Flux: Next-Generation Resource Management and Scheduling Infrastructure for Exascale Workflow and Resource Challenges

ADAC, March 25, 2019

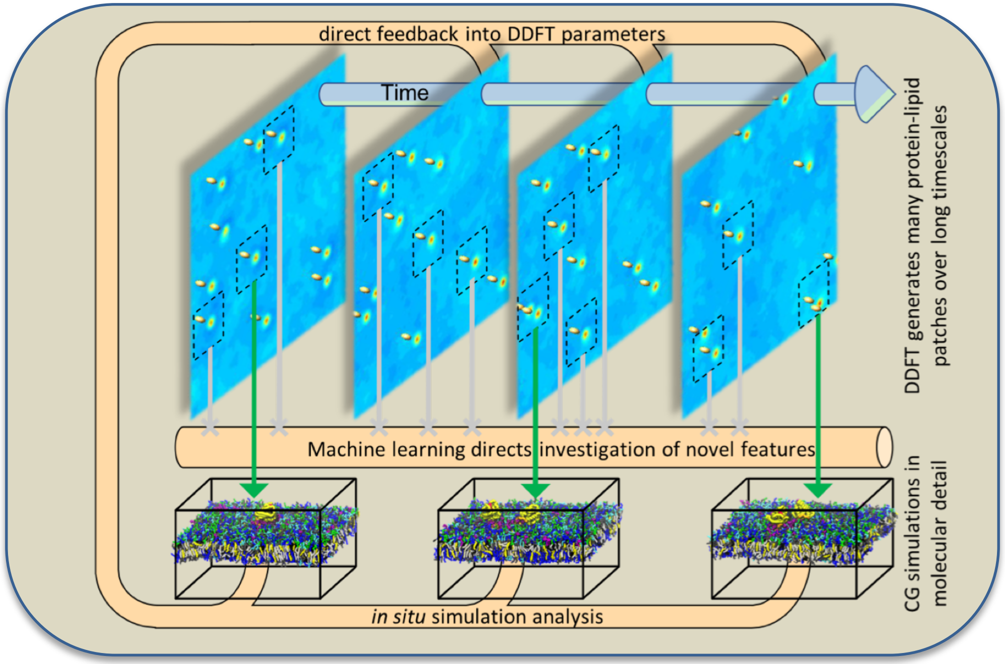
Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, Tapasya Patki, Thomas R. W. Scogland, Becky Springmeyer, and Michela Taufer

