An Analytical Performance Model of Generalized Hierarchical Scheduling

Journal Title
XX(X):1–18

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Stephen Herbein[1], Tapasya Patki[1],Dong H. Ahn[1], Sebastian Mobo[2], Clark Hathaway[2], Silvina Caíno-Lores[2], James Corbett[1], David Domyancic[1], Thomas R. W. Scogland[1], Bronis R. de Supinski[1], and Michela Taufer[2]

Abstract

High performance computing (HPC) workflows are undergoing tumultuous changes, including an explosion in size and complexity. Despite these changes, most batch job systems still use slow, centralized schedulers. Generalized hierarchical scheduling (GHS) solves many of the challenges that face modern workflows, but GHS has not been widely adopted in HPC. A major difficulty that hinders adoption is the lack of a performance model to aid in configuring GHS for optimal performance on a given application. We propose an analytical performance model of GHS, and we validate our proposed model with four different applications on a moderately-sized system. Our validation shows that our model is extremely accurate at predicting the performance of GHS, explaining 98.7% of the variance (i.e., an R^2 statistic of .987). Our results also support the claim that GHS overcomes scheduling throughput problems; we measured throughput improvements of up to $270\times$ on our moderately-sized system. We then apply our performance model to a pre-exascale system, where our model predicts throughput improvements of four orders of magnitude and provides insight into optimally configuring GHS on next generation systems.

Keywords

High performance computing, workflow performance, ensemble workflows, hierarchical scheduling, performance modeling

1 Introduction

The landscape of High Performance Computing (HPC) is rapidly changing. Current and expected future scientific workflows in HPC stray significantly from the traditional set of a few large, homogeneous, long-running applications. An emerging class is *ensemble workflows*, which execute several runs of the same application with varying inputs or resources, typically simultaneously. Many computational domains, including climate science Murphy et al. (2004), epidemics Eriksson et al. (2011), molecular dynamics Phillips et al. (2005), and fusion Gaffney et al. (2014) use ensembles. Our analysis of jobs on a major cluster at a US Department of Energy Laboratory shows that 48.1% of jobs were submitted as part of an ensemble (that is, 100 or more jobs submitted from the same user, with more than half submitted within one minute of each other).

Traditionally, ensembles consist of between 100 to 10,000 jobs Gyllenhaal et al. (2014); Dahlgren et al. (2015) but they have grown to as many as 100 million jobs on pre-exascale systems Peterson et al. (2019). The sheer scale of these workflows presents major challenges to existing HPC batch schedulers. One popular HPC scheduler crashes beyond 20,000 simultaneously running jobs Gyllenhaal et al. (2014). Fundamentally, today's schedulers largely remain stuck with a centralized paradigm in which a single scheduler manages every job and every resource "OLCF policy guide" (2019); Hines (2015); Barney (2017), resulting in scalability issues. Paradoxically, as HPC systems have grown larger and faster, their scheduling resources have mostly remained stagnant.

A plethora of workflow frameworks attempt to address the bottleneck that existing centralized schedulers impose "Computational data analysis workflow systems" (2020). by implementing a custom workflow scheduler. This solution to the scalability issues requires framework developers to invest considerable effort on code development and maintenance, which is often duplicated for every workflow framework. Ad-hoc solutions also place a significant burden on workflow users, who must now port their application from the system's default scheduler to the custom workflow scheduler.

Generalized Hierarchical Scheduling (GHS) addresses these scalability and customization challenges systematically, generically, and portably Ahn et al. (2020); Di Natale et al. (2019); Peterson et al. (2019). Figure 1 compares GHS to the typical HPC centralized scheduling paradigm. The GHS divide-and-conquer technique allows many scheduler instances to allocate jobs cooperatively on a distributed system. The root GHS scheduler spawns child sub-schedulers that control a subset of the resources and jobs. This scheduler

[1]Lawrence Livermore National Laboratory
[2]University of Tennessee, Knoxville

Corresponding author:
Stephen Herbein, Lawrence Livermore National Laboratory, Livermore, CA, USA
Email: herbein1@llnl.gov

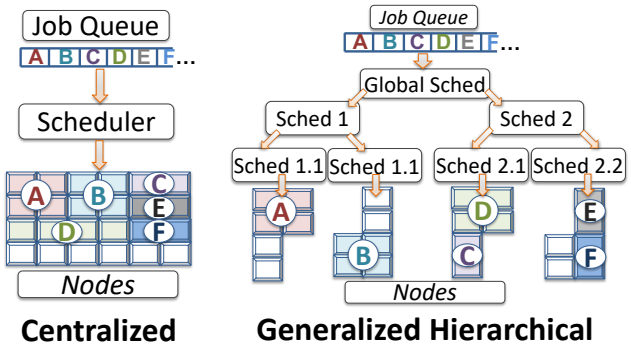


Figure 1. Centralized versus Generalized Hierarchical Scheduling

spawning continues recursively to a system or user-specified depth to form a tree of sub-schedulers. GHS improves overall job throughput by adding scheduler parallelism, simplifying the larger scheduling problem and limiting the communication required between sub-schedulers.

A challenge in GHS deployment and use is determining an effective scheduler tree to use for a given ensemble workflow on a given system. Before we can optimize the scheduler tree, we must first have a performance model that quantifies the benefits of GHS on various ensemble applications, the costs of creating a given scheduler tree, and the scale-dependent bottlenecks that schedulers face. In this paper, we address this performance modeling challenge. Our contributions are:

- A formal definition of GHS and its underlying principles;
- An analytical performance model to capture the performance properties of GHS;
- Validation of our analytical model on Flux Ahn et al. (2020), a production scheduling software that embodies GHS, with four different ensemble applications;
- Demonstration of our validated analytic model as a tool for GHS performance exploration on extreme-scale systems; and
- Initial insights into efficiently selecting the optimal GHS tree for applications running on pre-exascale systems

The rest of the paper is organized as follows. Section 2 defines hierarchical schedulers and the underlying principles of GHS. We then present our GHS performance model and its assumptions in Section 3. Sections 4 and 5 detail the validation of our model for a range of representative ensemble workloads. Section 6 projects pre-exascale GHS performance.

2 Generalized Hierarchical Scheduling

Traditional hierarchical scheduling attempts to overcome the limitations of centralized scheduling. In the hierarchical model, there are typically two hierarchy levels. At the top-level, a central meta-scheduler interacts with few local schedulers at the lower level to create efficient schedules that honor both global and local constraints. This model has emerged predominately in grid and cloud computing and is also taking hold in HPC. Example implementations include the cloud computing schedulers Mesos Hindman et al. (2011), YARN Vavilapalli et al. (2013) and Omega Schwarzkopf et al. (2013) as well as the grid schedulers Globus Foster and Kesselman (1997) and Legion Chapin et al. (1999). Examples from HPC include Parsl Babuji et al. (2019), Radical Pilot Merzky et al. (2018), Cram Gyllenhaal et al. (2014) and MPIwithinMPI Wozniak et al. (2019).

With a push towards larger and complex workflows on HPC, however, the traditional hierarchical schemes have also begun to be hard-pressed with respect to scalability and flexibility. In response, an alternative model that *generalizes* the concept of hierarchical scheduling emerged. In this model, any scheduler instance can spawn child instances to aid in scheduling, launching, and managing jobs. The parent scheduler instance grants a subset of its jobs and resources to each child. This parent–child relationship can extend to an arbitrary depth and width, creating many opportunities for scheduling parallelization. This alternative model has already proven to be effective on emerging HPC workflows Ahn et al. (2020); Di Natale et al. (2019); Peterson et al. (2019). Unfortunately, it lacks a formal definition. Thus, we first define its guiding principles with the goal of later using them for our analytic performance model.

2.1 Defining Principles

Three principles govern GHS in its purest form:

1. **Hierarchical Bounding**: A parent scheduler grants job and resource allocations to its children.
2. **Instance Effectiveness**: Each scheduler can be configured independently and is solely responsible for the most effective use of its resource set.
3. **Arbitrary Recursion**: The first two principles apply recursively from the top of the resource hierarchy (for instance, the entire HPC center) down to any arbitrarily small subset of resources.

Hierarchical Bounding improves scalability in two ways. First, it reduces the number of resources that each scheduler must consider, which improves the performance of each individual scheduler. Second, it enables schedulers to delegate work to child schedulers, spreading the load across many independent schedulers, improving collective performance.

Instance Effectiveness enables the customization of schedulers for specific workflows and bounds communication costs. The top-level scheduler may be a system-wide scheduler with a generic policy running by default, but it can create a child scheduler for each new workflow, which enables the workflows to customize the scheduler to their exact needs. This customization includes not only the scheduling policy and configuration parameters of the scheduler but also the number of children that the scheduler creates. This principle also bounds the communication costs of parallelization by only requiring schedulers to directly interact with their parent and children, as opposed to an all-to-all communication structure.

Arbitrary Recursion amplifies the scalability and flexibility provided by the previous two principles. It allows

for the creation of the appropriate number of schedulers for each workflow. Specifically, this principle enables large ensemble workflows to create more schedulers and thus improve throughput without incurring excessive parallelization overhead. Schedulers within GHS can even be dynamically created and destroyed depending on how workloads change over time.

2.2 Implementation Restrictions

While our principles describe GHS in its purest form, specific implementations can impose restrictions and provide a partial implementation of GHS. Commonly, implementations enforce homogeneous scheduler configuration across all levels or limit the nesting of schedulers to a fixed number of levels. For example, Mesos strongly enforces a limit of two hierarchy levels. Its top-level scheduler can spawn an arbitrary number of independent schedulers. However, these second-level schedulers, which usually target a specific workload type such as big data analytics, long-running services and batch jobs, cannot spawn other schedulers. In Mesos, system administrators configure both levels; it does not support user customization. Similarly, Parsl and Radical Pilot impose a limit of two hierarchy levels in which the second level has exactly one scheduler per node. Unlike Mesos, all second-level schedulers in Parsl and Radical Pilot must use a homogeneous configuration.

Orthogonally to Mesos, Parsl, and Radical Pilot, Flux does not impose any restrictions on the number of levels or the amount of configuration at each level. In this work, we choose Flux as our representative hierarchical scheduler because of its flexibility and lack of artificial restrictions on our GHS principles. While we use Flux to demonstrate and to validate our analytical model, our model is general; it captures the performance factors of any hierarchical scheduler. If Mesos, Parsl, or Radical Pilot are extended to remove their restrictions on homogeneity and number of levels, then our model will capture their performance and can be used for their optimization.

3 Modeling

The broad flexibility of GHS creates a challenge to find the combination of scheduler tree levels and total number of schedulers that maximizes performance for a given workflow. As the first step towards determining these scheduler configurations *a priori*, we propose an analytical model that characterizes the various GHS performance factors. For convenience and ease of reference, we list the input parameters to our model and their descriptions in Table 1.

