SC23
Denver, CO | i am hpc.

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

Tapasya Patki, Dong Ahn*, Daniel Milroy, Jae-Seung Yeom, Jim Garlick, Stephen Herbein*, Tom Scogland

Lawrence Livermore National Laboratory
*NVIDIA Corporation**

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

WORKS Workshop, SC'23

Nov 13, 2023

Tapasya Patki, Dong Ahn*, Daniel Milroy, Jae-Seung Yeom,
Jim Garlick, Stephen Herbein*, Tom Scogland

*NVIDIA Corporation**

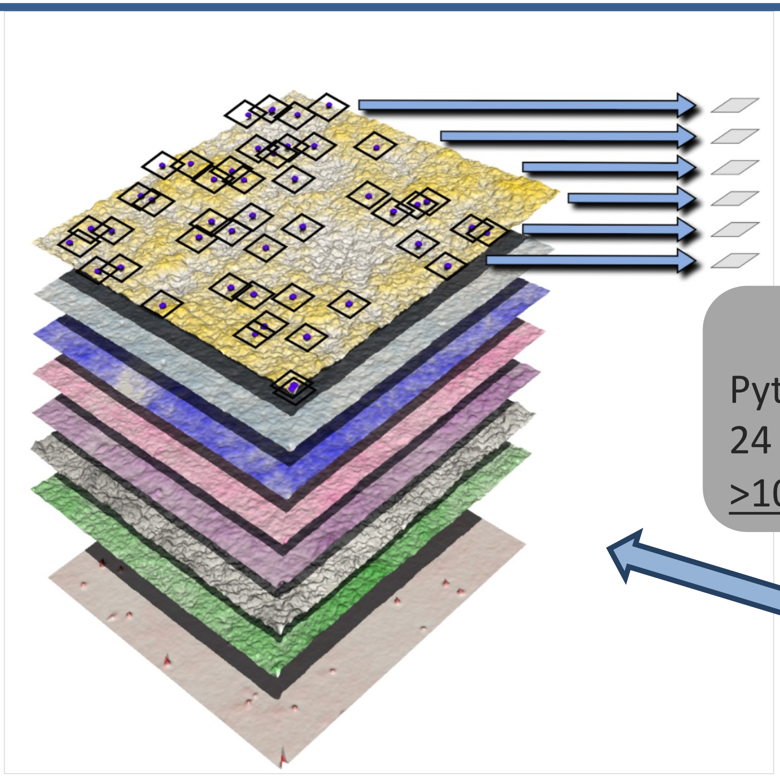
https://github.com/flux-framework/flux-sched

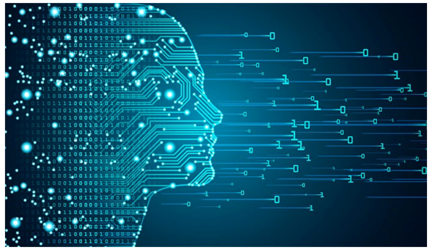
LLNL-PRES-844974

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

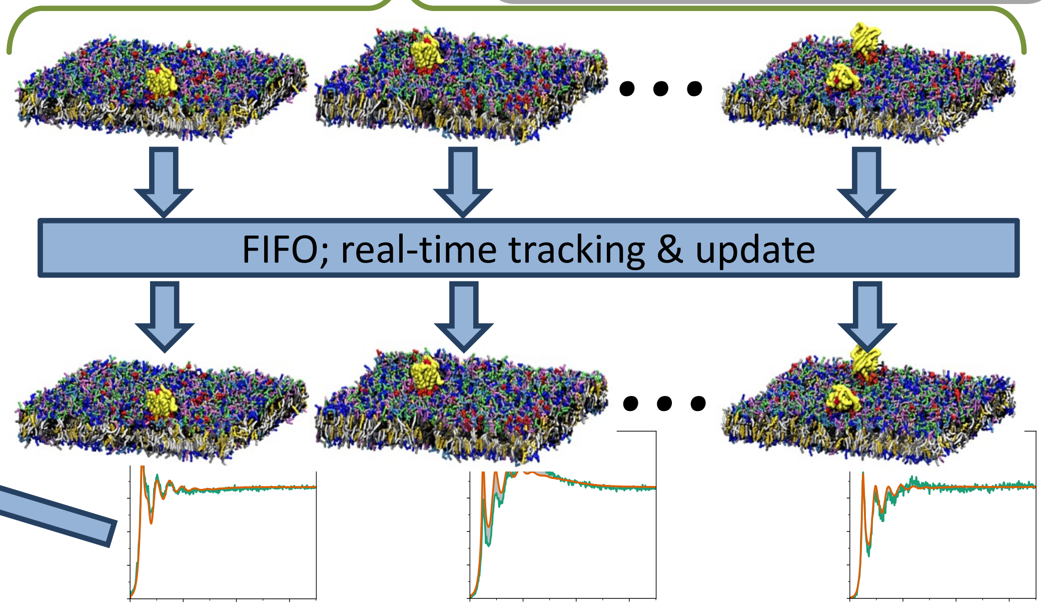
Sierra pre-exascale system is a wakeup call (MuMMI).



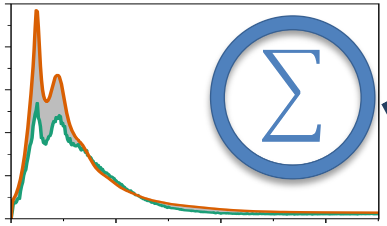


ML-based Selection
Python; ML frameworks; FAISS
24 CPU cores
>1000 decisions per minute

CG Setup
Python; Custom; GROMACS
24 CPU cores each 1.5 hr each



FIFO; real-time tracking & update



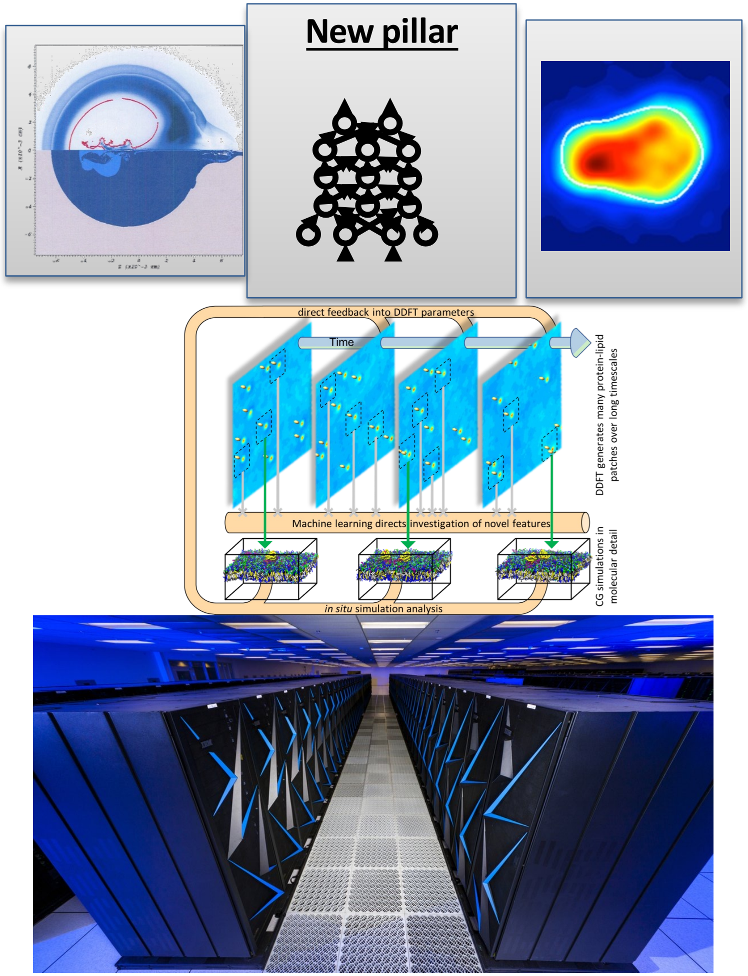
Single Macro Simulation
C++ (with MPI); MOOSE; ddcMD
24 CPU cores/node; 2400 MPI tasks
242 GB per day

Analysis Aggreg. & Feedback
Python; Custom
24 CPU cores
120,000 reads per cycle

CG Simulation
C++ (with CUDA); ddcMD
1 GPU + 1 CPU core each
1.04 μs and ~6.5 GB per day

In situ CG Analysis
Python; Custom
3 CPU cores each
>2K frames per day

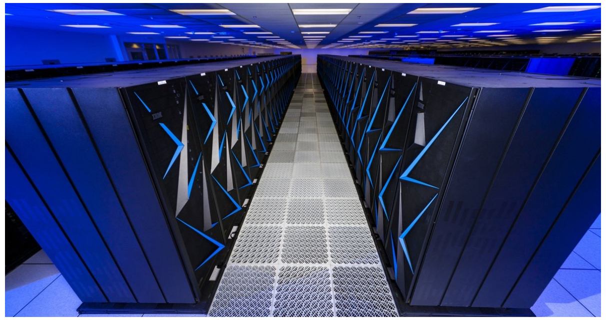
Trends towards complex workflows, extreme resource heterogeneity, and converged computing render traditional workload managers increasingly ineffective.

