Flux: Overcoming Scheduling Challenges for Exascale Workflows

Dong H. Ahn[*], Ned Bass[*], Albert Chu[*], Jim Garlick[*], Mark Grondona[*], Stephen Herbein[*], Helgi I. Ingólfsson[*], Joseph Koning[*], Tapasya Patki[*], Thomas R. W. Scogland[*], Becky Springmeyer[*] Michela Taufer[†]
[*]Lawrence Livermore National Laboratory, 7000 East Ave. Livermore, CA
{ahn1, bass6, chu11, garlick1, grondona1, herbein1, ingolfsson1, koning1, patki1, scogland1, springmeyer1}@llnl.gov [†]University of Tennessee, Knoxville. Knoxville, TN
taufer@utk.edu

***Abstract*—Many emerging scientific workflows that target high-end HPC systems require complex interplay with the resource and job management software (RJMS). However, portable, efficient and easy-to-use scheduling and execution of these workflows is still an unsolved problem. We present Flux, a novel, hierarchical RJMS infrastructure that addresses the key scheduling challenges of modern workflows in a scalable, easy-to-use, and portable manner. At the heart of Flux lies its ability to be seamlessly nested within batch allocations created by other schedulers as well as itself. Once a hierarchy of Flux instances is created within each allocation, its consistent and rich set of well-defined APIs portably and efficiently support those workflows that can often feature non-traditional execution patterns such as requirements for complex co-scheduling, massive ensembles of small jobs and coordination among jobs in an ensemble. Our evaluation of Flux on some of the emerging workflow efforts at Lawrence Livermore National Laboratory indicates that our approach can significantly address major workflow scheduling challenges: job throughput, co-scheduling, job coordination and communication and portability challenges. Further, our performance measurements on both synthetic and real-world ensemble-based workflows suggest that our solution can improve the job throughput performance of these scientific workflows by a factor of 48.**

I. Introduction

Scientific workflows continue to become more complex, and their execution patterns are also drastically changing. To exploit the ever-growing compute power of systems and upcoming exascale platforms, modern workflows increasingly employ multiple types of simulation applications coupled with *in situ* visualization, data analytics, data stores and machine learning [1], [2], [3], [4]. Furthermore, the current push towards rigorous verification and validation (V&V) and uncertainty quantification (UQ) [5] approaches often features simulations that involve enormously large numbers of short-running jobs (e.g., reduced models and 1D simulations), straying away from traditional long-running execution.

These trends have become ever more apparent on some of the most massive high-performance computing (HPC) systems, such as the Sierra [6] and Summit [7] machines, the new pre-exascale systems fielded at the world's largest supercomputing centers. Three major early science applications running on Lawrence Livermore National Laboratory (LLNL)'s Sierra, including [2], [8], for instance, now embrace non-traditional workflows. Additionally, our recent analysis on other large *production* clusters at LLNL shows that 48.1% of jobs involved the submission of at least 100 identical jobs by the same user with 27.8% submitted within one minute of each other, a pattern typically associated with V&V and UQ. Such workflows, often referred to as *ensemble-based*, are quickly becoming a norm.

Resource and job management software (RJMS) is central to enabling efficient execution of applications on HPC systems, and therefore is also the main interface for executing these complex workflows. However, recent trends towards new execution patterns, significantly complicate efficient (co-)scheduling and execution of their tasks. In particular, *centralized* techniques implemented within widely deployed RJMS including SLURM [9], IBM LSF [10], MOAB [11], or PBS Pro [12] no longer work well as they are fundamentally designed for the traditional paradigm: a few large, long-running, homogeneous jobs rather than ensembles composed of many, and often small, short-running heterogeneous tasks.

These limitations are already presenting greater technical challenges for exascale workflows, which will only worsen if not met properly. Four such key challenges are listed below.

1) *Throughput Challenge*: Large ensemble simulations require massive numbers of jobs that cannot comfortably be ingested and scheduled by the traditional approach;
2) *Co-scheduling Challenge*: Complex coupling requires sophisticated co-scheduling that the existing centralized approaches cannot easily provide;
3) *Job coordination and communication challenge*: Intimate interactions with RJMS is required to keep track of the overall progress of the ensemble execution, and existing approaches lack well-defined interfaces;
4) *Portability Challenge*: There has been a proliferation of ad hoc implementations of user-level schedulers as an attempt to tackle the above challenges. They are often non-portable and come with a myriad of side effects (e.g., millions of small files just to coordinate the current state of an ensemble).

In this paper, we present Flux, a novel resource management and scheduling infrastructure that overcomes the above challenges in a scalable, easy-to-use, portable, and cost-effective manner. At the core of Flux lies its ability to be seamlessly nested within batch allocations that are created by itself or other resource managers, along with allowing for user-level customization of scheduling policies and parameters. This *fully hierarchical* approach allows the target workflows to submit fewer jobs that resemble the traditional execution pattern to the low-level schedulers, most notably the native system scheduler, while more fine-grained scheduling is performed by a hierarchy of nested instances running within each allocation. Each level also allows customizable scheduling policies and parameters, addressing both the *throughput and co-scheduling challenges*.

In addition, Flux is designed from the ground up as a software framework with a rich set of well-defined APIs including job submission, job status and control, messaging, as well as input and output streaming APIs. Workflows can use any of these APIs to *facilitate communication and coordination of various tasks* to be executed within and across ensembles. Finally, to address *portability challenges*, its APIs are specifically designed to be consistent across different platforms. Creating an instance requires only the lower-level resource manager to provide the Process Management Interface (PMI), the de facto standard for MPI bootstrapping, or the user to provide a configuration.

Specifically, this paper makes the following contributions:

- Identification and discussions of specific exascale workflow scheduling challenges based on emerging practices at LLNL, one of the world's largest supercomputer centers;
- Novel hierarchical approaches for providing resource management and scheduling infrastructure at the user level to address the above challenges;
- Performance evaluations of our hierarchical approaches on up to one million short-running jobs using both synthetic and real simulation codes;
- Case studies and lessons learned from integrating our approaches to three distinct real-world workflow management systems targeting exascale computing;
- Discussions on techniques needed to address the remaining challenges.

Our evaluation of Flux on emerging workflow efforts at LLNL shows that our solution significantly overcomes all of the stated challenges. First, our case study on the Multiscale Machine-Learned Modeling Infrastructure (MuMMI) shows that Flux can efficiently co-schedule a new workflow that employs machine learning (ML) to couple a large macro-scale simulation with an ensemble of tens of thousands of micro-scale molecular dynamics (MD) simulations; starting and stopping during a run at high speed. Second, our integration with Merlin, a workflow management system designed to support next-generation ML on HPC, shows that Flux significantly enables not only co-scheduling of various task types within each ensemble but also its needs for high portability and task communication and coordination. Third, our performance measurements on both synthetic and real-world ensemble-based workflows suggest that our hierarchical scheduling approach can lead to $48\times$ performance improvement in terms of job throughput for these workflows.

II. A Motivational Example

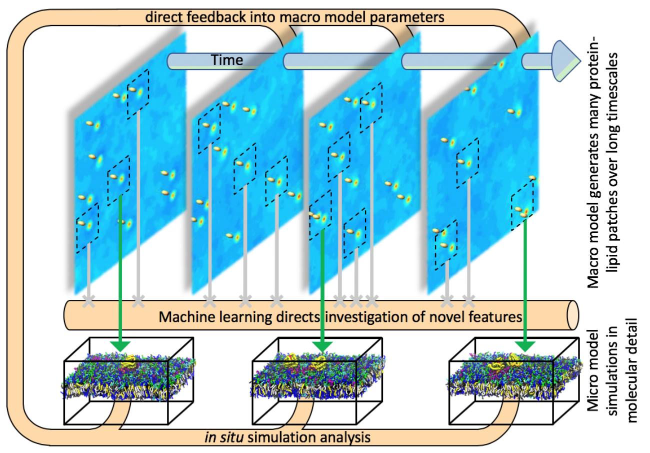
To motivate the need for our technology, we consider the Joint Design of Advanced Computing Solutions for Cancer (JDACS4C) Pilot 2 workflow as our motivational example, an early science application being run on LLNL's Sierra system. This workflow features non-traditional co-scheduling and execution patterns that present challenges to existing system schedulers.