Table 1. Input parameters to our analytical model

Parameter	Description
J	number of jobs
X	set of resources
x	resource requirements of a job
$Runtime(X_{usage})$	runtime of a job on a node that is X_{usage} percent utilized by other jobs
$AvgSchedRate(X)$	average decision rate, in jobs per second, that a given scheduler implementation achieves at steady-state when considering X resources
$Init(X)$	cost to start a new scheduler on X resources
$Shutdown(X)$	cost to terminate a scheduler on X resources

3.1 Simplifying Assumptions

Our model makes six simplifying assumptions. First, we assume that jobs are rigid in their resource request, that is they have fixed resource requirements over their entire life cycle. This assumption holds for most HPC jobs, with effective dynamic allocation support in resource managers and parallel runtimes being an active area of research Prabhakaran et al. (2018). Second, we assume that all jobs are submitted at the start of the workflow. This assumption is made by many HPC workflows, including all DAG-based Kwok and Ahmad (1999); Deelman et al. (2015) and most ensemble workflows. Third, we assume that jobs have no dependencies between them. Fourth, we assume that all jobs are homogeneous in their runtime and resource requirements. The third and fourth assumptions are common for our main motivation, ensemble workflows, since they, by definition, execute the same application with the same resources and time requirements many times in an embarrassingly parallel fashion.

Fifth, we assume that jobs and resources are distributed uniformly across every scheduler at a given level within the GHS tree. And finally, we assume that the schedulers within the GHS have low variance in scheduling performance such that the average scheduling rate at steady-state is representative of the scheduling rate over the lifetime of the workflow. The fifth and sixth assumptions concern the uniformity and consistency of the GHS implementation; and to the best of our knowledge, they hold for all existing GHS implementations.

While we make these six assumptions to simplify our model, we believe that our model as-is holds significant value to the community due to the prevalence and importance of ensemble workflows on modern HPC systems. As we described in the introduction, 48.1% of jobs run on a major cluster at a US Department of Energy Laboratory meet assumptions 1, 3, and 4, and approximately half of those jobs also meet assumption 2. So, we estimate that approximately a quarter of all jobs submitted to the analyzed cluster match the four job-related assumptions that we make. Furthermore, in Section 3.4, we describe in detail how our model can be generalized to additional workflows and schedulers by relaxing these six assumptions.

3.2 Single Scheduler Instance

To model GHS, we must understand and model the events on the critical path of a single scheduler instance. Table 2 lists the critical path events when a scheduler launches a job.

We group these events into three phases: initialization, scheduling, and shutdown. The initialization phase consists of the one-time events related to starting the scheduler ($e_0 - e_3$). The scheduling phase includes the events that occur for every job in the workflow ($e_4 - e_8$). The shutdown phase consists of one-time events related to terminating the

Table 2. Critical path events of a scheduler

Event	Description
e_0	scheduler instance starts
e_1	scheduler populates memory with resource information
e_2	scheduler ready for jobs
e_3	workflow is submitted to the scheduler
e_4	scheduler processes next job in queue
e_5	scheduler allocates resources for job
e_6	execution system launches job
e_7	job completes; execution system notifies scheduler
e_8	scheduler frees resources associated with job
e_9	all jobs completed; workflow completes
e_{10}	scheduler shuts down

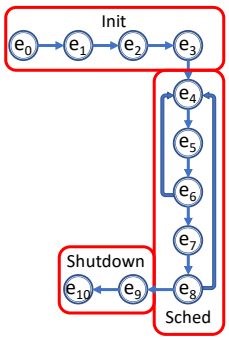


Figure 2. Single scheduler critical path events

scheduler ($e_9 - e_{10}$). Figure 2 shows this critical path for the launch of a workflow with multiple jobs.

A single scheduler model must accurately capture the initialization, scheduling, and shutdown phases. The initialization and shutdown phases have one-time, static costs and are easily measured by starting the scheduler, launching no jobs, and terminating it immediately. In Section 4, we discuss our methodology for this empirical measurement, which takes only a few seconds on a 32-node system.

Modeling the scheduling phase of a single scheduler ($Sched_1$) is decidedly more complex than the other two static cost phases. One of two factors limits the performance of the scheduling phase of ($Sched_1$): the scheduling decision rate or the resource capacity. Only one of these factors is the bottleneck for a given workflow, so we can define the cost of scheduling J jobs on X resources with a single scheduler ($Sched_1(J, X)$) as the maximum of these two potential limiting factors.

The first potential limiting factor is the rate at which the scheduler makes scheduling decisions. Because finding the minimal makespan for J arbitrarily-sized, rigid jobs on M identical machines is NP-Hard Wang and Cheng (1992), schedulers that attempt to make close-to-optimal decisions may suffer from a low scheduling rate. Many HPC schedulers support multiple scheduling policies (for instance, First-Come First-Served, EASY Backfilling, or Fair Share), each with their own performance implications, which increases the complexity of this factor. Additionally, the quality of the implementation matters for the performance of the scheduler.

When the decision rate is the limiting factor, the cost of scheduling in wall-clock time is $\frac{J}{AvgSchedRate(X)}$. J is the number of jobs in the workflow and $AvgSchedRate(X)$ is the average decision rate, in jobs per second, that a given scheduler implementation achieves at steady-state when considering X resources. The scheduling rate can also depend on many parameters of the implementation. For example, scheduling queue depth is a common parameter in scheduling algorithm implementations that limits the number of jobs the scheduler considers each scheduling loop, bounding the scheduling time, improving performance and improving consistency. Modeling the performance of every popular scheduling algorithm and their individual parameters is beyond the scope of this paper. Instead, $AvgSchedRate$ implicitly captures queue depth and other parameters of the scheduling algorithm implementation. We demonstrate how to determine this parameter empirically, cheaply and easily for most scheduling configurations in Section 4.

The other potential limiting factor is the resource capacity constraints of the system. Regardless of the scheduler's decision rate, if the system has few resources (such as cores, memory, or storage) or the workflow's jobs each require many resources, then the job scheduling rate will be low. The value of $Sched_1$ when resource capacity is the limiting factor is given by the expected makespan of scheduling all J jobs on X resources. As stated earlier, this problem in its most general form is NP-Hard.

However, because we assume a homogeneous set of jobs, the expected makespan is $\lceil \frac{J*x}{X} \rceil * Runtime(X_{usage})$ where J is the number of jobs in the workflow, x is the resource requirement of a job, X is the total resource capacity, and $Runtime(X_{usage})$ is the runtime of a job when run on a node that is X_{usage} percentage filled with other jobs.

Our definition of $Runtime$ includes runtime inflation due to inter-job interference or contention (for example, interference resulting from memory-bound applications running on the same node). Lacking any empirical data on runtime inflation, X_{usage} is set to zero so that $Runtime(X_{usage})$ is the runtime of the jobs when run in isolation. We address how to collect empirical data and derive the inflated application runtime from X_{usage} in Section 4.4.

We combine these two limiting factors to produce an equation that models the cost in wall-clock time of scheduling a set of jobs (J) on a set of resources (X) in an already running scheduler, as shown in Equation 1.

$$Sched_1(J, X) = max(\frac{J}{AvgSchedRate(X)}, \left\lceil \frac{J * r}{X} \right\rceil * Runtime(X_{usage})) \quad (1)$$

The models of the phases of the scheduler's critical path are combined to model the total end-to-end cost of scheduling J jobs on X resources with a single scheduler:

$$Makespan_1(J, X) = Init(X) + Sched_1(J, X) + Shutdown(X) \quad (2)$$

The makespan cost can easily be converted into job throughput: $JobThroughput_1 = \frac{J}{Makespan_1}$.

3.3 Modeling a Hierarchical Scheduler

We build our GHS performance model on top of and similarly to our single scheduler performance model.

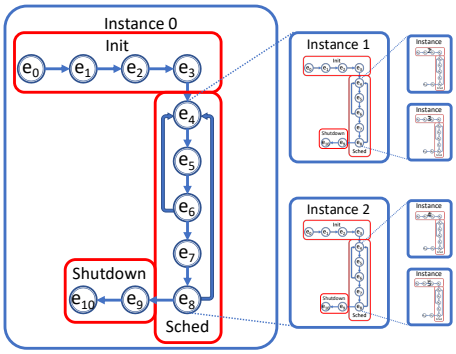
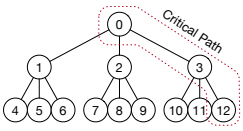


Figure 3. GHS critical path events



(a) Critical path of three scheduler levels

Time	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
$Inst_0$	1	1	2	2	3	3						
$Inst_1$			4	4	5	5	6	6				
$Inst_2$					7	7	8	8	9	9		
$Inst_3$							10	10	11	11	12	12

(b) Last scheduler at each level is on the critical path

Figure 4. Analysis of launching multiple nested instances

Figure 3 shows the critical path events for a hierarchical scheduler. This critical path analysis yields three key observations. First, we must capture the cost to start a hierarchical scheduler, which is the cost to launch the scheduler tree. Second, from the perspective of a higher-level instance, the nested scheduler instances are equivalent to jobs. Third, the critical event paths of each leaf scheduler are independent.

Figure 4 analyzes the cost associated with GHS tree creation, with the critical path of tree creation highlighted in red. Figure 4a shows how the root scheduler ($Inst_0$) schedules three nested scheduler instances that further schedule three additional schedulers. If it takes two time units to schedule and launch a nested scheduler, Figure 4b shows that the critical path consists of the scheduling cost of $Inst_0$ and the scheduling and startup cost of the last spawned scheduler ($Inst_3$). Thus, the cost of scheduling jobs under GHS is the cost to create the tree plus the job scheduling cost at only the last scheduler to be created (which is instance 12 in Figure 4a).

We further define these costs analytically in terms of three factors: the cost to create the GHS tree, ($Create$); the scheduling parallelism in the tree, (P); and the scheduling performance of single instance, $Sched_1$ (Equation 1).

To define the tree creation cost $Create(Tree)$, we use our second observation from the critical path analysis: launching J sub-schedulers is the same as launching J jobs. Thus, we define the cost to create the next level in the tree in terms of $Sched_1(J, X)$ where J is the number of sub-schedulers to launch and X is the number of resources managed by the current scheduler. Specifically, the total tree creation cost is:

$$\begin{aligned} Create(Tree_n, X) =& Sched_1(Tree_n, \tfrac{X}{Tree_n}) \\ &+ Create(Tree_{n-1}, \tfrac{X}{Tree_n}) \end{aligned} \quad (3)$$

where $Tree_i$ is the branching factor at the ith level (for example, for the tree in Figure 4a, $Tree_2$ and $Tree_1$ are 3 and $Tree_0$ is 1) and $\frac{X}{Tree_n}$ is the resource set given to each nested scheduler instance by its parent. The cost of creating a deeper tree is captured by additional recursion of the $Create$ function and creating a wider tree is captured by the increased size of $Tree_n$ passed to $Sched_1$.