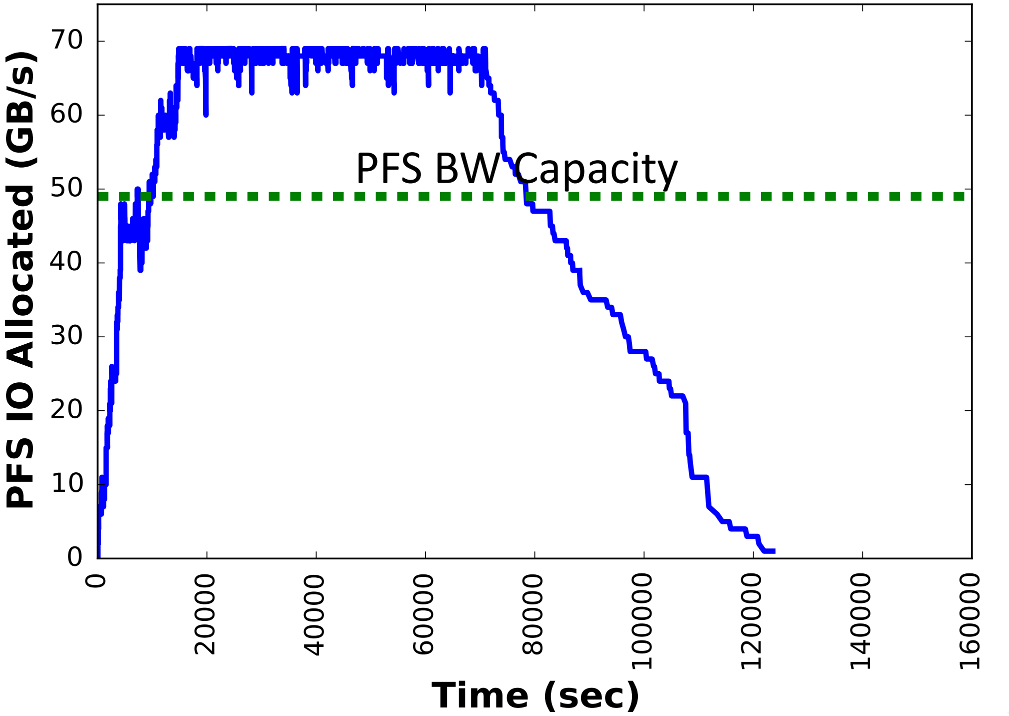


- Co-scheduling
- Job throughput
- Job communication/coordination
- Portability
- Extremely heterogenous resources

The changes in resource types are equally challenging.

- Problems are not just confined to the workload/workflow challenge.
- Resource types and their relationships are also becoming increasingly complex.
- Much beyond compute nodes and cores requiring partial occupancy and accounting…
  - GPGPUs, Burst buffers
  - I/O and network bandwidth, Power management
  - Variation
- Converged computing and disaggregated system designs require support for elasticity and dynamism





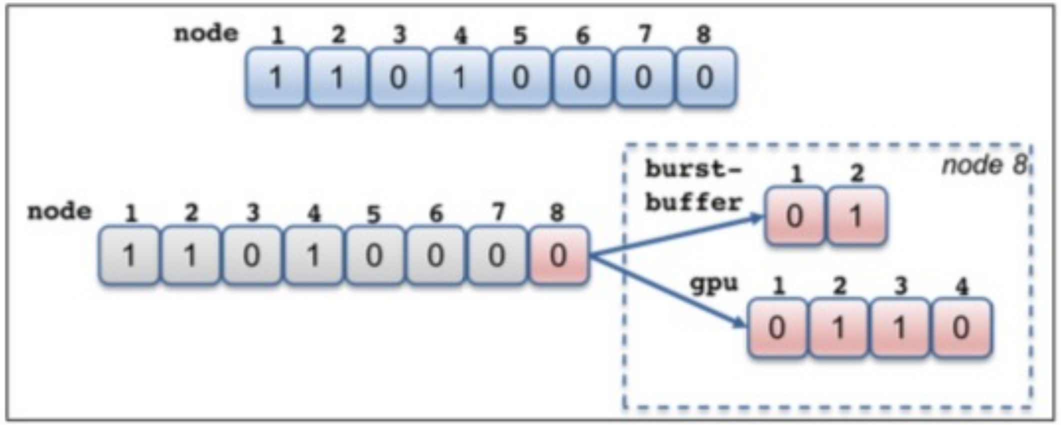
The traditional resource data models are largely ineffective to cope with these resource challenges.

- Resource Models: Internal representations and data structures used for managing resources (e.g. nodes, cores, memory, power)
- Node- or core-centric models are typical
  - Designed over 20 years ago when heterogeneity was uncommon, and memory was limited
- Pros: scheduling overhead and space complexity is low
- Cons:
  - Cannot represent resource relationships beyond physical hierarchy
  - Partial occupancy or level of detail for flow resources cannot be specified easily
  - Do not have a notion of containment or subsystems, e.g. allocating across a power or I/O subsystem hierarchy simultaneously
  - Do not support dynamic updates to resource pools

Incremental improvements are insufficient to address this gap for supporting advanced use cases.

- Approaches such as GRES plugins (SLURM) or custom resources (PBSPro) exist, but are still node-centric and cannot express complex resource relationships

- Scalability and management can become unwieldy
  - Every new resource type requires new a user-defined type
  - A new relationship requires a complex set of pointers cross-referencing different types.
  - Dynamic updating of resources is not supported
  - Cannot allocate through diverse hierarchies or resource pools simultaneously



Examples:

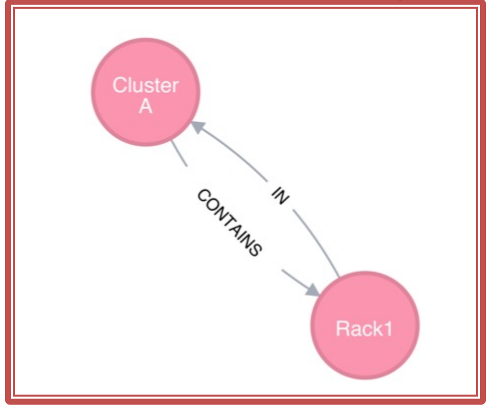
- *SLURM: bitmaps to represent a set of compute nodes, and GRES plugins for custom resources*
- *PBSPro: linked-list of nodes with custom resource definitions*

A graph-based resource model supports five key properties that address these challenges.

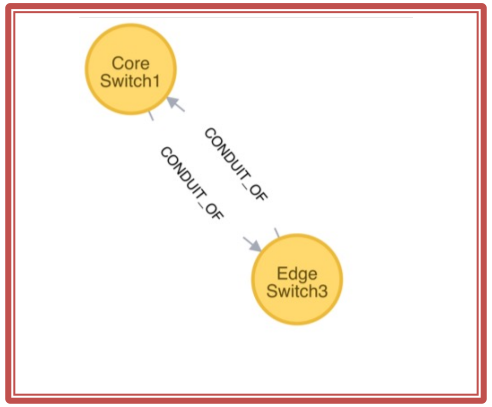
- Universality and Expressibility: Ability to model arbitrary and diverse resource types along with the various relationships between them
- Flexibility: Ability to support scheduling points at different levels of detail (eg. core, GPU, network bandwidth, power)
- Scalability: Ability to scale well and leverage parallelism across diverse setups, ranging from containers, to clouds, to supercomputers.
- Separations of Concerns: Ability to construct the resource model separately from the scheduling policy, allowing for support for scheduling policy customizations.
- Elasticity: Ability to update internal representations and data structures dynamically, to support moldability, malleability and variable capacity.

Fluxion pioneers and uses graph-based scheduling to manage complex combinations of extremely heterogenous resources.