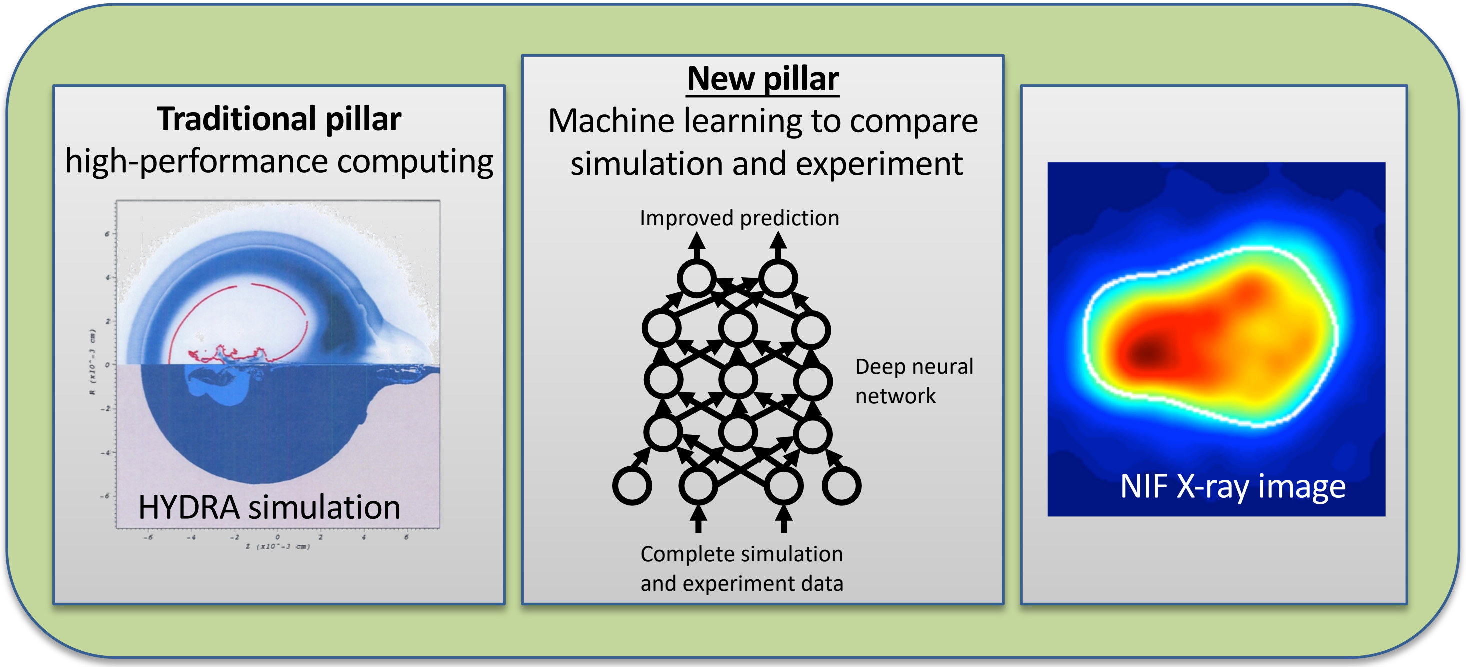
LLNL-PRES- 765450

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

Workflows on high-end HPC systems are undergoing significant changes.

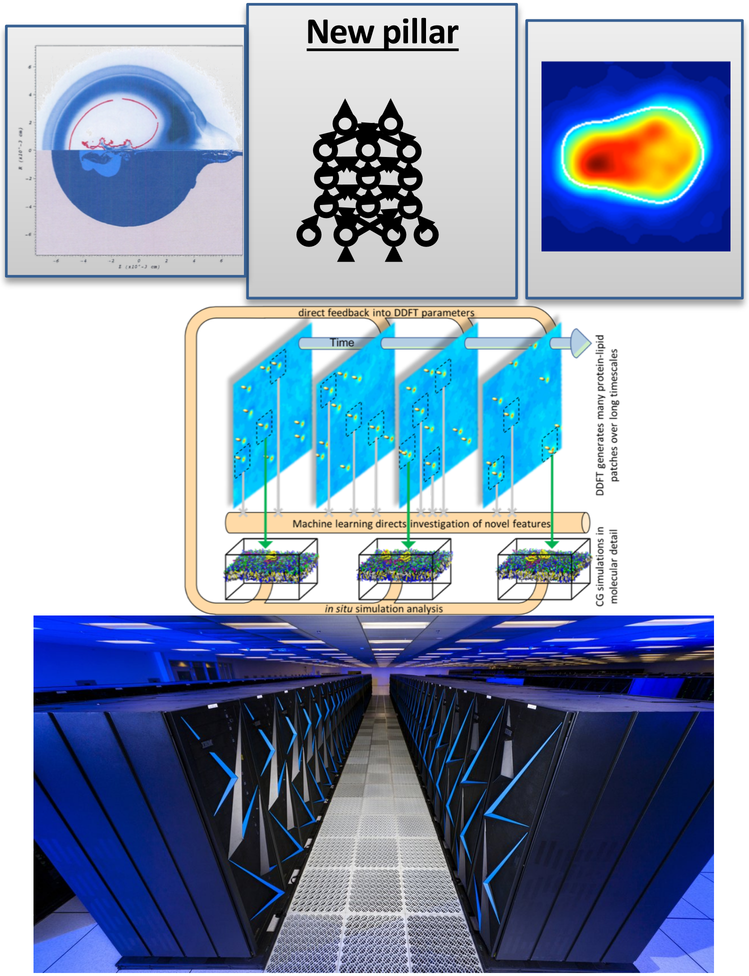




- Cancer Moonshot Pilot2 – **co-schedule many elements and ML continuously schedules, de-schedules and executes MD jobs.**
- In-situ analytics modules
- ~7,500 jobs simultaneously running

