LLNL-CONF-855103

Fluxion: A Scalable Graph-Based Resource Model for HPC Scheduling Challenges

T. Patki, D. H. Ahn, D. J Milroy, J. Yeon, J. Garlick, M. Grondona, S. Herbein, T. Scogland

September 28, 2023

18th Workshop on Workflows in Support of Large-Scale Science
Denver, CO, United States
November 12, 2023 through November 13, 2023

Fluxion: A Scalable Graph-Based Resource Model for HPC Scheduling Challenges

Tapasya Patki
Lawrence Livermore National Laboratory
United States

Dong H. Ahn
Nvidia Corporation
United States

Daniel J. Milroy
Lawrence Livermore National Laboratory
United States

Jae-Seung Yeom
Lawrence Livermore National Laboratory
United States

Jim Garlick
Lawrence Livermore National Laboratory
United States

Mark Grondona
Lawrence Livermore National Laboratory
United States

Stephen Herbein
Nvidia Corporation
United States

Thomas Scogland
Lawrence Livermore National Laboratory
United States

ABSTRACT

The current era of exascale supercomputing and the emergence of a computing continuum present several significant resource management challenges. These include, but are not limited to, management of complex and high-throughput scientific workflows, diverse resources such as power and network bandwidth, extremely heterogenous and dynamic systems, elasticity in user jobs, and disaggregated and converged environments. The resource models that underpin today's job scheduling frameworks reflect the node- (or core-) centric system architectures prevalent when the frameworks were designed. Consequently, they are not suited to capturing resource relationships or dynamism. The inability to express relationships or dynamism limits their applicability to the emerging multifaceted challenges in high-performance computing (HPC) and Cloud, as well as other converged environments. In this paper, we propose a scalable graph-based resource model to overcome these challenges, which allows for representation of complex, changing resource relationships and multiple containment hierarchies. We implement this model, Fluxion, in a production-quality scheduling framework, and evaluate its performance. Additionally, we present emerging and advanced scheduling use cases that are enabled by our model.

ACM Reference Format:
Tapasya Patki, Dong H. Ahn, Daniel J. Milroy, Jae-Seung Yeom, Jim Garlick, Mark Grondona, Stephen Herbein, and Thomas Scogland. 2023. Fluxion: A Scalable Graph-Based Resource Model for HPC Scheduling Challenges. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, Denver, CO, USA, 12 pages. https://doi.org/10.1145/3624062.3624286

1 INTRODUCTION

The next-generation of supercomputing requires support for complex resource types, diverse workloads, and portability to converged computing frameworks. Compute nodes on future systems are expected to have high levels of hardware resource heterogeneity, and flow resources such as network bandwidth, I/O bandwidth and power are becoming important thrust areas for exascale Workload types are changing from traditional long-running jobs, to a mix containing high-throughput computing and ensemble jobs, data-driven applications, containerized and converged applications, and elastic (moldable or malleable) applications. Newer applications are pushing boundaries of scientific knowledge through the use of large-scale coordinated workflows, in-situ workflows, *ensemble* simulations, and converged HPC and Cloud workflows. Effectively scheduling modern scientific workflows through these multi-dimensional problems is getting increasingly more difficult.

In this changing landscape, HPC resource managers and job schedulers relying on restrictive approaches to solve emerging problems create a technology gap. One such gap is in the design of *resource models*, which define how information about resource types and job requests is stored internally. Existing HPC schedulers use a node-centric or core-centric design. While such a design is ideal for traditional applications and is efficient in terms of scheduler performance, it falls short in supporting emerging HPC workloads. For example, it does not offer natural ways to express resource relationships. As a result, it cannot be used to represent complex resource constraints or to capture elasticity that is inherent to modern workloads. It also cannot be extended easily to represent extremely heterogeneous resources or flow resources such as power or network bandwidth. Such a design also limits expressibility for seamless scheduling across disaggregated resources or converged HPC and Cloud platforms [9, 28, 30].

There is a need to diverge from such node- or core-centric representations for future systems. We now identify a set of properties required for a resource model to successfully address the emerging challenges.

- *Universality and Expressibility* : Ability to model arbitrary and diverse resource types along with the various relationships between them.
- *Flexibility*: Ability to adapt to various system integrations, heterogeneous architectures, and support scheduling points at different levels of detail (eg. core, GPU, network bandwidth, power).
- *Scalability*: Ability to scale well and leverage parallelism across diverse setups, ranging from containers on a laptop, to clouds, to the world's fastest supercomputers.
- *Separations of Concerns*: Ability to define and construct the resource model separately from the scheduling policy, allowing for adaptive support for modern workflows, converged computing, and scheduling policy customizations.
- *Elasticity*: Ability to update internal representations and data structures dynamically, to support moldability, malleability and variable capacity.

In this paper, we propose a novel *graph-based* resource model, Fluxion, offering the aforementioned properties. This allows us to represent resources and their relationships precisely. The proposed model is extensible to future resource types, converged computing environments, disaggregated systems, and is able to support emerging workloads effectively. We utilize the open-source Flux framework for job management [1] to plug in our resource model.

The rest of the paper is organized as follows. Section 2 presents the limitations of existing resource models and related work. Sections 3 and 4 present the design and implementation of our graph-based resource model. Section 5 discusses how our model can be leveraged with ease for implementing advanced scheduling policies. In particular, we discuss various use-cases of our resource model, including representations for disaggregated systems, scheduling in converged environments, hierarchical scheduling, variation-aware scheduling, and novel storage solutions for exascale computing. Section 6 evaluates the performance of our graph-based resource model at various scales using optimizations such as level of detail control and pruning. Finally, Section 7 summarizes our findings.

2 LIMITATIONS OF EXISTING MODELS

Resource models refer to the internal representations and data structures used for managing resources and jobs in batch schedulers. Traditional resource models, such as those used in HPC or Cloud resource managers [3, 7, 11, 21, 22, 41, 43, 45], are often based on node-centric (or core-centric) designs and utilize bitmap-based [45], linked-list based [3], or simple static graph implementations [18, 26]. These designs emerged roughly twenty years ago when HPC systems as well as applications were far less complex than they are today and cloud technologies were not as commonplace. At that time, compute nodes were not as heterogeneous and management of flow resources (e.g., network and I/O bandwidth or power) and elasticity of jobs (moldability or malleability) were not key design considerations. Furthermore, reducing the time and space complexity of the scheduling algorithm itself was critical, due to the limited opportunity for scheduler parallelism. As a result, representations based on node-centric designs were preferred.

A major limitation of node- or core-centric models is the lack of natural support for resource *relationships*. They are not designed not accommodate complex resource requests, such as a request for a certain amount of power and network bandwidth together with a few cores and a single GPU, because these models do not have a natural notion of *containment* or different *subsystems* [26, 44]. Another limitation is the lack of support for dynamic updates and *granularity* control. Elasticity in resources as well as jobs, where the system resource or the job allocations change during job execution, is difficult for traditional models to support. Additionally, supporting disaggregated resources could lead to frequent updates to the scheduler code when rack types or relationships are modified.

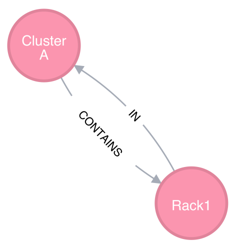
Several researchers have highlighted these limitations and proposed solutions in form of scheduling plugins along with addressing scalability challenges [12, 17, 18, 39]. For example, SLURM introduced the GRES plugin [37] and PBSPro supports custom resources [5], which allow resource and site customizations. However, these solutions do not capture relationships between resources well. Flow resources are typically handled through specialized plugins, such as power management or network-aware plugins [17, 40]. Such plugins enable simple use cases, but are not designed to interoperate, leading to scenarios where certain multi-level constraints cannot be supported. Scalability challenges can be addressed by considering workflow managers as well as hierarchical approaches [2, 4, 12, 13, 20, 21, 38, 41]. Many of these solutions focus on the scheduling policy or scalability aspects but do not consider limitations of fundamental design decisions of the resource model and internal representations. KubeFlow [26] and Poseidon/Firmament [18, 44] explore graph-based and relationship-based techniques, but their representations are simplistic and static and do not address the aforementioned challenges adequately. Addressing these challenges requires a universal, flexible, and comprehensive resource model.

