenges for exascale workflows. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 10–19. IEEE, 2018.

[2] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas 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*, May 2012.

[3] Amazon EC2. *https://aws.amazon.com/ec2/.*

[4] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive Parallel Programming in Python. *arXiv e-prints*, page arXiv:1905.02158, May 2019.

[5] Renu Bala. An improved heft algorithm using multi-criterian resource factors. *International Journal of Computer Science and Information Technologies*, 5(6), 2014.

[6] Derik Barseghian, Ilkay Altintas, Matthew B. Jones, Daniel Crawl, et al. Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50, Jan 2010.

[7] Angel M. Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20, 2019.

[8] Angel M. Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20, 2019.

[9] Saurabh Bilgaiyan, Santwana Sagnika, and Madhabananda Das. Workflow scheduling in cloud computing environment using cat swarm optimization. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 680–685, 2014.

[10] Blosc lossless compressor. Website: `https://www.blosc.org/`.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *IEEE Computing in Science and Engineering*, 15(6):36–45, November 2013.

[12] Shirley Browne, Christine Deane, George Ho, and Phil Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[13] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. Storage*, 7(3), October 2011.

[14] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. PMIx: Process management for exascale environments. In *Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI '17)*. ACM, 2017.

[15] Jieyang Chen, Qiang Guan, Zhao Zhang, Xin Liang, Louis Vernon, Allen McPherson, Li-Ta Lo, Patricia Grubel, Tim Randles, Zizhong Chen, and James Ahrens. Beeflow: A workflow management system for in situ processing across hpc and cloud systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1029–1038, 2018.

[16] Hyungmin Cho, Eric Cheng, Thomas Shepherd, Chen-Yong Cher, and Subhasish Mitra. System-level effects of soft errors in uncore components. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1497–1510, 2017.

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

[18] J. Dayal, J. Lofstead, G. Eisenhauer, K. Schwan, et al. Soda: Science-driven orchestration of data analytics. In *2015 IEEE 11th International Conference on e-Science*, pages 475–484, August 2015.

[19] Ewa Deelman, Karan Vahi, Mats Rynge, Gideon Juve, et al. Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing*, 20(1):70–76, 2016.

[20] Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739, 2016.

[21] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36. ACM, 2010.

[22] Ciprian Docan, Fan Zhang, Tong Jin, Hoang Bui, et al. Activespaces: Exploring dynamic code deployment for extreme scale data processing. *Concurr. Comput. : Pract. Exper.*, 27(14):3724–3745, September 2015.

[23] Matthieu Dorier, Zhe Wang, Utkarsh Ayachit, Shane Snyder, Robert Ross, and Manish Parashar. Colza: Enabling elastic in situ visualization for high-performance computing simulations. In *In Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium*, 2022.

[24] Matthieu Dorier, Orcun Yildiz, Tom Peterka, and Robert Ross. The challenges of elastic in situ analysis and visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '19, page 23–28. ACM, 2019.

[25] David Dossot. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[26] Ian Foster, Mark Ainsworth, Bryce Allen, Julie Bessac, Franck Cappello, Jong Youl Choi, Emil Constantinescu, Philip E. Davis, Sheng Di, Wendy Di, Hanqi Guo, Scott Klasky, Kerstin Kleese Van Dam, Tahsin Kurc, Qing Liu, Abid Malik, Kshitij Mehta, Klaus Mueller, Todd Munson, George Ostouchov, Manish Parashar, Tom Peterka, Line Pouchard, Dingwen Tao, Ozan Tugluk, Stefan Wild, Matthew Wolf, Justin M. Wozniak, Wei Xu, and Shinjae Yoo. Computing just what you need: Online data analysis and reduction at extreme scales. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 3–19. Springer International Publishing, 2017.

[27] William Fox, Devarshi Ghoshal, Abel Souza, Gonzalo P. Rodrigo, and Lavanya Ramakrishnan. E-hpc: A library for elastic resource management in hpc environments. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, WORKS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[28] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. ADIOS 2: The Adaptable Input Output System. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[29] A. Goswami, Y. Tian, K. Schwan, F. Zheng, et al. Landrush: Rethinking in-situ analysis for GPGPU workflows. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 32–41, May 2016.

[30] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Orti, Vicenc Beltran, and Antonio J. Pena. DMR API: Improving cluster productivity by turning applications into malleable. *Parallel Computing*, 78:54 – 66, 2018.

[31] Salomon Janhunen, Robert Hager, Seung-Hoe Ku, Choong-Seock Chang, et al. Integrated multi-scale simulations of drift-wave turbulence: coupling of two kinetic codes XGC1 and XGCa. In *APS Meeting Abstracts*, 2015.

[32] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of NetDB'11*, volume 11, pages 1–7, 2011.

[34] S Ku, CS Chang, and PH Diamond. 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):115021, 2009.