We now define the scheduling cost at the last scheduler created as a function of the scheduler parallelism in the tree. Based on our assumption that the resource and job allocations at each level of the tree are uniform, it is $Sched_1(\frac{J}{P(Tree)}, \frac{X}{P(Tree)})$, where $P(Tree)$ is the number of schedulers in a given tree that are applied to user workflow jobs. In our GHS implementation, only the leaves of the tree schedule user jobs. In this implementation, $P(Tree)$ is:

$$P(Tree_n) = \prod_{i=0}^{n} Tree_i \quad (4)$$

An implementation that uses every scheduler in the tree to schedule user jobs would define $P(Tree)$ as:

$$P(Tree_n) = \prod_{i=0}^{n} Tree_i + P(Tree_{n-1}) \quad (5)$$

Thus our overall GHS model is:

$$\begin{aligned} Sched_{tree}(J, X) =& Create(Tree) \\ &+ (Sched_1(\frac{J}{P(Tree)}, \frac{X}{P(Tree)})) \end{aligned} \quad (6)$$

3.4 Generalizing the Model

Modeling GHS as it applies to every possible workflow in HPC is outside the scope of this paper, but in this section, we describe how prior work can be integrated into our model to relax or remove the simplifying assumptions we make in Section 3.1.

The first four assumptions that we make constrain the job type and simplify the expected makespan calculation (i.e., $\lceil \frac{J*r}{R} \rceil * Runtime(R_{usage})$). In its most general form, the workflow makespan calculation is NP-Hard, but many heuristics and alternative calculations exist for this problem. Relaxing any or all of the first four assumptions in our model requires replacing our workflow makespan calculation with a suitable alternative. Table 3 provides references to existing alternatives that do not rely on a given assumption or combination of assumptions and thus can be used in place of our expected makespan calculation when necessary. For example, in the case of relaxing assumptions one through four, a discrete-event simulation of the workflow on X resources would be used to estimate the makespan of the workflow.

Generally speaking, the costs of the alternative calculations presented in Table 3 are polynomial with respect to the number of jobs, the number of resources, and the number of dependencies. The cost of applying these calculations directly to a workflow with millions of jobs running on hundreds of thousands of cores would be large. Fortunately, the divide-and-conquer approach that GHS uses makes this problem tractable by cutting down the number of jobs and

Table 3. Alternative calculations that can be used in place of our workflow makespan calculation

Assumption Relaxation	Reference
1: non-rigid jobs	Wang and Cheng (1992); Marchal et al. (2018)
2: jobs submitted over time	Manimaran and Murthy (1998)
3: inter-job dependencies	Min-You Wu et al. (2001)
4: heterogeneous jobs	Sahni (1976); Hochbaum and Shmoys (1987); Wang and Cheng (1992)
1+3+4: static DAG scheduling	El-Rewini and Lewis (1990); Cosnard et al. (2004); Jing-Chiou Liou and Palis (1998); Tao Yang and Gerasoulis (1994)
1+2+3+4: scheduler simulation	Simakov et al. (2018); Jokanovic et al. (2018); Klusáček et al. (2020); Rodrigo et al. (2018) Dutot et al. (2017); Eleliemy and Ciorba (2021)

resource per scheduler by a factor of $P(Tree_n)$. Furthermore, if the fifth simplifying assumption of a uniform distribution of jobs across the tree can be maintained, the heuristic need only be applied to one leaf scheduler, and the results can be mirrored across the other leaves due to the symmetry of the leaf schedulers.

Relaxing the fifth assumption requires modifying the recurrence relation in Equation 6 to give a number of jobs and resources to each child scheduler that is function of its position in the tree. If the GHS being modeled uses a static distribution of jobs, then these changes are sufficient, but if the GHS implements dynamic job distribution at runtime, an additional factor may need to be added to model the rate that jobs are routed and propagated through the tree. A dynamic job distribution would also help with the relaxation of the other assumptions, as the runtime load redistribution would mitigate errors from the workflow makespan estimation methods described above and referenced in Table 3. Finally, relaxing the sixth assumption on scheduler performance variability requires changing our $AvgSchedRate(X)$ factor to match the performance characteristics of the scheduler being modeled.

4 Model Validation

4.1 Experimental Setup

Applications: We validate our model using four applications. Each application is chosen to add incremental complexity to our modeling, enabling us to isolate and evaluate the accuracy of different factors within our model. The simplest application is an ensemble of *sleep 0* commands. Using *sleep 0* allows us to study the accuracy of our model without considering the costs associated with application runtime, scheduler-application contention, or inter-application contention. Our second application is an ensemble of *sleep 5* commands, which adds runtime as a factor while remaining free of any potential contention. Our third application is the *firestarter* processor stress test that is designed to saturate the execution units of the processor and constantly force the CPU into its highest power state Hackenberg et al. (2013). *Firestarter* supports an execution mode where it runs for a set time and then exits. This fixed-time execution mode allows us to study the contention that occurs between a CPU-intensive application and the scheduler without the runtime inflationary effects of inter-application contention. Lastly, our fourth application is the *stream* memory benchmark, which measures the sustainable memory bandwidth of a node McCalpin (1995). *Stream* executes a configurable number of iterations, each with a fixed amount of work. This fixed-work execution mode allows us to study all of the factors of our analytical model, including runtimes, scheduler-application contention, and inter-application contention.

Application Configurations: The *sleep 0* and *sleep 5* applications require no special flags or configuration when running. We configure the *firestarter* application to run for a fixed time of 5 seconds, to match the *sleep 5* application's runtime. We tune the *stream* application parameters so that it executes in approximately 5 seconds on an uncontended node. On our system, this means we configure *stream* to execute 70 iterations with an array size of 10,000,000 elements.

Hardware: We perform all of our validation tests on a 32-node cluster. Each node has 128 GB of RAM and dual-socket Intel Xeon CPU E5-2695 v4 processors, for a total of 36 cores and 72 hardware threads per node. All of the experiments below are run using all 32 nodes in the cluster.

Software: We use Flux as the representative hierarchical schedulerAhn et al. (2020). In particular, Flux has a utility called *flux-tree* that handles the creation of an arbitrary scheduler tree as well as the execution of an ensemble workflow across that tree. We use the default settings for Flux and its scheduler.

Model Input Parameters: Table 4 summarizes the values we use for our model's input parameters. For three of the parameters ($Init(X)$, $Shutdown(X)$, $AvgSchedRate(X)$), we measure their values on a single Flux instance with an X of 1 and 32 nodes. Specifically, for $Init + Shutdown$ we time how long it takes for a single Flux instance to startup and immediately shutdown. For $AvgSchedRate(X)$, we time how long it takes to schedule at least 3x the number of jobs that the given resources can handle (e.g., for a 1,152 core system, we schedule 3,456 single-core jobs) and use this time to calculate the average job throughput (i.e., $NumJobs/Makespan$). For a single node, we measure $Init + Shutdown$ to be 3.1 seconds and the $AvgSchedRate$ to be 4.5 jobs per second. For 32 nodes, we measure $Init + Shutdown$ to be 4.3 seconds and $AvgSchedRate$ to be 4.4 jobs per second. Given the relatively insignificant difference in the measured parameter values between 1 and 32 nodes, we make all three factors constant with respect to X for our evaluations in this section.

Scheduler Tree: For our real-world experiments, we test three different scheduler trees: *1*, *1x32*, and *1x32x36*. The scheduler trees that we choose map directly onto the hardware topology of our cluster and represent a single possible configuration within a large parameter space. The first tree, *1*, is a single scheduler instance running across all of the nodes. The second tree, *1x32*, is two levels deep with a single scheduler managing the entire cluster at the first level

Table 4. Model input values for our experimental setup

Parameter	Value
J	varies from 1 to 1 million
X	1152 cores
x	1 core
X_{usage}	0
$Runtime(0)$	*Sleep 0*: 0
	Sleep 5, Firestarter, Stream: 5
$AvgSchedRate(X)$	4.5
$Init(X) + Shutdown(X)$	4.3

and 32 schedulers at the second level, each managing a single node. Our third tree, *1x32x36*, is three levels deep, where the first two levels are the same as our *1x32* tree, and at the third level, there are 36 schedulers nested within each second level scheduler. Each third level scheduler is managing a single physical core.

4.2 Hierarchical Scheduling

We first begin by analyzing the real-world performance of hierarchical scheduling and its benefits for workflows. Table 5 presents the peak job throughput achieved with each tree and the percentage of the theoretical maximum throughput achieved (when applicable). For *Sleep 0*, the theoretical maximum throughput is infinite, and thus the percentage of maximum throughput is not meaningful. For *Sleep 5* and *Firestarter*, the runtime of these fixed-time applications is known and thus the theoretical maximum throughput, assuming a zero-overhead scheduler, can be calculated using the second factor in $Sched_1(J, X)$, which is $\lceil \frac{J*x}{X} \rceil * Runtime(X_{usage})$. The *Stream* application is a fixed-work application that experiences significant runtime inflation under contention. Therefore, its $Runtime(X_{usage})$ is not well defined analytically and the theoretical maximum throughput is unknown.

There are three important trends in these performance numbers. First, the use of hierarchical scheduling results in a massive throughput improvement over a single scheduler for all four applications. Second, hierarchical scheduling achieves a significant percentage of the theoretical maximum throughput for the two applications where the maximum theoretical throughput is known and finite. Third, and most important for the motivation of our analytical model, the deepest tree is not always necessary. For longer running applications or applications sensitive to inter-application contention such as *Stream*, a shallower tree may be more appropriate.

4.3 Analytical Model

We now apply our analytical model — built with the minimal amount of empirical data possible — and compare the accuracy against known performance values. As stated previously, we measured the $Init$, $Shutdown$, and $AvgSchedRate$ factors of a single Flux instance. For the runtime of our applications ($Runtime(X_{usage})$), we use 0 seconds for *Sleep 0* and 5 seconds for *Sleep 5*, *Firestarter*, and *Stream*, with the assumption that no inter-job interference occurs (such that $X_{usage} = 0$).

Figure 5 shows our analytical model's predictions when applied to all four applications for varying sized workflows and compares those model predictions against the actual performance numbers. The actual performance numbers are plotted with solid lines and marked with either red circles, blue triangles, or green squares for the *1*, *1x32*, and *1x32x36* trees respectively. The model predictions are plotted with dashed lines with the same color mapping as the real values. For both *Sleep 5* and *Firestarter*, the theoretical throughput limit is also plotted with a blacked dashed line. We ran each experiment three times and plot the median value with error bars for the minimum and maximum values, but the variance across our tests is small enough that the error bars are not visible behind the plot markers.

Using just these four measured constants (that is, $Init$, $Shutdown$, $AvgSchedRate$, and $Runtime$), our model is able to very accurately predict the performance of all scheduler trees for the *Sleep 5* and *Firestarter* applications and the shallower trees for the *Sleep 0* and *Stream* application. The exact R^2 values for the model on each application and scheduler tree are presented in Table 6. R^2 values can be interpreted as representing the percentage of explained variance and thus typically range between 0 and 1. In certain circumstances, R^2 can be negative due to large prediction errors. R^2 is defined as $1 - \frac{\sum_i e_i^2}{\sum_i (y_i - \bar{y})^2}$. When the prediction error (e_i) is significantly larger than the variance in the real data ($y_i - \bar{y}^2$), R^2 can be negative, as is the case for the predictions of *Sleep 0* and *Stream's* performance with the *1x32x36* topology.

