Enabling Performance Observability for Heterogeneous HPC Workflows with SOMA

Dewi Yokelson
University of Oregon
Oregon, USA
dewiy@uoregon.edu

Mikhail Titov
Brookhaven National Lab
New York, USA
mtitov@bnl.gov

Srinivasan Ramesh
NVIDIA
California, USA
srramesh@nvidia.com

Ozgur Ozan Kilic
Brookhaven National Lab
New York, USA
okilic@bnl.gov

Matteo Turilli
Rutgers University
New Jersey, USA
matteo.turilli@rutgers.edu

Shantenu Jha
Rutgers University
New Jersey, USA
shantenu.jha@rutgers.edu

Allen Malony
University of Oregon
Oregon, USA
malony@uoregon.edu

ABSTRACT

Heterogeneous workflows represent a promising approach for overcoming traditional application performance limitations and to accelerate scientific insight on high-performance computing (HPC) platforms. As HPC platforms grow in size and complexity, managing and optimizing workflow resources while maximizing scientific output assumes vital importance. Optimal workflow resource allocation requires high-quality and timely information about the state of the hardware resources, the status of the pending tasks, the performance of the tasks that have already been executed, and the current status of the workflow itself. A robust performance observability framework that captures and delivers this information can fundamentally improve the quality of decision-making within the workflow system, setting the stage for the adaptive execution of workflow tasks. We propose the use of SOMA, a service-based performance observability framework for such HPC workflows. With the RADICAL-Pilot runtime system as a development vehicle, SOMA demonstrates that service-based architectures coupled with an appropriate data model can serve the performance monitoring needs of large-scale ensemble workflows in a low-overhead fashion. Effective observability of workflow performance requires exporting, storing, and analyzing several types of performance data from across the application and workflow software stacks. Our study finds significant benefits in integrating observability frameworks as first-class citizens within an HPC workflow software stack. In this paper, we demonstrate how SOMA can simultaneously observe the performance states of the individual tasks, system hardware, and the workflow as a whole. Such information can then be employed to calculate better resource allocation and task configuration.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computer systems organization** → **Client-server architectures**.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673100

KEYWORDS

high-performance computing, performance monitoring, heterogeneous workflows, workflow monitoring

ACM Reference Format:
Dewi Yokelson, Mikhail Titov, Srinivasan Ramesh, Ozgur Ozan Kilic, Matteo Turilli, Shantenu Jha, and Allen Malony. 2024. Enabling Performance Observability for Heterogeneous HPC Workflows with SOMA. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673100

1 INTRODUCTION

HPC clusters have continued to grow in size, scale, and complexity, offering degrees of scale-out parallelism that few scientific applications can take full advantage of using traditional scaling strategies. In the past decade or so, the type and mix of scientific application workloads requiring HPC resources have undergone a paradigm shift, moving away from the traditional single program multiple data (SPMD) model to include a set of heterogeneous tasks working towards a common scientific goal [9]. The recent advances in machine learning (ML) and artificial intelligence (AI) technologies have served to motivate their integration into traditional scientific applications, accelerating this move away from traditional message-passing-based SPMD execution models. Heterogeneous workflows, touted to be the new HPC "application" [9], have emerged as a promising approach to deploying scientific applications on HPC platforms. Heterogeneous workflows is an umbrella term, covering the broad spectrum of multi-task execution, from ensemble computing, wherein several instances of the same (parallel) task are launched on a set of HPC resources to workflows where several instances of different types of tasks asynchronously execute with varying degrees of concurrency.

The breadth of HPC workflows spans a variety of scientific domains, including, for example, molecular dynamics, ML applications, material science, climate scienceand drug discovery. Today, workflows executing on HPC platforms orchestrate 100s to 10000s of individual tasks [18]. The tasks can be heterogeneous in terms of the application they execute and the computing resources they are mapped to. As HPC workflow scale continues to grow, managing workflow resources efficiently while simultaneously optimizing for the scientific output or performance of the workflow assumes vital importance. We argue that the ability of the workflow system to

dynamically adapt task execution based on the information about the tasks that have been completed or the state of the hardware resources would represent an important step towards enabling optimal workflow resource management.

Consider, for example, the simplest case of a workload consisting of a set of MPI tasks scheduled to execute on a set of HPC resources. Most workflow systems place the onus of deciding on the resource requirements of the individual tasks on the user. The user supplies this information *a priori* to the workflow system, typically during the task creation step. However, if the task does not scale well, assigning too many or too little resources to each individual task may lead to poor use of the assigned HPC workflow resources. The optimal strategy might well be to run more or less tasks, each at a smaller or larger scale. In another scenario, involving the concurrent coupling of machine learning (ML) tasks to traditional ensemble simulations[22], the allocation of computing resources to the two task types is not always apparent, and a misconfigured allocation can accrue vast performance penalties. In both these scenarios, enabling adaptive decision-making within the workflow system would require the *timely* availability of all the *necessary* performance information.

Several software challenges need to be addressed before the promise of adaptive workflows is realized in full effect. This research work focuses on the challenges involved in enabling robust performance observability of the workflow execution. First, there is the challenge of understanding what data to collect and make available online to the workflow such that observability is enabled. The second challenge pertains to choosing the appropriate data model for monitoring and analysis. Third, different types of data may require different instrumentation strategies and sampling frequencies that must serve the goal of observability while not incurring significant overheads during their measurement and transport. Fourth, there is the related challenge of exporting, storing, and making available the monitoring data online in a timely fashion to enable adaptive decision-making. Lastly, there is the question of how to assign resources to the monitoring system, given that it must run as a part of the workflow. This includes choosing the suitable interaction model between the workflow and the monitoring system.

This paper focuses on the use of SOMA, (Service-based Observability, Monitoring, and Analysis) [48] in a new approach — for HPC workflows. SOMA's ability to monitor and enable adaptive workflow execution is demonstrated through integration with RADICAL-Pilot [33], an HPC pilot-enabled [46] runtime system, integrated with multiple workflow systems including RADICAL-EnTK (Ensemble Toolkit) [8] and Parsl [3, 4], and that can be deployed on exascale platforms [5]. Current projections suggest that scientific workflows for HPC can comprise between 10^6–10^9 tasks [9]. At this scale, performance monitoring is likely to be treated as a first-class citizen of the workflow and also require a non-negligible amount of computing resources [18].

In summary, the key contributions of this work are: (1) The design and implementation of SOMA for service-based workflow monitoring with RADICAL-Pilot (RP) as an exemplar use-case, (2) Experiments demonstrating that SOMA can enable holistic observability of workflow performance, and (3) Experiments demonstrating the costs and benefits of enabling workflow observability through a service-based monitoring architecture.

This paper is organized as follows: Section 2 gives background on the architecture of RADICAL-Pilot and SOMA and explains how they are integrated for workflow monitoring. Section 3 describes the experiments. Section 4 presents the results of those experiments, demonstrating the effectiveness of enabling such monitoring functionality. We discuss related works in section 5, and lastly, we present conclusions and future work in section 6.

2 SOFTWARE ARCHITECTURE

This section discusses the software architecture necessary to monitor heterogeneous HPC workflows, including our existing technologies and what new functionality was required to implement for this study. Section 2.1 describes the architecture of RP and the ensemble environment. Section 2.2 discusses SOMA and the associated service components is given. Section 2.3 presents the novel work on how SOMA is integrated into RP and the special considerations needed to support HPC workflows (as opposed to traditional MPI-based HPC applications).

2.1 RADICAL-Pilot

RADICAL-Pilot (RP) [32] is a Python implementation of the pilot paradigm and architectural pattern [24, 46]. Pilot systems enable users to submit jobs to HPC platforms and then use those resources to execute the application's tasks. Those tasks are directly scheduled via the pilot-system without queueing in the platform's batch system. In that way, it is possible to achieve high-throughput task execution on HPC, avoiding the limits imposed by a centralized, multi-tenant batch queue [46].