3 A SCALABLE GRAPH-BASED RESOURCE MODEL

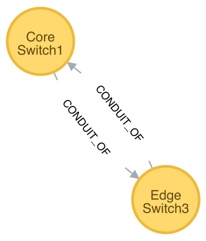
We now present the key concepts of our novel graph-based resource model, *Fluxion*. We further discuss its implementation in Section 4.

3.1 Representing Resources using a Graph

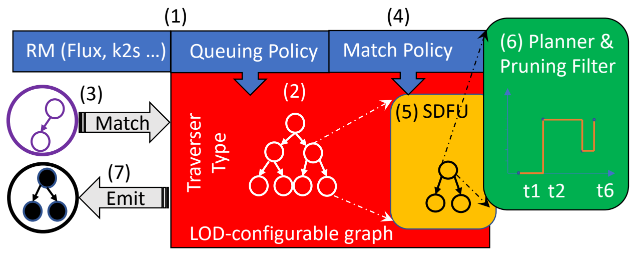
Our resource model combines two basic concepts to be able to *universally* represent resource types as well as to *express* various relationships among them: a resource pool and a graph. The *resource pool* is a group of one or more indistinguishable resources of the same kind, for example, cores, memory, or network bandwidth. Each resource in a pool is interchangeable with any other resource in the same pool, and thus, the resources in the pool are collectively represented as a quantity. We may model, for example, 512 GB of compute node memory as 16 memory-resource pools each with 32GB memory chunks, or with 8 memory resource pools of 64 GB chunks each. One of the benefits of introducing the pool concept is flexibility in describing a resource with different *levels of detail*, especially for flow resources such as network bandwidth or power. When a resource needs to be described at *coarse* granularity, it can be pooled together at a higher level. When *fine* granularity is required, the resource can be promoted to its own individual pool. Because a pool is a superset of a singleton case, a singleton resource such as a compute core can be modeled as a pool of size one.



(a) Containment



(b) Network



(c) The Tool Flow of Fluxion

Figure 1: Left and center figures are examples of graph-based modeling, and the right figure depicts flow of Fluxion scheduling.

Our second concept borrows from graph theory to describe *relationships* between individual resource pools. A directed graph consists of a set of vertices and edges, where each vertex represents an individual resource pool and an edge represents a directed relationship. Edges in the resource graph have a *type* as well as a *subsystem* name, such that the union of the set of all edges with the same name and the set of all vertices connected by these edges represents a unique resource subsystem, allowing for *containment*. Figures 1a and 1b show how our system can model various use cases, such as the `contains` relationship between a cluster resource and a compute rack or a `conduit-of` relationship between an InfiniBand (IB) core switch and an edge switch. Overall, we can represent arbitrary types of resources (universality) and many relationships, including deep hierarchies (expressibility).

3.2 Scheduling with the Graph-Based Resource Model

Our resource model, shown in Figure 1c, is designed to interoperate with and enable job scheduling of emerging HPC and Cloud resource managers (RM) such as Flux [1, 19] and Kubernetes [11] (see Step (1)). During initialization, Fluxion populates an in-memory *resource graph store* (see Step (2)), composed of vertices that represent compute or other resources, and edges that represent the relationships among those resources. This step also includes the selection of the graph's levels of detail (LOD) and traversal type. After initialization, Fluxion can receive requests from users, represented as an *abstract resource request graph* from the underlying RM (Step (3)). Job resource requirements can be expressed in terms of both node-local resources such as quantity of compute cores and memory, or in terms of higher-level resources such as compute racks, network switches, total power, etc. The abstract resource request graph becomes the input for the selected *traverser* (shown in Step (2)). The traverser walks the resource graph store in a pre-defined walking order (such as a depth-first walk) and *matches* the user request to the system resources to determine a job allocation.

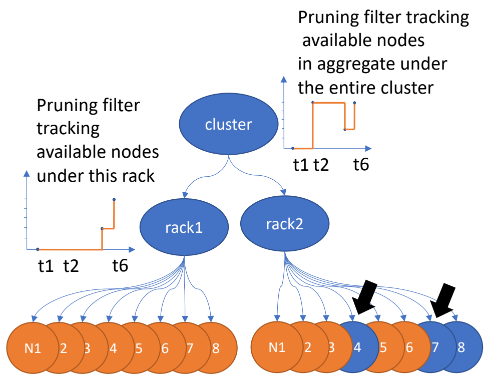
As shown in Step (4), the best matching criteria is determined by the RM-provided *match policy*, a callback plugged into the traverser, which is invoked for well-defined *visit* events, such as preorder or postorder visits. The callback evaluates how well the current resource vertex graph matches with the incoming request using a user- or admin-specified *scoring* mechanism (for example, an ID-based, a locality-aware, or a performance class based score).

Our resource model must also provide *efficient time management*: it must keep track of the state changes of resources over time in support of various queuing, reservation, and backfilling policies [15]. The model accomplishes this by directly integrating a highly efficient resource-time state tracking and search mechanism into each vertex, called *Planner*, (Step (6), Section 4.1). Even after we judiciously choose the right levels of detail for our resource graph store, the matching process for allocations requires further optimization. Thus, Fluxion utilizes *pruning filters* for scalability, as shown at Step (6). Pruning filters are installed at higher-level resource vertices (such as compute-rack-level vertices) to keep track of the aggregate amounts of available lower-level resources (e.g., individual cores). Fluxion also introduces a novel Scheduler-Driven Filter Update (SDFU) algorithm (Step (5)) to update these filters. Finally, once Fluxion determines the best matching resource subgraph, it is *emitted* as a selected resource set at (7), which can then be allocated to the user for their application by the underlying RM. The RM can then make use of this resource set to contain, bind and execute the target program(s) within those resources.

3.3 Level of Detail (LOD) Control

Our graph-based resource model supports many ways to increase or decrease the levels of detail when scheduling (e.g. core-level, memory-level, power-unit-level or rack-level). This control provides the scheduling policy an ability to make a trade-off between scheduling granularity, scalability, and performance. First, our *resource pool* concept allows flexibility in resource representations at different levels of detail. When a resource needs to be described at *coarse* granularity for higher performance, it can be pooled together with other resources of the same type as a single graph vertex. Conversely, when finer granularity is required, it can be promoted to its own vertex. Next, we provide additional coarsening and refinement control by allowing the addition or removal of certain resource vertices and edges dynamically if needed. Third, our model organizes a total graph into a set of subsystems. A simple scheduler may require a single subsystem such as containment subsystem; more complex schedulers may require multiple subsystems, such as containment and power and network. Our model allows the scheduler to name those subsystems as part of graph initialization, and Fluxion exposes only the subset of vertices and edges belonging to the subsystem of interest. We refer this technique as *graph filtering*.

Figure 2: Pruning and Scheduler-Driven Filter Update


Pruning filter tracking available nodes in aggregate under the entire cluster
Pruning filter tracking available nodes under this rack
cluster
t1 t2 t6
rack1
rack2
t1 t2 t6
N1 2 3 4 5 6 7 8
N1 2 3 4 5 6 7 8

3.4 Graph Pruning and Smart Updates

Searching a large graph in its entirety is not scalable; thus, our model incorporates *graph pruning* techniques which can aggressively reduce the search space. Fluxion supports configuring certain resource vertices to track aggregated or summary information about the subgraph underneath. For example, one or more pruning filters can be embedded into resource vertices of higher-level resource types (e.g., rack, node) to keep track of the currently available amounts of lower-level resources (e.g. cores, memory) in aggregate, which reside at a subtree rooted at each higher level vertex. If a higher level resource vertex has already been allocated exclusively, the traverser can also prune further descent to its subtree. Both of these filters use Planner, described in Section 4.1.