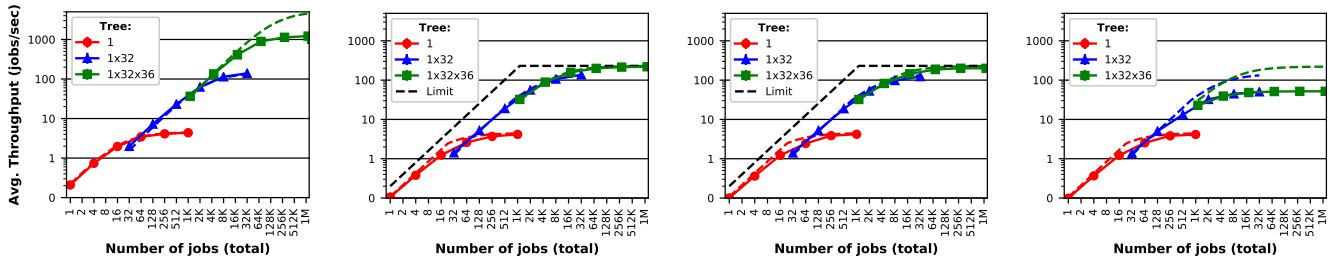
Unsurprisingly, the error in the *Stream* predictions are due to the inflated runtime caused by inter-application contention. The model predicts a throughput curve similar to the one for *Sleep 5* and *Firestarter*. In reality, as the system becomes filled with *Stream* process, the memory bus of each node becomes saturated and the node memory bandwidth becomes the bottleneck. Despite the scheduler adding more *Stream* jobs to the system, the overall job throughput does not improve due to the fixed-work nature of *Stream*. With each new *Stream* process, the overloaded memory bandwidth remains the same but is split between yet another process.

Potentially more surprising is the error in the *Sleep 0* predictions. *Sleep 0* is our simplest application with the fewest factors to consider, so one might expect that the predictions for this application should be the best. Interestingly, due to its lack of an associated computation time, massive numbers of *Sleep 0* applications are simultaneously injected into a node, hitting what appears to be some resource bottleneck. Specifically, with 36 schedulers each trying to fork/exec new processes at their $AvgSchedRate$ in parallel, some node resources become overwhelmed and we observe a node-level bottleneck quite similar in effect to the memory bottleneck that *Stream* encountered. We theorize that the bottleneck is formed at the system software level including a Unix shell program or the Operating System (OS) itself, such as with a fork/exec rate or with the process scheduler in the OS kernel.

The source of both of these errors are our chosen values for X_{usage} and $Runtime(X_{usage})$, which assume that the average runtime of the applications does not change with respect to the contention on the system from other applications.

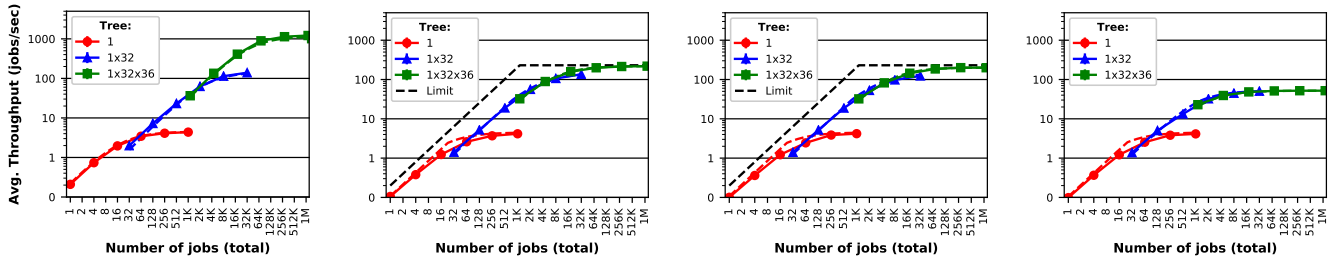
Table 5. Performance results of applying hierarchical scheduling to all four applications

Application Tree	Sleep 0 Throughput	% of Max	Sleep 5 Throughput	% of Max	Firestarter Throughput	% of Max	Stream Throughput	% of Max
1	4.5	-	4.2	2.0	4.2	2.0	4.1	-
1x32	143.1	-	136.0	59.0	123.0	53.4	50.4	-
1x32x36	1218.5	-	218.7	94.9	200.4	87.0	51.9	-



(a) Sleep 0 **(b)** Sleep 5 **(c)** Firestarter **(d)** Stream

Figure 5. Observed performance versus our **analytical model's** predicted performance of four applications on 32 node under various scheduler trees



(a) Sleep 0 **(b)** Sleep 5 **(c)** Firestarter **(d)** Stream

Figure 6. Observed performance versus our **hybrid model's** predicted performance of four applications on 32 node under various scheduler trees

Table 6. R^2 values of our analytical model's predictions

Application Tree	Sleep 0	Sleep 5	Firestarter	Stream
1	0.997	0.902	0.893	0.903
1x32	0.995	0.996	0.995	-3.555
1x32x36	-11.631	0.989	0.914	-134.177
Overall	-4.895	0.997	0.978	-11.159

4.4 Hybrid Model

To improve the accuracy of our analytical model, we must quantify the runtime inflation that occurs when applications interfere with one another on a resource constrained system. There are multiple ways one might go about modeling the runtime inflation caused by this inter-application contention. For example, if the total data movement performed by *Stream* and the available memory bandwidth on a node are known, one could estimate the runtime of the application assuming that bandwidth is evenly distributed across all running jobs. The same is true for *Sleep 0* and the rate at which the operating system can fork and exec new processes. One might also turn to the wealth of knowledge on application performance modeling in the literature. There exists many complex and highly accurate models for estimating application performance degradation due to resource contention for many classes of applications and resources Martinasso and Méhaut (2011); Dwyer et al. (2012); Yang et al. (2016).

For this work, we take the most direct route of empirically measuring the average runtime of our applications on a fully loaded node. To handle the case where a node is only partially full of jobs, we use a linear interpolation between the two average runtimes. Specifically, we use $Runtime(X_{usage}) = ((Runtime(1) - Runtime(0)) * X_{usage}) + Runtime(0)$ where X_{usage} is percent utilization of the node the application is running on, $Runtime(1)$ is the runtime on a fully saturated node, and $Runtime(0)$ is the runtime on an empty node.

To capture the empirical value for $Runtime(1)$, we pre-create a 1 by 36 scheduler tree on a single node, run 4 waves of jobs (144 total jobs), and measure the average runtime of the jobs. This entire process takes about a minute per application. Table 7 presents the updated input parameter values for our model based on these measurements. Although simple, this method is very accurate for the types of ensemble applications that we are concerned with, as we will see in our results. It has one primary shortcoming; it does not capture the contention that occurs outside of a node.

Figure 6 shows our analytical model's predictions when applied to all four applications using our new empirical

Table 7. Updated model input values for our hybrid model

Parameter	Value
X_{usage}	$max(1, \frac{J*x}{X})$
$Runtime(0)$	*Sleep 0*: 0
	Sleep 5, Firestarter, Stream: 5
$Runtime(1)$	*Sleep 0*: 1
	Sleep 5: 5.23
	Firestarter: 5.69
	Stream: 22.2
$Runtime(X_{usage})$	$((Runtime(1) - Runtime(0)) * X_{usage}) + Runtime(0)$

Table 8. R^2 values of our analytical model's predictions using the updated average application runtime

Application Tree	Sleep 0	Sleep 5	Firestarter	Stream
1	0.997	0.902	0.894	0.906
1x32	0.995	0.996	0.995	0.999
1x32x36	0.973	0.996	0.998	0.974
Overall	0.987	0.998	0.999	0.998

measure for the average job runtime. The line and marker colors are the same as in Figure 5. Table 8 includes the exact R^2 values for the model using the updated average runtime for each application and scheduler tree.

It is evident from both the figure and the R^2 values that the predictions using the new average job runtime are greatly improved. Our hybrid model achieves R^2 values of at least 0.987 across all four applications. The only detectable error that still remains is in the one-level tree as the single scheduler transitions from job limited to scheduler decision rate limited. The hybrid model slightly overpredicts the scheduler performance at the knee of the curve.

5 Application to Hierarchical Ensembles

As mentioned previously, ensemble workflows have swelled to encompass millions of jobs, and we have demonstrated how our model can be applied to improve the job throughput of these homogenous ensembles. Pure job throughput is not the only challenge facing these emerging HPC workflows. The latest generation of ensemble workflows have also become hierarchical, consisting of many runs of 1D, 2D, and 3D simulations Peterson et al. (2019). One use case for these hierarchical ensembles is the training of a neural network on the 1D simulations followed by the use of transfer learning to train on higher fidelity simulations Humbird et al. (2020). With each increase in simulation dimensionality, the resource requirements of the simulations increase and the number of simulations executed decreases. In this section, we demonstrate how our model can be applied to these heterogeneous hierarchical ensembles despite our simplifying assumption of homogeneous jobs.

Our fourth simplifying assumption constrains our model to jobs that are homogenous in their runtime and resource requirements. At first glance, this may seem to disqualify our model from application in the case of a hierarchical ensemble with multiple simulation types and varying resource requirements. The key to applying our model is that the jobs must only be homogenous within a given scheduler tree, and nothing in the model or implementation

Table 9. Model input values for Laghos ensemble

Parameter	Value
J	varies from 1 to 256,000
X	1,152 cores
x	6 cores
$Runtime(0)$	5.0
$Runtime(1)$	5.5
$AvgSchedRate(X)$	4.5
$Init(X) + Shutdown(X)$	4.3

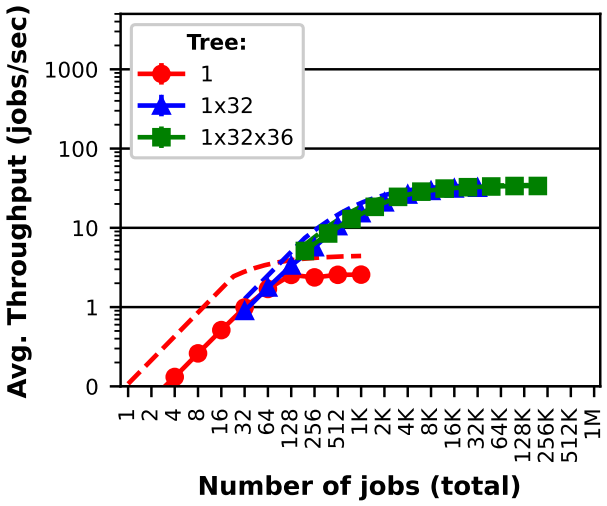


Figure 7. Observed performance versus our hybrid model's predicted performance of Laghos on 32 node under various scheduler trees

prevents us from creating multiple scheduler trees per workflow. In this way, we can create a scheduler tree for each dimensionality of simulation within the hierarchical ensemble. These scheduler trees can then be executed simultaneously or serially depending on the workflows needs and the resources available.