Distinctively, RP supports executing heterogeneous executable or function tasks on HPC resources. Both types of tasks can be single/multi-core/GPU/node and MPI/OpenMP. Executable tasks are programs that run as self-contained entities, while function tasks are functions or methods written in a specific programming language. Currently, RP utilizes a dedicated subsystem called RAPTOR [31] to execute Python functions at a very large scale. Uniquely, RP supports the concurrent execution of heterogeneous executable and Python function tasks on up to 193,000 cores and 27,600 GPUs [33].

RP implements two abstractions: Pilot and Task. Pilots are placeholders for computing resources, where resources are represented independently of architecture and platform details. Tasks are units of work specified by a program's executable or a language-specific function/method, alongside resource and execution environment requirements. Fig. 1 depicts RP's architecture with two subsystems (white boxes), each with several components (purple and yellow boxes). Purple components manage pilots and tasks, while yellow components enable communication and coordination. Subsystems can execute locally or remotely, communicating over TCP/IP and enabling multiple deployment scenarios.

Numbers in Fig. 1 show resource acquisition and task execution processes. PilotManager uses PSI/J [13] to queue a pilot job on an HPC platform's batch system (Fig. 1 ①–②). Once scheduled, the job bootstraps RP's Agent and the Agent's Updater notifies RP's Client that tasks can be executed (Fig. 1 ③). Upon notification, the client's TaskManager queues the available tasks onto the client's Scheduler and, after staging files when required, tasks are queued

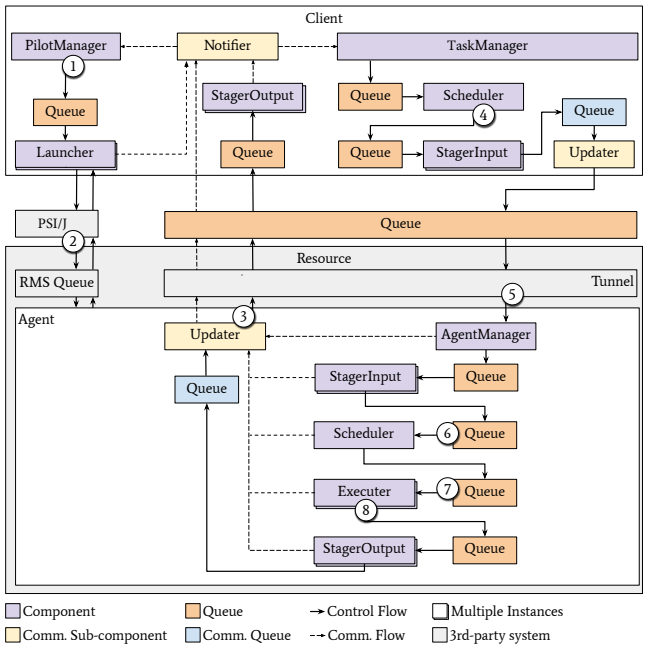

Figure 1: RADICAL-Pilot (RP) architecture has two main components: Client and Agent. The Client may execute on an HPC platform's compute node or remotely, such as on the user's workstation. Agent always executes on a compute node. The Client and Agent subcomponents manage resource acquisition and task execution on those resources. Numbers indicate the RP's execution process.

to the Agent's Scheduler (Fig. 1 ④–⑥). The Agent's scheduler assigns tasks to suitable portions of the available resources and then queues those tasks to an Executor (Fig. 1 ⑦). The Agent's Executor places each task on the assigned resources, sets up their execution environment, and launches each task for execution (Fig. 1 ⑧).

RP is designed and implemented as a building block [43]. In that way, RP can be more easily integrated with software tools independently developed by third-party engineering teams. Integration can utilize RP's public or private application programming interfaces (APIs). For example, RP has been successfully integrated with Parsl [3], Swift [44] and PanDA [30] via its public API, but also with PMIx [42, 45], Flux [36], Hadoop and Spark [23] via its internal API. This paper uses RP's private API to integrate it with SOMA.

2.2 SOMA

SOMA is a service-based framework for monitoring HPC applications [48]. SOMA's service implementation builds upon the Mochi [39] HPC microservice framework. Microservice architectures rely on the principle of composability to scale out and incrementally integrate advanced functionality. Mochi microservices employ high-performance RPC libraries that can use modern HPC network hardware to transport data efficiently and quickly through remote direct memory access (RDMA). By building upon existing state-of-the-art HPC software frameworks, SOMA can take direct and seamless advantage of the improvements made to these frameworks, while cleanly separating functionality and performance. This design choice arguably makes SOMA a more maintainable monitoring service than other *ad hoc* implementations.

HPC workflows require the capture of data from different sources across the workflow and application software stacks to enable observability. Not only are their storage needs different, but data from these sources may need to be monitored appropriately, directly impacting their computational needs within the SOMA service. In response, SOMA takes a leaf out of SERVIZ [37], a workflow-ready visualization service, to enable the partitioning of monitoring service resources into one or more independent "instances", each of which is responsible for monitoring data from one source.

2.2.1 Service Design and Implementation. SOMA's functionality is split up into a *client stub* and a *service library*. The client stub exposes the SOMA monitoring API and is responsible for translating the API calls into remote procedure call or local function calls as appropriate. The client stub can run within the address space of the component being instrumented (application or middleware) and require no additional computational resources to execute. The client stub can also be implemented in a separate binary that does run on additional resources. This is used to collect metrics unrelated to the application, both are demonstrated in this paper.

Typically, the SOMA service executes on a set of dedicated resources *outside* the application or workflow component being monitored. This clean separation between the client and service libraries allows significant flexibility in determining where SOMA's service instances execute while being completely transparent to the calling client. Previous work [48] has explored the benefits of running SOMA's service instances on the unused cores within the application's compute node. However, for HPC workflows, we choose to dedicate a part of the workflow's total computing resources to running the SOMA service instances.

2.2.2 Data Model. SOMA makes use of Conduit [12] to represent the data for monitoring. Conduit was chosen for the generality of its hierarchical approach to describing scientific data for transport between different software components within an application. Conduit has found adoption within the HPC community for integrating visualization capabilities into scientific HPC applications [16, 21]. In particular, this hierarchical data model is used for workflow monitoring, enabling the division of different monitoring data into their separate logical namespaces that can be combined during analysis. Examples of Conduit data models are shown in section 2.3.2.

2.3 Workflow Monitoring Using SOMA

RP introduced the concept of services in its latest API implementation and SOMA is treated as a first-class citizen within RP. A first-class citizen means that a SOMA task is able to be scheduled and run as any other application task would. This helps support SOMA client binaries that run outside of the application namespaces for collecting different metrics. This section enlists the special considerations required to integrate RP and SOMA and the methodologies used to capture various types of monitoring data from across the workflow and application software stacks.

2.3.1 Service Support Inside RADICAL-Pilot. Integrating SOMA and RP required two main capabilities: (1) scheduling and launching SOMA components on dedicated and shared resources and (2) enabling data exchange between SOMA and RP. SOMA can use a set of dedicated resources to run its service instances. The SOMA service is treated as a service task within RP. While the service task can specify its resource requirements like any other regular RP application task, the SOMA service task needs to be scheduled *before* any application tasks. Recall that this stems from the need for the SOMA service instances to make their RPC addresses publicly known within the workflow for clients to connect. RP enables such capability by scheduling the service tasks immediately after bootstrapping its Agent component but before any other task. Service tasks communicate their state to RP for the consumers of those services to know where, when, and whether they are available. RP's Agent components (see Fig. 1) exchange data via queues implemented with ZeroMQ [14]. Each component gets its inputs via a queue and pushes its output to another component's queue. That enables RP to integrate third-party components via well-defined interfaces and a unified coordination infrastructure.

Service tasks are also special concerning their scope. While regular application tasks execute and go out of scope, thereby releasing their computing resources, service tasks are long-running, i.e., execute for the entire workflow duration. Once the workflow is completed, service tasks are shut down through an appropriate control command from RP. Fig. 2 depicts the timeline of events and the RP-SOMA interaction model during workflow execution.