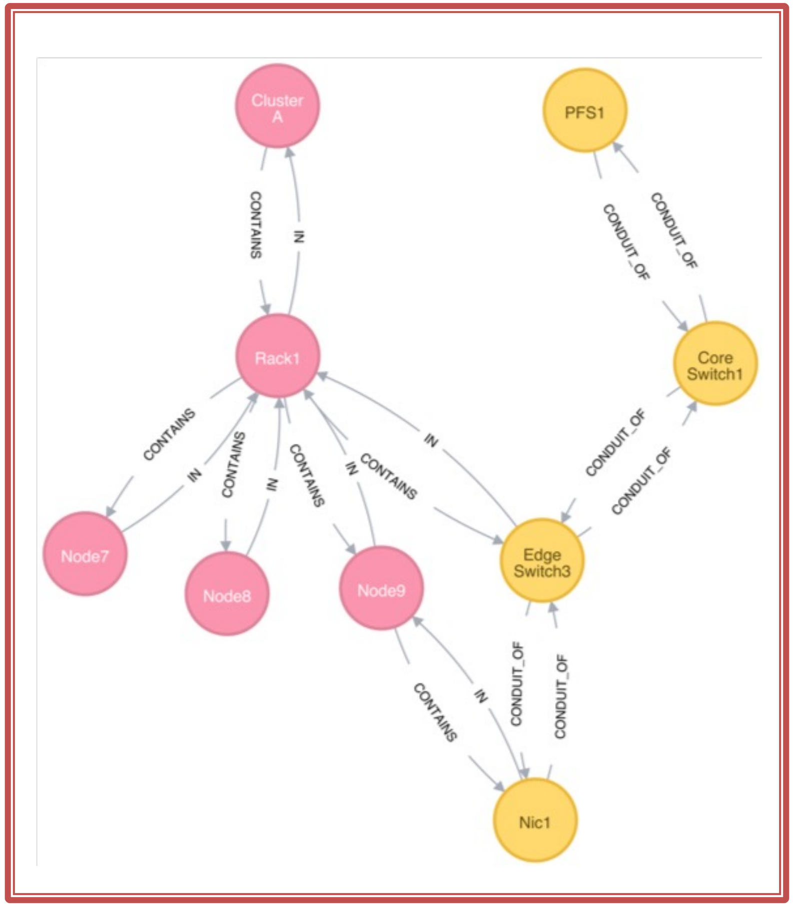
Containment subsystem



Network subsystem



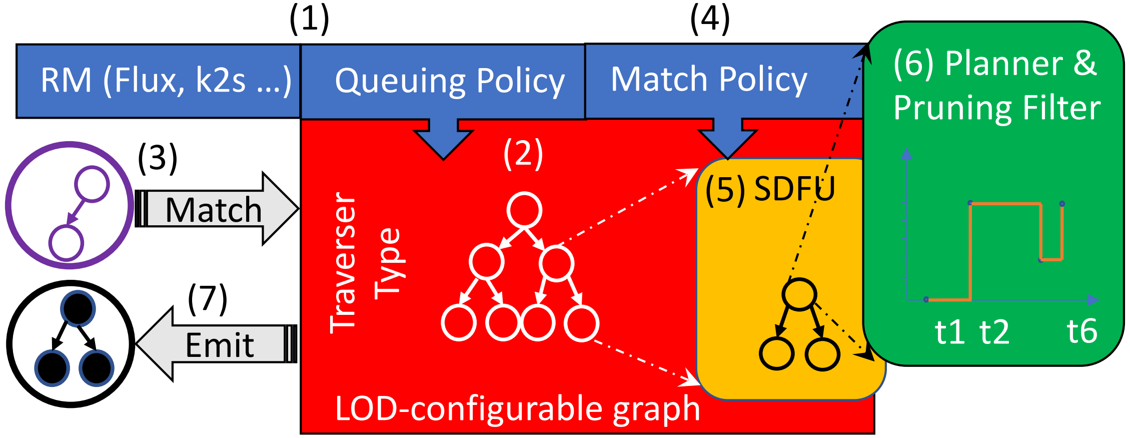
Containment and I/O subsystems



- Elevate resource relationships (edges) to an equal footing with resources (vertices)
- Resource Pool: group of indistinguishable resources (e.g. cores), can be viewed as coarse or fine grained
- Graph:
  - Vertex represents a resource pool
  - Edge has a type and subsystem attached

End-to-end scheduling flow with Fluxion

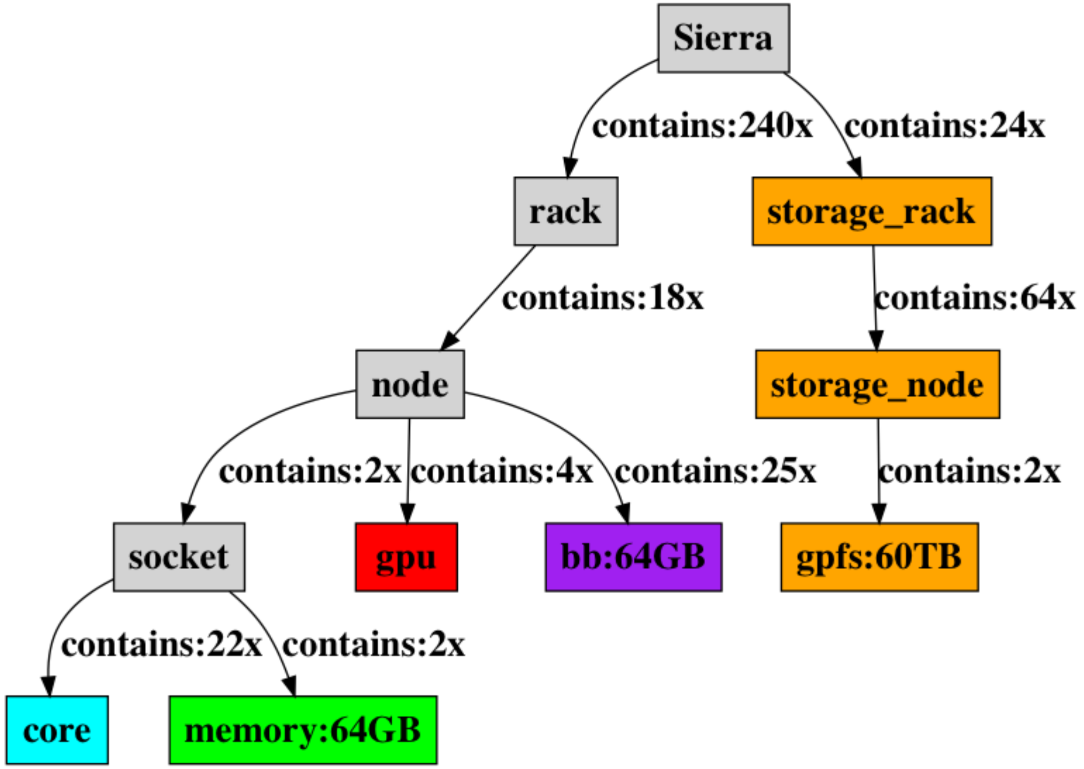
- In-memory *resource graph store* is populated with available resources (shown in Step 2), along with the level of detail and traversal type (e.g. depth-first)
- User's request is obtained as a *request graph* (Step 3)
- *Matching policy* (Step 4) callback is invoked on visit events (e.g. pre-order or post-order), and includes a scoring mechanism for ranking matches
- *Planner* allows for resource time tracking (like a calendar)
- Pruning filters and Scheduler Driven Filter Updates (SDFU) allow for better scalability



Fluxion's graph-based resource model can integrate with many resource managers, such as Flux and Kubernetes

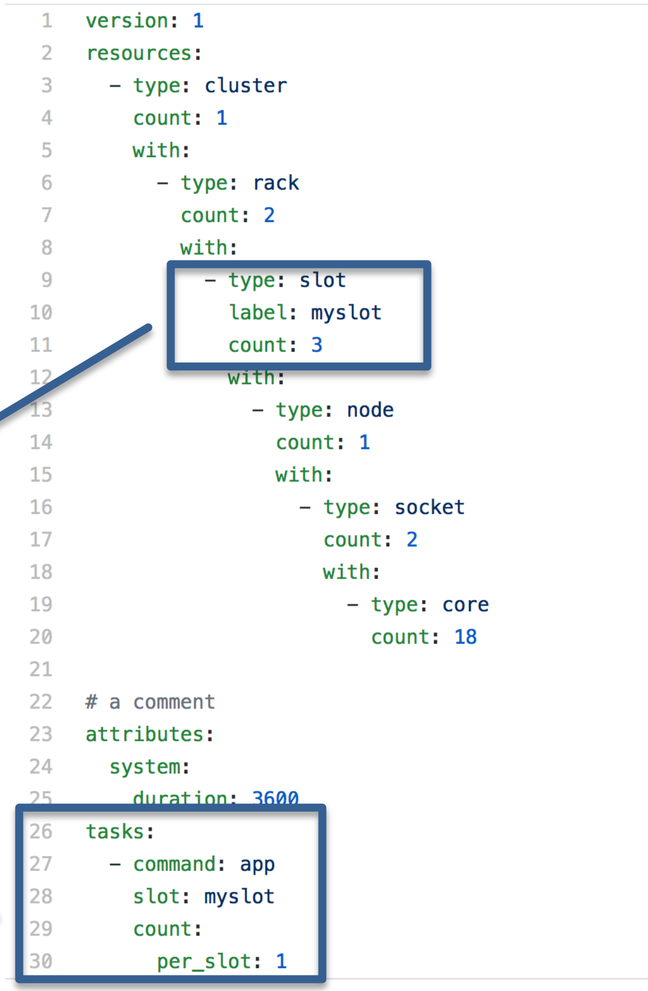
Fluxion uses Level of Detail (LOD) control to improve expressibility and scalability of graph models.

- Resource pools combined with subsystems enable different granularities of scheduling easily
  - E.g., select whether scheduling occurs at the node-level, rack-level, gpu-level or storage-node-level
- Coarse granularity
  - Higher performance
  - Pool together resources of the same type as a single vertex
- Finer granularity
  - Promote subdivisions of resources to their own vertex
- Graph filtering allows for selecting relevant subsystems in complex schedulers with multiple subsystems (e.g. containment and power)



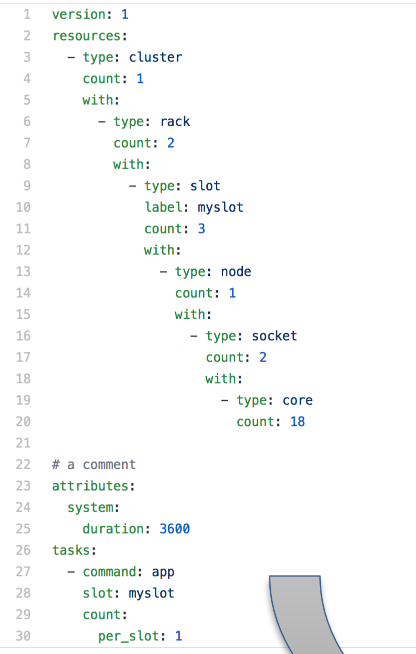
Fluxion’s graph-oriented canonical job-spec allows for a highly expressive user resource requests specification.