Aggressive graph pruning can drastically reduce the search space. However, if the performance overhead of maintaining the summary of the current subgraph state accurately is high, this reduces the benefits of pruning. To minimize this overhead, we build a novel technique called Scheduler-Driven Pruning Filter Updates (SDFU) into Fluxion. When Fluxion selects the best matching resource vertices it must also update the vertices that contain pruning filters. Figure 2 provides a simple example to show how our model provides pruning filters and scheduler-driven filter updates. Orange filled circles denote the busy nodes, and blue filled ones denote the idle ones. The graphs show node availability at various time points at each level in the resource graph strore (`cluster` and `rack` in this example). The process for maintaining these time points is explained in detail in Section 4.1. At the topmost resource vertex (`cluster`), we first find the minimum time point where an incoming job's resource request (e.g., 2 nodes for a duration of 1 unit) can be satisfied, which is determined to be time `t2`. When the traverser visits the `rack` level (`rack1`) it looks up the vertex's pruning filter and determines that no nodes are available in its subtree at `t2` so further subtree descent is pruned. Next, the traverser walks to `rack2` and finds the requested nodes and associated duration can be met in aggregate using its own pruning filter so it descends and finds two nodes to reserve at `t2`: compute node `N4` and `N7`. Once compute node `N4` and `N7` are selected, Fluxion updates the resource aggregates and time points at the ancestor vertices (i.e., `rack2` and `cluster`) that contain these two nodes appropriately.

3.5 Separation of Concerns Facilitates Adaptability

Our graph-based resource model and Fluxion implementation abstracts out the code complexity associated with managing many resource types and relationships. The scheduling policies can remain a generic and independent codebase providing relevant graph algorithms (such as graph traversals with predefined walking order and match evaluation callback scores on well-defined traversal events), and do not have to provide an underlying resource representation. Similarly, general queuing and backfilling policies can interoperate with the resource model separately. This decoupling significantly facilitates adaptability of scheduling to HPC, Cloud, and other distributed frameworks. For example, this separation of concerns allow our resource model to be plugged into a Cloud computing RM such as Kubernetes to providing HPC awareness into its scheduling [33].

4 IMPLEMENTATION OF FLUXION

We now discuss details of an open-source, extensible software implementation embodying our novel resource model, *Fluxion*, described in Section 3. In particular, we discuss the details of the Planner as well as the algorithmic implementation of the operations on the graph-based resource model.

4.1 Planner: Scalable Scheduled Time Points Management

Planner offers an abstraction similar to a physical calendar planner. It can keep track of resource state changes for each vertex in the resource graph store, and allows for updates and querying of resource states. Planner provides common query operations such as finding an *earliest fit* for an allocation, or checking resource satisfiability at a given time. Time management is based on a concept referred to as a *span*, which is similar to how one would mark activities in a real calendar with durations. Figure 3 shows an example of resource state changes with Planner. Here, we consider an unnamed resource pool (such as memory), whose schedulable quantity is 8, and three incoming job requests. The vertical axes represent the available resource amount and the horizontal axes are the time ticks. Each job request is represented as a tuple consisting of the amount of the resource, duration and the scheduled time point. For example, `<8,1,0>` in Figure 4a denotes that the request uses 8 units of the resource for 1 unit of duration at time point 0. When this job is added to Planner, a span of duration of 1, consisting of two scheduled points (`t0` and `t1`) is created, and the currently available resource amount for each of the two scheduled points is updated: at `t0`, no resources are available as a total of 8 units of resource are allocated to the job. And at `t1`, the available resource count grows to be 8 again as the job completes. Similarly, the scheduled time points are updated as two other jobs are added: `<3,3,1>` and `<7,1,6>` as shown in Figures 4b and 4c. Figure 4d shows 4 time points where a new job can be scheduled and how Planner can be used for incoming requests. Here, a common query could be: can a request of 5 resource units for a duration of 2 be planned at time points such as `t1` or `t6`? The answer is `yes` for `t1` given `p1`, while `no` for `t6` given `p3`. Another common query is: given a job with 6 resource units for 1 duration unit, what is the earliest timed point at

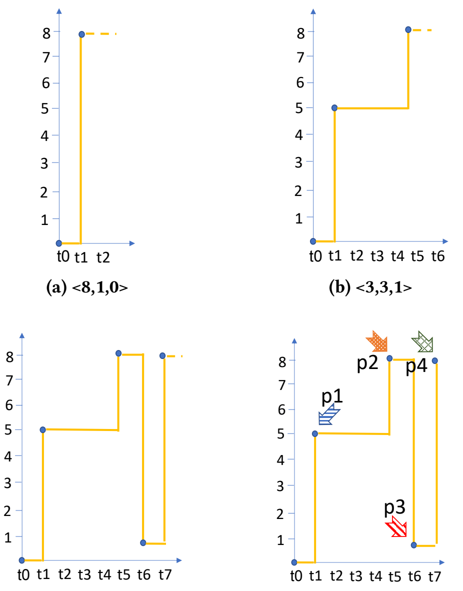


Figure 3: Planner Example

(a) <8,1,0> **(b) <3,3,1>** **(c) <7,1,6>** **(d) SchedPts**

which it can be scheduled? How about a duration of 2? The answer for the 1-duration case is `t5` given `p2`, and for the 2-duration case is `t7` given `p4`.

Planner uses two efficient red-black (RB) self-balancing binary search trees for scalability: a) a scheduled-point (SP) tree, where the key is the time field of the scheduled point; and b) an earliest-time (ET) resource *augmented* tree, where the key is the remaining amount of resource. When a new span is created, Planner inserts or updates the data of two scheduled points corresponding to the start and end times of the span in both trees. The SP tree allows time-based queries such as finding the available resource amount at time `t` to be performed in $O(logN)$ (*allocations* when resources are available), and the ET tree finds the earliest time at which a request can be met with $O(logN)$ (future *reservations*).

While the SP tree uses a standard search algorithm, our ET tree search technique is novel. It uses a data structure referred to as an *augmented tree*, in which each node in the tree not only stores the main key-value pair (the key is the remaining resource quantity and the value is the scheduled time point), but also additional data (the `earliest-at` time for the underlying subtree). When a new scheduled point is inserted, we traverse the tree to query the resource availability, and also update the subtree with the `earliest-at` time for each traversing node if the new time point is earlier than the current subtree value. This allows for a quick retrieval of the earliest scheduled time point for the subtree.

Algorithm 1 shows this pseudo-code for a resource request (*request*) on an ET tree whose root is *etRoot* and FINDEARLIESTAT is the entry function. This is one, novel step in the overall procedure for determining when a request can be allocated and is utilized for making reservations. A request consists of a resource specification as well as a desired execution duration. To search for the earliest time in the future at which the request can be satisfied,

Algorithm 1 Earliest Time Resource Augmented Tree Search Algorithm

```
function RIGHTET($rbNode$)                          ▷ Right branch subtree also satisfies
    $data \leftarrow rbNode_{data}$
    $rightVtx \leftarrow rbNode_{right}$
    $rightData \leftarrow rightRbNode_{data}$
    return MIN($data_{time}, rightData_{subtreeEarliestAt}$)

function FINDETPOINT($anchor, earliestAt$)
    Binary search of the subtree rooted at anchor;
    Locate a rbNode whose scheduled time = $earliestAt$;
    Return that rbNode's data;

function FINDANCHOR($request, etRoot, anchor$)
    $rbNode \leftarrow etRoot_{rbNode}$
    $earliestAt \leftarrow MAXTIME$
    $rightEarliestAt \leftarrow MAXTIME$
    while $rbNode \neq NULL$ do                     ▷ Begin binary search
        $data \leftarrow rbNode_{data}$
        if $request \leq data_{remaining}$ then
            $rightEarliestAt \leftarrow$ RIGHTET($rbNode$)
            if $rightEarliestAt < earliestAt$ then
                $earliestAt \leftarrow rightEarliestAt$     ▷ Current best time
                $anchor \leftarrow rbNode$                  ▷ Current best rbNode
            $rbNode \leftarrow rbNode_{left}$               ▷ Search the left subtree
        else
            $rbNode \leftarrow rbNode_{right}$              ▷ Search the right subtree
    return $earliestAt$

function FINDEARLIESTAT($request, etRoot$)
    $anchor \leftarrow NULL$                        ▷ returned by FINDANCHOR
    $earliestAt \leftarrow$ FINDANCHOR($request, mtRoot, anchor$)
    return FINDETPOINT($anchor, earliestAt$)
```

Fluxion begins at the resource graph store root, where it finds the earliest time point at which the aggregate counts of all requested resources can be satisfied (PLANNERMULTIAVAILTIMEFIRST). It does this by iteratively querying the root planner for *each* resource type (for example, the memory resource pool has a separate planner than the network resource pool). For each resource type planner, PLANNERAVAILTIMEFIRST finds the earliest time when that type of resource is available for the specified `duration` on `or after` the current query time (`at`).