Initially, RP is scheduled via the HPC platform's batch system as a pilot job [46] (Fig. 2 ①). That allows us to execute RP Client that, in turn, executes RP Agent on one or more compute nodes (Fig. 2, solid arrows). In this way, we avoid consuming resources on the cluster's login node, in accordance to the HPC platform usage policies. Once the RP Agent bootstraps (Fig. 2 ②), it first schedules and launches the SOMA service (Fig. 2 ③), then it schedules the RP monitoring task, one for the entire workflow, co-located with the service (Fig. 2 ④). Next, RP schedules the hardware monitoring tasks, one on each available compute node (Fig. 2 ⑤). Both monitoring tasks run a SOMA client communicating with the SOMA server via RPC (Fig. 2, dotted arrows). Finally, once the monitoring infrastructure bootstrap is completed, RP proceeds to schedule the task of the workflow application Fig. 2 ⑥. Note that each application task can also run a SOMA client to enable the SOMA service to receive asynchronous application information. SOMA Clients can be launched and stopped via RP's task pre/post execution capabilities and/or the task executable can be wrapped in a script that launches the SOMA client.

2.3.2 Capturing Monitoring Data. SOMA's ability to split its service task resources between several instances and the concept of logical namespaces is crucial in workflow monitoring. Monitoring data is divided into four namespaces — workflow, hardware, performance, and application. The total of N SOMA service processes within the service task is divided appropriately among four instances, each supporting the compute and storage needs of one namespace. For this work specifically, two new namespaces were introduced — workflow and hardware — the performance namespace was improved on from previous work, and the application

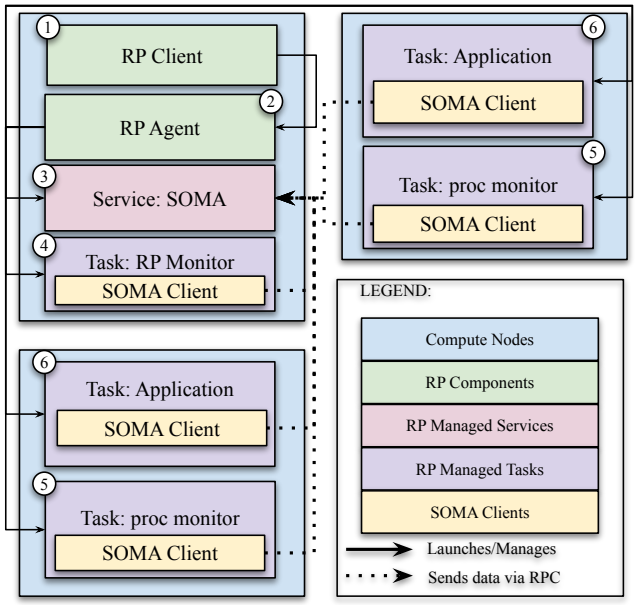

Figure 2: Example layout of how RP and SOMA components can be arranged and interact on the compute nodes of a pilot job. RP Client and Agent schedule and manage services and tasks on the available resources. A SOMA server is launched as a service, and each monitoring and application task has a SOMA client per rank. Monitoring tasks collect local node hardware and workflow profile data and send them to the SOMA server via a SOMA client.

namespace is not used. The methodology for capturing data for each of these namespaces is as follows.

Workflow Namespace: For the first of the two new namespace implementations we created a model for RP's workflow states. RP's components function as a state machine — the lifecycle of each component, including application tasks, proceeds through a set of predictable states through the execution. For example, a task proceeds through the NEW, SCHEDULED, EXECUTING, and DONE/FAILED states. The transitions between states are further broken down by timestamped events that indicate a new state, shown in Listing 1 are the events within the EXECUTING state. Likewise, the Pilot component and the RP Agents have appropriate state transitions. Snap-shotting these state transitions and collecting statistics on the total number of pending tasks, completed tasks, and so on can provide valuable insight into the overall performance of the workflow, to be used for subsequent analysis (online or offline). SOMA captures workflow-level information through a service client task launched on a single compute node within the workflow. This client task launched as a daemon, periodically reads the appropriate profile files generated by RP, summarizes basic statistics about the workflow from this data, and publishes the same to the SOMA service processes (using RPC) at a configurable monitoring frequency. This client task runs for the duration of the workflow. See Listing 1 for an example of the Conduit data model implemented to represent this workflow data.

```
RP: /* Top-level SOMA namespace */
    task.000000: /* Task name tag */
        /* Timestamp:       "event" */
        1698435412.6060030: "launch_start"
        1698435412.9642950: "exec_start"
        1698435412.9717330: "rank_start"
        1698435427.9775150: "rank_stop"
        1698435427.9850750: "exec_stop"
        1698435428.0583980: "launch_stop"
    ...
```

Listing 1: Conduit::Node of the RP workflow data model

Hardware Namespace: For the second of the two new namespaces we created a data model for certain hardware utilization metrics. Basic information about the state of the hardware, gathered periodically by reading `/proc/` is captured by SOMA client tasks, which can be scheduled on reserved cores on each compute node within the workflow. RP launches these client tasks before application tasks are scheduled and run for the duration of the workflow. Granted, such information can be captured directly by the application task in an alternative design. However, there is the issue of task scope. Application tasks come and go, but the hardware information is valid throughout the workflow execution. Therefore, we captured this information through long-running special SOMA client tasks. See Listing 2 for an example of the Conduit data model implemented to represent some `/proc/` data.

```
PROC: /* Top-level SOMA namespace */
    cn4302: /* Hostname tag */
        3824813742052238: /* Timestamp */
            Uptime: 49902
            Num Processes: 3
            Available RAM: 8422
            stat:
                cpu: 10749 865 685 9293 999 745
                cpu0: 4698 591 262 8953 612 449
                ...
```

Listing 2: Conduit::Node of the hardware data model

Performance Namespace: Traditional sources of performance information, such as MPI counters and application profiles, are captured by integrating the TAU [26] performance system with the application. We have developed a TAU plugin that captures the measurements enabled by the execution and converts the TAU profiles internally to a Conduit::Node through SOMA. While the plugin runs in the application's address space, it creates a separate client object and connects to the SOMA instances reserved for monitoring the performance namespace data. Updates to the Conduit::Node data model for TAU include adding the hostname tag (compute node name) as well as a task identifier in the filename. These additions allow for the properly attributing the TAU profile to the correct heterogeneous workflow tasks. This was not needed or considered previously as the TAU plugin was only being used for a single application, not a heterogeneous workflow.

Application Namespace: In addition to capturing performance data from traditional sources, the application may have useful custom information to be monitored, i.e., the *scientific rate-of-progress or figure-of-merit* self-reported by the application. For example, a molecular dynamics code might want to capture the atom-timesteps per second as the figure of merit. Unlike the other three namespaces, capturing this data typically requires application instrumentation with SOMA's API and is not the focus of this paper.

3 EXPERIMENTS

The primary function of the SOMA monitoring service is to enable *online access to performance data*. In order to demonstrate that such functionality is indeed useful in the context of HPC workflows, we designed three main thrusts of experiments: (1) the observation of the ensemble performance state at different granularity described in section 3.1; and (2) using performance data to explore potential changes to the workflow or application configuration, as well as (3) an overhead analysis, both in section 3.2. All experiments were run on the Summit supercomputer, its architecture details are specified in further detail online [35].

For both the OpenFOAM [11, 41] experiments in Section 3.1, and DeepDriveMD mini-app experiments [19] described in Section 3.2, we adapted existing RADICAL-Pilot workflow scripts by incorporating SOMA monitoring functionality. Any existing RADICAL-Pilot workflow could make use of SOMA monitoring in a similar manner. In addition, the run scripts can serve as examples for any workflow to use RADICAL-Pilot with SOMA monitoring. For workflows that are not managed with RADICAL-Pilot, SOMA could still be used to measure all except the workflow namespace.