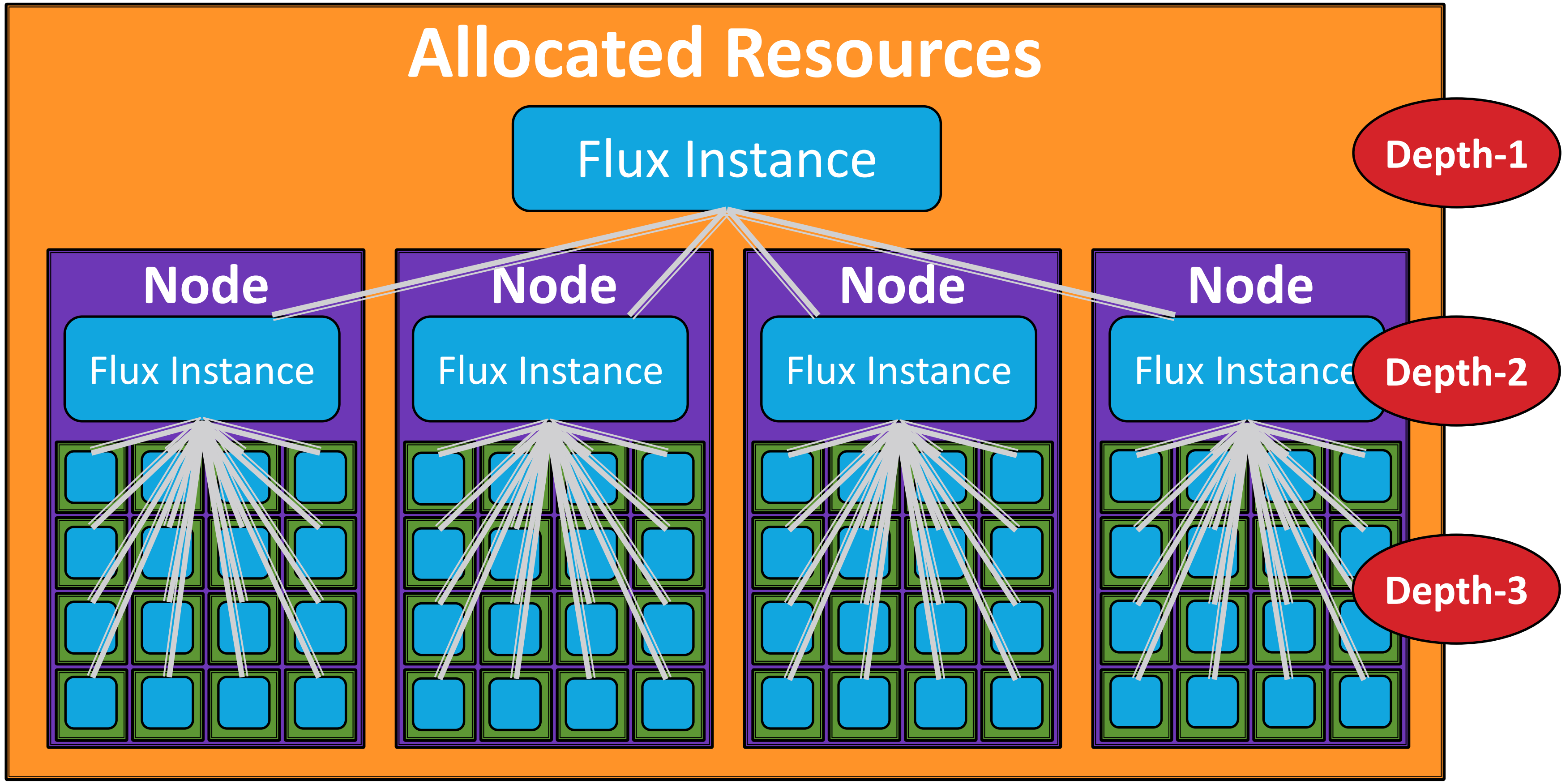
- Machine Learning Strategic Initiative (MLSI) – **1 billion short-running jobs!**
- Similar needs for co-scheduling heterogenous components