A. The Multiscale Machine-Learned Modeling Infrastructure

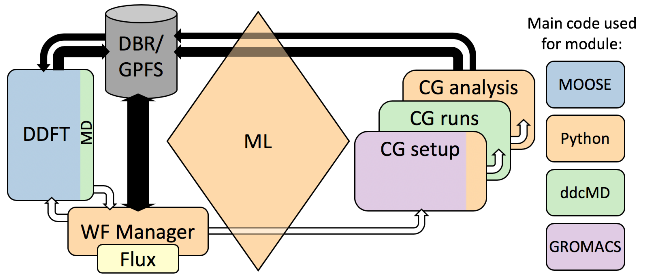
The JDACS4C program is a partnership between the United States (US) Department of Energy (DOE) and the National Cancer Institute (NCI) to advance cancer research using emerging exascale HPC capabilities. The Pilot 2 project within JDACS4C seeks to develop effective HPC simulation methods to uncover the detailed characterizations of the behavior of RAS on cellular membranes. The RAS protein family are small GTPase signaling proteins involved with control of cell division and growth. Mutations leading to increased RAS activity contribute to a wide range of cancers and up to 30% of human cancers are linked to mutations in the ras gene family [13]. RAS proteins typically signal their downstream effectors when bound to the lipid bilayer of cellular membranes and currently no drugs exist that inhibit RAS activity. Resolving RAS membrane dynamics and aggregation is a difficult task as macroscale length and time scales are needed; yet, microscale molecule-level details are required to capture protein-protein and protein-lipid interactions.

To resolve RAS structure and dynamics on cellular membranes, the Pilot 2 team developed MuMMI [8] that can sample at the macroscale with effective microscale resolution. MuMMI is an adaptive multiscale infrastructure that directly couples molecular-detail MD simulations to a cellular-scale macro simulation. Figure 1a depicts the MuMMI workflow. A macro model, in this case, an 1 μm x 1 μm realistic eight lipid-type plasma membrane mimic is simulated with 300 RAS molecules. This novel macro model is coupled with an ML module that drives the sampling of *patches*, small neighborhoods around an RAS molecule. These patches are then used to instantiate and run corresponding MD simulations. Additionally, several *in situ* processing components are required to control the running of particular MD simulations and to provide feedback to the macro model for parameter refinement.

Fig. 1b presents the MuMMI modules, code framework and how they are connected. The macro model is simulated at biologically relevant and experimentally accessible time and length scale, but with rather low resolution. The finite element solver MOOSE [14] is used to evolve the continuum lipid dynamics, based on a dynamic density functional theory (DDFT) model. The DDFT dynamics are then coupled to a Langevin particle model running on ddcMD [15] that allows for the evolution of discrete RAS proteins on the membrane. The ML is then used to direct instigation and investigation of coarse-grained (CG) particle simulations of the most "interesting" patches of the macro simulation. Its novel


direct feedback into macro model parameters
Time
Macro model generates many protein-lipid patches over long timescales
Machine learning directs investigation of novel features
Micro model simulations in molecular detail
in situ simulation analysis

(a) MuMMI couples macro and micro scale simulations using ML.



(b) MuMMI modules and code framework

Fig. 1: The Multiscale Machine-Learned Modeling Infrastructure (MuMMI) leverages ML to drive the effective sampling of a large length and time scale simulation for further molecular-level, detailed simulations on heterogeneous HPC resources.

features provide an intelligent sampling of the macro-scale simulation space, which then further leads to a detailed exploration that is not achievable using only brute force calculations. Furthermore, the *in situ* analyses of the CG simulations provide feedback back into the macro model: i.e., the vast sampling carried out at the CG level heals the initial macro model parameters in real time.

B. *Complex Execution Patterns Examplified by MuMMI*

At each time step of the macro model within MuMMI, 300 patches are extracted (one centered at each RAS protein) and compared to all patches previously explored using MD simulations. Whenever computing resources become available, the most unusual new patch (i.e., the patch with the largest distance to its neighbors in latent space) is taken and a new corresponding MD simulation is created and then executed. Therefore, the framework discussed above crucially relies on the ability to automatically instantiate MD simulations, monitor them on the fly, and provide feedback to the macro model. RAS orientations are selected from pre-constructed libraries based on their state, randomly rotated in the membrane plain, and pulled to the membrane surface.

The CG setup module is Python-based and uses the GROMACS MD package [16] for minimization and initial equilibration before ddcMD is used for production simulations (CG run) on GPU resources. A new ddcMD version was developed that implements the Martini force field [17], [18], adding an atom padding technique, and implements the entire MD loop in CUDA. This implementation offloads the entire computation to the GPU with nearly no CPU resource requirements, freeing those resources for other MuMMI modules. Every calculation step necessary for Martini now runs on the GPU via CUDA kernels, including the integrator and constraint solver, such that particles are only communicated back to the host for I/O purposes and never for calculations of the particle forces or movement. This leaves the CPUs tasked with only managing the order and launches of the aforementioned kernels. While the MD simulations are running, analysis modules (CG analysis) are actively monitoring the simulation analyzing all outputted frames on the fly and less frequently storing frames for offline analysis. The online analysis modules continuously accumulate data of interest for analysis and provide on-the-fly updates to the macro model parameters based on the newest CG simulation data.

The MuMMI workflow manager (WF Manager) connects all these different components. The WF manager consumes all generated macro model patches, initiates the ML infrastructure to score patches, and schedules selected patches for simulation setup and execution runs. All generated macro model patches are fed to the ML, which maintains a priority queue of candidate patches. Additionally, a small buffer of already setup simulations is maintained so available GPU resources can be utilized as soon as they become available. When new resources become available, top patch candidates are selected for simulation setup or simulation execution. MuMMI schedules jobs through Maestro [19], [20] that can use a wide range of back-end HPC resource managers and batch-job schedulers to start and stop jobs. The current workflow is coordinated through an in-memory data broker and a network file system. Fast data transfer and messaging are handled through the IBM® DataBroker (DBR) [21], which implements a fast, system-wide in-memory key-value store and files are shared through the IBM Spectrum Scale^{TM} (GPFS) parallel file system.

When running at scale on a heterogeneous resource machine like Sierra, the different MuMMI modules are run on different resources to utilize the machine better. The macro model (50-1000 nodes), ML and WF manager (one node), DBR (50-100 nodes), and GROMACS (CG setup) simulations (one node each) are all run on CPUs only. While, in the case of Sierra, four separate ddcMD simulations are run on each node, one for each GPU as well as an accompanying CG analysis (CPUs only) for each simulation. Therefore, a typical node is running nine separate MuMMI jobs. In order for this to work well at scale on Sierra with 4000 nodes, the job execution system needed to manage up to 36,000 simultaneous jobs and continually re-schedule work as microscale jobs complete and new simulations takeover those resources. Overall, MuMMI's execution pattern featuring complex co-scheduling of high volume of jobs has proven to be difficult even for arguably most advanced HPC resource managers and schedulers to handle.

III. Challenges in Workflow Scheduling

MuMMI exemplifies the many (co-)scheduling and execution challenges faced by emerging workflows. They include co-scheduling of coupled simulations at different scales (i.e., macro models-based simulations with several thousand MD simulations, coordination between CPU and GPU runs), the use of an ML module to schedule and execute simulations dynamically at a high rate, and the use of data store to coordinate the data flow between different tasks.