The most straightforward use case is one that allows a user to understand useful metrics about the state of the workflow and make an observation about entities. SOMA allows users to observe the following metrics: hardware utilization across the pilot job, runtime performance of the workflow (current and average task throughput), performance statistics across tasks (distribution of load imbalance, task performance variation, etc), and monitoring the task state transitions. We show that SOMA enables measuring each of those metrics from different sources: hardware metrics, task-level TAU performance data TAU, and RP event-based states.

Experiment	Tuning	Overload
Number of Tasks	4	80
Number of Nodes	10	10
Number of MPI Ranks	20, 41, 82, 164	
Monitors	proc, rp, tau	
SOMA Ranks Per Namespace	1	

Table 1: OpenFOAM Experiment Summary

3.1 OpenFOAM Workflow

We run the existing real-scale workflow from the Exascale Additive Manufacturing (ExaAM) project [10], part of the Exascale Computing Project. The ExaAM project has developed a suite of exascale-ready computational tools to model the process-to-structure-to-properties (PSP) relationship for additive manufactured (AM) metal components. The target workflow contains simulations for the melt pool physics and uses AdditiveFOAM [11], an extension of OpenFOAM [41] for AM processes.

We created a corresponding workflow using RADICAL-Pilot [5] and will refer to it as the OpenFOAM workflow. We run this workflow on Summit, monitoring it with our SOMA service. Each compute node of Summit has 44 physical cores, two of which are reserved for the system, leaving 42 available to the user. We run four

different task configurations within the OpenFOAM workflow, with either 20, 41, 82 or 164 MPI ranks per task, and with each rank using one physical core. We turn hardware multi-threading off, and reserve one core per node for the SOMA hardware monitoring client. Thus, each task utilizes between 0.5 compute node to 4 compute nodes, respectively. Initially, we run one instance of each task configuration across a workflow with 4 nodes, we refer to this as the "tuning" run, and then we run 20 instances of each across 10 nodes, which we refer to as the "overloaded" run, see Table1. In both cases we allocate one extra node (for 5, and 11 total) that is reserved for running the RADICAL-Pilot Agent and SOMA service.

Monitoring Setup: In these experiments we use the following three SOMA clients to observe the workflow and collect metrics. The first is workflow task information via our RP monitoring client for SOMA, launched with one client per workflow. At configurable intervals, the SOMA client gathers RP-managed task status information, calculates the time spent in each state, and sends it via RPC to the SOMA service. This client setup is shown in Figure 2 by the pink square ③. We schedule the RP monitoring client onto the same node as the RP client and agent, sharing the node resources. This is an additional node that is not used for any simulation-related tasks, but only by RP and its associated SOMA monitoring client. The second source of metrics is from the hardware monitoring client shown in Figure 2 by the squares ④. At configurable intervals, that client reads memory and CPU usage data from `/proc/`. This monitoring client runs on one physical core on each node.

The third and final data source for these OpenFOAM experiments is the TAU [26] performance profiling plugin for SOMA, which samples the running application to gather performance data, e.g., time spent in each function or MPI metrics. In this case, the TAU plugin is executed without any need for instrumentation of the OpenFOAM code. The plugin uses the `tau_exec` functionality to sample the code and publishes the sampled performance profiles to the SOMA server. This isn't specifically denoted in Fig. 2, but is similar to the squares ⑤, where a SOMA client is created within the same task space as the application. However, from OpenFOAM's perspective, it is just being profiled by TAU, and all SOMA functionality is encapsulated by the TAU plugin.

3.2 DeepDriveMD Miniapp Workflow

With the increase in heterogeneous machine learning workflows as an acceleration technique for domain science, the ability to monitor and improve such a workflow with SOMA is important. The DeepDriveMD workflow mini-apps [19] models the computational patterns and behaviors of a Deep Learning oriented Moleculer Dynamics simulation workflow using DeepDriveMD (deep-learning driven molecular dynamics) [22].

The DeepDriveMD workflow mini-apps consists of phases, each phase made up of four stages. The stages are (1) Simulation, (2) ML Training, (3) Model Selection, and (4) Agent (Inference), which must be run in order. The baseline workflow uses 12 simulation tasks and 1 task each for training, selection, and agent. We do not modify the number of tasks except for the training tasks as explicitly stated in Table 2. The simulation, training, and agent stages use both CPU cores and one GPU resource per task, whereas the selection stage runs only on the CPU. This is configured and managed by the RADICAL-EnTK (Ensemble Toolkit), which is a higher-level abstraction of RADICAL-Pilot functionality [5]. We use EnTK to schedule n number of phases in a row, within m number of concurrent pipelines, this can be done for any combination of $n \times m$, see Fig. 3. We run three different experiment setups with the DeepDriveMD workflow mini-apps, summarized in Table 2.

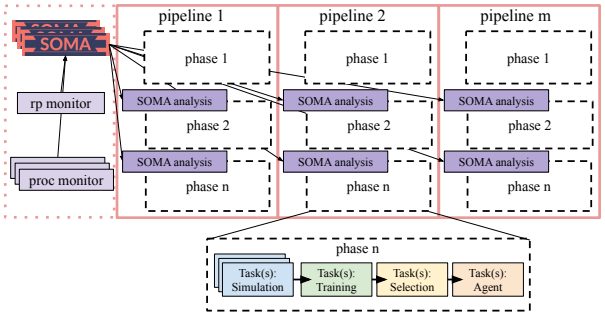

Figure 3: An illustration of how RADICAL-EnTK can launch SOMA monitoring tasks, one or more pipelines, and one or more phases of the DeepDriveMD mini-app workflow. Each phase is one full DDMD mini-app workflow comprised of multiple tasks, shown in the inset.

The first experiment is a tuning study, where we alter the number of cores allocated per task. We run $n = 6$ phases, and $m = 1$ pipelines. We can compare each phase of the workflow for the effects the core assignment has on the runtime of each stage of the workflow. Details of the configurations for this experiment can be found in Table 2 under the "Tuning" column.

The second DeepDriveMD mini-app experiment involves conducting SOMA analysis to identify free resources during runtime. While EnTK cannot make adaptive changes during runtime, we can use SOMA to learn more about how to configure the workflow during each successive phase. We run $n = 4$ phases, and $m = 1$ pipelines. The setups that were run during the second experiment can be found in table 2 under the "Adaptive" column. The number of training tasks listed in the Table are set *a priori*, as in the original workflow, but we conduct online SOMA analysis to be available between phases. Fig. 3 depicts where the SOMA analysis fits within the flow of the adaptive experiments.

The third experiment set for the DeepDriveMD mini-app was scaling it up to more compute nodes to ensure SOMA monitoring could keep pace. In Scaling A and B we run $n = 1$ phases for $m = a$ pipelines of the workflow, where a is the number of application nodes. This creates an over-subscription during the simulation stage, where each simulation stage (per pipeline) requires 12 GPUs, but there are only 6 available per node. In Scaling A we vary the ratio of SOMA server ranks to pipelines, and in Scaling B we keep the ratio steady, see Table 2.

Monitoring Setup: In the DeepDriveMD Miniapp experiments, we implemented data collection from two sources. Similar to the OpenFOAM workflow, we initialized one RADICAL-Pilot monitoring client per workflow, scheduled onto the same node as the RADICAL-Pilot agent. Also similarly, we initialize one `/proc/` monitoring client per compute node for collection of CPU usage data,

Experiment	Phases (n)	Pipelines (m)	Application Nodes	SOMA Nodes	Cores / Sim Task	Num Train Tasks	Cores / Train Task	SOMA Ranks / Namespace	Monitoring Freq. (s)
Tuning	6	1	2	1	1,3,7	1	1,3,7	1	60
Adaptive	4	1	2	1	6	1,2,4,6	1	1	60
Scaling A	1	64	64	1,2,4	3	1	7	16,32,64	60
Scaling B	1	64,128,256,512	64,128,256,512	4,7,13,25	3	1	7	64,128,256,512	60,10

Table 2: DeepDriveMD Mini-app Experiment Summary

which takes a reading every minute, and calculates the current CPU utilization online. We did not use the TAU monitoring client in the DeepDriveMD Miniapp experiments. In most experiments, we sampled and published the performance data every 60 seconds. In Scaling B, to push the limits of SOMA, we increased the frequency to every 10 seconds. We also run baseline workflows with no SOMA nodes or monitoring for comparison.