The low-fidelity 1D simulations used in these hierarchical ensembles are typically single-core applications, which are covered extensively in Section 4. Higher dimensionality simulations are almost exclusively multi-core applications, which Section 4 does not cover. To capture the effectiveness of our model on higher dimensionality simulations, we demonstrate our model applied to a multi-core hydrodynamics miniapp, Laghos Dobrev et al. (2012). Laghos is bundled with many different problems and meshes that it can be executed with. We use the primary problem of interest for Laghos, the Sedov blast, with the bundled 3D mesh *cube_12_hex* and its associated parameters "Laghos" (2021). We configured each simulation of Laghos to use 6 cores so that each node of the cluster execute multiple (i.e., 6) Laghos simulations without wasting cores.

The parameters for our analytical performance model as applied to Laghos are listed in Table 9. We use the same 32 node cluster and values for $AvgSchedRate(X)$ and $Init(X) + Shutdown(X)$ as those used in Section 4. We also use the technique described in Section 4.4 for determining the values of $Runtime(0)$ and $Runtime(1)$ for Laghos.

The model performs well in capturing the performance of multi-core Laghos on a 3D mesh under the various scheduler

trees with a slight overprediction of throughput for the one-level tree. The larger resource requirements of Laghos limit the necessary depth of the hierarchy such that a two-level tree is sufficient to achieve the maximal job throughput of 32 jobs per second on 1,152 cores. This represents a 7x speedup over the 4.5 jobs per second achievable with a one-level tree. The results from the previous section demonstrated that our model can help achieve a roughly 50x speedup on single core applications like *Firestarter*. Combined, these results validate that despite our simplifying assumptions, our analytical model is suitable for applying to a complex, heterogenous, hierarchical ensemble workflow.

6 Identifying Concerns at Extreme Scale

One of the goals of our performance modeling is to allow scheduler implementors and workflow users to reason about the benefits of GHS without running massively large numbers of experiments, a prohibitively expensive task. To meet this objective, the parameter values of our model must be easy to obtain for the given platform and scales; yet the produced model must be capable of identifying the areas of concerns to guide further investigations. We now present a case study to discuss the effectiveness of our model as such an easy-to-use reasoning tool.

Our case study targets pre-exascale systems such as Summit at Oak Ridge National Laboratory "Summit" (2018) and Sierra at Lawrence Livermore National Laboratory "Sierra" (2018). These systems each have approximately 4,500 nodes. Each node has 2 IBM Power9 processors with 22 cores each, for a total of 44 cores per node and approximately 198,000 cores per system. Summit has 6 NVIDIA Volta V100 GPUs per node and Sierra has 4 per node, for an approximate total of 27,000 and 18,000 GPUs respectively. While the GPUs provide the bulk of computational capability of these systems, we focus on the CPUs in our modeling because under our GHS scenario the schedulers run on the compute node CPUs.

The first step to produce a model for these system is to update the input parameter values as shown in Table 10. For three of the parameters ($Init(X)$, $Shutdown(X)$, $AvgSchedRate(X)$), we experimentally measure their values on a single Flux instance with an X of 1 and 12 nodes from the Sierra supercomputer. For a single node, we measure $Init + Shutdown$ to be 2.15 seconds and $AvgSchedRate$ to be 3.6 jobs per second. For 12 nodes, we measure $Init + Shutdown$ to be 3.4 seconds and $AvgSchedRate$ to be 3.3 jobs per second. Given the relatively insignificant difference at these scales, we make all three factors constant with respect to X. For the $Runtime(1)$ values, we perform the same empirical method described in Section 4.4 for *Sleep 0*, *Sleep 5*, and *Stream* on a single node of Sierra. We use the $Runtime(1)$ values from our test cluster for the *Firestarter* application as it does not support the Power9 architecture. For the *Stream* application, its $Runtime(0)$ when performing 70 iterations on a Sierra node is only 2.2 second. To bring the runtime in line with the runtimes of *Sleep 5* and *Firestarter*, we configured *Stream* to perform 145 iterations when running on a Sierra node, which brings the $Runtime(0)$ value back to approximately 5 seconds.

Table 10. Input values for pre-exascale system exploration

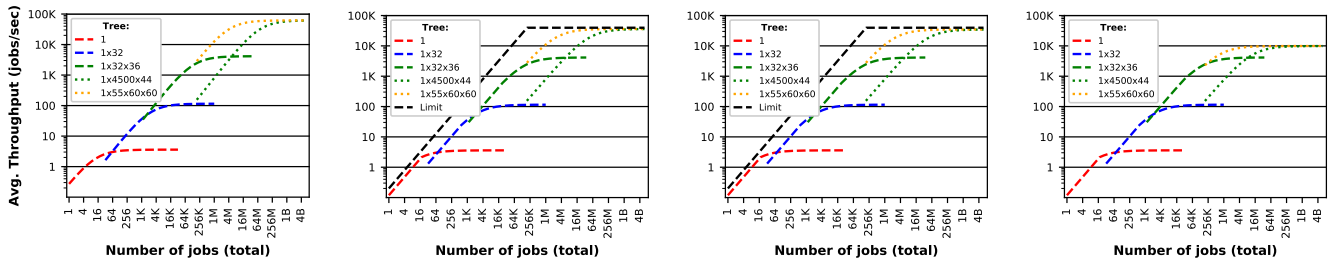
Parameter	Value
J	varies from 1 to 4 billion
X	198,000 cores
x	1 core
X_{usage}	$max(1, \frac{J*x}{X})$
$Runtime(0)$	*Sleep 0*: 0 *Sleep 5, Firestarter, Stream*: 5
$Runtime(1)$	*Sleep 0*: 3.20 *Sleep 5*: 5.48 *Firestarter*: 5.69 *Stream*: 20.0
$Runtime(X_{usage})$	$((Runtime(1) - Runtime(0)) * X_{usage}) + Runtime(0)$
$AvgSchedRate(X)$	3.6
$Init(X) + Shutdown(X)$	3.4

It is worth noting that the empirical values we use (that is, $Init(X)$, $Shutdown(X)$, and $AvgSchedRate(X)$ have two known sources of error that affect the precision of our model. The first source is that we do not have empirical data for the values of $AvgSchedRate(X)$ and $Init(X) + Shutdown(X)$ for an X of 198,000 cores. In our previous experiments, we assumed that these values were constant with respect to R, but for large enough values of X, we expect to see a decrease in $AvgSchedRate$ and an increase in $Init + Shutdown$. To some extent, these errors will cancel each other out. The decreased single scheduler decision rate will bias the model towards larger trees, and the increased scheduler creation cost will bias the model towards smaller trees. A user that wants an exact model of GHS for these systems can run the empirical methods that we describe in Section 4.4 at full scale to get the exact values of $AvgSchedRate(X)$ and $Init(X) + Shutdown(X)$ for Summit or Sierra.

The second source of error is in the empirical method that we proposed in Section 4.4 for capturing the runtime inflation of applications due to resource bottlenecks. As we mentioned previously, this empirical method does not capture the effects of bottlenecks on resources external to a node, like networks and filesystems. Given the nature of the ensemble applications we are using, we do not expect this error to affect the predictions we present in this paper, but users of our model may want to apply our model to their own applications which may perform significant amounts of network or disk I/O. At the scale of thousand of nodes and hundreds of thousands of cores, even launching a python interpreter can cause significant filesystem contention Frings et al. (2013).

Nevertheless, the baseline model without the above two error terms corrected should meet our goal of informing the relative impacts of each error source on the scheduler implementation and workflow when compared to an experimental data.

With an expanded system size, the question becomes: what is the best scheduler tree for the new system? While determining the optimal tree is outside the scope of this paper, our model does allow us for reasoning about general tree shapes that would perform better than others. The



(a) Sleep 0 **(b)** Sleep 5 **(c)** Firestarter **(d)** Stream

Figure 8. Our hybrid model's predicted performance of four applications on a 4,500 node pre-exascale system under various scheduler trees

performance of two new scheduler trees in addition to the same trees is presented in Figures 5 and 6. The first extra tree, 1x4500x44, matches the hardware topology of the system (that is, 4500 nodes and 44 cores per node) much like the 1x32x36 tree matches the hardware of our test cluster. The second extra tree, 1x55x60x60, has the exact same number of leaf schedulers as the 1x4500x44 tree, but the branching factor at each level is more uniform. We expect that this more uniform tree will be less expensive to create and thus perform better overall.

Figure 8 shows the job throughput that our hybrid model predicts a GHS with the five chosen scheduler trees would achieve on a pre-exascale system. The lines are colored based on the number of levels in the tree: 1-level is red, 2-level is blue, 3-level is green, and 4-level is orange. The trees used in previous results are plotted in bold dashed lines, and the new additional trees are plotted in a sparser dotted line. Table 11 presents the peak predicted job throughput for each tree, the improvement over a single scheduler (1-Level), and the percentage of theoretical maximum throughput that is predicted to be achievable (when applicable).

There are three key observations that we make from these results. First, the one- and two-level trees perform about the same as on the 32 node system, except for a slight bump in the *Stream* application. This is expected as these trees were not resource limited but instead limited by the scheduler parallelism, which has remained constant across these experiments.

Second, the 1x32x36 tree achieves roughly 10x the job throughput across all three applications due to the significantly expanded system. On the 32 node cluster with 1,152 schedulers, the bottleneck was resources, but on a 198,000 core system that resource bottleneck is removed, increasing performance. Note that these smaller trees outperform the larger trees on ensembles with fewer than 256,000 jobs. Intuitively this makes sense because the system is not fully saturated with jobs until ensembles with at least 198,000 jobs are run. So the optimal scheduler tree for these workflows would be one that is quick to create.

Third, the two new trees 1x4500x44 and 1x55x60x60 both have high percentage of theoretical maximum throughput, achieving a predicted 95% and 87% on the *Sleep 5* and *Firestarter* applications respectively. Despite having the same number of schedulers and a similar predicted peak throughput, the 1x55x60x60 tree performs better than the 1x4500x44 tree for a wide range of ensemble sizes. The more uniform distribution of branching factors across the levels significantly reduces the tree creation cost from 1,022 seconds down to 55 seconds, over an 18x improvement. This reduced cost enables the 1x55x60x60 to perform better on smaller ensembles.

Interestingly, this motivates the claim that even deeper trees make sense on large-enough systems. If system node and core counts continue to rise, the optimal tree depth will most likely increase as well. It also reveals a heuristic for picking the right tree. A tree should have as uniform a branching factor across levels as possible to minimize the creation cost while still achieving the desired level of scheduler parallelism.

Overall, the findings from this case study significantly narrowed down the areas of further investigations for scheduler developers and workflow users.

7 Related Work

Modeling of Tree-based Distributed Systems: Goehner *et al.* Goehner et al. (2013); Goehner (2011) presented Libi, which is a framework for efficient bootstrapping of parallel software using a tree of launchers. They model the performance of Libi when using a sequential tree and a k-ary tree of launchers and propose a greedy algorithm for optimizing launch trees. This is analytically similar to our $Create(Tree, X)$ equation.