Key challenges in emerging workflow scheduling include…



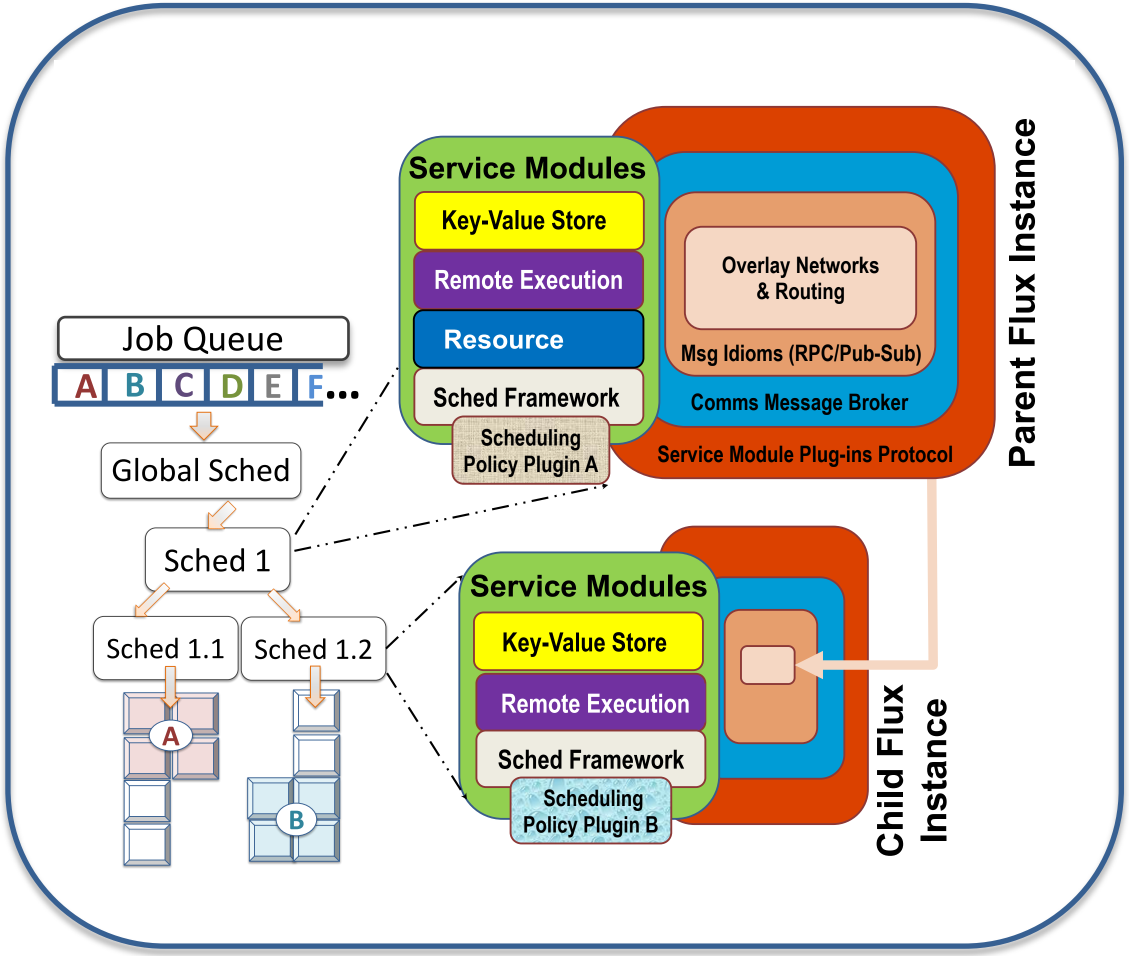
- Co-scheduling challenge
- Job throughput challenge
- Job communication/coordination challenge
- Portability challenge

Flux provides a new scheduling model to meet these challenges.



Our "Fully Hierarchical Scheduling" is designed to cope with many emerging workload challenges.

Flux is specifically designed to embody our fully hierarchical scheduling model.



Flux framework

Techniques	Challenges
Scheduler Specialization	Co-scheduling challenge
Scheduler Parallelism	Job throughput challenge
Rich API set	Job communication/coordination challenge
Consistent API set	Portability challenge

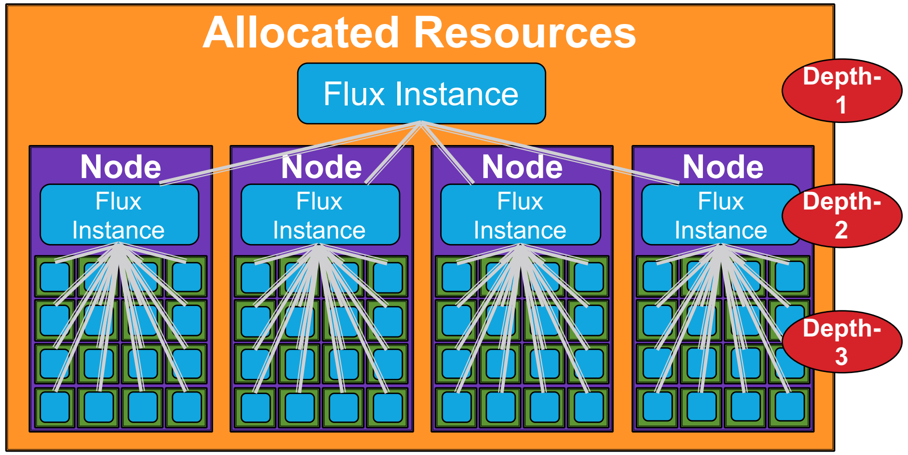
Scheduler specialization solves the co-scheduling challenge.