4 RESULTS

The three data sources: RP task information, hardware (proc), and TAU performance profiles, give us a rich understanding of the online performance of the OpenFOAM workflow. In sections 4.1, and 4.2 we discuss the OpenFOAM results at two levels of granularity, local to each task and global to the entire workflow.

For the DeepDriveMD mini-app experiments we use the collected data from the hardware (proc) and rp monitors. In section 4.3 we first discuss the results from the tuning and adaptive workflows. We then look at the scaling workflows and illustrate our discoveries about the overhead and effects of SOMA monitoring at scale.

4.1 OpenFOAM Task Scaling Analysis

Fig. 4 shows the results of strong-scaling the OpenFOAM tasks from one to four nodes. Without any *a priori* knowledge of the application execution time we ran multiple configurations. Since we capture performance from 20 different instances of each configuration, we see a variation in the total execution time, but are able to calculate averages. An interesting observation that can be made here is that there is limited benefit to scaling the OpenFOAM tasks beyond two nodes. This information can be used to inform RADICAL-Pilot which MPI task configurations to use, and is fundamental in enabling runtime adaptivity. RP could collect information about MPI task performance, and utilize that information to change the task description, adjusting the number of ranks of each type of task in the workflow. As shown by our experiments, that would allow to utilize the available resources better, thus reducing the total time to completion of the entire workflow. Additionally, from monitoring with the TAU SOMA integration we can also gather the related MPI metrics that correspond for further analysis. We zoom in on one instance of a task for clarity in Fig. 5. We observe that a large portion of time for each rank is spent in `MPI_Recv()` and `MPI_Waitall()`. While this may not be as easy for on-the-fly adaption, it can be used for further simulation tuning.

As with the number of ranks per task, RP could utilize the information about the physical location of the MPI ranks across nodes to make adaptive scheduling decisions. Fig. 6 shows our results from comparing the execution times when the physical location of the MPI ranks can differ. In the case of 20, and 41 MPI ranks —

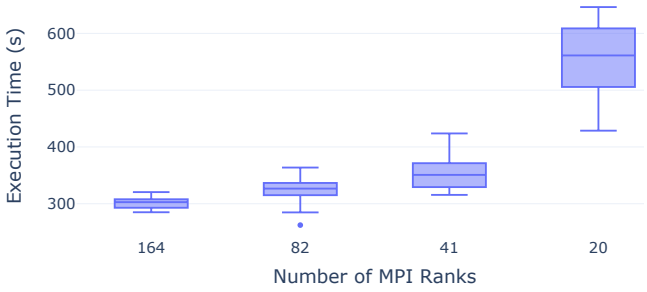

Figure 4: A scaling study of different OpenFOAM configurations. We vary the number of MPI ranks and run 20 instances of each configuration in one RP managed workflow.

where all ranks could fit on one compute node — RP has the option to schedule all of these on one compute node, or spread the ranks out across any available CPU cores of the allotted nodes. Fig. 6 shows a distribution of 20 and 41 rank tasks running on different node distributions based on what was available during the overloaded run. Along the x-axis we see whether each task was run on a single node, or split across up to 5 nodes. For the 20 rank runs we actually see an execution time improvement as the ranks are spread across more nodes. This is possibly due to the fact that the smaller rank runs were typically scheduled later in the workflow, when resources are less utilized. This can be seen in Fig. 8 where non-green sections indicate unused resources. However, in the 41 rank case our performance improvement is not quite as remarkable as we may be approaching an upper bound.

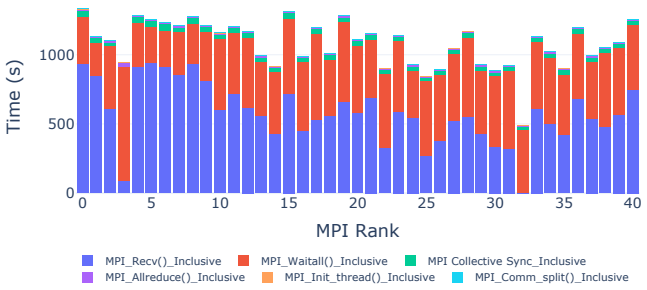

Figure 5: From the TAU SOMA plugin we have access to TAU profile information such as the time spent in MPI calls here to observe load balancing.

4.2 OpenFOAM Workflow Analysis

Observation of the workflow from a global perspective offers insights into areas where scheduling of future tasks could be improved. Results from our overloaded scenario — with twenty instances of each task configuration — shown in Fig. 8. In Fig. 8 we can see the resource utilization from the perspective of RP. It marks resources (cores in this case) as either available or unavailable, but does not look at percent utilized. The green color on the graph indicates a resource has a task running on it, and the purple sections show when it is scheduling a new task to be run on that resource. The resources are well used, but using this data in combination with the CPU utilization could lead to even more informed decisions. Currently, RP schedules a task as soon as there are enough free resources, i.e., CPU cores and/or GPUs. Based on the online information about overall CPU (or GPU) utilization, RP could adapt its scheduling decisions, prioritizing the use of the free CPUs on a node with comparably lower overall CPU utilization. The benefits of this approach are better appreciated with a run with a lower overall resource utilization.

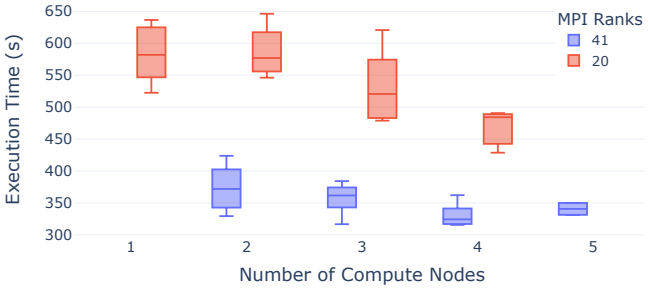

Figure 6: Comparison of the execution time of OpenFOAM tasks when configured with 20 or 41 ranks and scheduled onto a different number of compute nodes.

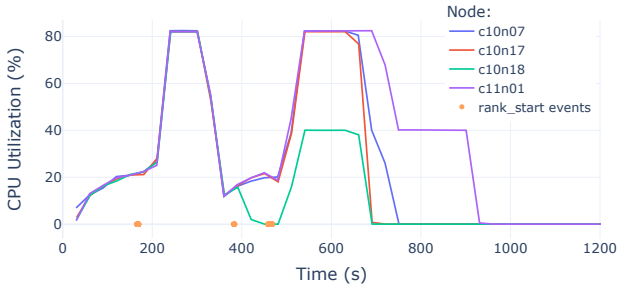

Figure 7: CPU utilization for an OpenFOAM Tuning workflow. Each colored line shows the CPU utilization on a different compute node, measured every 30 seconds by the SOMA hardware monitoring client. The orange dots indicate when the SOMA RP monitor observed from RP that a task is starting.