[35] Seung-Hoe Ku, C. S. Chang, J. Seo, J. Lang, and S. Parker. XGC1 total-f simulation of electrostatic edge turbulence and neoclassical physics with kinetic electrons and ions. In *APS Division of Plasma Physics Meeting Abstracts*, volume 54 of *APS Meeting Abstracts*, page BP8.154, October 2012.

[36] Kevin Lee, Norman W. Paton, Rizos Sakellariou, Ewa Deelman, Alvaro A. A. Fernandes, and Gaurang Mehta. Adaptive workflow processing and execution in pegasus. *Concurrency and Computation: Practice and Experience*, 21(16):1965–1981, 2009.

[37] Ke Liu, Hai Jin, Jinjun Chen, X. Liu, Dong Yuan, and Yun Yang. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *Int. J. High Perform. Comput. Appl.*, 24:445–456, 2010.

[38] IBM Spectra LSF. `https://www.ibm.com/support/knowledgecenter/en/SSWRJV_10.1.0/lsf_welcome/lsf_welcome.html`, 2019.

[39] Milos Madi, Danijel Markovi, and Miroslav Radovanovi. Comparison of meta-heuristic algorithms for solving machining optimization problems. In *FACTA UNIVERSITATIS. Series: Mechanical Engineering*, 2013.

[40] Mohammad Masdari, Sima ValiKardan, Zahra Shahi, and Sonay Imani Azar. Towards workflow scheduling in cloud computing: A comprehensive analysis. *Journal of Network and Computer Applications*, 66:64–82, 2016.

[41] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[42] Kshitij Mehta, Bryce Allen, Matthew Wolf, Jeremy Logan, Eric Suchyta, Swati Singhal, Jong Y. Choi, Keichi Takahashi, Kevin Huck, Igor Yakushin, Alan Sussman, Todd Munson, Ian Foster, and Scott Klasky. A codesign framework for online data analysis and reduction. *Concurrency and Computation: Practice and Experience*, 34(14):e6519, 2022.

[43] Microsoft Azure. *https://azure.microsoft.com/en-us/*.

[44] MPI 4.0 specification (draft version). https://www.mpi-forum.org/docs/.

[45] NVIDIA CUDA profiling tools interface documentation. https://docs.nvidia.com/cupti/index.html.

[46] Eduardo Ogasawara, Jonas Dias, Vitor Silva, Fernando Chirigati, et al. Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341, 2013.

[47] Valerio Pascucci, Jim Ahrens, E. Wes Bethel, Hank Childs, Jian Huang, Ken Joy, Quincey Koziol, Jay Lofstead, Jeremy Meredith, Kenneth Moreland, George Ostrouchov, Michael Papka, Venkatram Vishwanath, Matthew Wolf, Nicholas Wright, and Kesheng Wu. Report from the DOE ASCR 2011 workshop on exascale data management, analysis, and visualization. Technical report, DOE ASCR, Feb. 2011.

[48] Portable Batch System. `https://github.com/pbspro/pbspro/`, 2019.

[49] S. Perarnau, J. A. Zounmevo, M. Dreher, B. C. Van Essen, et al. Argo NodeOS: Toward unified resource management for exascale. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 153–162, 2017.

[50] Mustafizur Rahman, Rafiul Hassan, Rajiv Ranjan, and Rajkumar Buyya. Adaptive workflow scheduling for dynamic grid and cloud computing environment. *Concurrency and Computation: Practice and Experience*, 25(13):1816–1842, 2013.

[51] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[52] Ranjit Singh Sarban Singh and Sarbjeet Singh. Score based deadline constrained workflow scheduling algorithm for cloud systems. In *CloudCom 2013*, 2013.

[53] Swati Singhal, Alan Sussman, Matthew Wolf, Kshitij Mehta, and Jong Youl Choi. DYFLOW: A flexible framework for orchestrating scientific workflows on supercomputers. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21. ACM, 2021.

[54] SLURM job scheduler. `https://slurm.schedmd.com/`, 2019.

[55] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088, July 2008.

[56] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022.

[57] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[58] Matteo Turilli, Andre Merzky, Vivek Balasubramanian, and Shantenu Jha. Building blocks for workflow system middleware. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 348–349. IEEE, 2018.

[59] G. Vallee, C. E. A. Gutierrez, and C. Clerget. 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)*, pages 43–48, 2019.

[60] Michael Wilde, Mihael Hategan, Justin M. Wozniak, et al. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37:633–652, 2011.

[61] Fuhui Wu, Qingbo Wu, and Yusong Tan. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, 71(9):3373–3418, 2015.

[62] Hu Wu, Zhuo Tang, and Renfa Li. A priority constrained scheduling strategy of multiple workflows for cloud computing. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 1086–1089, 2012.

[63] Zeromq: An open-source universal messaging libraray. *https://zeromq.org/*, 2020.

[64] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, et al. 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, 2013.