Park *et al.* Ju-Young L. Park et al. (1996) created a communication model for multi-level multicast trees and provide dynamic programming techniques to find the analytically optimal multicast tree. They based their model on LogP Culler et al. (1993), which is itself a general performance model of parallel computing. Despite being focused on optimizing the performance of multicast network operations, their performance modeling is similar to our modeling of $Create(Tree_n, X)$. Their model could potentially be used to optimize $Tree_n$ with the objective of minimizing the $Create$ cost. Another similarity is that their model is parameterized such that benchmarks can be run on the system and used to set values within the model for a given parallel computer. With both Libi and the multicast model, a major differentiator from our work is that they do not consider jobs, inter-job interference, or the resources X.

Alternative Scheduling Models: At a high-level, scheduling solutions can be broken down into the centralized, hierarchical, and decentralized Schwarzkopf et al. (2013); Hussain et al. (2013). We have already covered the centralized and hierarchical scheduling at length but

Table 11. Predicted performance results of applying hierarchical scheduling on a pre-Exascale system

Application Tree	Sleep 0 Throughput	% of Max	Sleep 5 Throughput	% of Max	Firestarter Throughput	% of Max	Stream Throughput	% of Max
1	3.6	-	3.6	0.1	3.6	0.1	3.6	-
1x32	115.0	-	115.0	0.3	115.0	0.3	115.0	-
1x32x36	4132.3	-	4132.3	10.4	4132.3	10.4	4132.3	-
1x4500x44	61131.0	-	35876.4	90.6	34561.3	87.3	9875.8	-
1x55x60x60	61836.1	-	36118.1	91.2	34785.6	87.8	9894.1	-

have not covered decentralized scheduling. In decentralized scheduling, there are multiple schedulers cooperatively scheduling without a central leader. Decentralized schedulers coordinate their actions via atomic operation on a distributed key-value store (DKVS). In the simplest configuration, a decentralized scheduler will perform a DKVS operation for every job allocation, but in a more optimized architecture, jobs and resources are partitioned between schedulers and DKVS operations are used for work and resource stealing Wang et al. (2015). Decentralized schedulers have a higher throughput and fault-tolerance than centralized schedulers, but as the number of decentralized schedulers increases, so does the communication overhead and the rate of conflicts in the DKVS atomic operations. Decentralized schedulers optimized for HPC jobs, such as Slurm++, have demonstrated real-world scalability up to 500 nodes and 10 decentralized schedulers and simulated scalability up to 65,536 nodes and 64 decentralized schedulers Wang et al. (2015).

GHS and decentralized scheduling both buck the trend of centralized scheduling, improving throughput and scalability. The main differentiator between the two scheduling models is their architectural structure. In GHS, the architectural structure is a tree rather than a fully-connected topology. This tree-based structure enables both the top-down enforcement of complex scheduling constraints and policies (via the *hierarchical bounding principle*) as well as the user-customization of schedulers within the tree (via the *instance effectiveness principle*). To the best of our knowledge, no prior work has proposed or demonstrated a method for user-specialization of individual schedulers within a decentralized scheduler while maintaining global policies and constraints.

8 Discussion and Conclusion

In this work, we formally defined generalized hierarchical scheduling (GHS) and presented an analytical performance model for it. We showed how our analytical model can be combined with empirical values that are quick and easy to obtain to form an extremely accurate hybrid model. We validated our model against four different applications and achieved R^2 values greater than 0.98 across them.

Using our validated model, we explored the performance of GHS on large pre-exascale systems and a broad range of workflow sizes. Our model shows that as systems and workflows increase in size, so too should the GHS tree. At the largest scales, our model predicts that a four-level scheduler achieves up to 95% of peak theoretical throughput and four orders of magnitude higher throughput than a single, centralized scheduler. Our model also highlights that the highest performing GHS tree depends on multiple factors including system size, scheduler decision rate, scheduler initialization cost, and average application runtime.

We anticipate several uses of our analytical performance model. One potential usage enables GHS users to pinpoint bottlenecks in their GHS trees. Users can determine if system resources, the amount of scheduler parallelism, scheduler tree creation, or single scheduler performance limits performance. As an example, a user can determine that an artificial GHS limit on tree depth to two levels impacts performance and, thus, motivates adoption of a more flexible GHS implementation. Another potential use case is an automated GHS scheduler tree optimizer. After the relatively inexpensive collection of data (approximately 1 node-minute), our hybrid model can be used to quickly explore the possible GHS parameter space, weigh the benefits of increased parallelism against the costs of launching additional schedulers, and ultimately predict the optimal scheduler tree depth and width. This initial prediction of the optimal scheduling tree for a workflow could then be further optimized using actual runs of the workflow.

Supplemental material

All of the scripts and code used in this research can be found online on GitHub: https://doi.org/10.5281/zenodo.5168133. The repository includes the scripts to generate the experiments, automatically submit them to a Slurm cluster, analyze the results, and then generate the figures and tables seen in the paper. How to use the scripts is documented in the README found within the repository.

Acknowledgements

The authors acknowledge the advice and support from the Flux team at LLNL: Jim Garlick, Mark Grondona, Al Chu, Daniel Milroy, Christopher Moussa, Don Lipari, Ned Bass, and Becky Springmeyer.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE- AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. LLNL-JRNL-825636.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Declaration of conflicting interests

The Author(s) declare(s) that there is no conflict of interest.

Author Biographies

Stephen Herbein is a computer scientist in Livermore Computing at Lawrence Livermore National Laboratory. His research interests include batch job scheduling, HPC+Cloud convergence, and parallel IO. He is a part of the Flux team, developing next-generation IO-aware and multi-level schedulers for HPC. Stephen has been at LLNL since 2018. He earned his Ph.D., M.S., and B.S. degrees in Computer Science from the University of Delaware in 2018, 2016, and 2014. He is a member of both the IEEE and the ACM.

Tapasya Patki is a Computer Scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory and the co-PI for the ECP Argo PowerStack project. Her current research involves the design and implementation of exascale system software (such as vendor-neutral portable libraries and next-generation HPC resource managers), with a specific focus on power awareness and application performance optimization under multiple constraints. Broadly, she is interested in power-constrained supercomputing, network topology research, performance modeling and analysis, converged computing, and HPC system software. Tapasya earned her Ph.D. in Computer Science from The University of Arizona under the advisement of Dr. David Lowenthal (2015). Her dissertation research focused on exploring the potential of hardware overprovisioning in power-constrained, high-performance computing.

Dong H. Ahn is a computer scientist. He has worked for Livermore Computing (LC) at Lawrence Livermore National Laboratory since 2001 and currently leads the next-generation computing enabling (NGCE) project within the ASC ATDM sub-program. During this period, Dong has worked on several code-development-tools and next-generation resource management and scheduling software framework projects with a common goal to provide highly capable and scalable tools ecosystems for large computing systems.

Sebastian Mobo is a third-year undergraduate student in the Department of Electrical Engineering and Computer Science and the Department of Mathematics at the University of Tennessee, Knoxville. His expected B.S. graduation date is Spring 2022. He is an Undergraduate Research Assistant in the Global Computing Lab under Dr. Michela Taufer. His interests include systems programming, operating systems development, statistical learning, and machine learning / AI.

Clark Hathaway is a fourth-year undergraduate student in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His expected B.S. graduation date is Fall 2021. He is a Research Assistant in the Global Computing Lab under Dr. Michela Taufer. His interests include machine learning and cryptographic, decentralized databases.

Silvina Caíno-Lores is a Post-Doctoral Research Associate in the University of Tennessee-Knoxville, as a member of the Global Computing Laboratory. She obtained her PhD in Computer Science and Technology in 2019 at the Carlos III University of Madrid (Spain). Her research interests include cloud computing, in-memory computing and storage, HPC scientific simulations, and data-centric paradigms. Her recent works and active collaborations focus on the area of convergence between HPC and Big Data analytics at the application and platform layers.

James Corbett is a software developer at LLNL, which he joined in 2019. While there he has worked on a variety of projects including Flux, the geophysics simulation GEOSX, and the data exchange library Conduit. However, he has spent the majority of his development time on Themis, an ensemble workflow manager that integrates with Flux.

David Domyancic is a computer scientist at LLNL. He works on Themis and the Uncertainty Quantification Pipeline (UQP).

Thomas R. W. Scogland is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research centers around the design and development of runtime systems and programming models for scientific applications dealing with heterogeneous and many-core architectures, heterogeneous memory systems, and scheduling in heterogeneous systems. He is an active developer on the flux resource manager and RAJA framework libraries, lead on the CAMP and DESUL library proejcts, chair of the OpenMP accelerator subcommittee, LLNL WG21 C++ committee representative and participates in numerous other research and development efforts. Tom holds a B.S. in Computer Science from Purdue University, and an M.S. and Ph.D. in Computer Science from Virginia Tech.

As Chief Technology Officer (CTO) for Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL), **Bronis R. de Supinski** formulates LLNL's large-scale computing strategy and oversees its implementation. He frequently interacts with supercomputing leaders and oversees many collaborations with industry and academia. Previously, Bronis led several research projects in LLNL's Center for Applied Scientific Computing. He earned his Ph.D. in Computer Science from the University of Virginia in 1998 and he joined LLNL in July 1998. In addition to his work with LLNL, Bronis is a Professor of Exascale Computing at Queen's University of Belfast and an Adjunct Associate Professor in the Department of Computer Science and Engineering at Texas A&M University. Throughout his career, Bronis has won several awards, including the prestigious Gordon Bell Prize in 2005 and 2006, as well as two R&D 100s. He is an IEEE Fellow.

Michela Taufer is an ACM Distinguished Scientist and holds the Jack Dongarra professorship in high performance computing with the Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. Her research interests include high-performance computing, volunteer computing, scientific applications, scheduling and reproducibility challenges, and in situ data analytics. Dr. Taufer received her Ph.D. in Computer Science from the Swiss Federal Institute of Technology (ETH) in 2002.

References

(2018) Sierra. https://hpc.llnl.gov/hardware/platforms/sierra. Retrieved July 30, 2018.

(2018) Summit. https://www.olcf.ornl.gov/summit/. Retrieved July 30, 2018.

(2019) Olcf policy guide. https://www.olcf.ornl.gov/for-users/olcf-policy-guide/. Retrieved January 29, 2019.

(2020) Computational data analysis workflow systems. https://github.com/common-workflow-language/

common-workflow-language/wiki/Existing-Workflow-systems. Retrieved April 19, 2020.

(2021) Laghos: High-order lagrangian hydrodynamics miniapp. hhttps://github.com/CEED/Laghos. Retrieved June 01, 2021.

Ahn DH, Bass N, Chu A, Garlick J, Grondona M, Herbein S, Ingolfsson H, Koning J, Patki T, Scogland TRW, Springmeyer B and Taufer M (2020) Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* DOI:10.1016/j.future.2020.04.006.

Babuji Y, Woodard A, Li Z, Katz DS, Clifford B, Kumar R, Lacinski L, Chard R, Wozniak JM, Foster I, Wilde M and Chard K (2019) Parsl: Pervasive parallel programming in python. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19. ISBN 9781450366700, p. 25–36. DOI:10.1145/3307681.3325400. URL https://doi.org/10.1145/3307681.3325400.