Figs. 7 and 8(bottom) provide a simplified version of the results above where we launched only one instance of each task type. With

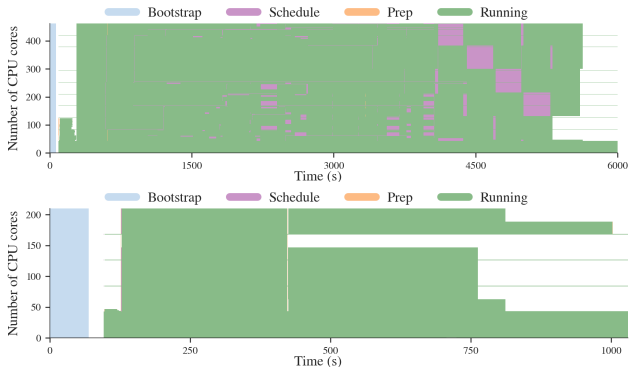

Figure 8: RP resource utilization for the OpenFOAM workflows. Top: Overload workflow. Bottom: Tuning workflow. Light blue: RP bootstrap time. Purple: RP task scheduling time. Green: task running time. White space: unused resources, a measure of RP scheduling optimization based on information provided by SOMA.

fewer nodes and tasks it is easier for a human to observe the specific behavior. We can see on the CPU utilization graph (Fig. 7 that as a rank starts, there is a corresponding spike in CPU utilization. However, we can clearly see an imbalance in the utilization on each node in roughly the latter half of the runtime, which offers room for better scheduling decisions on a second go-around. As in Fig. 8(top), in Fig. 8(bottom) green represents resource utilization, and white space denotes free cores, or resources that could potentially be utilized. In this case, once the 164 rank task completed using all of the cores, the other tasks were scheduled to use the cores simultaneously.

4.3 DeepDriveMD Mini-app Analysis

The tuning experiment results — where we have 6 phases of the workflow and change the number of cores per simulation and training task — are in Fig. 9. We see that even when changing the number of cores that can be used per task, CPU utilization remains low. This is due to the fact that most of the work for the two longest stages, simulation and training, is done on the GPU. Therefore, learning that the effect of using fewer CPU cores per task was minimal, we next explored parallelizing the training tasks in order to use more GPUs per node. To parallelize the training tasks appropriately, we also resized the data and added additional `MPI_Reduce` calls to the tasks. While the full-scale DeepDriveMD workflow does not yet support such parallelization, we determined this a good use of the mini-app, to model potential changes without having to implement it in the full-scale workflow unless deemed worthwhile. SOMA can calculate the current CPU utilization online, but the integration to utilize that feedback with RP is still in progress.

Fig. 10 depicts the pipeline execution times from the Scaling A experiment where we increased the number of pipelines to SOMA ranks from 1:1 to 8:1 to pinpoint any significant bottlenecks. Because we allocated extra nodes for SOMA and did not necessarily need to use every single core on those nodes for SOMA ranks we ran in two configurations. The *shared* configuration allowed

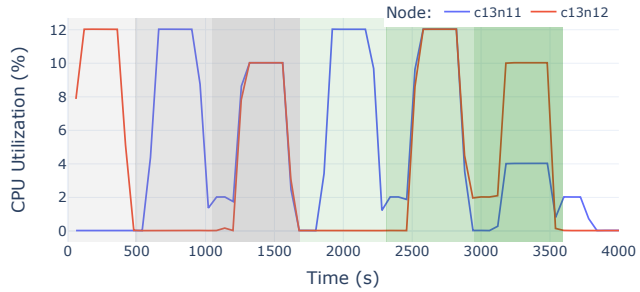

Figure 9: CPU utilization for the DeepDriveMD mini-app Tuning experiment shows. Gray background shows 7 cores per training task, and green shows 3 cores. Shading changes from light to dark for 1, 3, and 7 cores per simulation task.

RADICAL-Pilot to make use of any free cores on the SOMA nodes. The *exclusive* configuration reserved the nodes only for SOMA ranks and did not allow RADICAL-Pilot to schedule any application tasks on any available cores. Because of the oversubscription of simulation tasks in the workflow, each pipeline requires 12 GPUs during the simulation stage but there are only 6 per node. In the *shared* cases, RADICAL-Pilot was able to make use of the some of the free GPUs and cores on the SOMA nodes as SOMA runs on CPU-only and did not completely fill the allocated SOMA nodes. This caused more variance in overall execution times but reduced the execution time for many of the pipelines.

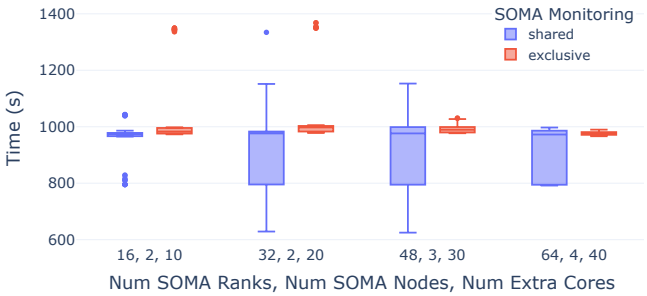

Figure 10: Runtimes for 64 pipelines of the DDMD Mini-app Scaling A experiment workflow. The oversubscription of GPUs causes more variability in scheduling and execution times in the shared configuration, but the ratio of SOMA ranks to pipelines does not have much effect.

Fig. 11 demonstrates the distribution of pipeline execution times when we kept the ratio of SOMA ranks to pipelines at 1:1 and scaled up SOMA ranks and nodes, and application pipelines and nodes. Details of our configurations are given by the Scaling B column in table 2. We again make use of the flexible integration between RADICAL-Pilot and SOMA to run in both the *shared* and *exclusive* configurations described previously. We compare these with the baseline *none* configuration which was only *m* DDMDmini-app workflow pipelines running on *m* applications nodes with no SOMA monitoring or SOMA nodes. Again, we can actually see potential benefit from running with extra SOMA nodes in the *shared* configuration as RADICAL-Pilot's opportunistic scheduling system can make use of the available cores. The higher outliers in the *shared* configurations are due to the fact that RADICAL-Pilot is non-deterministic in scheduling and may make an inefficient placement during runtime that delays one or more pipelines.

Furthermore, when pushing SOMA to a higher monitoring frequency — every 10 seconds, up from every 60 seconds — we do start to see increased overhead costs. This is shown in Fig. 11 under the *frequent-exclusive* label. In these results, the *frequent-exclusive* experiments are also exclusive in configuration, i.e., the extra cores on the SOMA nodes cannot be used by the application tasks. This gives us a better direct comparison to the *exclusive* execution times because only the monitoring frequency changes. Some of this overhead can be mitigated when we run in the *frequent-shared* configuration. This allows for RADICAL-Pilot to utilize any extra cores during scheduling, thus increasing the variability, but reducing some execution times. When comparing the worst performance, *frequent-exclusive*, with the baseline we see approximately 1.4, 3.4, 3.2, and 4.6 percent runtime overhead for 64, 128, 256, and 512 nodes respectively. In the *shared* configuration, we actually see a reduction in runtime for 64, 128, and 256 nodes (6.5, 3.8, 1.1 percent, respectively) and an overhead cost of about 1.8 percent at 512 nodes.

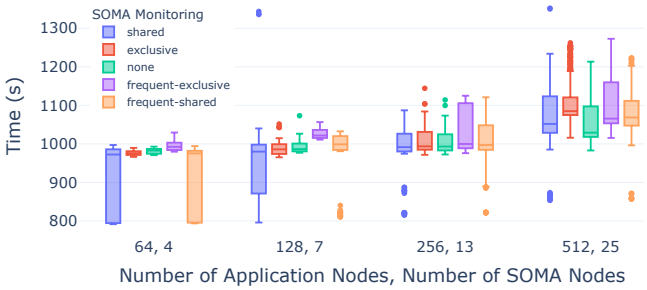

Figure 11: Distribution of runtimes for each concurrent pipeline of the DDMD Mini-app Scaling B experiment workflow. Shared indicates RADICAL-Pilot can schedule application tasks on the free cores on the nodes reserved to SOMA. Exclusive means that nodes were reserved for SOMA only. In the "none" case, the number of SOMA nodes is zero, and we run the pipelines without any extra nodes. Frequent indicates the increased publication rate. (Note the x-axis application nodes is on a log scale).