Computing when a specific resource planner can satisfy the request at the query time involves Algorithm 1. Fluxion begins with a loop (AVAILAT) with a starting scheduled point equal to FINDEARLIESTAT(`request`) shown in Algorithm 1. Then, it checks if the time value of the returned earliest point is before `at` (which can occur when the top-most loop (containing PLANNERMULTIAVAILTIMEFIRST) advances the `at` time). If so, the loop enters the next iteration and finds the next point that satisfies the request. Once the time value of the returned point is equal to or after `on or after`, Fluxion checks if the span from that point through the duration of the request can satisfy the request by iteratively searching the SP tree for a later scheduled point (SpanOk). If the next point is after `at + duration`, then the span satisfies the request, and the `at` time is returned up the call stack. SpanOk also determines if the resource request is greater than the available resources of the next scheduled point. If so, SpanOk exits and the next loop iteration of AvailAt begins with the next point in the ET tree.

4.2 Mapping User Input for Graph Matching

Fluxion leverages Flux's canonical job specification for user input, which is a graph-oriented domain-specific language based on YAML and is defined to express the resource requirements along with other attributes.The resource section of a `jobspec` in YAML format is the main input to Fluxion, and represents the abstract resource request graph. Figure 4 shows the graphical representations of the resource sections of three simple job specifications. In all three examples, each vertex except for `slot` represents a physical hardware resource type along with its requesting quantity: its label is the type name and quantity delimited with a colon. Each edge represents `contains` relationship between two connecting vertices. A box-shaped vertex means that the corresponding resource need to be exclusively allocated, and a circular-shaped vertex denotes that the resource can be shared between different jobs. The `slot` is the only vertex type which does not represent a physical resource constraint and thus is drawn in its own shape (diamond). Each `slot` vertex specifies that the subgraph under it is the resource *shape* in which the program processes will be contained, bound, and executed, and all of the subgraph vertices under it must be *exclusively* allocated to those processes. Figure 5a, for instance, requests an exclusive allocation of 1 `slot` that contains 2 `sockets`, each with 5 `cores`, 1 `gpu` and 16 allocable `memory` units (e.g., 16 GBs) within a compute `node` that can be shared with other jobs. By contrast, Figure 5b has a higher `rack`-level constraint, as it requests 4 `slots` each containing 2 compute `nodes` each with at least 22 `cores` and 2 `gpus`. Because of the high level constraint, those `slots` (and thus `nodes`) must be spread across 2 compute `racks`. Finally, Figure 5c requests an exclusive allocation of 128 I/O `bandwidth` units (e.g., 128 GBs) within a parallel file system (`pfs`) which is in the same `zone` as `cluster` that contains these `nodes`.

5 EMERGING USE CASES

Our model enables several advanced scheduling use cases. We provide a summary of these use cases, as detailed discussions on each of these use cases would involve individual research papers and are not the focus of this foundational resource modeling paper.

5.1 Near Node Flash Storage on the Upcoming El Capitan Supercomputer

Modern scientific workflows, such as those based on Artificial Intelligence/Machine Learning (AI/ML), in-situ analysis and reductions, and complex multi-component simulations are changing and stressing the parallel file systems (for I/O) in supercomputers in new ways. For example, AI/ML workflows require training phases, which perform a very large number of small and random reads. Existing parallel file systems, such as Lustre, often encounter contention while supporting such a workflow, leading to low performance as well as high performance variability. Local filesystems with directly attached low-latency storage devices are necessary to maintain high performance for such modern scientific applications. Additionally, the ability to *dynamically configure* storage, to offer either node-local storage or common storage across all compute nodes, is an important criteria for upcoming supercomputers.

Near Node Flash, also known as *rabbit*, provides a disaggregated chassis-local storage solution which is designed to support a wide range of use cases, including resolving network bursts, optimizing input, and even running analysis processes. Prominent and upcoming supercomputers, such as the El Capitan supercomputer at Lawrence Livermore National Laboratory, are being designed to have a multi-tiered storage system centered around such rabbits, where each rabbit node has a processor and a collection of SSDs. There will be one rabbit per compute chassis, which contains some small fixed number of compute nodes. Rabbits will allow the SSDs to be dynamically configured to offer either node-local storage or storage common to all compute nodes in a job. Node-local storage is connected by PCIe, whereas global storage is over the network.

The fact that rabbits can be dynamically configured to be different kinds of storage makes them extremely challenging to schedule with traditional resource managers such as SLURM or PBS. If a job asks for a global file system, the scheduler needs to be aware that it can pick any rabbit. If a job asks for node-local storage, the scheduler needs to be aware that when it picks compute nodes for the job, it must pick compute nodes whose rabbit has enough storage available. If a job asks for both global storage and node-local storage, the scheduler needs to make multi-tier decisions. An additional complication is that that users can allocate rabbits independently of jobs, which is useful for when users want to keep a file system around for multiple jobs to use at different times. This requires the scheduler to support resource allocations which have no compute resources associated with them. There are further constraints about the number and types of storage that can be combined on a single rabbit. For instance, there is a limit on the number of NVMe namespaces supported by each SSD, so trying to create many different file systems on a single rabbit will fail. The scheduler needs to be aware of this constraint. Similarly, the Lustre file system underpins the global rabbit storage, and one of the Lustre servers that runs on a rabbit requires a unique IP. This means there can only be one such server per rabbit.

Fluxion's graph-based resource model, along with Flux, handles all of the rabbit scheduling complexity, and supports all aforementioned use cases of node-local, global, and storage-only allocations scalably. The rabbit nodes are modeled simply a vertex in the resource graph, with edges from both "rack" (or chassis) and "cluster", representing the fact that rabbits can be either a cluster-level or a rack-level resource. SSD namespaces can be added vertices associated with each rabbit, and each rabbit has a single "IP" vertex to ensure that two Lustre file systems are not scheduled on the same rabbit.