Barney B (2017) SLURM and Moab. https://computing.llnl.gov/tutorials/moab. Retrieved August 22, 2017.

Chapin SJ, Katramatos D, Karpovich JF and Grimshaw AS (1999) The Legion resource management system. In: *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP)*.

Cosnard M, Jeannot E and Yang T (2004) Compact dag representation and its symbolic scheduling. *Journal of Parallel and Distributed Computing* 64(8): 921 – 935. DOI:https://doi.org/10.1016/j.jpdc.2004.05.001. URL http://www.sciencedirect.com/science/article/pii/S0743731504000693.

Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, Subramonian R and von Eicken T (1993) Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.* 28(7): 1–12. DOI:10.1145/173284.155333. URL https://doi.org/10.1145/173284.155333.

Dahlgren TL, Domyancic D, Brandon S, Gamblin T, Gyllenhaal J, Nimmakayala R and Klein R (2015) 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)*.

Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, Mayani R, Chen W, da Silva] RF, Livny M and Wenger K (2015) Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46: 17 – 35. DOI:https://doi.org/10.1016/j.future.2014.10.008. URL http://www.sciencedirect.com/science/article/pii/S0167739X14002015.

Di Natale F, Bhatia H, Carpenter TS, Neale C, Kokkila Schumacher S, Oppelstrup T, Stanton L, Zhang X, Sundram S, Scogland TRW, Dharuman G, Surh MP, Yang Y, Misale C, Schneidenbach L, Costa C, Kim C, D'Amora B, Gnanakaran S, Nissley DV, Streitz F, Lightstone FC, Bremer PT, Glosli JN and Ingólfsson HI (2019) A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362290. DOI:10.1145/3295500.3356197. URL https://doi.org/10.1145/3295500.3356197.

Dobrev VA, Kolev TV and Rieben RN (2012) High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing* 34(5): B606–B641. DOI:10.1137/120864672. URL https://doi.org/10.1137/120864672.

Dutot PF, Mercier M, Poquet M and Richard O (2017) Batsim: A realistic language-independent resources and jobs management systems simulator. In: Desai N and Cirne W (eds.) *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing. ISBN 978-3-319-61756-5, pp. 178–197.

Dwyer T, Fedorova A, Blagodurov S, Roth M, Gaud F and Pei J (2012) A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. pp. 1–11.

El-Rewini H and Lewis T (1990) Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing* 9(2): 138 – 153. DOI: https://doi.org/10.1016/0743-7315(90)90042-N. URL http://www.sciencedirect.com/science/article/pii/074373159090042N.

Eleliemy A and Ciorba FM (2021) A resourceful coordination approach for multilevel scheduling. *CoRR* abs/2103.05809. URL https://arxiv.org/abs/2103.05809.

Eriksson H, Raciti M, Basile M, Cunsolo A, Fröberg A, Leifler O, Ekberg J and Timpka T (2011) A cloud-based simulation architecture for pandemic influenza simulation. *AMIA Annu Symp Proc* 2011: 364–373.

Foster I and Kesselman C (1997) Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications* 11(2): 115–128.

Frings W, Ahn DH, LeGendre M, Gamblin T, de Supinski BR and Wolf F (2013) Massively parallel loading. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS. ISBN 9781450321303, p. 389–398. DOI:10.1145/2464996.2465020. URL https://doi.org/10.1145/2464996.2465020.

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

Goehner J, Arnold D, Ahn D, Lee G, [de Supinski] B, LeGendre M, Miller B and Schulz M (2013) Libi: A framework for bootstrapping extreme scale software systems. *Parallel Computing* 39(3): 167 – 176. DOI:https://doi.org/10.1016/j.parco.2012.09.003. URL http://www.sciencedirect.com/science/article/pii/S0167819112000774. High-performance Infrastructure for Scalable Tools.

Goehner JD (2011) *Efficiently bootstrapping extreme scale software systems*. Master's Thesis, The University of New Mexico.

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

Hackenberg D, Oldenburg R, Molka D and Schöne R (2013) Introducing firestarter: A processor stress test utility. In: *Proceedings of the International Green Computing Conference*. pp. 1–9.

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S and Stoica I (2011) Mesos: A platform for fine-grained resource sharing in the data center. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

Hines J (2015) Moab scheduling tweak tightens titan's workload. https://www.olcf.ornl.gov/2015/10/13/moab-scheduling-tweak-tightens-titans-workload/. Retrieved January 29, 2019.

Hochbaum DS and Shmoys DB (1987) Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM* 34(1): 144–162. DOI:10.1145/7531.7535. URL https://doi.org/10.1145/7531.7535.

Humbird KD, Peterson JL, Spears BK and McClarren RG (2020) Transfer learning to model inertial confinement fusion experiments. *IEEE Transactions on Plasma Science* 48(1): 61–70. DOI:10.1109/TPS.2019.2955098.

Hussain H, Malik SUR, Hameed A, Khan SU, Bickler G, Min-Allah N, Qureshi MB, Zhang L, Yongji W, Ghani N, Kolodziej J, Zomaya AY, Xu CZ, Balaji P, Vishnu A, Pinel F, Pecero JE, Kliazovich D, Bouvry P, Li H, Wang L, Chen D and Rayes A (2013) A survey on resource allocation in high performance distributed computing systems. *Parallel Computing* 39(11): 709–736.

Jing-Chiou Liou and Palis MA (1998) A new heuristic for scheduling parallel programs on multiprocessor. In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques*. pp. 358–365.

Jokanovic A, D'Amico M and Corbalan J (2018) Evaluating slurm simulator with real-machine slurm and vice versa. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. pp. 72–82. DOI:10.1109/PMBS.2018.8641556.

Ju-Young L Park, Hyeong-Ah Choi, Natawut Nupairoj and Ni LM (1996) Construction of optimal multicast trees based on the parameterized communication model. In: *Proceedings of the ICPP Workshop on Challenges for Parallel Processing*, volume 1. pp. 180–187 vol.1. DOI:10.1109/ICPP.1996.537159.

Klusáček D, Soysal M and Suter F (2020) Alea – complex job scheduling simulator. In: Wyrzykowski R, Deelman E, Dongarra J and Karczewski K (eds.) *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing. ISBN 978-3-030-43222-5, pp. 217–229.

Kwok YK and Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys* 31(4): 406–471. DOI:10.1145/344588.344618. URL https://doi.org/10.1145/344588.344618.

Manimaran G and Murthy CSR (1998) An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 9(3): 312–319.

Marchal L, Simon B, Sinnen O and Vivien F (2018) Malleable task-graph scheduling with a practical speed-up model. *IEEE Transactions on Parallel and Distributed Systems* 29(6): 1357–1370.

Martinasso M and Méhaut JF (2011) A contention-aware performance model for hpc-based networks: A case study of the infiniband network. In: Jeannot E, Namyst R and Roman J (eds.) *Euro-Par 2011 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-23400-2, pp. 91–102.

McCalpin JD (1995) Stream: Sustainable memory bandwidth in high-performance computers. https://www.cs.virginia.edu/stream/.

Merzky A, Turilli M, Maldonado M, Santcroos M and Jha S (2018) Using Pilot Systems to Execute Many Task Workloads on Supercomputers. *arXiv e-prints* : arXiv:1512.08194.

Min-You Wu, Wei Shu and Jun Gu (2001) Efficient local search far dag scheduling. *IEEE Transactions on Parallel and Distributed Systems* 12(6): 617–627.

Murphy JM, Sexton DM, Barnett DN, Jones GS, Webb MJ, Collins M and Stainforth DA (2004) Quantification of modelling uncertainties in a large ensemble of climate change simulations. *Nature* 430: 768–772.

Peterson JL, Anirudh R, Athey K, Bay B, Bremer PT, Castillo V, Di Natale F, Fox D, Gaffney JA, Hysom D, Ade Jacobs S, Kailkhura B, Koning J, Kustowski B, Langer S, Robinson P, Semler J, Spears B, Thiagarajan J, Van Essen B and Yeom JS (2019) Merlin: Enabling machine learning-ready HPC ensembles. *arXiv e-prints* : arXiv:1912.02892.

Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kalé L and Schulten K (2005) Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26(16): 1781–1802.

Prabhakaran S, Neumann M and Wolf F (2018) Efficient fault tolerance through dynamic node replacement. In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID. pp. 163–172.

Rodrigo GP, Elmroth E, Östberg PO and Ramakrishnan L (2018) Scsf: A scheduling simulation framework. In: Klusáček D, Cirne W and Desai N (eds.) *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing. ISBN 978-3-319-77398-8, pp. 152–173.

Sahni SK (1976) Algorithms for scheduling independent tasks. *Journal of the ACM* 23(1): 116–127. DOI:10.1145/321921.321934. URL https://doi.org/10.1145/321921.321934.

Schwarzkopf M, Konwinski A, Abd-El-Malek M and Wilkes J (2013) Omega: flexible, scalable schedulers for large compute clusters. In: *SIGOPS European Conference on Computer Systems (EuroSys)*. pp. 351–364. URL http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

Simakov NA, Innus MD, Jones MD, DeLeon RL, White JP, Gallo SM, Patra AK and Furlani TR (2018) A slurm simulator: Implementation and parametric analysis. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Cham: Springer. ISBN 978-3-319-72971-8, pp. 197–217.

Tao Yang and Gerasoulis A (1994) Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems* 5(9): 951–967.

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B and Baldeschwieler E (2013) Apache Hadoop YARN: Yet another resource negotiator. In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*.

Wang K, Zhou X, Qiao K, Lang M, McClelland B and Raicu I (2015) Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC. ISBN 9781450335508, p. 219–222. DOI:10.1145/2749246.2749249. URL https://doi.org/10.1145/2749246.2749249.

Wang Q and Cheng KH (1992) A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing* 21(2): 281–294. DOI:10.1137/0221021. URL https://doi.org/10.1137/0221021.

Wozniak JM, Dorier M, Ross R, Shu T, Kurc T, Tang L, Podhorszki N and Wolf M (2019) Mpi jobs within mpi jobs: A practical way of enabling task-level fault-tolerance in hpc workflows. *Future Generation Computer Systems* 101: 576 – 589. DOI:10.1016/j.future.2019.05.020. URL https://doi.org/10.1016/j.future.2019.05.020.

Yang X, Jenkins J, Mubarak M, Wang X, Ross RB and Lan Z (2016) Study of intra- and interjob interference on torus networks. In: *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems*, ICPADS. pp. 239–246.

References

(2018) Sierra. https://hpc.llnl.gov/hardware/platforms/sierra. Retrieved July 30, 2018.

(2018) Summit. https://www.olcf.ornl.gov/summit/. Retrieved July 30, 2018.

(2019) OLCF policy guide. https://www.olcf.ornl.gov/for-users/olcf-policy-guide/. Retrieved January 29, 2019.

(2020) Computational data analysis workflow systems. https://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems. Retrieved April 19, 2020.

(2021) Laghos: High-order lagrangian hydrodynamics miniapp. hhttps://github.com/CEED/Laghos. Retrieved June 01, 2021.