- Graph-oriented resource requests
  - Express the resource requirements of a program to the scheduler
  - Express program attributes such as arguments, run time, and task layout, to be considered by the execution service
- cluster->racks[2]->slot[3]->node[1]->sockets[2]->core[18]
- *slot* is the only non-physical resource type
  - Represent a schedulable place where program process or processes will be spawned and contained
- Referenced from the tasks section

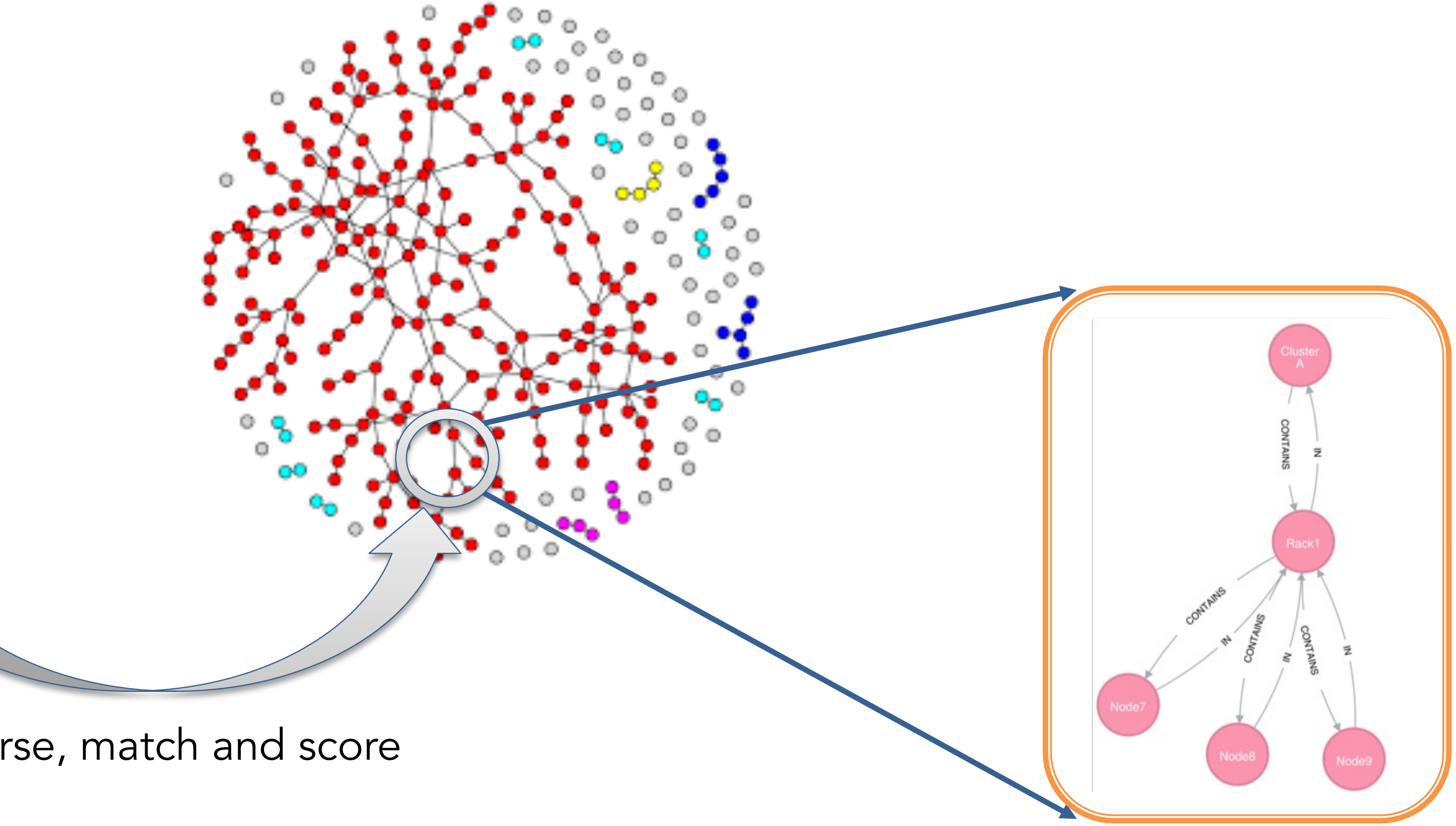


```yaml
version: 1
resources:
  - type: cluster
    count: 1
    with:
      - type: rack
        count: 2
        with:
          - type: slot
            label: myslot
            count: 3
            with:
              - type: node
                count: 1
                with:
                  - type: socket
                    count: 2
                    with:
                      - type: core
                        count: 18

# a comment
attributes:
  system:
    duration: 3600
tasks:
  - command: app
    slot: myslot
    count:
      per_slot: 1
```

Fluxion maps complex scheduling problems into graph matching problems and allows for ranking between options.



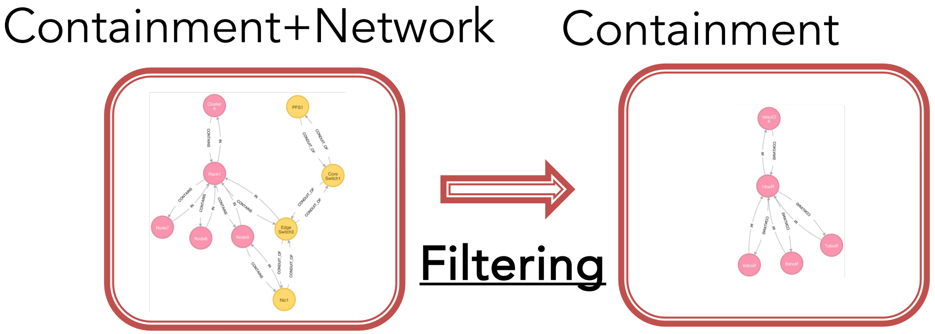
```yaml
version: 1
resources:
  - type: cluster
    count: 1
    with:
      - type: rack
        count: 2
        with:
          - type: slot
            label: myslot
            count: 3
            with:
              - type: node
                count: 1
                with:
                  - type: socket
                    count: 2
                    with:
                      - type: core
                        count: 18

# a comment
attributes:
  system:
    duration: 3600
tasks:
  - command: app
    slot: myslot
    count:
      per_slot: 1
```



Traverse, match and score

Fluxion uses graph filtering and pruning to manage the graph complexity and optimize graph search.

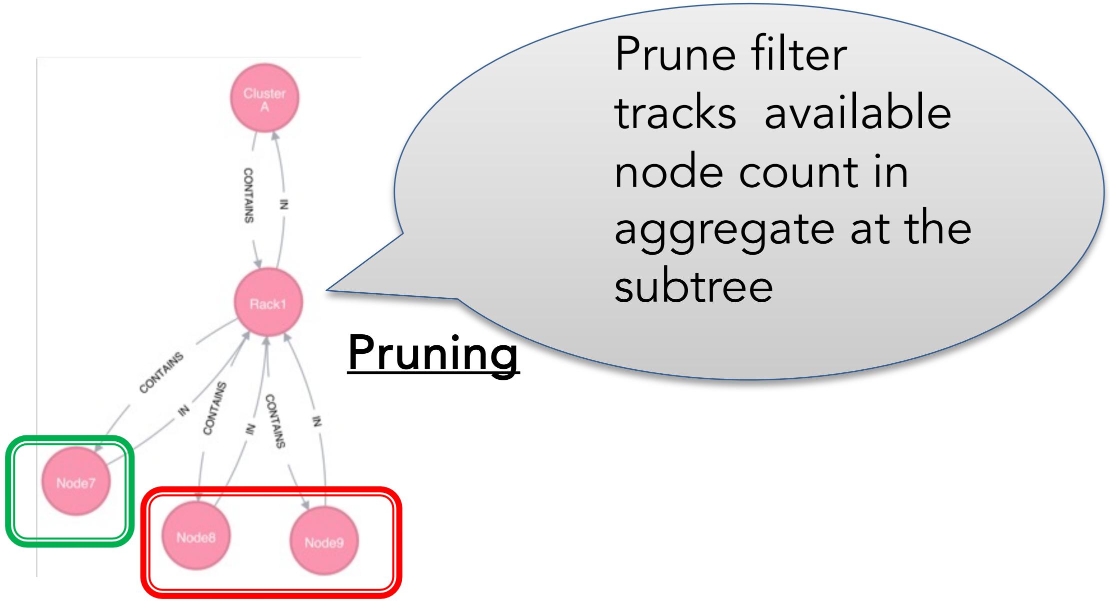
- The total graph can be quite complex
  - Two techniques to manage the graph complexity and scalability
- Filtering reduces graph complexity
  - The graph model needs to support schedulers with different complexity
  - Provide a mechanism by which to filter the graph based on what subsystems to use
- Pruned search increases scalability
  - Fast RB tree-based planner is used to implement a pruning filter per each vertex.
  - Pruning filter keeps track of summary information (e.g., aggregates) about subtree resources.
  - Scheduler-driven pruning filter update