5.2 Performance Variability Aware Scheduling

Our resource model can be used effectively for designing advanced scheduling policies. As an example, we consider the well-established issue of processor manufacturing variation in power-constrained

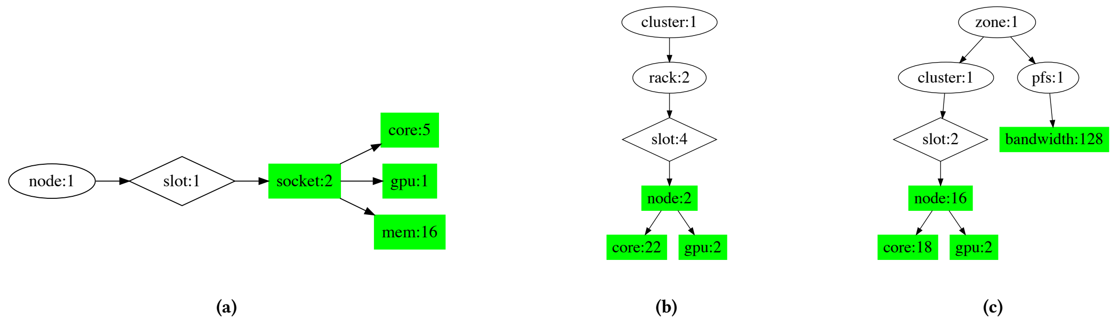


Figure 4: Abstract Resource Request Graph Examples:
(a) Node-centric constraints, (b) Simple global constraints, (c) I/O constraints

supercomputing [23, 34]. Manufacturing high-performance processor architectures that behave efficiently under power constraints is becoming increasingly challenging during the lithography process. In the past, significant processor level performance improvements could be obtained by increasing the transistor density on a die without impacting its power. This was accomplished by shrinking the feature size of transistors. However, with the end of Dennard scaling, shrinking a transistor below a certain level leads to increased leakage currents and heat dissipation, resulting in significant power variation in modern processors [10, 24]. This variation in power can be catastrophic to performance, as it translates directly into variation in CPU frequencies. As a result, processors with the same microarchitecture that are expected to be homogenous can exhibit significant differences in CPU performance, making applications sensitive to node placement and impacting their execution times drastically and causing load imbalance.

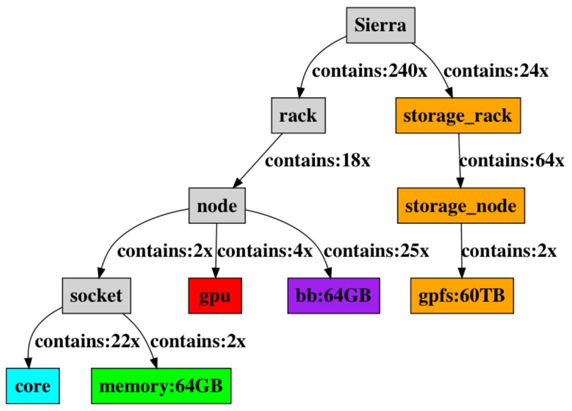
Manufacturing variation can manifest in two ways for applications. First, *rank-to-rank* variation can occur within the application resulting in unforeseen slowdowns and load imbalance. Second, *run-to-run* variation can occur, wherein subsequent executions of the same application get distinct physical allocations, and as a result exhibit differences in performance and a lack of reproducibility. We focus on mitigating *rank-to-rank* application performance variation. Typically, these variations can be categorized deterministically by profiling all the nodes in the HPC system at bring-up time across a diverse set of benchmarks. Such benchmarking allows for ranking nodes within the system in terms of performance and power efficiency. For the scope of the discussion in this paper, we assume that we have a distribution of nodes that can be binned into a few *performance classes* in advance for a HPC system, where each performance class contains a set of nodes that depict relatively similar performance on the selected benchmarks. Determining these performance classes is an orthogonal research problem that we do not address in this paper.

Given such performance classes and node distributions, we can then design a policy to mitigate rank-to-rank variation – an application's ranks are allocated either to the same performance class if possible; and if not, the application's ranks are placed in a manner that they are not spread across a wide set of performance efficiency classes. Such a policy can be easily implemented and represented with our novel resource model, by adding a vertex for an associated performance class for each compute node in the resource store graph. Based on each performance class, a score and a matching policy can be determined to mitigate variation in application ranks. We show the scheduler performance scaling results from a variation-aware policy we implemented based on this idea of performance classes in Section 6.3.

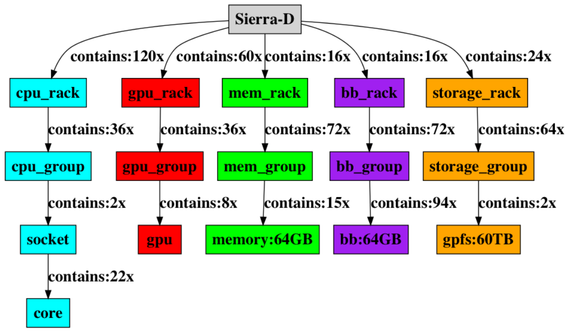
5.3 Converged Computing: Cloud, Fog, and Edge

Scheduling techniques for a computing continuum [8, 9, 29] are an active area of research. Traditional HPC schedulers are inherently challenged by newer trends toward complex scientific workflows that require the use of heterogeneous compute hardware and demand elasticity as well as portability. Similarly, the cloud, fog and edge computing communities have been exploring options for higher performance and resiliency for their elastic and portable workflows. A converged environment, bringing together the best of these communities, is rapidly emerging, along with a need for expressible and performant resource models.

Kubernetes [11], the de facto standard for lifecycle orchestration of containerized applications, and their software defined networks, is a prominent example of cloud technology integrating HPC techniques. The focus of Kubernetes has been the declarative management of loosely-coupled services, a design choice that has performance implications for containerized, orchestrated HPC workloads. The resource model and scheduler used by Kubernetes is simplistic in comparison to the sophisticated expression capabilities of Fluxion. Recent work to standardize interfaces for plugin schedulers includes Fluence, an embodiment of Fluxion, to provide HPC-grade scheduling for Kubernetes and high-performance for MPI-based containerized workloads [31, 32]. Deploying Fluence on cloud resources results in much higher application performance for MPI-based containerized workloads [31]. As Kubernetes continues to develop rapidly, the Fluxion model may offer even greater benefits to elastic, resilient, and dynamic workflows running in the cloud.



(a) Sierra Supercomputer



(b) Sierra Supercomputer (Disaggregated View)

Figure 5: Example Modeling for Traditional and Emerging Architectures

5.4 Management of Disaggregated Systems

Resource disaggregation has been proposed for HPC as well as Cloud systems recently to improve overall utilization. Promising proposals for resource disaggregation include *specializing* system racks such that the resources of a same type are populated into each different rack type, and connected with a high-performance network such as an optical network [6, 16, 35, 42]. With our resource model, scheduling such a disaggregated architecture becomes fundamentally the same as scheduling with a traditional containment hierarchy. Figures 5a and 5b show a comparison between a traditional containment-based supercomputer and an equivalent disaggregated supercomputer. The disaggregated supercomputer contains specialized racks for CPUs, GPUs, memory, and burst-buffers. With traditional resource models that are node-centric in their allocations, managing internal representations and data structures for such a disaggregated system is challenging and high-overhead, both in terms of developer time as well as complexity of user requests. This is because traditional models cannot easily represent or schedule across different subsystem hierarchies (such as disaggregated racks or scheduling only across the GPU-racks), and cannot coarsen the levels of detail when scheduling. Our graph-based resource model enables this seamlessly, providing universality, expressibility, flexibility, scalability and the much required separation of concerns between the data structures and scheduling policies.

5.5 Enabling Elasticity

Elasticity for user applications [28, 36] as well as variable capacity in system resources [27, 46] introduces significant adaptive scheduling challenges, including growing and shrinking of user allocations as well as system resources. Through separation of concerns, universality and expressibility, our model can enable dynamic updates to the system resource graph store as well as to user resource request graphs, allowing us to schedule across subsystems as well as resources that are added or removed even after system initialization – a feature that existing resource models cannot support easily.

5.6 Fully Hierarchical Scheduling

Our resource model is also designed to support a fully hierarchical scheduling model [19] for extreme scalability. Currently, only the Flux RM offers this model with multi-levels, and other resource managers such as PBSPro [4] or Omega [41] offer this in a limited capacity. Under the Flux design, any instance can spawn child instances to aid in scheduling, launching, and managing jobs. The parent instance grants a subset of its jobs and resources to each child. This parent-child relationship, can extend to an arbitrary depth and width, enabling high throughput as well as customized scheduler specialization [20].

6 EXPERIMENTAL RESULTS

We now evaluate the performance of Fluxion. Our experiments were conducted on a single node of a 11.3 petaflop supercomputer, Corona, which consists of 291 nodes, out of which 285 are batch nodes. It consists of 170 AMD Naples and 121 AMD Rome CPUs. Each node has 48 cores per node, and total system memory is 127,488 GB, connected through HDR InfiniBand.