This section further further characterizes the key scheduling and execution challenges such as the ones shown in the MuMMI workflow. Our analysis is based on our direct interactions with three distinct workflow management software development teams at LLNL, namely the JDACS4C MuMMI workflow, Uncertainty Quantification Pipeline (UQP) [22], and the Merlin workflow that supports extreme-scale machine learning [2], as well as interviews with developers of other workflow management software such as PSUADE UQ framework [23] and end users who have created ad hoc schedulers for their workflows. While each of these workflows often addresses entirely different domains of science, they exhibit common scheduling issues. As briefly highlighted in Section I, they are referred to as throughput, co-scheduling, job coordination/communication, and portability challenges.

A. Throughput Challenge

Many workflows feature large ensembles of small, short-running jobs, which can create thousands or even millions of jobs that need to be rapidly ingested and scheduled. For the JDACS4C Pilot 2 example presented in the previous section, several thousand MD simulations need to be run successfully with a quick turnaround time to facilitate the refinement of parameters in the macro model and produce microscale results. In the case of the UQP, building a surrogate model can require tens to hundreds of thousands of simulation executions to adequately sample the simulation's input parameter space. Such ensemble workloads are becoming a norm rather than an exception on high-end HPC systems.

Traditional RJMS in most cloud and HPC centers today are based on centralized designs. Cloud schedulers such as Swarm and Kubernetes [24], [25] and HPC schedulers such as SLURM, MOAB, PBSPro and IBM LSF [9], [11], [10], [12] are implemented using this model. This model often fails to cope with rapid job ingestion, and because of this, a site imposes a cap on the number of jobs submitted at once and allowed in the scheduler. The cap then requires workflow managers to throttle the rate of their job submissions to match the ingestion rate, artificially decreasing the job throughput of the workload.

Furthermore, this pattern can also lead to shared resource thrashing and exhaustion. For example, the Sequoia supercomputer at LLNL, which has 1.6 million cores, encountered several scale-up problems when users tried to run about 1500 small UQ jobs (1-4 MPI tasks each) at the same point in time in 2014. While SLURM and IBM's control software managed to expand their limits to about 3-5K simultaneously executing jobs after fine-tuning various configuration parameters for some cases, several rare errors still kept cropping up. Eventually, LLNL created a temporary solution by building CRAM, a library that packs many small jobs into a single large job [26]. Unfortunately, libraries such as CRAM are not the panacea for centralized schedulers, and even well-engineered centralized solutions can suffer from several scalability and resiliency issues.

B. Co-scheduling Challenge

Coupling in complex workflows requires co-scheduling of different components. In the example we presented earlier, the CPU and GPU workloads need to be co-scheduled effectively. Additionally, data need to be communicated to the host when necessary, and support for *in situ* analysis, as well as online techniques, require other jobs to be active on the node. More specifically, in MuMMI four different types of jobs need to be scheduled on CPUs only and one job type on both GPU and CPU. Two of these jobs are always run in tandem with four pairs running on every node. One of the CPU jobs runs across a good fraction of the nodes, while other single node jobs are started as needed; fully utilizing all resources. Moreover, this decision is dynamically determined by an ML guided workflow, a completely new execution pattern.

Most traditional schedulers do not allow for such customization, making it challenging to utilize resources well. Co-scheduling can offer several utilization and job throughput benefits, as well as allow for customization of application kernels and efficient co-existence of multiple workflow components. Current schedulers offer little or no support for sharing multiple kinds of jobs within an allocation or for customizing resource allocations such as cores or GPUs (or others, such as burst buffers). If at all, only fixed mechanisms for requesting allocations exist, and users cannot tune these from one application to the next or leverage their domain knowledge about the resource utilization of their application.

C. Job coordination and communication Challenge

Modern scientific workflows depend on data transfer between various components of a framework. For example, as we showed in Fig. 1b, the information about novel patches triggers additional micro model simulations, which in turn are used for further parameter refinement. Multiple such simulations need to be analyzed on the fly and their information is aggregated and weighted to update the macro model parameters, which requires intimate coordination and communication between jobs as well as within the job.

Existing schedulers have limited support for ingesting, storing/retrieving job output or job status information, often requiring inefficient communication through file systems. Many workflow managers, such as UQP circumvent these issues by having jobs create an empty file whenever they start or complete. This allows UQP to track the state of every job in the workflow, but it is at the cost of creating a large and unnecessary metadata load on the target file system, infringing on the performance of both the workflow itself and the entire system.

D. Portability Challenge

One of the common problems with emerging workflow management systems is that they have to be ported to a wide range of RJMS. With no common infrastructure for supporting their scheduling, the task of porting m workflows to n environments amounts to an $m \times n$ effort. Often, those point solutions are non-portable, and even if a solution is ported on a new platform, they can often come with a multitude of side-effects (e.g., creating too many files for ensemble status checking). The more complex the target workflow is, the more difficult porting would become because a new scheduler may not provide all of the advanced features that the workflow might have used in its previously tested schedulers.

Scientists and developers often need to rewrite their scripts from scratch in order to adapt to a new environment, potentially introducing several scripting/setup bugs, requiring additional testing, underutilized resource allocations and reducing overall productivity. For example, in the MuMMI RAS multiscale simulation campaign, moving from a cluster that uses IBM's LSF and `jsrun` to another that relies on SLURM can be challenging regarding setup cost. Also, being able to leverage different heterogeneous resources, including GPUs and burst buffers, often requires new flags and configuration parameters to be specified. This often results in ad hoc solutions for application scheduling.

IV. Flux

The Flux framework is a suite of projects, tools and libraries that can be used to provide both site- or user-level resource managers and schedulers for large HPC centers. Flux provides a fully hierarchical software framework architecture to allow for seamless nesting of resource manager and scheduler instances in a highly scalable and customizable manner. The main foundation of this framework is a scalable tree-based overlay network. In fact, a Flux instance is a complete instantiation of this overlay network along with resource management and scheduling service modules that are dynamically loaded into this overlay network, leveraging its various communication idioms including *publish-subscribe*, *request-reply* and *push-pull* as well as asynchronous event handling. A Flux instance can spawn one or more child Flux instances that can manage a subset of the parent's resources, and such a nesting can further recurse.

Flux's software architecture is highly modular. The core resource-management services such as heartbeat, remote execution and key-value store are provided by a main

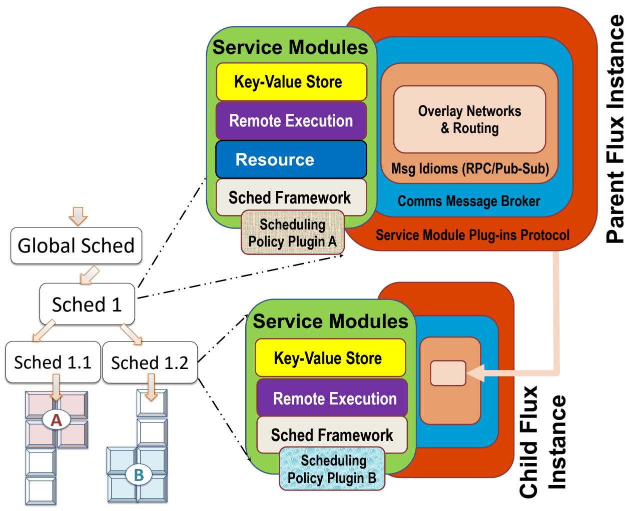


Fig. 2: Flux framework

component called *flux-core* while the job scheduling service is abstracted out into a separate component referred to as *flux-sched*. At runtime, scheduling modules in *flux-sched* are dynamically loaded into a Flux instance and handles all the functionality common to scheduling for that instance. [1] *flux-sched* has an ability to load one or more scheduling plugins that provide specific scheduling behaviors. They can be user-defined or administrative, allowing the owner of the Flux instance to specialize scheduling behaviors and policies that are tailored to the target workflow. Fig. 2 shows the modular architecture of Flux, and also depicts how the Flux network can be organized to manage two schedulers at different levels of the hierarchy, with a parent Flux instance and a child Flux instance. We will discuss Flux's fully hierarchical scheduling model in detail in the subsections below.