- Traditional approach
  - A single site-wide policy being enforced for all jobs
  - No support for user-level scheduling with distinct policies

- Flux enables both system- and user-level scheduling under the same common infrastructure.

- Give users the freedom to adapt their scheduler instance to their needs.
  - Instance owners can choose predefined policies different from system-level policies.
    - FCFS/backfilling
    - Scheduling parameters (queue depth, reservation depth etc)
  - Create their own policy plug-in

Scheduler parallelism solves the throughput challenge.



- The centralized model is fundamentally limited.
- Hierarchical design facilitates scheduler parallelism.
- Deepening the scheduler hierarchy allows for higher levels of scheduler parallelism
- Implementation used in our scalability evaluation:
  - Submit each job in the ensemble individually to the root
  - The jobs are distributed automatically across the hierarchy.

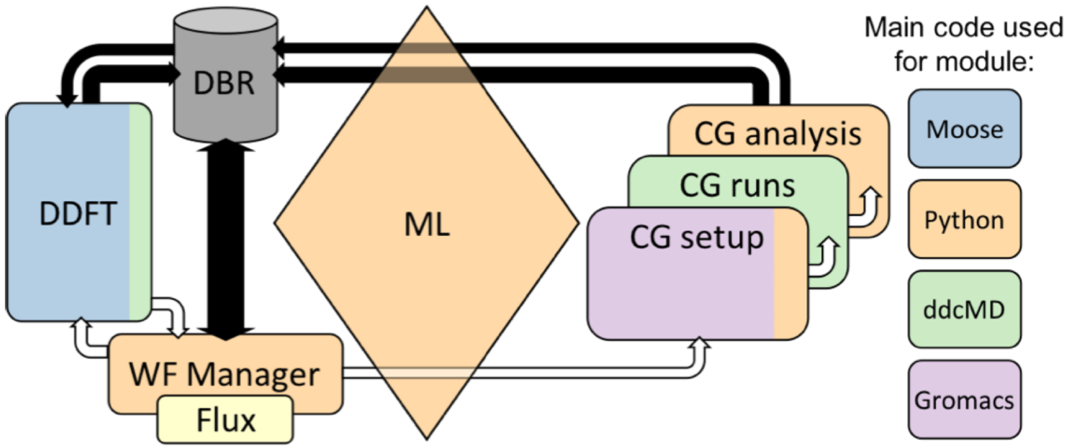
A rich API set enables easy job coordination and communication.

- Jobs in ensemble-based simulations often require close coordination and communication with the scheduler as well as among them.
  - Traditional CLI-based approach is too slow and cumbersome.
  - Ad hoc approaches (e.g., many empty files) can lead to many side-effects.
- Flux provides well-known communication primitives.
  - Pub/sub, request-reply, and send-recv patterns
- High-level services
  - Key-value store (KVS) API
  - Job status/control (JSC) API
  - KZ stdout/stderr stream API

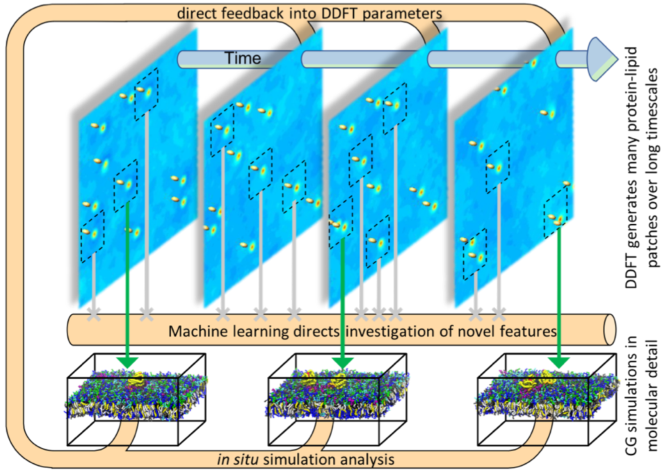
A consistent API set facilitates high portability.