Containment+Network

Containment

Filtering



Prune filter tracks available node count in aggregate at the subtree

Pruning

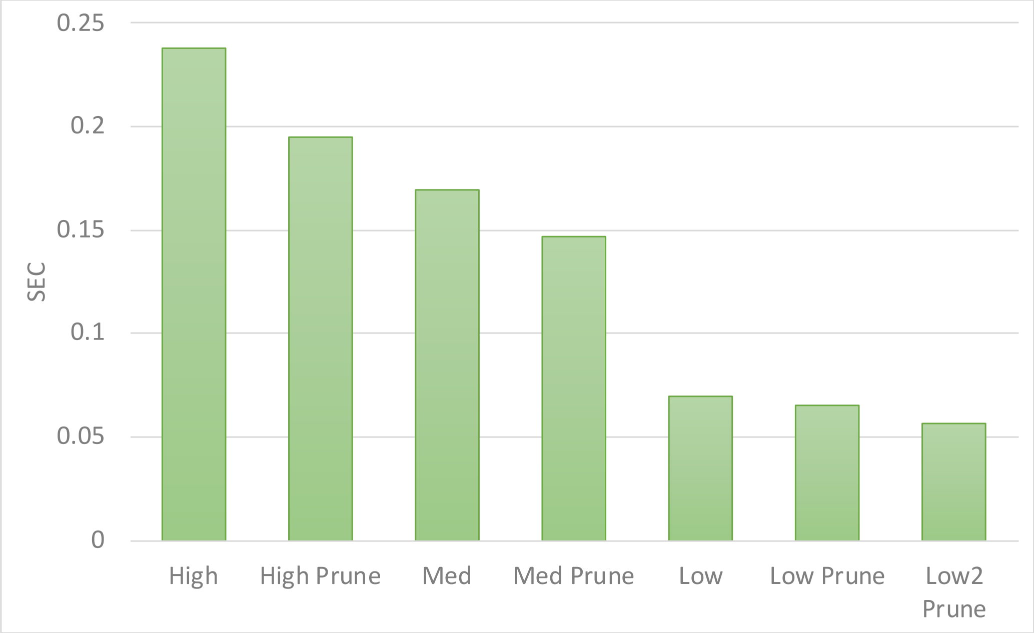
Scalability Results: Level of Detail along with Pruning

Evaluate a 1008 compute node system with four levels of detail:

- High LOD:
  - 56 compute racks, 18 nodes, with 2 sockets.
  - 20 cores, 2 GPUs, 8 memory (16GB each), 8 burst-buffers (BB) (100 GB) per socket
- Med LOD:
  - Same system, but remove socket-level detail
  - 40 cores, 4 GPUs, 8 memory (32 GB) and 8 BB (200 GB) per node
- Low LOD:
  - Remove rack-level vertices
  - Create a new core-pool of 5 cores each, 4 memory (64 GB) and 4 BB (400 GB) per node
- Low2 LOD:
  - Similar to Low, but doesn't remove rack vertices

- Job request:
  - 10 cores, 8 GB memory, 1 BB
  - Repeat until system is fully allocated

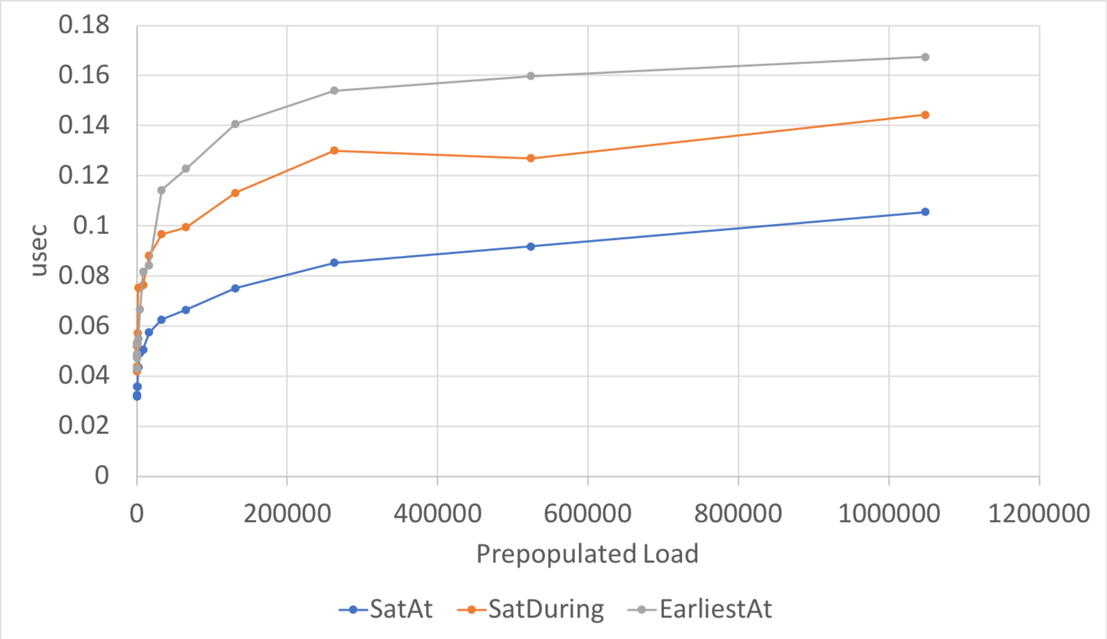
Time taken for matching all job requests with varying LOD, and with and without pruning



Scalability Results: Planner scalability

- Evaluate with 128 units of an unnamed resource with maximum time of 12 hours.
- Up to 1 million prepopulated spans with <r,d> (resource amount, duration) drawn from a uniform distribution of (1,128) and (1s, 43200s)
- SatAt:
  - How quickly can a new request R with increasing amounts of *r* and unit duration be satisfied at a random time t?
- SatDuring:
  - How quickly can a new request R with increasing amounts of both *r* and *d* be satisfied at a random time t?
- EarliestAt:
  - How quickly can we find the earliest fit for a new request R with increasing amounts of *r* ?

Planner performance with different span counts and query types



Use Case 1: The Fluence (FKA KubeFlux) plugin brings HPC-grade scheduling and improved performance to Kubernetes.

K8s Scheduling Framework plugin based on Fluxion scheduler.

Architectural change from monolithic to gRPC-based

- Improves maintainability, separation of concerns

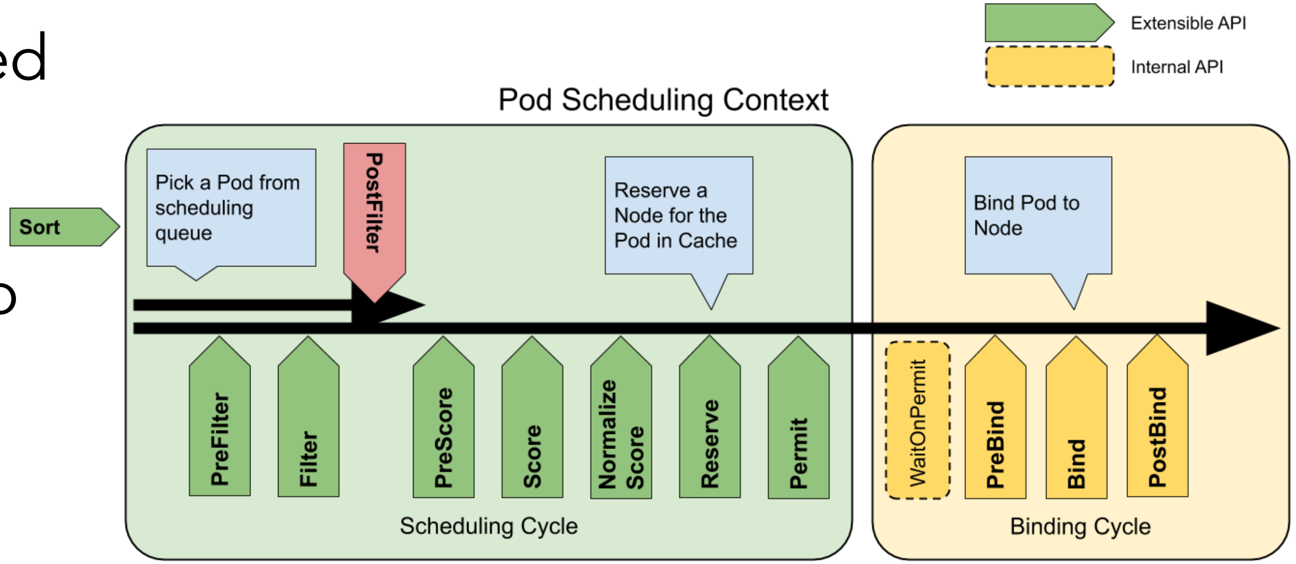


image: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

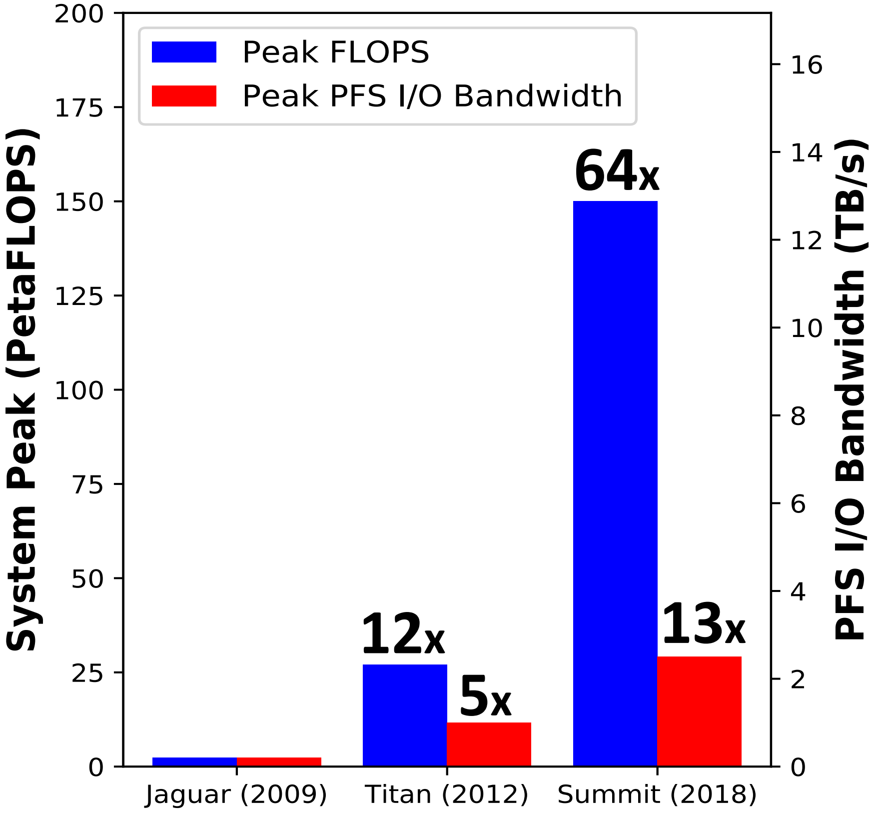
More placement control and functionality