6.1 Tradeoffs of Using Different Levels of Detail

We use `resource-query` to evaluate the effect of different levels of detail. This is a command-line utility that reads in a resource-graph generation recipe written in the GRUG (Generating Resources Using GraphML) format, populates the resource graph store, and takes as input an abstract resource request graph written in Flux's canonical job specification. After a traversal, it selects the best-matching resources in accordance with a selection policy. The utility can be executed on a single compute node, and can simulate a system resource graph store of up to thousands of nodes.

We create four GRUG files, each of which configures a medium-size system with 1008 compute nodes using different levels of detail: `High`, `Med`, `Low`, and `Low2`. Graphs created by these GRUG files consist of a set of vertices and edges, where each resource vertex (except for the root) is configured to have in and out edges to its child and parent in the containment subsystem, labeled `contains` and `in` respectively.

The `High` LOD models this system such that both global- and node-local-level resource constraints can be considered for matching. Specifically, the system is configured to have one cluster vertex connected to 56 compute rack vertices, each then connecting to 18 compute node vertices. Each compute node vertex has in/out edges from/to 2 socket vertices, each of which have edges to 20 core vertices, 2 GPU vertices, 8 memory pool vertices representing 16GB of memory, and 8 burst-buffer vertices each modeling 100GB of SSD storage.

The `Med` LOD configures the same size system but coarsens this model at the node-local level by removing the socket vertices and reducing the schedulable granularity of memory and burst buffers: each compute node vertex now has in/out edges from/to 40 core vertices, 4 GPU vertices, 8 memory pool vertices each representing 32GB of memory, and 8 burst-buffer pool vertices each representing 200GB of SSD storage.

`Low` further coarsens this model both at the global and node-local levels. It removes the rack vertices and compute cores are also federated into core resource pool vertices, each representing 5 cores. Memory and burst buffer granularities are reduced such that we have 4 memory pool vertices each representing 64GB and 4 burst-buffer pool vertices each modeling 400 GB of SSD storage. `Low2` is identical to `Low` except that it does not remove the rack vertices.

We also created a job specification (abstract resource request graph) that requests 10 cores, 8GB memory, 1 burst buffer on a node and issued the `match allocate` command within `resource-query` until the configured system becomes fully allocated. Finally, we ran this test with and without a pruning filter configured with the core resource type and measure the matching time for each request, and report the average values.

Figure 7a shows the effect of different levels of detail on the average match performance. The y-axis represents time taken for matching all requests, and the x-axis represents the levels of detail with and without pruning. Lower values are better on this graph. As we coarsen the resource model, the match performance and pruning filter improves as expected. We observe that pruning performance is slightly better when the GRUG includes the rack-level vertices, as the pruning is done at a higher level (Low Prune vs. Low2 Prune).

6.2 Performance of Planner-based Time Management

We empirically evaluate the performance of the Planner discussed in Section 4.1. A total of 128 units of unnamed resource is planned for a maximum time of 12 hours. We begin by inserting a set of pre-populated spans (going up to 1 million spans, which would represent 2 million scheduled points) into a single Planner object. Each request is a tuple, denoted as $< r, d >$, where r is the amount of resource requested (sampled from a uniform distribution between 1 to 128) and d is the duration of the request (sampled from the uniform distribution between 1 to 43200, or 12 hours). We use conservative backfilling. We have the following tests:

- `SatAt`: How quickly can a new request R with increasing amounts of r with a unit duration be satisfied at a *random* time t? That is, $R :< r, 1 >$, where r increases from 1 to 128 with powers of two.
- `SatDuring`: How quickly can a new request R with durations from [1, 12h] be satisfied at a random time t? Or $R :< r, d >$, where r is from 1 to 128 with powers of two and d is sampled from 1 to 43200 (12h)
- `EarliestAt`: How quickly can we find the earliest fit for a new request R with increasing amounts of r? Or $R :< r, 1 >$, where r increases from 1 to 128 with powers of two.

Figure 7b shows the measured performance for these three tests. The x-axis shows the pre-populated load (spans), and the y-axis represents time taken for the associated Planner query. As expected, the query performance trends are logarithmic. This suggests that not only the standard binary tree algorithm but also our minimum-time resource tree algorithm described in Algorithm 1 are logarithmic with respect to the numbers of the existing spans in Planner.

6.3 Results from Variation-Aware Scheduling Case Study

For our evaluation, we first took a snapshot of the job queue from a 2604-node production cluster, `quartz`, comprising of Intel Broadwell nodes, with 36 cores per node. Our job queue snapshot comprised of 467 user jobs, including jobs that were currently executing and jobs that were pending in the queue.

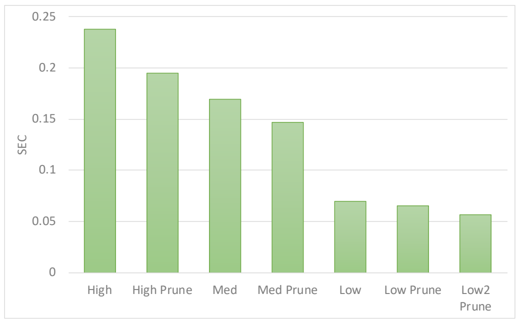
The `quartz` cluster is organized in 42 racks, with 62 nodes per rack, with a total of 2604 nodes. Using two simple benchmarks, we collected single-node variation data under a socket-level power cap (of 50 W) to study performance variability resulting from manufacturing differences across different nodes in the cluster, where each single-node experiment was repeated five times. We used NAS MG (Class C) [14], a multi-grid algorithm, and LULESH [25], a code that approximates hydrodynamics equations discretely, for benchmarking. We observed a 2.47x performance difference between the slowest and the fastest node for MG, and a 1.91x difference for LULESH. For this study, we were only able to obtain data for 2469 nodes of quartz. For simplification, we considered only 39 full racks, or 2418 nodes.

We grouped the nodes based on a combined time score for each node (t_{norm_i}, for node i) based on the median values of the two benchmarks obtained on each node, and created five performance classes. We show the grouping in Equation 1, where p represents a performance class, and a lower value for p means a more efficient set of nodes (top 10% nodes are performance class 1, 10-25% are performance class 2, and so on). Figure 7(a) depicts a histogram of the 2418 nodes across 5 performance classes based on the ranges specified in Equation 1 based on our real-world dataset. We pick these specific ranges just for demonstration of our *Fluxion* scalability.

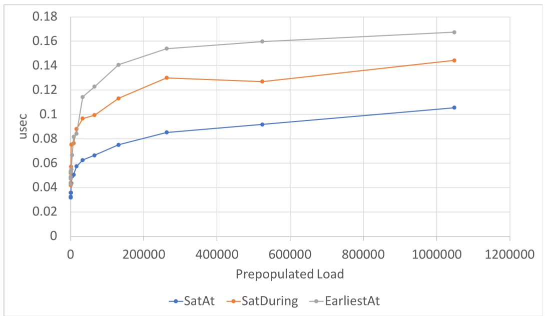
$$p = \begin{cases} 1, & \text{if } 0 \leq t_{norm_i} \leq 0.10 \\ 2, & \text{if } 0.10 < t_{norm_i} \leq 0.25 \\ 3, & \text{if } 0.25 < t_{norm_i} \leq 0.40 \\ 4, & \text{if } 0.40 < t_{norm_i} \leq 0.60 \\ 5, & \text{if } 0.60 < t_{norm_i} \leq 1.0 \end{cases} \tag{1}$$

We then randomly chose 200 of the jobs from the job queue as an initial trace, and used the node count and the duration from