A. Scheduler Parallelism for Throughput Challenge

The hierarchical design of Flux provides ample parallelism to overcome the job throughput challenge present in traditional scheduling techniques. Under the hierarchical design of Flux, any Flux instance can spawn child instances to aid in scheduling, launching, and managing jobs. The parent Flux instance grants a subset of its jobs and resources to each child. This parent-child relationship, depicted in Fig. 2, can extend to an arbitrary depth and width, creating a limitless opportunity for parallelization while avoiding the high communication overhead of other distributed schedulers (e.g., fully connected graphs of schedulers and all-to-all communications). Parent and child instances communicate using the Flux communication overlay network described further in Section IV-C.

Our current implementation of hierarchical Flux consists of three main design points: the scheduler hierarchy, the

[1]A Flux instance loaded with scheduler modules will be referred to as a Flux scheduler instance or simply a scheduler instance when this distinction can elucidate our concepts.

resource assignment, and the job distribution. For the scheduler hierarchy, our implementation supports a hierarchy of schedulers with a fixed size and shape. Ensemble workflow managers or users specify the exact hierarchy size and shape using JSON, which our implementation parses and uses to launch the corresponding scheduler hierarchy automatically. For the resource assignment, by default, our implementation assigns a uniform number of resources to schedulers at each level in the hierarchy (e.g., all of the leaf schedulers are allocated the same number of cores). Non-uniform assignments of resources are possible but require careful consideration when distributing jobs.

To minimize the changes required for workflows to leverage hierarchical Flux, the workflow manager submits each job in the ensemble individually at runtime to the root scheduler instance (as it would with a traditional scheduler), and then, the jobs are distributed automatically across the hierarchy. In this configuration, it may seem that the root instance will become a bottleneck, but the work required to map and send a job to a child scheduler is significantly less than the work required to schedule and launch a job. After a job is submitted, the root instance in the hierarchy must only consider tens to hundreds of children, while a traditional scheduler must consider thousands of cores as well as all other jobs in the queue. Additionally, the job distribution at the root instance can overlap with the scheduling and launching of jobs at the leaf instances. For the job distribution, our implementation, by default, uses round-robin to distribute jobs uniformly across the scheduler hierarchy, but other distribution policies are supported and can be implemented by users.

B. *Scheduler Specialization Solves Co-scheduling Challenge*

Flux's user-driven, customizable approach to scheduling provides inherent support for co-scheduling. Flux's flexible design allows users to decide whether or not co-scheduling should be configured and also lets users choose their own scheduling policies within the scope of an instance. With the help of the job submission API, several tasks can efficiently coexist on a single node without any restrictions on their number, type, or resource requirements. This allows for submission and tuning at all possible levels of heterogeneity within a node (and across nodes), including individual cores, a set of cores, sockets, GPUs, or burst buffers.

Users can also choose a policy within their Flux instance. These can be simple policies, such as first-come-first-serve/back-filling, and the infrastructure can be easily extended to incorporate complex policies for advanced management of resources such as I/O or power or multiple constraints. Traditional resource managers do not provide any such capability or extensible design to users, resulting in underutilized resources and limited throughput. While some workflows need exclusive scheduling per node, other workflows may need co-scheduling or different distributions of jobs between the resources available on a node. Traditional RJMS software has no support for user-level scheduling, which Flux addresses by design, giving users the freedom to adapt to their instance to the needs and characteristics of their particular application.

C. *Rich APIs for Easy Job Coordination and Communication*

Flux provides various communication idioms and APIs to help solve the job coordination and communication challenge. To support coordination within and across both Flux instances and jobs, Flux provides primitives that encapsulate the *publish-subscribe* (pub/sub), *request-reply*, and *push-pull* communication patterns. These primitives allow individual jobs within a workflow to synchronize without the use of ad hoc methods like empty file creation on a POSIX-compliant file system. Flux also provides several high-level services that jobs and workflows can leverage: an in-memory key-value store (KVS) and a job status/control (JSC) API.

The KVS provided by Flux enables jobs and workflow managers to retrieve and store information scalably. One example KVS use-case for workflows is accessing job provenance data. All of a job's metadata is stored in Flux's KVS, including the resources requested, the environment variables used, and the contents of `stdout` and `stderr`. The storage of the `stdout` and `stderr` enables workflow managers to inspect a job's output easily without requiring expensive file system accesses. A specific feature of Flux's KVS, watcher callbacks, enables workflow managers to ingest and analyze a job's output efficiently as it is being generated. Advanced workflows can leverage this real-time output analysis to detect job failures as they happen and take corrective actions, such as re-submitting the job for execution.

Traditional schedulers provide limited access to job status information, most commonly through a slow and cumbersome command line interface (CLI). Many workflow managers work around this interface by tracking job states via extraneous file creation. Flux's JSC provides a fast, programmatic way to receive job status updates, eliminating the use of the slow CLI and tracking via the file system. JSC users can subscribe to real-time job status updates, which are sent whenever a job changes its state (e.g., from *running* to *completed*). This allows workflow managers to stay up-to-date on the state of their jobs with minimal overhead and without degrading file system performance.

D. *Consistent API Set for High Portability*

To serve as the common, portable scheduling infrastructure, Flux offers two main characteristics: 1) its APIs are consistent across different platforms and 2) the porting and optimization effort of Flux itself for a new environment is small. Creating a Flux instance on a given environment only requires the lower-level resource manager to provide the Process Management Interface (PMI), or the user to provide a configuration. Because PMI is the de facto standard for MPI bootstrapping, the system resource managers (including Flux itself) on a majority of HPC systems directly offer this interface or else provide other variant interfaces such as PMIx on top of which PMI can be easily implemented.

V. Evaluating Performance and Scalability

To demonstrate how Flux, with its fully hierarchical design, addresses the throughput challenge, we measure the scheduler throughput on real-world and stress-test ensemble workflows.

We measure throughput as the average number of jobs ingested, scheduled, and launched per second (the higher, the better). We schedule the workflows using three different hierarchies: depth-1, depth-2, and depth-3 [2]. The depth-1 hierarchy only has a single scheduler instance that schedules every job in the workflow, similar to existing schedulers like SLURM and Moab. For the depth-2 hierarchy, we create a root scheduler with one child scheduler for every node allocated to the workflow, and we distribute the jobs equally among the lowest level of schedulers (i.e., the leaf schedulers). For the depth-3 hierarchy, we extend the hierarchy by adding one scheduler for every core allocated to the workflow, and as with the previous hierarchy, we distribute the workflow's jobs equally among the leaf schedulers. Our throughput evaluations on both workflows use 32 nodes of an Intel Xeon E5-2695v4 cluster, each node with 36 physical cores and 128 GB of memory.

To demonstrate the effects of hierarchical Flux on a real-world workflow, we generated an ensemble workflow with the Uncertainty Quantification Pipeline (UQP) [22]. Our UQ ensemble simulates a semi-analytical inertial confinement fusion (ICF) stagnation model that predicts the results of full ICF simulations [27], [28], [29]. UQ ensembles with this semi-analytical model typically consist of tens of thousands of runs, but the scientists' goal is to execute millions of jobs.