- Gang scheduling
- GPU support
- Topology awareness of Availability Zones (AZs)

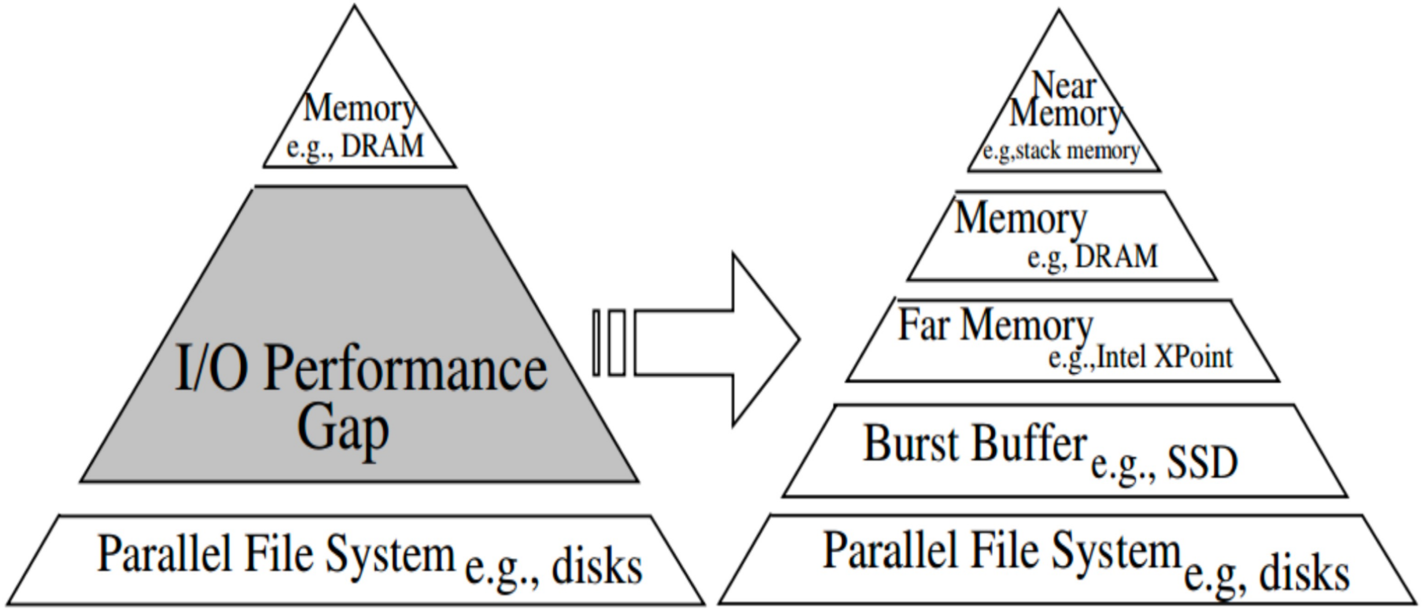
Easier deployment

- Automation through Helm
- Export of Golang modules for easier distribution

Use Case 2: Tiered Storage in HPC with Rabbits

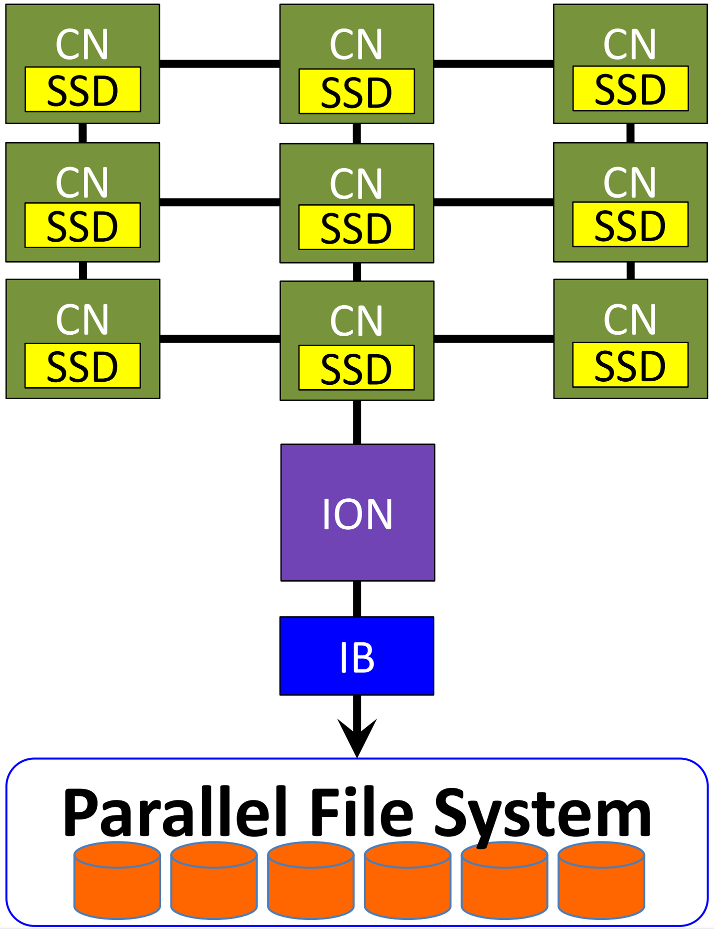


Source: Lucy Nowell (DOE)