5 RELATED WORK

The monitoring of HPC applications has historically been limited to "system-level" monitoring — typically available to HPC system administrators to monitor the health of the cluster and view job-level statistics and hardware usage over time. Examples of such monitoring services include LDMS [1], Ganglia [28], and XDMoD [40]. ZeroSum[15] is special in this regard, designed to operate in between the application and system layer and focused on optimizing the environment configuration of the HPC application. Typically, such services operate in the background (deployed as daemons on

the compute nodes) and lie outside the scope of the HPC application — both in terms of online, real-time access to the monitoring data and the configuration of the monitoring service itself. Recent work that has made use of the LDMS monitoring data by employing active learning-based frameworks to diagnose performance anomalies during runtime includes [2], and they focus on reducing the amount of labeled data required for accurate diagnostics. We envision that work in this vein could be a future integration with the SOMA framework, as a consumer of the performance metrics in order to improve online decision-making, especially at scale.

As HPC workloads have evolved to include complex heterogeneous workflows [9, 18], the need for online monitoring has grown in importance. Prior work for HPC workflow monitoring has enabled user-based workflow steering [29], integration of machine-level data into workflow monitoring [40], integration of ML-based techniques into distributed workflows for minimizing resource wastage [7], and specialized workflow monitoring systems to detect execution anomalies [27]. OSU INAM is a tool that provides online introspection into application performance through a visualization dashboard, focusing on monitoring Infiniband network traffic and MPI communication between nodes [20]. Some of these metrics overlap with what we can collect from the TAU plugin, i.e. MPI message size, but they include more network specific metrics whereas we prioritize hardware and workflow states in this work.

There have also been proposed solutions for online performance analysis for individual applications, such as TAUoverMRNet [25], TAUoverSupermon [34], MONA [47], and Score-P [17]. The idea of combining application metrics with current hardware and system activity is useful in that it presents a much more robust picture, and may help account for perforomance variations or anomalies. However, with the focus of these studies on individual applications, it remains to be seen if they are capable or effective in monitoring a more complex workflow, i.e., if they can measure and analyze any metrics specific to workflow performance.

Bader et al. [6] present an architectural blueprint for categorizing monitoring data from the HPC workflow and application stacks, closely resembling the concept of namespaces presented in this work. SYMBIOMON [38] introduces a service-based monitoring infrastructure for coupled HPC applications. SOMA represents a generalized design and data model for service-based monitoring of heterogeneous HPC workflows. In particular, SOMA's implementation and architecture are specially geared towards *real-time online monitoring* to enable adaptive execution of the workflow.

6 CONCLUSION AND FUTURE WORK

The increased complexity of HPC application workflows versus traditional simulation codes necessitate a different approach to performance measurement and analysis, one that observes the dynamic task execution, workflow scheduling, and resource allocation actions throughout the application's lifetime. Performance objectives of turnaround time are expanded to include makespan and utilization, especially in large many-task scenarios where resource management, critical paths, and scheduling efficiency are paramount. It is no longer reasonable to rely on a post-mortem performance analysis and tuning strategy because the computation environment for workflow execution can not be assumed to be constant and the same as prior runs. A performance monitoring system is needed to capture real-time tasks and system performance data together with workflow operation information and ideally provide in situ processing for runtime decision actuation.

The paper describes how the SOMA monitoring framework is integrated with a HPC many-task workflow system based on RADICAL-Pilot (RP). In this environment, we developed multiple observability sources that connect to SOMA and provide information about RP workflow states, resource usage, system performance, and application tasks. Different interfaces to access the information leverage robust measurement tools, such as TAU for granular task-level performance metrics. Once in SOMA's possession, the data gathered can be processed and analyzed online. The analysis of this data can inform decisions for choosing better configurations of the workflow in future runs.

The experiments we performed and presented are intended to highlight functional capabilities of SOMA and to convey the possibilities for more scalable and sophisticated translation of our research work. The results in section 4 demonstrate a solid foundation for realtime monitoring and the opportunity for in situ analysis and feedback to RP to guide dynamic scheduling and resource assignment. In future work, we plan to extend SOMA's support to develop adaptive workflows in RADICAL-Pilot. The idea is to analyze performance metrics together with scientific progress measures to make smart scheduling and configuration decisions, including the altering of the workflow configuration on-the-fly.

REFERENCES

[1] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. 2014. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 154–165.

[2] Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K. Coskun. 2024. Runtime Performance Anomaly Diagnosis in Production HPC Systems Using Active Learning. *IEEE Transactions on Parallel and Distributed Systems* (2024), 1–14. https://doi.org/10.1109/TPDS.2024.3365462

[3] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. 2022. RADICAL-pilot and Parsl: Executing heterogeneous workflows on HPC platforms. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 27–34.

[4] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin Wozniak, Ian Foster, Mike Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. https://doi.org/10.1145/3307681.3325400

[5] Jonathan Bader, Jim Belak, Matthew Bement, Matthew Berry, Robert Carson, Daniela Cassol, Stephen Chan, John Coleman, Kastan Day, et al. 2023. Novel Approaches Toward Scalable Composable Workflows in Hyper-Heterogeneous Computing Environments. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, 2097–2108. https://doi.org/10.1145/3624062.3626283

[6] Jonathan Bader, Joel Witzke, Soeren Becker, Ansgar Lößer, Fabian Lehmann, Leon Doehler, Anh Duc Vu, and Odej Kao. 2022. Towards Advanced Monitoring for Scientific Workflows. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2709–2715.

[7] Jonathan Bader, Nicolas Zunker, Soeren Becker, and Odej Kao. 2022. Leveraging Reinforcement Learning for Task Resource Allocation in Scientific Workflows. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 3714–3719.

[8] Vivekanandan Balasubramanian, Antons Treikalis, Ole Weidner, and Shantenu Jha. 2016. Ensemble Toolkit: Scalable and Flexible Execution of Ensembles of Tasks. In *2016 45th International Conference on Parallel Processing (ICPP)*, Vol. 00. 458–463. https://doi.org/10.1109/ICPP.2016.59

[9] Tal Ben-Nun, Todd Gamblin, Daisy S Hollman, Hari Krishnan, and Chris J Newburn. 2020. Workflows are the new applications: Challenges in performance, portability, and productivity. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 57–69.

[10] Robert Carson, Matt Rolchigo, John Coleman, Mikhail Titov, Jim Belak, and Matt Bement. 2023. Uncertainty Quantification of Metal Additive Manufacturing Processing Conditions Through the Use of Exascale Computing. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)*. Association for Computing Machinery, 380–383. https://doi.org/10.1145/3624062.3624103

[11] John Coleman, Kellis Kincaid, Gerald L. Knapp, Benjamin Stump, and Alexander J. Plotkowski. 2023. ORNL/AdditiveFOAM: Release 1.0. https://doi.org/10.5281/zenodo.8034098

[12] Cyrus Harrison, Matthew Larsen, Brian S. Ryujin, Adam J. Kunen, Arlie G. Capps, and Justin Privitera. 2022. Conduit: A Successful Strategy for Describing and Sharing Data In Situ. *2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2022), 1–6.

[13] Mihael Hategan-Marandiuc, Andre Merzky, Nicholson Collier, Ketan Maheshwari, Jonathan Ozik, Matteo Turilli, Andreas Wilke, Justin M Wozniak, Kyle Chard, Ian Foster, et al. 2023. PSI/J: A Portable Interface for Submitting, Monitoring, and Managing Jobs. In *IEEE International Conference on eScience*. IEEE.