Fig. 3a shows the scheduler throughput of the three hierarchies when applied to variably sized real-world UQ ensemble workflows. For each ensemble size, we perform the test three times and present the min, max, and median job throughput values. As we increase the ensemble size, the throughput of the depth-1 scheduler plateaus at 10 jobs/sec, artificially limiting the overall performance of the ensemble workflow and creating idle resources. By adding additional levels to the scheduler hierarchy (i.e., depth-2 and depth-3) and thus increasing the scheduler parallelism, we can improve the peak job throughput by an order of magnitude. With a job throughput of 100 jobs/sec, the scheduler is no longer on the critical path of the workflow and the compute resources are 100% utilized. After the scheduler throughput enhancements provided by hierarchical scheduling, the ensemble workflow's critical path now consists primarily of the ensemble application's runtime.

To demonstrate the throughput capabilities of hierarchical Flux, unrestrained by the workflow application's runtime, we created a *stress-test ensemble workflow* in which each job exits immediately after it launches (i.e., has a negligible runtime). Fig. 3b shows the throughput of Flux on this stress-test workflow. As before, for each ensemble size, we perform the test three times and present the min, max, and median job throughput values. No longer limited by the workflow application's runtime, the depth-2 and depth-3 hierarchies achieve a peak throughput of 370 jobs/sec and 760 jobs/sec, respectively. These represent a $23.5\times$ and $48\times$ increase over the job throughput achieved by the traditional, depth-1 scheduler.

[2]Our model supports additional levels. In our evaluation, we use a one-to-one mapping between hardware and scheduler levels.

VI. Enabling Emerging Workflow Management with Flux

In this section, we describe how we improve the scheduling and execution of real-world production workflows using Flux. Our study targets both the workflow in MuMMI already described in Section II and the Merlin workflow.

A. Specialization Addresses Challenges in MuMMI

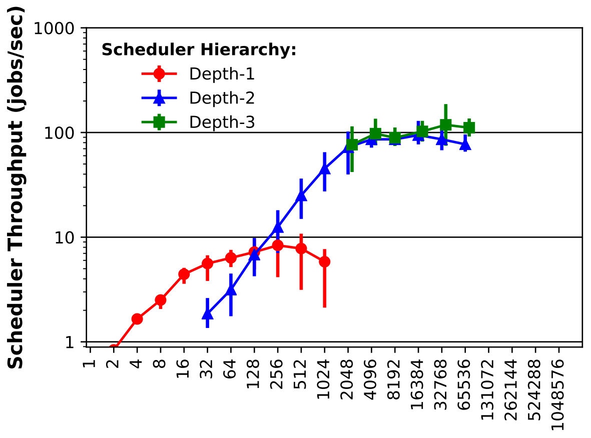
Fig. 1b shows how the Flux infrastructure interacts with the rest of the JDACS4C Pilot 2 multiscale infrastructure. MuMMIs workflow manager instantiates the ML module, which implements the latent space, and uses Maestro [19], [20] to start and stop jobs accordingly. To handle the volume of jobs and the required co-scheduling of resources, the team developed a Maestro adapter to Flux.

The workflow manager is closely coupled to the macro simulation, progressing each simulation frame as it is generated. Each frame is decomposed into 300 patches, one for each RAS in the simulations. Each patch is transformed into the latent space and scored, based on its distance from those patches that have already been simulated at the micro scale. The workflow manager maintains a priority queue of the top n candidate patches, and when new resources become available, the queue is re-evaluated and a set of new patches for CG MD simulation set-up (create micro sims) and/or production runs (micro sims, ddcMD) scheduled.

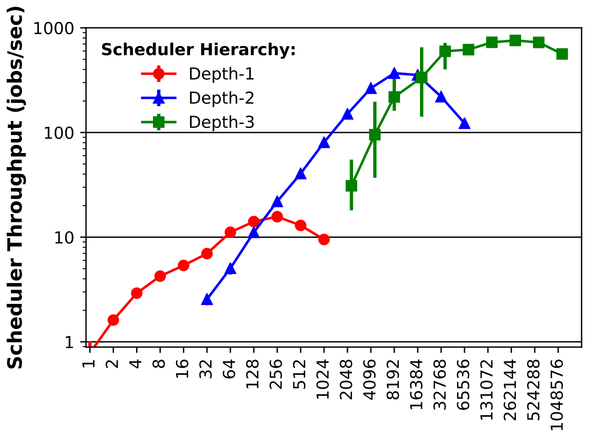
This means that the primary scheduling objective required from Flux is a simple first-come, first-served (FCFS) policy tailored for high-throughput workload. Leveraging Flux's ability to specialize the scheduling policy and parameters for each instance, MuMMI instantiates the preexisting FCFS scheduling plugin with a scheduling parameter that further optimizes the scheduler performance for high-throughput workload. We specifically set the depth of the queue to one so that the scheduler does not have to look ahead for later jobs to schedule, an optimization of the FCFS policy that can improve resource utilization without having to break the definition of FCFS. (If the blocked highest priority job requires a compute node without GPU while the next job requires a node with GPU, the latter job can be scheduled without affecting the schedulability of the first job.)

Production runs of MuMMI have used this flexibility to tune the scheduler for higher performance for the particular workload. That is, considering only a fewer jobs when making a decision of what to run, which would be inappropriate for a center-wide scheduler that must maintain fairness, but that provides significant performance benefits for this application and its high quantity of concurrent jobs.

Furthermore, as described in Section II, MuMMI must schedule different type of jobs on a machine with heterogeneous resources. Fig. 4 shows the resource utilization for a typical, 2,040-node MuMMI run on Sierra. The MuMMI workflow launches four different kinds of jobs on CPUs only, and an additional type of job on some CPUs and GPUs. A single dedicated node is used for the root of the Flux master instance as well as 24 CPU cores of another node to the MuMMI workflow manger (Fig. 4, gray and



(a) UQ workflow



(b) Stress test workflow

Fig. 3: Job throughput (in jobs/sec, on a logarithmic scale) for the depth-1, depth-2, and depth-3 scheduler hierarchies for fixed-size clusters and differing numbers of total jobs (on a logarithmic scale)

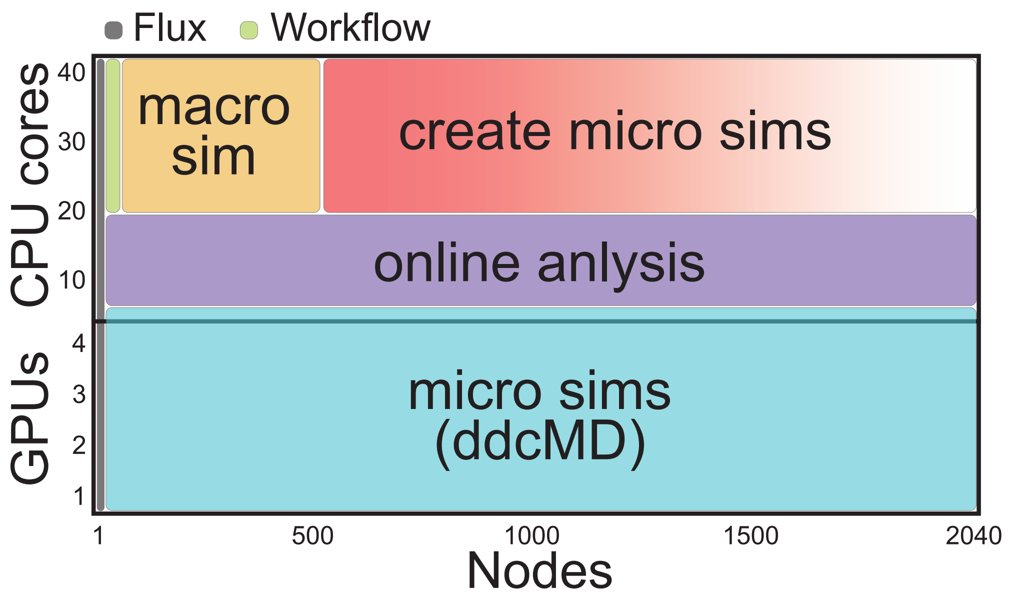


Fig. 4: MuMMI utilization of heterogeneous resources: the CPU/GPU resource used by different MuMMI jobs are shown for a typical run of 2,040 nodes on Sierra.