- Flux's APIs are consistent across different platforms
- Effort for porting and optimizing Flux itself for a new environment is small
  - Linux
  - Require the lower-level system to provide the Process Management Interface (PMI)

Scheduler specialization addresses co-scheduling challenges in Cancer Moonshot Pilot2 on Sierra



Pilot2 workflow architecture

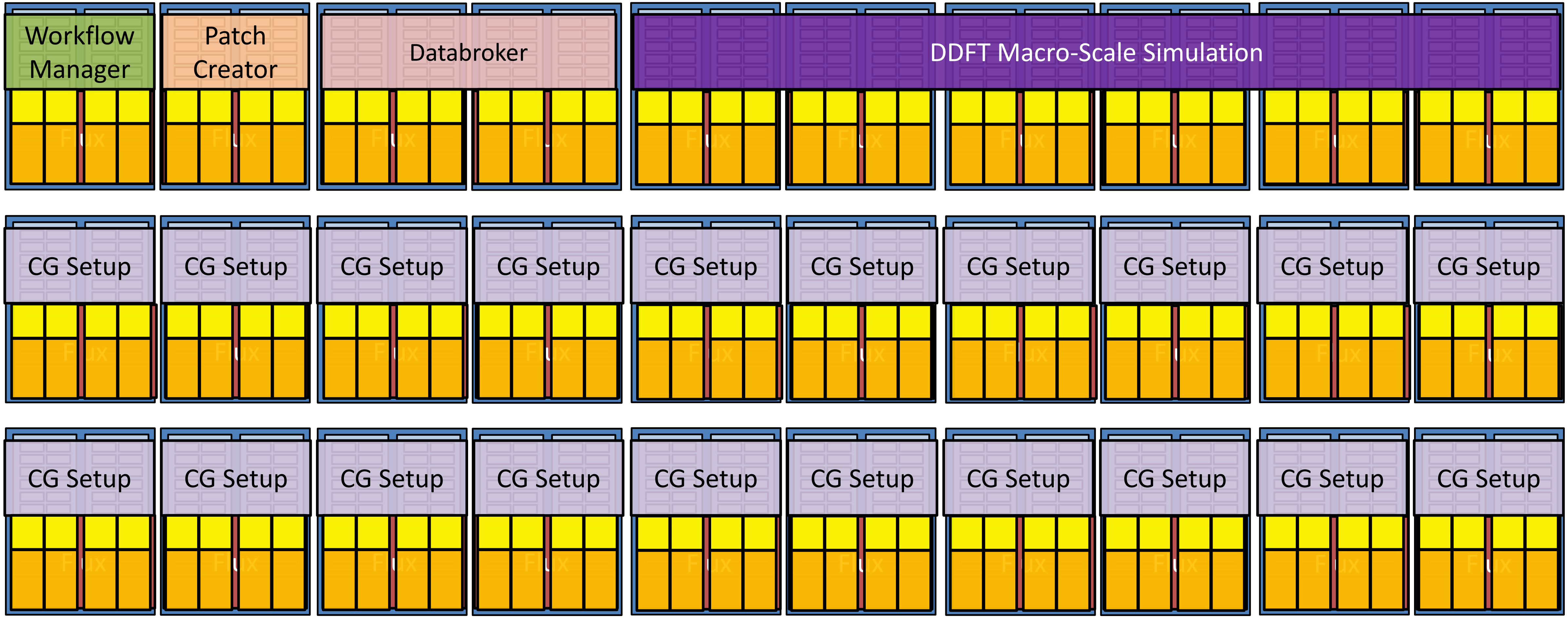


- The machine-learning module evaluates the top *n* candidate patches for MD simulations.
- Integrate Flux into Maestro workflow manager to start and stop jobs accordingly
- Maestro adapter to Flux
  - Specialize the policy to be non-node-exclusive scheduling for complex co-scheduling
  - At least 5 different logically separate jobs, each with CPU and/or GPU requirements on every node
  - Handle the volume of jobs using simple hierarchical scheduling

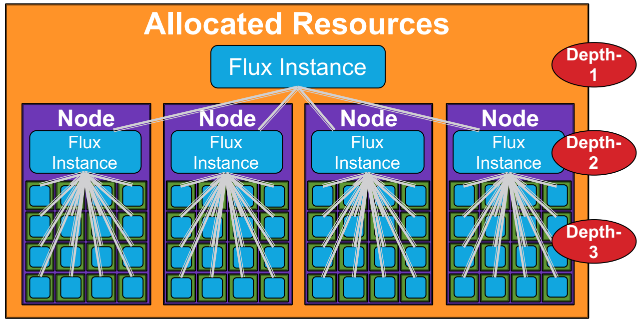
Scheduler specialization solves the co-scheduling challenge