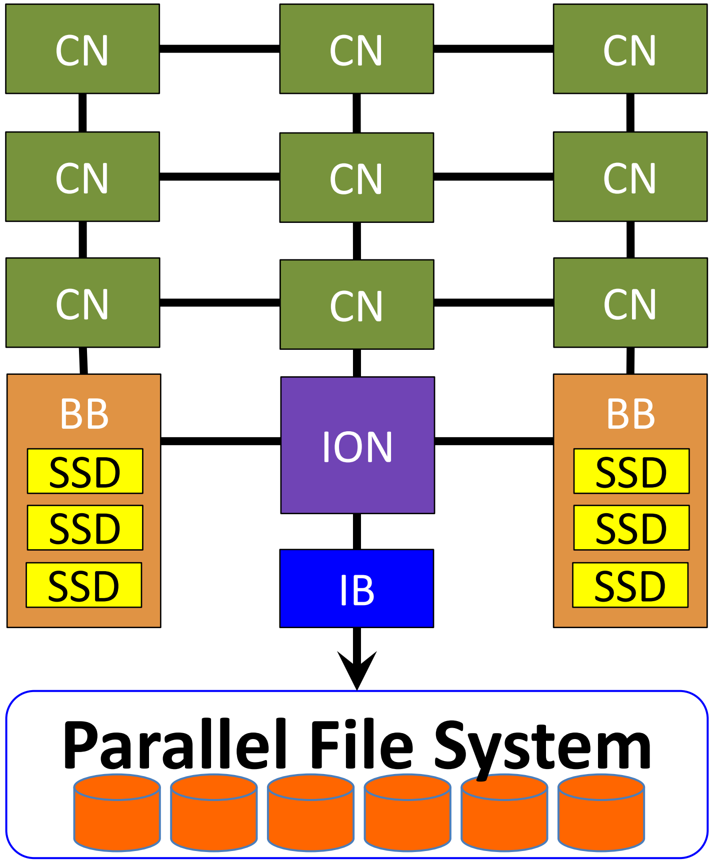


Burst Buffer Architectures

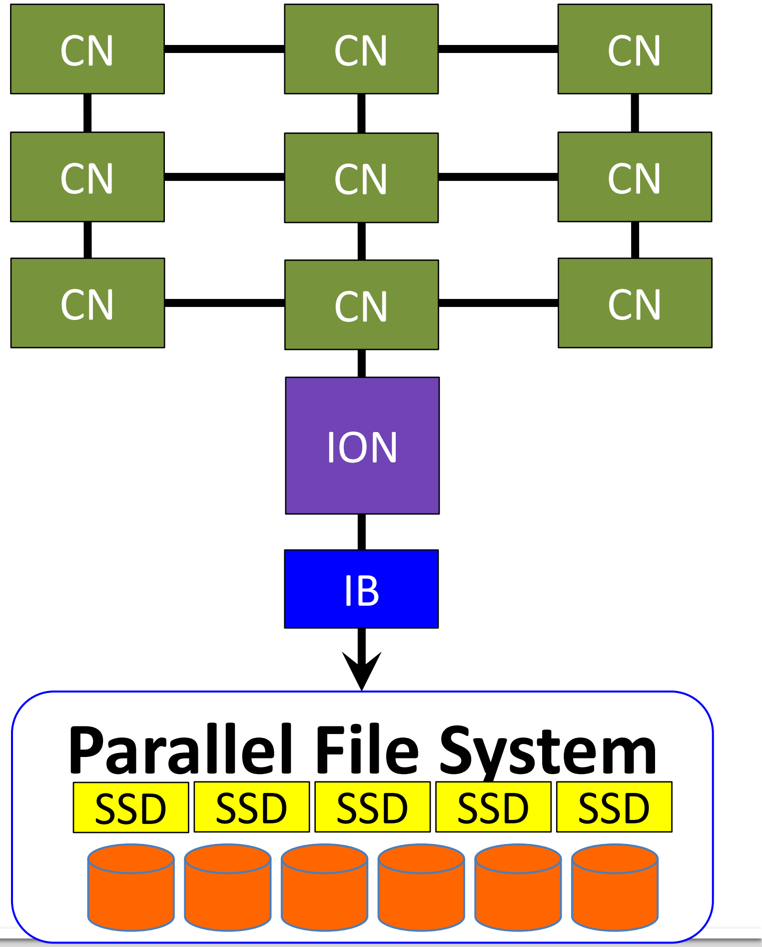
Node-local BB



Remote, shared BB



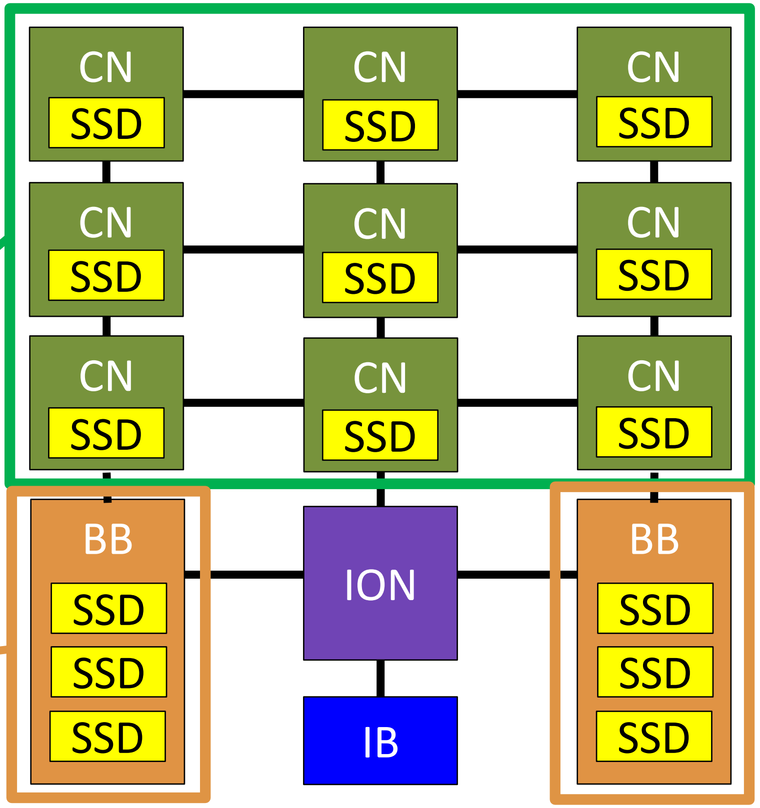
Filesystem BB



Example of Tiered Storage Request

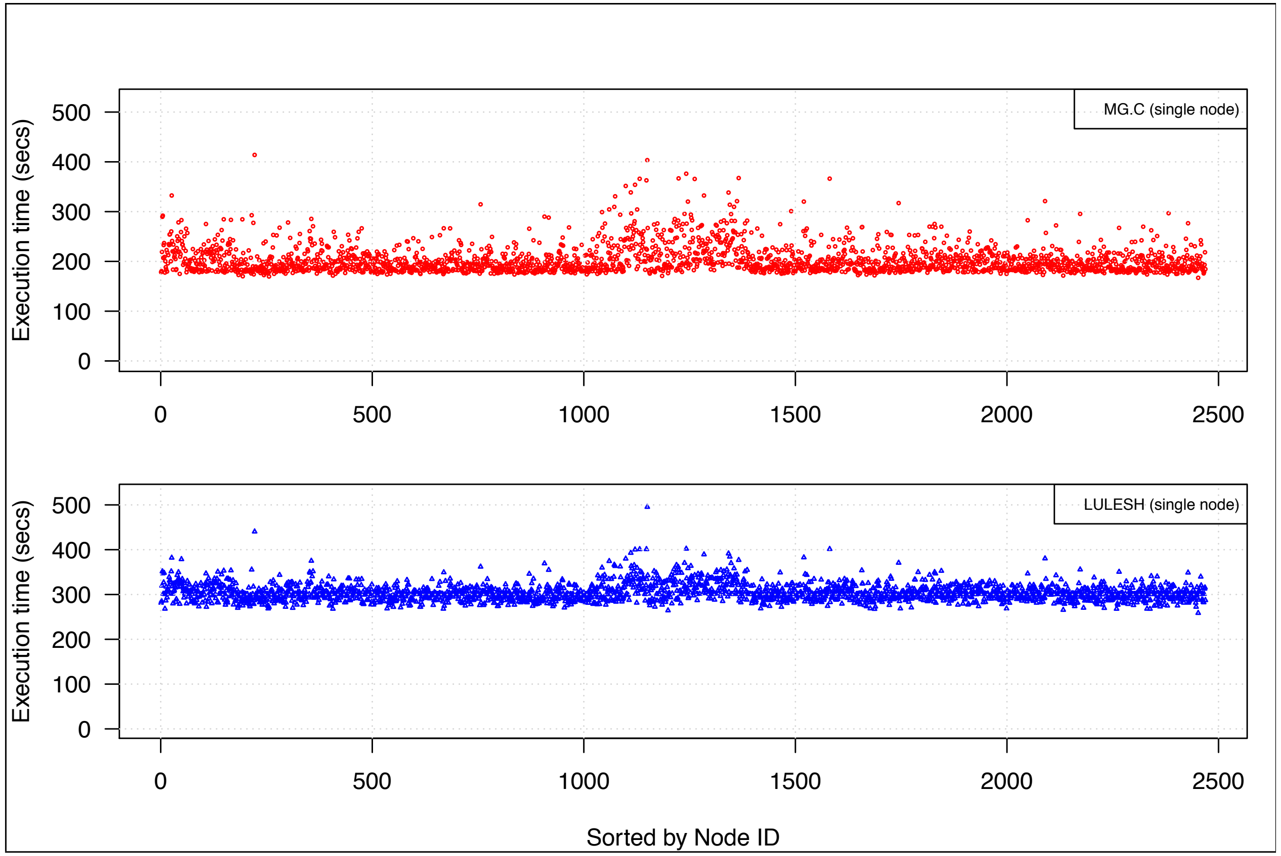
```
resources:
  - type: node
    count: 9
    with:
      - type: slot
        count: 1
        label: default
        with:
          - type: core
            count: 2
      - type: storage
        count: 1
        unit: terabytes
        label: node-local-scratch
  - type: storage
    count: 4
    unit: terabytes
    label: PFS-cache
```

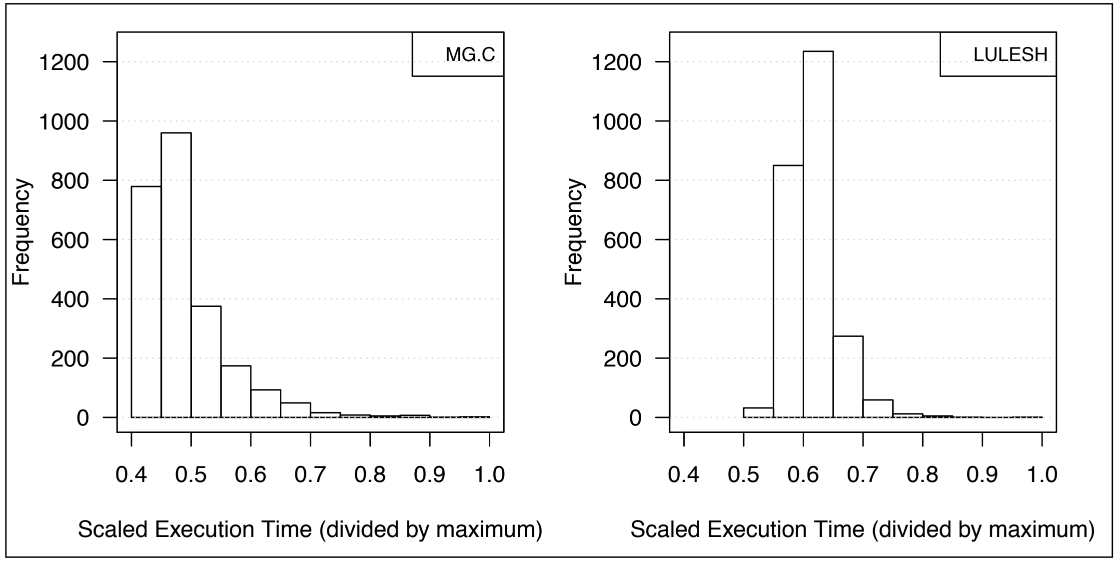
```
attributes:
  storage:
  - label: node-local-scratch
    mode: scratch
    granularity: per-node
    stage-in:
      list: /path/to/stage-in-listing
  - label: PFS-cache
    data-layout: striped
    mode: cache
    stage-in:
      directory: /path/to/PFS
```