green, respectively). Other resources are split between the macro simulation, creation of CG MD micro simulations, the online analysis modules and the production macro simulation run using ddcMD (Fig. 4, yellow, red, purple and blue, respectively). This means that each node normally has nine running jobs, one pure CPU (e.g. macro model or create micro simulation) and eight jobs for the four different micro simulations running on that node (one ddcMD on each of the four GPUs and one online analysis for each ddcMD simulation). Flux automatically discovers CPUs and GPUs using `libhwloc` upon being instantiated and uses them for scheduling. Therefore, setting the scheduling granularity to CPU/GPU-level instead of exclusive node-level fulfilled this co-scheduling requirement. Additionally, to speed up scheduling and reduce the number of jobs needed to be scheduled and maintained by the master Flux scheduler instance, a child Flux instance was launched on every node and the eight jobs related to the four micro simulations running on each node are managed through this local instance.

Overall, during a simulation restart the MuMMI workflow using Flux can fill all available resources on Sierra in less than 1 hour, reaching a steady-state. Running on all of Sierra, 4,000 nodes, at peak all GPUs and CPUs were utilized, 16,000 GPUs and 176,000 CPU cores, running a full MuMMI multiscale simulations with 16,000 CG MD simulation running at the same time. The JDACS4C Pilot 2 team ran a huge RAS multiscale simulation campaign on Sierra, described in [8]. 300 RAS proteins were simulated on a square μm plasma membrane at the macro scale for over 150 µs. Over 116,000 patches were selected from the macro simulation and simulated at CG MD micro level with a total aggregated time of 200 ms, orders of magnitude greater than comparable studies. In total, the multiscale simulation campaign used 5.6 million GPU hours, the online analysis processed over 400,000,000 simulation frames, and for each simulation every 2 ns a snapshot was saved for later offline analysis resulting in over 320 TB of data

B. Easy, Scalable Interaction with the Scheduler for Merlin

The Merlin workflow is a component of the Machine Learning Strategic Initiative (MLSI) [2] at LLNL. Merlin's goal is to provide a python-based workflow that is adaptable and efficient. This workflow runs an ensemble of simulations and records the results while concurrently running machine learning on the results as they become available. The machine-learned model then helps steer the ensemble of simulations as it improves with more data.

The workflow executes a variety of tasks to generate and analyze the data. The first of these is defining the ensemble of simulations. This ensemble consists of a set of samples spanning the domain needed for creating a unique set of data describing the domain. A simulation executable task will accept the sample set as input parameters and produce data for the machine-learning model. The simulation can range from a simple ordinary differential equations (ODE) to a massively parallel MPI rad-hydro simulation. These simulations may also be run on many different platforms with different resource managers and schedulers, where scheduling and launching the simulations in a general manner becomes difficult.

The first version of the Merlin MPI parallel launch used a simple python-based subprocess call to take a set of MPI parameters such as the number of nodes and tasks and map them onto the commands needed for a SLURM or IBM LSF launch. This became a maintenance issue when each new batch system required a set of runtime parameters that do not map one to one between the various launch systems. In the case of `jsrun` for LSF, the system did not handle nested launches where there was one `jsrun` call for the allocation and a subsequent `jsrun` call for the simulation. Some parallel runs need GPU support and few CPU cores, while others require only CPU cores. This requirement puts the onus on the workflow to schedule resources for the various types of parallel jobs.

In Merlin, Flux solves both the nesting issue and co-scheduling issue through the use of a single Flux instance. Jobs can be co-scheduled because this single instance is tracking all of the resources with a GPU/CPU-level scheduling policy. Nesting is not an issue due to this single instance.

In the Flux-based launch system for Merlin, the python subprocess call was replaced with a Flux `rpc_send` with a `job.submit` command that includes the environment and resource request for this job. This Flux instance can be augmented with a callback function that will be invoked on each status change of the submitted job so the workflow can be informed on all stages of the job submission: `submitted`, `completed`, `canceled`, and `failed`. This information can be sent back through the Merlin workflow to inform the system on the state of the simulation task. This Flux interface is independent of the native job launcher and provides a single interface for the user to configure a simulation launch.

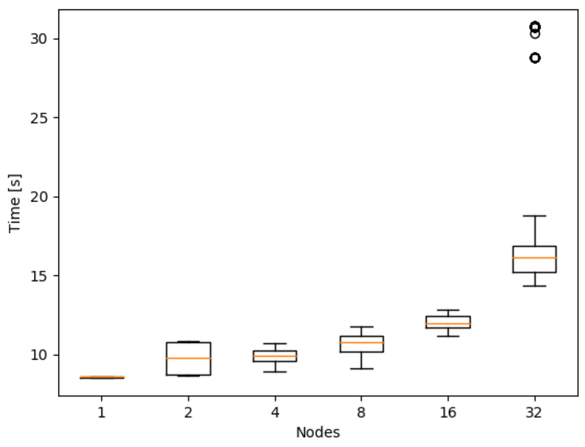


Fig. 5: Flux Latency

The Merlin tool has a number of components that can add performance overhead in terms of starting the tasks within the workflow. To characterize the latency for the Flux component of the workflow, we define a simple timing procedure. Our procedure is listed below:

- Compile a simple MPI hello world;
- Run the MPI hello world with varying input parameters;
- Run a python script to collect the job running overhead of Flux from its key value store service.

We use the Rztopaz machine at LLNL to measure the overheads. Rztopas is composed of 748 compute node, each with two Intel Xeon E5-2695 CPUs and is connected with Intel Omni-Path network. Each batch allocation runs a set of Merlin worker threads to request tasks from the task server that will be run on that allocation. An Rztopaz compute node has a total of 36 compute cores so Merlin runs 36 worker threads per node. On eight nodes, for instance, 288 threads will be run. The latency is defined as the difference between the time immediately before Flux's job launch command (`flux-wreck`) is invoked to the create time for each job. This difference is added up for all the jobs that are scheduled and executed in the end. The number of jobs or samples, run on each node is the same as the number of worker threads, so all samples can optimally run at the same time on the allocation without having to contend for compute cores among them. This also means that we use a weak scaling test where we increase the number of jobs as we increase the number of nodes being used for our experiments. Ideally, the experiment at each compute node count contains the simulations that would be completed in eight minutes if no scheduling or other overheads are added.

Fig. 5 shows that Flux's scheduling and launch performance overhead only account for less than 5 percent at 32 nodes in this configuration. Fig. 5 also shows that the increase in node count results in an increase of the max time for jobs to create. The time increase is relatively uniform (and as expected) up to 16 nodes: the more jobs need to be launched and some serial bottlenecks are hit. However, the 32-node run shows a number of jobs with large discrepancies from the average. We theorize that these are related to either the task server system within the Merlin workflow or other Flux-related latency. More experiments will need to be performed to determine the cause. Regardless, as part of future work, we plan to reduce the Flux-attributed overhead by exploring more Flux scheduler parallelism, leveraging a deeper Flux hierarchy than a single instance being used in this experiment.

C. Only Small Effort is Required to Gain Portability

We also evaluate the two main characteristics described in Section IV-D by integrating Flux into Merlin across two environments: LLNL's large clusters with Intel Xeon E5-2695v4, RedHat Enterprise Linux (RHEL) 7, and SLURM being resource manager and scheduler; and LLNL's Sierra pre-exascale system with a completely different environment: IBM POWER little endian, RHEL7, IBM JSM being the resource manager and LSF the system scheduler.