Hardware
- Node
- Socket
- CPU
- GPU

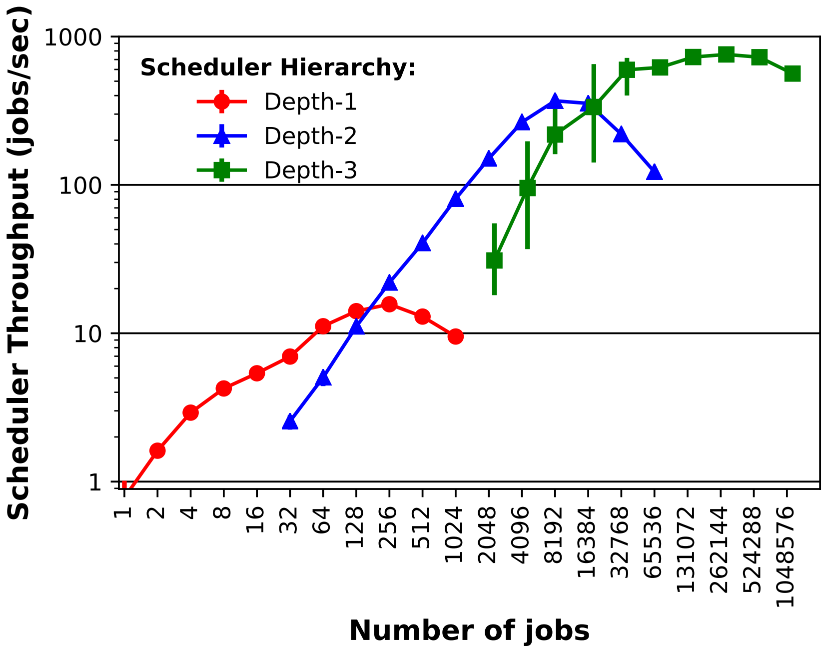
Jobs
- CG Analysis
- CG Run
- CG Setup
- DDFT
- Workflow Manager
- Patch Creator
- Databroker
- Flux



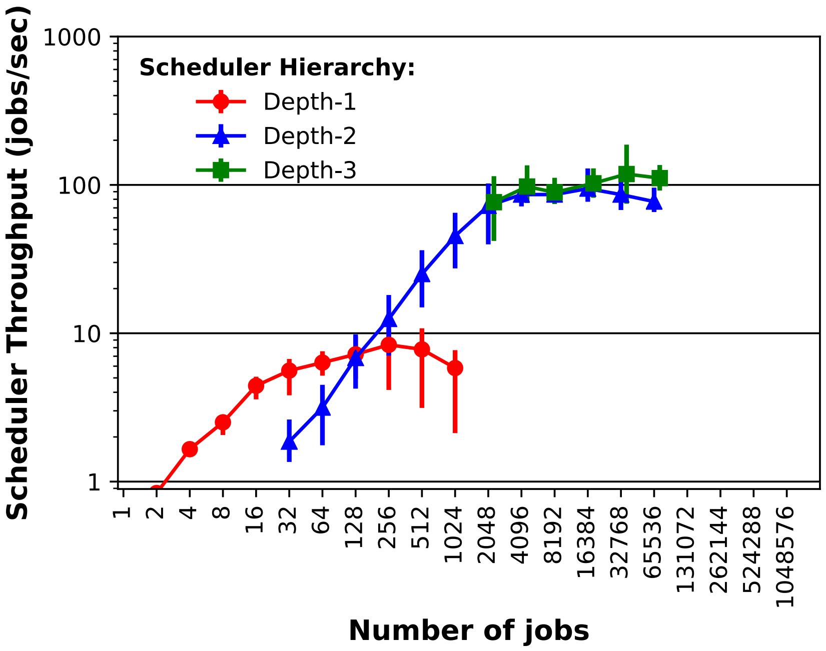
Deepening scheduler hierarchy can significantly improve job throughput.



- Depth-1: allocation level scheduler only
- Depth-2: spawns additional node-level schedulers
- Depth-3: further spawns core-level schedulers