[14] Pieter Hintjens. 2013. ZeroMQ: Messaging for Many Applications. https://api.semanticscholar.org/CorpusID:190148604

[15] Kevin Huck and Allen Malony. 2023. ZeroSum: User Space Monitoring of Resource Utilization and Contention on Heterogeneous HPC Systems. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, New York, NY, USA, 685–695. https://doi.org/10.1145/3624062.3624145

[16] Seif Ibrahim, Thomas Stitt, Matthew Larsen, and Cyrus Harrison. 2019. Interactive in situ visualization and analysis using Ascent and Jupyter. *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2019). https://api.semanticscholar.org/CorpusID:211040657

[17] Thomas Ilsche, Mario Bielert, and Christian von Elm. 2022. Bridging the Gap between Application Performance Analysis and System Monitoring. *2022 IEEE International Conference on Cluster Computing (CLUSTER)* (2022), 611–615. https://api.semanticscholar.org/CorpusID:252997742

[18] Shantenu Jha and Allen D. Malony. 2021. Dynamic and Adaptive Monitoring and Analysis for Many-task Ensemble Computing. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 637–641. https://api.semanticscholar.org/CorpusID:238751313

[19] Ozgur Ozan Kilic, Tianle Wang, Matteo Turilli, Mikhail Titov, Andre Merzky, Line Pouchard, and Shantenu Jha. 2024. Workflow Mini-Apps: Portable, Scalable, Tunable & Faithful Representations of Scientific Workflows. arXiv:2403.18073 [cs.DC]

[20] Pouya Kousha, D. KamalRajS., Hari Subramoni, Dhabaleswar Kumar Panda, Heechang Na, Trey Dockendorf, and Karen A. Tomko. 2020. Accelerated Real-time Network Monitoring and Profiling at Scale using OSU INAM. *Practice and Experience in Advanced Research Computing* (2020). https://api.semanticscholar.org/CorpusID:220666832

[21] Matthew Larsen, James Paul Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization* (2017). https://api.semanticscholar.org/CorpusID:3561850

[22] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. 2019. DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 12–19.

[23] Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, and Shantenu Jha. 2016. Hadoop on HPC: Integrating Hadoop and pilot-based dynamic resource management. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1607–1616.

[24] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. 2012. P: a Model of Pilot-abstractions. In *2012 IEEE 8th International Conference on E-Science*. IEEE, 1–10.

[25] Allen D. Malony, Srinivasan Ramesh, Kevin A. Huck, Chad Wood, and Sameer S. Shende. 2019. Towards Runtime Analytics in a Parallel Performance System. *2019 International Conference on High Performance Computing & Simulation (HPCS)* (2019), 559–566. https://api.semanticscholar.org/CorpusID:221590339

[26] Allen D. Malony, Sameer S. Shende, Wyatt Spear, Chee Wai Lee, and Scott Biersdorff. 2011. Advances in the TAU Performance System. In *Parallel Tools Workshop*.

[27] Anirban Mandal, Paul Ruth, Ilya Baldin, Dariusz Krol, Gideon Juve, Rajiv Mayani, Rafael Ferreira Da Silva, Ewa Deelman, Jeremy Meredith, Jeffrey Vetter, et al. 2016. Toward an end-to-end framework for modeling, monitoring and anomaly detection for scientific workflows. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1370–1379.

[28] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[29] Marta Mattoso, Jonas Dias, Kary ACS Ocana, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vitor Silva, and Daniel De Oliveira. 2015. Dynamic steering of HPC scientific workflows: A survey. *Future Generation Computer Systems* 46 (2015), 100–113.

[30] Andre Merzky, Pavlo Svirin, and Matteo Turilli. 2019. PanDA and RADICAL-Pilot Integration: Enabling the Pilot Paradigm on HPC Resources. In *EPJ Web of Conferences*, Vol. 214. EDP Sciences, 03057.

[31] Andre Merzky, Matteo Turilli, and Shantenu Jha. 2022. Raptor: Ravenous throughput computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 595–604.

[32] Andre Merzky, Matteo Turilli, Manuel Maldonado, Mark Santcroos, and Shantenu Jha. 2019. Using Pilot Systems to Execute Many Task Workloads on Supercomputers. In *Job Scheduling Strategies for Parallel Processing: 22nd International Workshop, JSSPP 2018, Vancouver, BC, Canada, May 25, 2018*, Dalibor Klusáček, Walfredo Cirne, and Narayan Desai (Eds.). Springer, 61–82.

[33] Andre Merzky, Matteo Turilli, Mikhail Titov, Aymen Al-Saadi, and Shantenu Jha. 2021. Design and performance characterization of RADICAL-pilot on leadership-class platforms. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 818–829.

[34] Aroon Nataraj, Matthew J. Sottile, Allen Morris, Allen D. Malony, and Sameer S. Shende. 2007. TAUoverSupermon : Low-Overhead Online Parallel Performance Monitoring. In *European Conference on Parallel Processing*. https://api.semanticscholar.org/CorpusID:18418017

[35] OLCF. 2024. Summit System Overview. https://docs.olcf.ornl.gov/systems/summit_user_guide.html#system-overview. Accessed: 2024-4-30.

[36] RADICAL Development Team. 2023. RADICAL-pilot: Integration with Flux. https://github.com/radical-cybertools/radical.pilot/blob/devel/src/radical/pilot/agent/executing/flux.py, last accessed 2023-08-07.

[37] Srinivasan Ramesh, Hank Childs, and Allen D. Malony. 2022. SERVIZ: A Shared In Situ Visualization Service. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[38] Srinivasan Ramesh, Robert B. Ross, Matthieu Dorier, Allen D. Malony, Philip H. Carns, and Kevin A. Huck. 2021. SYMBIOMON: A High-Performance, Composable Monitoring Service. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 332–342.

[39] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, et al. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Comp. Sci. and Tech.* 35 (2020), 121–144.

[40] Jeanette Sperhac, Robert L DeLeon, Joseph P White, Matthew Jones, Andrew E Bruno, Renette Jones Ivey, Thomas R Furlani, Jonathan E Bard, and Vipin Chaudhary. 2020. Towards Performant Workflows, Monitoring and Measuring. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–9.

[41] The OpenFOAM Foundation. 2023. OpenFOAM, open source software for Computational Fluid Dynamics (CFD). https://openfoam.org

[42] Mikhail Titov, Matteo Turilli, Andre Merzky, Thomas Naughton, Wael Elwasif, and Shantenu Jha. 2022. RADICAL-pilot and PMIx/PRRTE: Executing heterogeneous workloads at large scale on partitioned HPC resources. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 88–107.

[43] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevakos, and Shantenu Jha. 2019. Middleware Building Blocks for Workflow systems. *Computing in Science & Engineering* 21, 4 (2019), 62–75.

[44] Matteo Turilli, Feng Liu, Zhao Zhang, Andre Merzky, Michael Wilde, Jon Weissman, Daniel S Katz, and Shantenu Jha. 2016. Integrating Abstractions to Enhance the Execution of Distributed Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 953–962.

[45] Matteo Turilli, Andre Merzky, Thomas Naughton, Wael Elwasif, and Shantenu Jha. 2019. Characterizing the Performance of Executing Many-tasks on Summit. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 18–25.

[46] Matteo Turilli, Mark Santcroos, and Shantenu Jha. 2018. A Comprehensive Perspective on Pilot-job Systems. *Comput. Surveys* 51, 2 (2018), 43.

[47] Matthew Wolf, Julien Dominski, Gabriele Merlo, J. Choi, Greg Eisenhauer, Stéphane Ethier, Kevin A. Huck, Scott Klasky, Jeremy S. Logan, Allen D. Malony, and Chad Wood. 2019. Scalable Performance Awareness for In Situ Scientific Applications. *2019 15th International Conference on eScience (eScience)* (2019), 266–276. https://api.semanticscholar.org/CorpusID:214596196

[48] Dewi Yokelson, Oskar Lappi, Srinivasan Ramesh, Miikka S. Väisälä, Kevin Huck, Touko Puro, Boyana Norris, Maarit Korpi-Lagg, Keijo Heljanko, and Allen D. Malony. 2024. SOMA: Observability, monitoring, and in situ analytics for exascale applications. *Concurrency and Computation: Practice and Experience* (June 2024), e8141. https://doi.org/10.1002/cpe.8141