We first designed and implemented our initial integration, on one of LLNL's Intel Xeon E5-2695v4 Linux clusters. Then, we ported and customized Flux on Sierra while Merlin is being tested on the Intel Xeon Linux systems. Flux's porting efforts on Sierra are mainly threefold.

- Port a PMI library to PMIX because the PMI library, though a de facto standard, was not bundled with IBM's Spectrum MPI distribution;
- Compile our own `libhwloc` library to ensure GPUs are correctly discovered and used in our scheduling (The system-provided `libhwloc` was misconfigured such that its discovered GPU was not marked as Co-Processor, an attribute required for any scheduler to identify its element as a schedulable compute entity);
- Create an MPI plug-in within Flux for IBM Spectrum MPI to hide the passing of various environment variables to each MPI job in order to assist its bootstrapping.

While they require some communications with IBM, once the proper porting path is set, implementing these required changes was trivial.

Once Flux has been ported, porting Merlin code to Flux on the new platform required only minimal changes. While Merlin still uses Sierra's resource manager specific launcher (`jsrun`) to bootstrap a Flux instance per each batch allocation, once the instance is bootstrapped, Merlin uses the same Flux API and commands to perform its workflow. Further, Flux has been installed in public locations on both environments to further assist other workflows with portability.

VII. Related Work

This section presents a summary of the existing system and user-level solutions to workflow scheduling.

A. System-level Solutions

System-level solutions can be broken down into centralized, limited hierarchical, and decentralized schedulers. Centralized schedulers use a single, global scheduler that maintains and tracks the full knowledge of jobs and resources to make scheduling decisions. This scheduling model is simple and effective for moderate-size clusters, making it the state of the practice in most cloud and HPC centers today. Cloud schedulers such as Swarm [24] and Kubernetes [25] and HPC schedulers such as SLURM [9], MOAB [11], IBM LSF [10], and PBSPro [12] are centralized. While simple, these centralized schedulers are capped at tens of jobs/sec [30], provide limited to no support for co-scheduling of heterogeneous tasks [31], have limited APIs, and cannot be easily nested within other system schedulers.

Limited hierarchical scheduling has emerged predominantly in grid and cloud computing. This scheduling model uses a fixed-depth scheduler hierarchy that typically consists of two levels. The scheduling levels consist of independent scheduling frameworks stacked together, relying on custom-made interfaces to make them interoperable. Example implementations include the cloud computing schedulers Mesos [32] and YARN [33] as well as the grid schedulers Globus [34] and HTCondor [35]. Efforts to achieve better scalability in HPC have resulted in this model's implementation in some large HPC centers. For example, at LLNL multiple clusters are managed by a limited hierarchical scheduler that uses the MOAB grid scheduler on top of several SLURM schedulers, each of which manages a single cluster [36]. While this solution increases throughput over centralized scheduling, it's ultimately limited by its shallow hierarchy and the capabilities of the scheduling frameworks used at the lowest levels. In the case of LLNL example, all of the co-scheduling, coordination, and portability limitations of SLURM still apply.

Decentralized scheduling is the state-of-the-art in theoretical and academic efforts, but, contrary to centralized scheduling, it has not gained traction. To the best of our knowledge, decentralized schedulers are not in use in any production environment. Sparrow [37], in cloud computing, and SLURM++ [38] and Swift/T [39], in HPC, are existing decentralized schedulers. In decentralized scheduling, multiple schedulers each manage a disjoint subset of jobs and resources. The schedulers are fully connected and thus can communicate with every other scheduler. In this model, a scheduler communicates with other schedulers when performing work stealing and when allocating resources outside of its resource set (i.e., resources managed by another scheduler). Despite providing higher job throughput, decentralized schedulers suffer from many of the same problems as centralized schedulers: little to no support for co-scheduling of heterogeneous tasks and limited APIs. Additionally, cloud schedulers commonly make assumptions about the types of applications being run to improve performance. For example, Sparrow assumes that a common computational framework, such as Hadoop or Spark, is used by most of the jobs, enabling the use of long-running framework processes and lightweight tasks over short-lived processes and large application binaries [37].

B. User-level Solutions

User-level solutions can be broken down into application-level runtimes and workflow managers. Application-level runtimes work by offloading a majority of the task ingestion, scheduling, and launching from the batch job scheduler onto a user-level runtime. These application-level runtimes are typically much simpler and less sophisticated than the complex system-level schedulers described in VII-A but in exchange provide extremely high throughput. For example, CRAM provides no support for scheduling (i.e., once a task completes, the resources remain idle until all other tasks have completed), tasks requiring GPUs, or an API to query the status of tasks, but it can launch ~1.5 million tasks in ~19 minutes, resulting in an average job throughput of ~1,200 jobs/sec [40].

Workflow managers are designed to ease the composition and execution of complex workflows on various computing infrastructures, including HPC, grid, and cloud resources [41]. Example workflow managers include Pegasus [42], DAGMan [43], and the UQ Pipeline [22]. Workflows can be represented as a directed acyclic graph (DAG), as is the case with Pegasus and DAGMan, or a parameter sweep, as is the case with the UQP. Once the workflow has been specified by the user, the workflow manager handles moving data between and submitting the tasks to the various computing resources. Workflow managers provide an interface for users to track the status of their workflow, and provide portability across many types of computing infrastructures. While the use of a workflow manager can improve the overall workflow throughput by taking advantage of multiple, independent computing resources (e.g., clusters), they do not improve the job throughput or co-scheduling capabilities of any individual computing resource. Additionally, to submit and manage jobs in a portable way, many workflow managers incur expensive side-effects, such as the creation of millions of job status files [44].

VIII. Conclusion

Emerging scientific workflows present several system-level challenges. These include, but are not limited to, throughput, co-scheduling, job coordination/communication and portability across HPC systems. In this paper, we took a deep dive into upcoming workflows and described these four specific challenges that are becoming increasingly commonplace across modern workflows. Specifically, we show three workflow examples, MuMMI, the Uncertainty Quantification Pipeline, and the MLSI Merlin workflow. We then presented Flux, a hierarchical and open-source resource management and scheduling framework, as a common infrastructure that can address these challenges flexibly and efficiently. The core of Flux lies in its ability to be nested seamlessly within batch allocations created by other schedulers as well as itself. Once a hierarchy of Flux instance is created within each allocation, the rich set of well-defined, platform-independent APIs efficiently support advanced workflows that can often feature non-traditional execution patterns. Our results show the performance and functionality benefits of our approach as applied to various exascale workflow challenges. Future work involves performing diverse explorations in the directions of the workflow challenges that we presented in this paper, which includes developing a deeper understanding on the effect of scheduling specialization on more diverse sets of workflows, as well as enriching our scheduling infrastructure to support heterogeneous and multi-constraint resources with the help of an advanced data model.

Acknowledgment

We thank JDACS4C Pilot 2 team. Part of this work has been supported by the JDACS4C program established by the U.S. Department of Energy (DOE) and the National Cancer Institute (NCI) of the National Institutes of Health. For computing time, we thank Livermore Computing (LC) and Livermore Institutional Grand Challenge. This work was performed under the auspices of the U.S. Department of Energy by LLNL under contract DE-AC52-07NA27344 (LLNL-CONF-756663).

References

[1] S. H. Langer, B. Spears, J. L. Peterson, J. E. Field, R. Nora, and S. Brandon, "A hydra uq workflow for nif ignition experiments," in *Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, ser. ISAV '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ISAV.2016.6

[2] J. L. Peterson, "Machine learning aided discovery of a new nif design," Lawrence Livermore National Laboratory, August 2018.