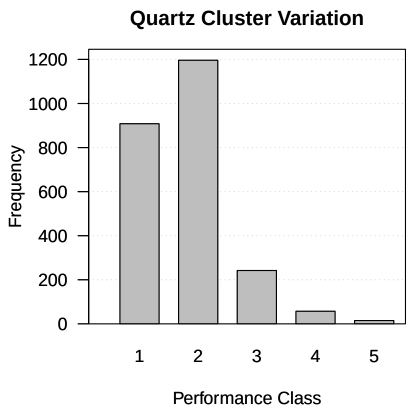
Figure 6: Performance of Fluxion



(a) Impact of LOD and Pruning



(b) Planner Performance



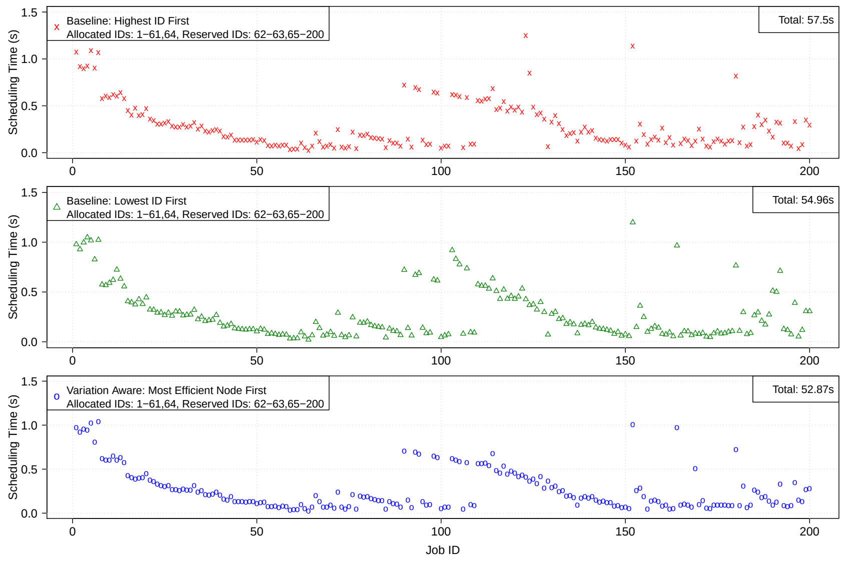


Figure 7: (a) Performance Classes, (b) Scheduling overhead (including pruning).

these jobs to generate YAML-based `jobspec` files as input. Our goal here was to measure the scheduling time required using our graph-based model. For evaluating scheduling overhead, we also chose two baseline policies from `resource-query` to compare with our `var-aware` policy. The first baseline policy prefers nodes with higher IDs, and the second prefers nodes with lower IDs. These baseline policies represent how most production HPC clusters operate today, where they do a simple compute node ID based sorting when assigning nodes. Note that all three policies use backfilling.

Figure 7(b) depicts the time taken in seconds to schedule each of the 200 jobs with the three policies. It also shows the total time taken to finish the entire job queue on the top right. It is important to note that when the cluster was empty, the matcher had to do more computation to determine the best allocations for jobs, resulting in a higher scheduling time that can be seen in the first few jobs. However, once steady state was in progress, the scheduling time decreased significantly. In our runs, out of the 200 jobs, 62 jobs were scheduled immediately and for the rest, resources were reserved into the future. All three policies exhibited similar scheduling times, although our variation-aware policy was about 10% faster than the policy that picked the highest ID first. This 10% time improvement is specific for our selected node distribution and this particular job trace, and should not be considered as a general result.

$$\begin{aligned} P_j &:= \{p_a | a \in n \wedge allocated(a, j)\} \\ fom_j &= max(P_j) - min(P_j) \end{aligned} \tag{2}$$

Next, we evaluate the effectiveness of our variation-aware policy by calculating the *figure of merit* of each job in our trace, as

Table 1: Comparison of the three policies in terms of rank-to-rank variation. The table shows the number of jobs with a certain value of figure of merit. Having many jobs with a zero or one figure of merit value is considered good, because this indicates that the jobs were allocated similar performing resources and variation was minimized.

Policy	$fom = 0$	$fom = 1$	$fom = 2$	$fom = 3$	$fom = 4$
HighestID	66	54	47	27	6
LowestID	79	34	43	33	11
Variation-aware	184	7	8	1	0

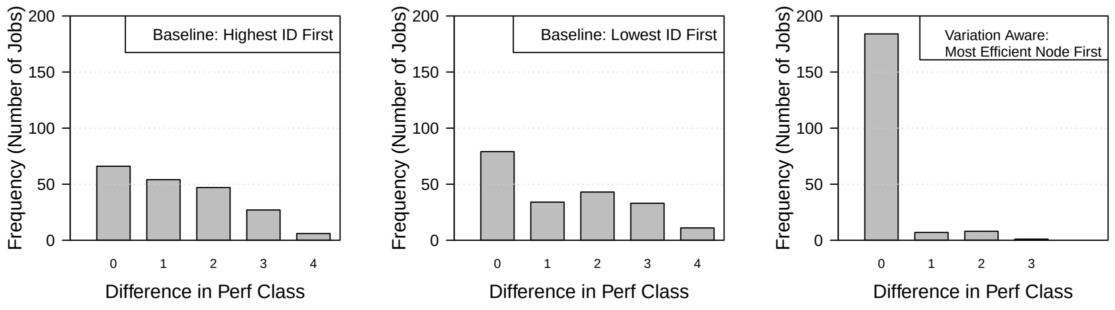


Figure 8: Results of the variation-aware policy depicting significant reduction in performance variation

shown in Equation 2. Rank-to-rank variation for an application can be minimized by ensuring that the allocated nodes span as few performance classes as possible. If $allocated(a, j)$ returns true when node a has been allocated to job j, we can determine the figure of merit a single application as shown in Equation 2. Here, P_j is the set comprising of the performance class associated with each node that is allocated to the job. When fom_j is zero, it means that the application will exhibit little or no variation as it is scheduled on similar performing nodes. A good scheduling policy will try to maximize the number of jobs that have a zero or low fom_j. We can thus gauge the effectiveness of a policy by looking at the number of jobs for which the difference in performance classes was zero. It is important to note here that the *number of performance classes* chosen plays an important role, and we assume that a reasonable number of classes is chosen. In our case, we chose 5 performance classes, as depicted in Equation 1. If there was only a single performance class, fom_j would always be zero and will fail to capture the high amount of variation that jobs incur. If we had too many performance classes, achieving a zero fom_j will not be possible for a large number of cases, but a low fom_j should suffice instead. We don't explore the impact of these choices in this paper, and plan to include this in our future work.

The figure of merit results are presented as a histogram in Figure 8, and the detailed data is shown in Table 1. For our discussion, the range of values for figure of merit is from zero through four, as we have a maximum of five performance classes. As can be observed, our variation based policy reduces rank-to-rank variation by 2.8x and 2.3x compared to the policies that select the highest and the lowest IDs, respectively. Additionally, it does not schedule any jobs with a figure of merit value of 4, and only one job with a figure out merit value of 3, resulting in significant reductions in variation within a job. As part of our future work, we will evaluate more job traces and node distributions, along with techniques to determine performance classes with machine learning.

7 CONCLUSIONS

We presented a novel graph-based resource model and its implementation, Fluxion, for HPC and converged computing scheduling. Our model addresses the limitations of existing node-centric models, and supports converged computing environments, disaggregated systems, and complex and elastic modern workflows. We described the novel architecture, procedures for improving scalability, and algorithms comprising Fluxion and discussed how these contributions allow Fluxion to schedule resources efficiently. We discussed several emerging use cases which our model enables and evaluated its scalability. Future work involves optimizing our model's implementation and applying it to scheduling environments with dynamic resources and hierarchies. The powerful combination of a directed-graph resource model with fully-hierarchical scheduling and architectural separation of concerns allows Fluxion to schedule virtually any resource types, including types not yet devised.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-855103) and was supported by the LLNL-LDRD Program under Project No. 22-ERD-041. Authors also wish to thank James Corbett, Al Chu, Don Lipari, Christopher Moussa, Chris Morrone, and Becky Springmeyer from the Flux team at Lawrence Livermore National Laboratory.