We can use the Fluxion to allocate these new storage tiers with 0 code changes

Use Case 3: Variation-aware scheduling with Fluxion: Addressing Manufacturing Variability, Processor Aging, and inherent heterogeneity



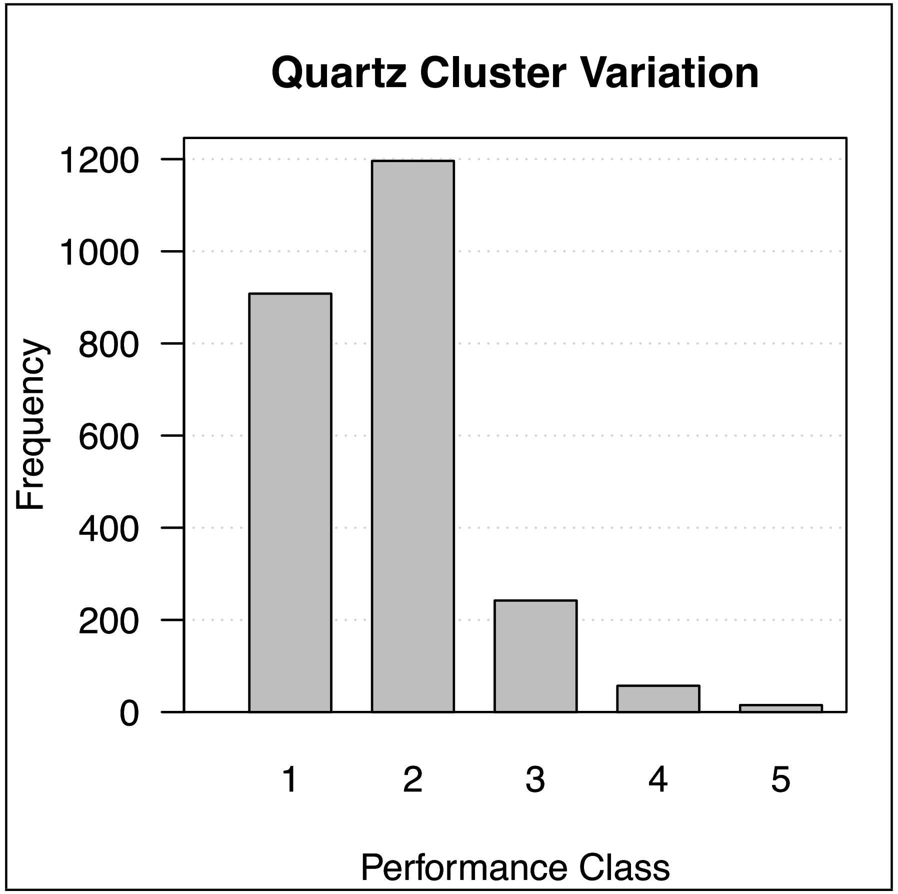


- Real world example under power constraints: Quartz cluster, 2469 nodes, 50 W CPU cap
- 2.47x difference between the slowest and the fastest node for MG
- 1.91x difference for LULESH.

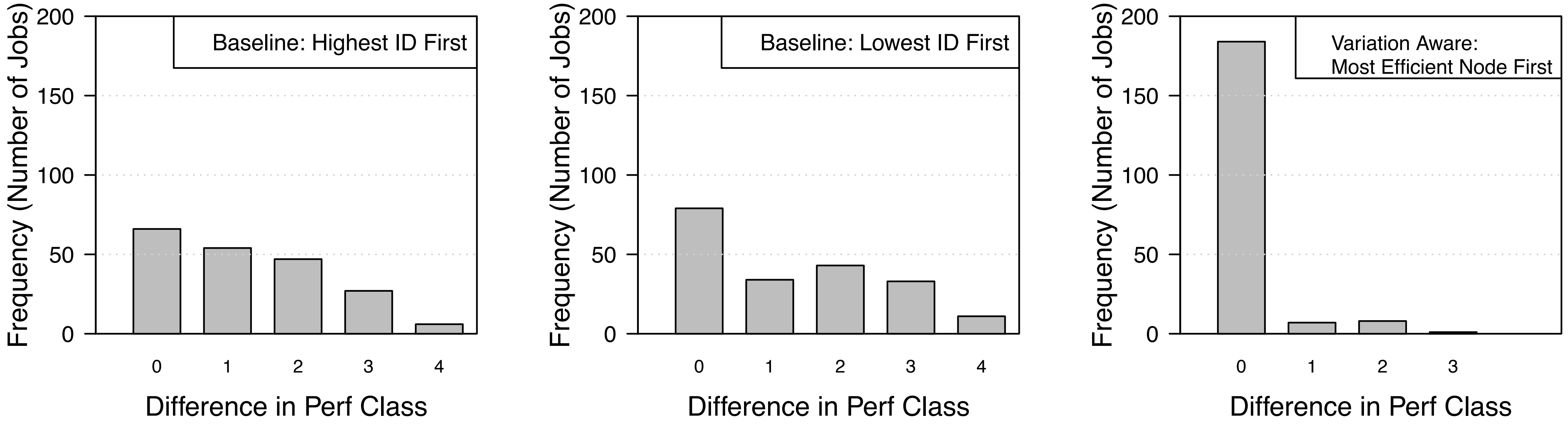
https://github.com/flux-framework/flux-sched/tree/master/resource/policies

Example: Statically determining node performance classes

- Ranking every processor is not feasible
- Statically create bins of processors with similar performance instead
  - Techniques for this can be simple or complex
  - How many classes to create, which benchmarks to use, which parameters to tweak
  - Our choice: 5 classes, LULESH and MG, 50 W cap
- Mitigation
  - Rank-to-rank: minimize spreading application across multiple performance classes
  - Run-to-run: allocate nodes from same set performance classes to similar applications



Variation-aware scheduling results in 2.4x reduction in rank-to-rank variation in applications with Flux

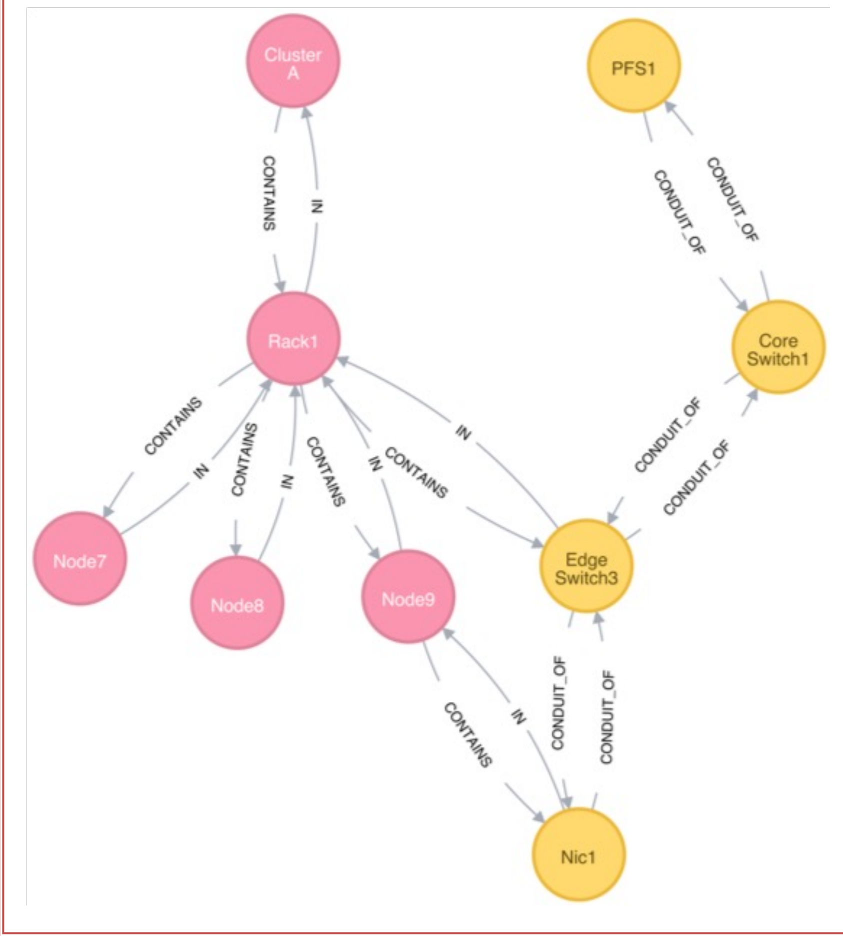


Flux's graph-based resource model easily and effectively enables this variation-aware scheduler optimization

Conclusions

- Fluxion is a graph-based resource model that addresses scheduling challenges in the exascale era and beyond
- Elevates resource relationships to an equal footing with resources to allow for representation of diverse resource sets and subsystems
- Supports expressibility, flexibility, separation of concerns and elasticity in a scalable manner
- Implementations within Flux and Kubernetes allow for support of converged computing in addition to traditional HPC

https://github.com/flux-framework/flux-sched



Thank you!
Questions?

Lawrence Livermore
National Laboratory