Ahn DH, Bass N, Chu A, Garlick J, Grondona M, Herbein S, Ingolfsson H, Koning J, Patki T, Scogland TRW, Springmeyer B and Taufer M (2020) Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* DOI:10.1016/j.future.2020.04.006.

Babuji Y, Woodard A, Li Z, Katz DS, Clifford B, Kumar R, Lacinski L, Chard R, Wozniak JM, Foster I, Wilde M and Chard K (2019) Parsl: Pervasive parallel programming in python. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19. ISBN 9781450366700, p. 25–36. DOI:10.1145/3307681.3325400. URL https://doi.org/10.1145/3307681.3325400.

Barney B (2017) SLURM and Moab. https://computing.llnl.gov/tutorials/moab. Retrieved August 22, 2017.

Chapin SJ, Katramatos D, Karpovich JF and Grimshaw AS (1999) The Legion resource management system. In: *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP)*.

Cosnard M, Jeannot E and Yang T (2004) Compact dag representation and its symbolic scheduling. *Journal of Parallel and Distributed Computing* 64(8): 921 – 935. DOI:https://doi.org/10.1016/j.jpdc.2004.05.001. URL http://www.sciencedirect.com/science/article/pii/S0743731504000693.

Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, Subramonian R and von Eicken T (1993) Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.* 28(7): 1–12. DOI:10.1145/173284.155333. URL https://doi.org/10.1145/173284.155333.

Dahlgren TL, Domyancic D, Brandon S, Gamblin T, Gyllenhaal J, Nimmakayala R and Klein R (2015) 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)*.

Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, Mayani R, Chen W, Ferreira da Silva R, Livny M and Wenger K (2015) Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46: 17 – 35. DOI:https://doi.org/10.1016/j.future.2014.10.008. URL http://www.sciencedirect.com/science/article/pii/S0167739X14002015.

Di Natale F, Bhatia H, Carpenter TS, Neale C, Kokkila Schumacher S, Oppelstrup T, Stanton L, Zhang X, Sundram S, Scogland TRW, Dharuman G, Surh MP, Yang Y, Misale C, Schneidenbach L, Costa C, Kim C, D'Amora B, Gnanakaran S, Nissley DV, Streitz F, Lightstone FC, Bremer PT, Glosli JN and Ingólfsson HI (2019) A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362290. DOI:10.1145/3295500.3356197. URL https://doi.org/10.1145/3295500.3356197.

Dobrev VA, Kolev TV and Rieben RN (2012) High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing* 34(5): B606–B641. DOI:10.1137/120864672. URL https://doi.org/10.1137/120864672.

Dutot PF, Mercier M, Poquet M and Richard O (2017) Batsim: A realistic language-independent resources and jobs management systems simulator. In: Desai N and Cirne W (eds.) *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing. ISBN 978-3-319-61756-5, pp. 178–197.

Dwyer T, Fedorova A, Blagodurov S, Roth M, Gaud F and Pei J (2012) A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. pp. 1–11.

El-Rewini H and Lewis T (1990) Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing* 9(2): 138 – 153. DOI: https://doi.org/10.1016/0743-7315(90)90042-N. URL http://www.sciencedirect.com/science/article/pii/074373159090042N.

Eleliemy A and Ciorba FM (2021) A resourceful coordination approach for multilevel scheduling. *CoRR* abs/2103.05809. URL https://arxiv.org/abs/2103.05809.

Eriksson H, Raciti M, Basile M, Cunsolo A, Fröberg A, Leifler O, Ekberg J and Timpka T (2011) A cloud-based simulation architecture for pandemic influenza simulation. *AMIA Annu Symp Proc* 2011: 364–373.

Foster I and Kesselman C (1997) Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications* 11(2): 115–128.

Frings W, Ahn DH, LeGendre M, Gamblin T, de Supinski BR and Wolf F (2013) Massively parallel loading. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS. ISBN 9781450321303, p. 389–398. DOI:10.1145/2464996.2465020. URL https://doi.org/10.1145/2464996.2465020.

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

Goehner J, Arnold D, Ahn D, Lee G, [de Supinski] B, LeGendre M, Miller B and Schulz M (2013) Libi: A framework for bootstrapping extreme scale software systems. *Parallel Computing* 39(3): 167 – 176. DOI:https://doi.org/10.1016/j.parco.2012.09.003. URL http://www.sciencedirect.com/science/article/pii/S0167819112000774. High-performance Infrastructure for Scalable Tools.

Goehner JD (2011) *Efficiently bootstrapping extreme scale software systems*. Master's Thesis, The University of New Mexico.

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

Hackenberg D, Oldenburg R, Molka D and Schöne R (2013) Introducing firestarter: A processor stress test utility. In: *Proceedings of the International Green Computing Conference*. pp. 1–9.

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S and Stoica I (2011) Mesos: A platform for fine-grained resource sharing in the data center. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

Hines J (2015) Moab scheduling tweak tightens titan's workload. https://www.olcf.ornl.gov/2015/10/13/moab-scheduling-tweak-tightens-titans-workload/. Retrieved January 29, 2019.

Hochbaum DS and Shmoys DB (1987) Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM* 34(1): 144–162. DOI:10.1145/7531.7535. URL https://doi.org/10.1145/7531.7535.

Humbird KD, Peterson JL, Spears BK and McClarren RG (2020) Transfer learning to model inertial confinement fusion experiments. *IEEE Transactions on Plasma Science* 48(1): 61–70. DOI:10.1109/TPS.2019.2955098.

Hussain H, Malik SUR, Hameed A, Khan SU, Bickler G, Min-Allah N, Qureshi MB, Zhang L, Yongji W, Ghani N, Kolodziej J, Zomaya AY, Xu CZ, Balaji P, Vishnu A, Pinel F, Pecero JE, Kliazovich D, Bouvry P, Li H, Wang L, Chen D and Rayes A (2013) A survey on resource allocation in high performance distributed computing systems. *Parallel Computing* 39(11): 709–736.

Jing-Chiou Liou and Palis MA (1998) A new heuristic for scheduling parallel programs on multiprocessor. In: *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques*. pp. 358–365.

Jokanovic A, D'Amico M and Corbalan J (2018) Evaluating slurm simulator with real-machine slurm and vice versa. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. pp. 72–82. DOI:10.1109/PMBS.2018.8641556.

Ju-Young L Park, Hyeong-Ah Choi, Natawut Nupairoj and Ni LM (1996) Construction of optimal multicast trees based on the parameterized communication model. In: *Proceedings of the ICPP Workshop on Challenges for Parallel Processing*, volume 1. pp. 180–187 vol.1. DOI:10.1109/ICPP.1996.537159.

Klusáček D, Soysal M and Suter F (2020) Alea – complex job scheduling simulator. In: Wyrzykowski R, Deelman E, Dongarra J and Karczewski K (eds.) *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing. ISBN 978-3-030-43222-5, pp. 217–229.

Kwok YK and Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys* 31(4): 406–471. DOI:10.1145/344588.344618. URL https://doi.org/10.1145/344588.344618.

Manimaran G and Murthy CSR (1998) An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 9(3): 312–319.

Marchal L, Simon B, Sinnen O and Vivien F (2018) Malleable task-graph scheduling with a practical speed-up model. *IEEE Transactions on Parallel and Distributed Systems* 29(6): 1357–1370.

Martinasso M and Méhaut JF (2011) A contention-aware performance model for hpc-based networks: A case study of the infiniband network. In: Jeannot E, Namyst R and Roman J (eds.) *Euro-Par 2011 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-23400-2, pp. 91–102.

McCalpin JD (1995) Stream: Sustainable memory bandwidth in high-performance computers. https://www.cs.virginia.edu/stream/.

Merzky A, Turilli M, Maldonado M, Santcroos M and Jha S (2018) Using Pilot Systems to Execute Many Task Workloads on Supercomputers. *arXiv e-prints* : arXiv:1512.08194.

Min-You Wu, Wei Shu and Jun Gu (2001) Efficient local search far dag scheduling. *IEEE Transactions on Parallel and Distributed Systems* 12(6): 617–627.

Murphy JM, Sexton DM, Barnett DN, Jones GS, Webb MJ, Collins M and Stainforth DA (2004) Quantification of modelling uncertainties in a large ensemble of climate change

simulations. *Nature* 430: 768–772.

Peterson JL, Anirudh R, Athey K, Bay B, Bremer PT, Castillo V, Di Natale F, Fox D, Gaffney JA, Hysom D, Ade Jacobs S, Kailkhura B, Koning J, Kustowski B, Langer S, Robinson P, Semler J, Spears B, Thiagarajan J, Van Essen B and Yeom JS (2019) Merlin: Enabling machine learning-ready HPC ensembles. *arXiv e-prints* : arXiv:1912.02892.

Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kalé L and Schulten K (2005) Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26(16): 1781–1802.

Prabhakaran S, Neumann M and Wolf F (2018) Efficient fault tolerance through dynamic node replacement. In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID. pp. 163–172.

Rodrigo GP, Elmroth E, Östberg PO and Ramakrishnan L (2018) Scsf: A scheduling simulation framework. In: Klusáček D, Cirne W and Desai N (eds.) *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing. ISBN 978-3-319-77398-8, pp. 152–173.

Sahni SK (1976) Algorithms for scheduling independent tasks. *Journal of the ACM* 23(1): 116–127. DOI:10.1145/321921.321934. URL https://doi.org/10.1145/321921.321934.

Schwarzkopf M, Konwinski A, Abd-El-Malek M and Wilkes J (2013) Omega: flexible, scalable schedulers for large compute clusters. In: *SIGOPS European Conference on Computer Systems (EuroSys)*. pp. 351–364. URL http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

Simakov NA, Innus MD, Jones MD, DeLeon RL, White JP, Gallo SM, Patra AK and Furlani TR (2018) A slurm simulator: Implementation and parametric analysis. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Cham: Springer. ISBN 978-3-319-72971-8, pp. 197–217.

Tao Yang and Gerasoulis A (1994) Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems* 5(9): 951–967.

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B and Baldeschwieler E (2013) Apache Hadoop YARN: Yet another resource negotiator. In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*.

Wang K, Zhou X, Qiao K, Lang M, McClelland B and Raicu I (2015) Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC. ISBN 9781450335508, p. 219–222. DOI:10.1145/2749246.2749249. URL https://doi.org/10.1145/2749246.2749249.

Wang Q and Cheng KH (1992) A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing* 21(2): 281–294. DOI:10.1137/0221021. URL https://doi.org/10.1137/0221021.

Wozniak JM, Dorier M, Ross R, Shu T, Kurc T, Tang L, Podhorszki N and Wolf M (2019) Mpi jobs within mpi jobs: A practical way of enabling task-level fault-tolerance in hpc workflows. *Future Generation Computer Systems* 101: 576 – 589. DOI:10.1016/j.future.2019.05.020. URL https://doi.org/10.1016/j.future.2019.05.020.

Yang X, Jenkins J, Mubarak M, Wang X, Ross RB and Lan Z (2016) Study of intra- and interjob interference on torus networks. In: *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems*, ICPADS. pp. 239–246.