REFERENCES

[1] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. 2014. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In *Proc. of the 10th Intl. Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*.

[2] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. 2021. RADICAL-Pilot and Parsl: Executing Heterogeneous Workflows on HPC Platforms. arXiv.

[3] Altair. [n. d.]. PBSPro: An HPC workload manager and job scheduler for desktops, clusters, and clouds. https://github.com/PBSPro/pbspro.

[4] Altair. 2023. Hierarchical Scheduling for High-throughput Computing Workloads in PBSPro. https://www.altair.com/resource/hierarchical-scheduling-for-high-throughput-computing-workloads.

[5] Altair. 2023. Using PBS Professional Hooks: Examples and Benefits. https://www.scientific-computing.com/sites/default/files/PBS_TechPaper_hooks_08.29.12.pdf.

[6] Marcelo Amaral, Jordà Polo, David Carrera, Nelson Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D'Amora, Alaa Youssef, and Malgorzata Steinder. 2021. DRMaestro: Orchestrating Disaggregated Resources on Virtualized Data-Centers. *J. of Cloud Computing* (Mar 2021).

[7] Amazon, Inc. 2023. AWS Batch Scheduler. https://docs.aws.amazon.com/batch/latest/userguide/what-is-batch.html.

[8] Gabriel Antoniu, Patrick Valduriez, Hans-Christian Hoppe, and Jens Krüger. 2021. Towards Integrated Hardware/Software Ecosystems for the Edge-Cloud-HPC Continuum.

[9] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and M. Parashar. 2019. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The Intl. J. of High Performance Computing Applications* (2019).

[10] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *Micro, IEEE* (Nov 2005).

[11] Cloud Native Computing Foundation. [n. d.]. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[12] Yiqin Dai, Yong Dong, Kai Lu, Ruibo Wang, Wei Zhang, Juan Chen, Mingtian Shao, and Zheng Wang. 2022. Towards Scalable Resource Management for Supercomputers. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*.

[13] Ewa Deelman, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. 2019. The Evolution of the Pegasus Workflow Management Software. *Computing in Science and Engineering* (2019).

[14] Rob F. Van der Wijngaart and Haoqiang Jin. 2003. *NAS Parallel Benchmarks*. Technical Report.

[15] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. 1997. Theory and Practice in Parallel Job Scheduling. In *Proc. of the Job Scheduling Strategies for Parallel Processing*.

[16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. [n. d.]. Network Requirements for Resource Disaggregation. In *12th USENIX Conf. on Operating Systems Design and Implementation*.

[17] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. 2017. Topology-Aware Resource Management for HPC Applications. In *Proc. of the 18th Intl. Conf. on Distributed Computing and Networking*.

[18] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*. 99–115.

[19] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, T Patki, Thomas R. W. Scogland, Becky Springmeyer, and Michela Taufer. 2018. Flux: Overcoming Scheduling Challenges for Exascale Workflows. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 10–19.

[20] Stephen Herbein, Tapasya Patki, Dong H Ahn, Sebastian Mobo, Clark Hathaway, Silvina Caíno-Lores, James Corbett, David Domyancic, Thomas RW Scogland, Bronis R de Supinski, and Michela Taufer. 2022. An analytical performance model of generalized hierarchical scheduling. *The Intl. J. of High Performance Computing Applications* 36, 3 (2022).

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing. In *USENIX Conf.e on Networked Systems Design and Implementation*.

[22] IBM. 2020. IBM LSF Scheduler. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0.

[23] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. 2015. Analyzing and Mitigating the Impact of Manufacturing Variability in Power-constrained Supercomputing. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[24] Sudhakar Jilla. 2013. Minimizing The Effects of Manufacturing Variation During Physcial Layout. *Chip Design Magazine* (2013). http://chipdesignmag.com/display.php?articleId=2437.

[25] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[26] KubeFlow. [n. d.]. The Machine Learning Toolkit for Kubernetes. https://www.kubeflow.org/docs/components/pipelines/v1/concepts/graph/.

[27] An-Dee Lin, Chung-Sheng Li, Wanjiun Liao, and Hubertus Franke. 2018. Capacity Optimization for Resource Pooling in Virtualized Data Centers with Composable Systems. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[28] Feng Liu. 2018. Elastic Scheduling in HPC Resource Management Systems. https://hdl.handle.net/11299/202169. [University of Minnesota].

[29] Khaled Matrouk and Kholoud Alatoun. 2021. Scheduling Algorithms in Fog Computing: A Survey. *Intl. J. of Networked and Distributed Computing* (2021).

[30] Satoshi Matsuoka, Jens Domke, Mohamed Wahib, Aleksandr Drozd, and Torsten Hoefler. 2023. Myths and Legends in High-Performance Computing. arXiv.

[31] Daniel J. Milroy, Claudia Misale, Giorgis Georgakoudis, Tonia Elengikal, Abhik Sarkar, Maurizio Drocco, Tapasya Patki, Jae-Seung Yeom, Dong H. Gutierrez, Carlos Eduardo Arango Ahn, and Yoonho Park. 2022; in press. One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC. In *2022 4th Intl. Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*.

[32] Claudia Misale, Maurizio Drocco, Daniel J. Milroy, Carlos Eduardo Arango Gutierrez, Stephen Herbein, Dong H. Ahn, and Yoonho Park. 2021. It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler. In *3rd Intl. Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC*. 10–16.

[33] Claudia Misale, Daniel J. Milroy, Carlos Eduardo Arango Gutierrez, Maurizio Drocco, Stephen Herbein, Dong H. Ahn, Zvonko Kaiser, and Yoonho Park. 2022. Towards Standard Kubernetes Scheduling Interfaces for Converged Computing. In *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*, Jeffrey Nichols, Arthur 'Barney' Maccabe, James Nutaro, Swaroop Pophale, Pravallika Devineni, Theresa Ahearn, and Becky Verastegui (Eds.). Springer Intl. Publishing, Cham, 310–326.

[34] Dmitry A. Nikitenko, Felix A. Wolf, Bernd Mohr, Torsten Hoefler, Konstantin S. Stefanov, Vadim V. Voevodin, Alexander S. Antonov, and Alexandru Calotoiu. 2021. Influence of Noisy Environments on Behavior of HPC Applications. *Lobachevskii J. of Mathematics* 42 (2021), 1560 – 1570.

[35] Ivy Peng, Roger Pearce, and Maya Gokhale. [n. d.]. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd Intl. Symp. on Computer Architecture and High Performance Computing*.

[36] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kalé. 2015. A Batch System with Efficient Scheduling for Malleable and Evolving Applications. In *29th IEEE Intl. Parallel and Distributed Processing Symp.*

[37] SchedMD. 2023. Generic Resource Scheduling. https://slurm.schedmd.com/gres.html.

[38] SchedMD. 2023. High Throughput Computing in SLURM. https://slurm.schedmd.com/high_throughput.html.

[39] SchedMD. 2023. SLURM Heterogeneous Jobs: Limitations. https://slurm.schedmd.com/heterogeneous_jobs.html#limitations.

[40] SchedMD. 2023. Slurm Power Management. https://slurm.schedmd.com/power_mgmt.html.

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conf. on Computer Systems (EuroSys)*.

[42] Jason Taylor. 2015. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *Optical Fiber Communications Conf.*

[43] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th Annual Symp. on Cloud Computing*.

[44] Deepak Vij and Shivram Shrivastava. [n. d.]. Poseidon-Firmament Scheduler: Flow Network Graph Based Scheduler. kubernetes.io/blog/2019/02/06/poseidon-firmament-scheduler-flow-network-graph-based-scheduler/.

[45] Andy Yoo, Morris Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science)*, Vol. 2862. 44–60.

[46] Chaojie Zhang and Andrew A. Chien. 2021. Scheduling Challenges for Variable Capacity Resources. In *Job Scheduling Strategies for Parallel Processing*.