[3] D. Wang1, X. Luo2, F. Yuan1, and N. Podhorszki, "A data analysis framework for earth system simulation within an in-situ infrastructure," *Journal of Computer and Communications*, vol. 5, no. 14, pp. 76–85, Dec. 2017. [Online]. Available: http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jcc.2017.514007

[4] M. Dorier, J. M. Wozniak, and R. Ross, "Supporting task-level fault-tolerance in hpc workflows by launching mpi jobs inside mpi jobs," in *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, ser. WORKS '17. New York, NY, USA: ACM, 2017, pp. 5:1–5:11. [Online]. Available: http://doi.acm.org/10.1145/3150994.3151001

[5] D. Higdon, R. Klein, M. Anderson, M. Berliner, C. Covey, O. Ghattas, C. Graziani, S. Habib, M. Seager, J. Sefcik, P. Stark, and J. Stewart, "Uncertainty quantification and error analysis," U.S. Department of Energy, Office of National Nuclear Security Administration, and the Office of Advanced Scientific Computing Research, Tech. Rep., Jan 2010.

[6] L. L. N. Laboratory, "Sierra," https://hpc.llnl.gov/hardware/platforms/sierra, Lawrence Livermore National Laboratory, August 2018, retrieved July 30, 2018.

[7] O. R. N. Laboratory, "Summit," https://www.olcf.ornl.gov/summit/, Oak Ridge National Laboratory, August 2018, retrieved July 30, 2018.

[8] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. K. Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P.-T. Bremer, J. N. Glosli, and H. I. Ingólfsson, "A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer," in *To appear in Supercomputing '19: The International Conference for High Performance Computing*, ser. SC '19, 2019.

[9] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2003.

[10] "IBM spectrum LSF," https://www.ibm.com/, 2017, retrieved April 03, 2017.

[11] "The Moab workload manager," http://www.adaptivecomputing.com/, 2017, retrieved April 03, 2017.

[12] "PBSPro: An HPC workload manager and job scheduler for desktops, clusters, and clouds," https://github.com/PBSPro/pbspro, Altair, 2018, retrieved August 8, 2018.

[13] I. A. Prior, P. D. Lewis, and C. Mattos, "A comprehensive survey of ras mutations in cancer," *Cancer Research*, vol. 72, no. 10, pp. 2457–2467, 2012.

[14] "MOOSE," https://moose.inl.gov/SitePages/Home.aspx.

[15] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond cpu millennium: A micron-scale atomistic simulation of kelvin-helmholtz instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, 2007.

[16] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19 – 25, 2015.

[17] S. J. Marrink, H. J. Risselada, S. Yefimov, D. P. Tieleman, and A. H. de Vries, "The MARTINI Force Field: Coarse Grained Model for Biomolecular Simulations," *The Journal of Physical Chemistry B*, vol. 111, no. 27, pp. 7812–7824, Jul. 2007. [Online]. Available: https://pubs.acs.org/doi/10.1021/jp071097f

[18] T. A. Wassenaar, H. I. Ingólfsson, R. A. Böckmann, D. P. Tieleman, and S. J. Marrink, "Computational Lipidomics with insane: A Versatile Tool for Generating Custom Membranes for Molecular Simulations," *Journal of Chemical Theory and Computation*, vol. 11, no. 5, pp. 2144–2155, May 2015.

[19] F. D. Natale, "Maestro workflow conductor (maestrowf)," https://github.com/LLNL/maestrowf, Lawrence Livermore National Laboratory, August 2018, retrieved Aug 11, 2018.

[20] T. S. Carpenter, C. A. Lopez, C. Neale, C. Montour, H. I. Ingólfsson, F. Di Natale, F. C. Lightstone, and S. Gnanakaran, "Capturing Phase Behavior of Ternary Lipid Mixtures with a Refined Martini Coarse-Grained Force Field." *Journal of Chemical Theory and Computation*, vol. 14, no. 11, pp. 6050–6062, Nov. 2018.

[21] L. Schneidenbach, C. Misale, B. D'Amora, and C. Costa, "Ibm data broker," https://github.com/IBM/data-broker, 2019.

[22] T. L. Dahlgren, D. Domyancic, S. Brandon, T. Gamblin, J. Gyllenhaal, R. Nimmakayala, and R. Klein, "Poster: Scaling uncertainty quantification studies to millions of jobs," in *Proceedings of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2015.

[23] L. L. N. Laboratory, "Non-intrusive uncertainty quantification: Psuade," https://computation.llnl.gov/projects/psuade-uncertainty-quantification/, Lawrence Livermore National Laboratory, August 2018, retrieved August 3, 2018.

[24] "Swarm: a docker-native clustering system," https://github.com/docker/swarm, Docker Inc., 2017, retrieved April 03, 2017.

[25] "Kubernetes by Google," http://kubernetes.io, 2017, retrieved April 03, 2017.

[26] J. Gyllenhaal, T. Gamblin, A. Bertsch, and R. Musselman, "Enabling high job throughput for uncertainty quantification on bg/q," in *IBM HPC Systems Scientific Computing User Group*, ser. ScicomP'14, Chicago, IL, 2014.

[27] J. Gaffney, P. Springer, and G. Collins, "Thermodynamic modeling of uncertainties in NIF ICF implosions due to underlying microphysics models," *Bulletin of the American Physical Society.*, October 2014.

[32] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011,

[28] J. Gaffney, D. Casey, D. Callahan, E. Hartouni, T. Ma, and B. Spears, "Data driven models of the performance and repeatability of NIF high foot implosions," *Bulletin of the American Physical Society.*, November 2015.

[29] "Inertial confinement fusion," https://en.wikipedia.org/wiki/Inertial_confinement_fusion, Wikipedia, 2017, retrieved August 22, 2017.

[30] K. Wang, "Slurm++: A distributed workload manager for extreme-scale high-performance computing systems," http://www.cs.iit.edu/~iraicu/teaching/CS554-S15/lecture06-SLURM++.pdf, Feb 2015.

[31] "SLURM heterogeneous jobs: Limitations," https://slurm.schedmd.com/heterogeneous_jobs.html#limitations, SchedMD, Dec 2017, retrieved August 8, 2018.

pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[33] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[34] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of High Performance Computing Applications*, vol. 11, no. 2, pp. 115–128, Jun. 1997. [Online]. Available: http://dx.doi.org/10.1177/109434209701100205

[35] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[36] B. Barney, "Slurm and moab," https://computing.llnl.gov/tutorials/moab, Lawrence Livermore National Laboratory, August 2017, retrieved August 22, 2017.

[37] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[38] X. Zhou, H. Chen, K. Wang, M. Lang, and I. Raicu, "Exploring distributed resource allocation techniques in the SLURM job management system," Illinois Institute of Technology, Department of Computer Science, Tech. Rep., 2013.

[39] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, ser. CCGrid, May 2013, pp. 95–102.

[40] J. Gyllenhaal, T. Gamblin, A. Bertsch, and R. Musselman, "Enabling high job throughput for uncertainty quantification on BG/Q," in *IBM HPC Systems Scientific Computing User Group (ScicomP)*, May 2014.

[41] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, no. 3, pp. 171–200, Sep 2005. [Online]. Available: https://doi.org/10.1007/s10723-005-9010-8

[42] E. Deelman, G. Singh, M. hui Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, Dec 2005.

[43] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, *Workflow Management in Condor*. London: Springer London, 2007, pp. 357–375. [Online]. Available: https://doi.org/10.1007/978-1-84628-757-2_22

[44] S. Hebrein, T. Patki, D. H. Ahn, D. Lipari, T. Dahlgren, D. Domyancic, and M. Taufer, "Poster: Fully hierarchical scheduling: Paving the way to exascale workloads," in *Proceedings of the 29th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2017.