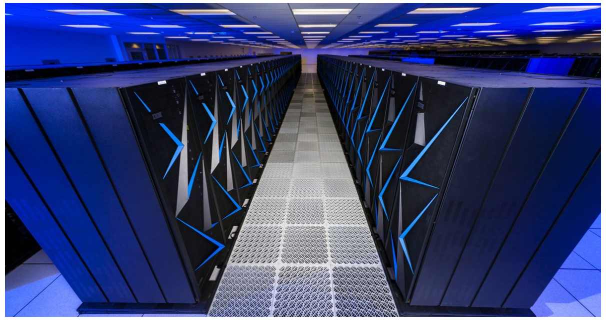
Stress test

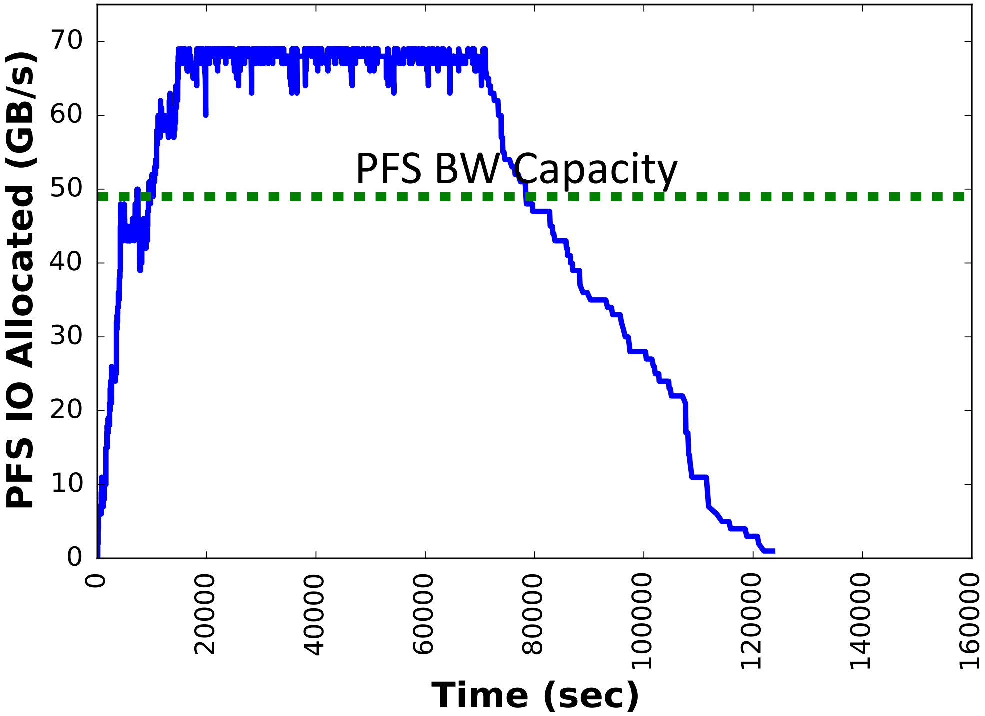


UQ workflow

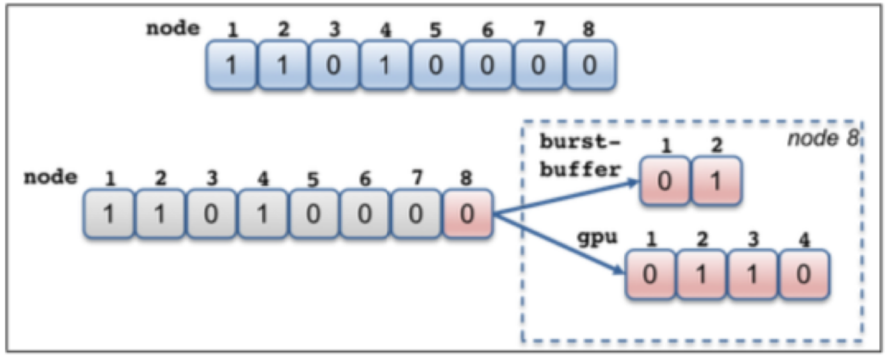
The changes in resource types are equally challenging.

- 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…
  - GPGPUs
  - Burst buffers
  - I/O and network bandwidth
  - Network locality
  - Power





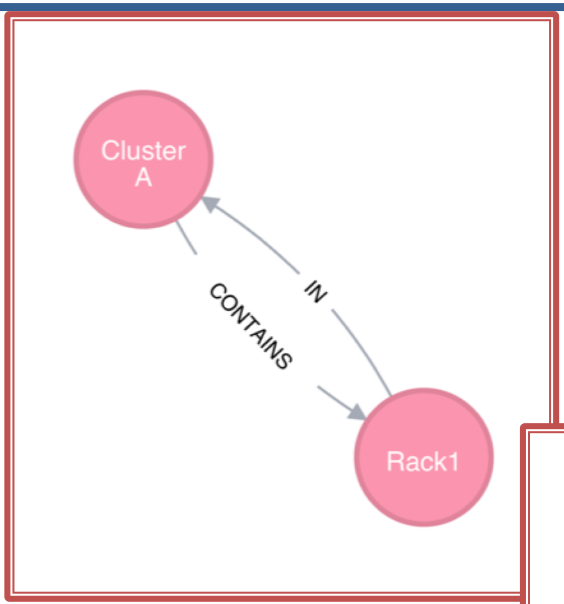
The traditional resource data models are largely ineffective to cope with the resource challenge.



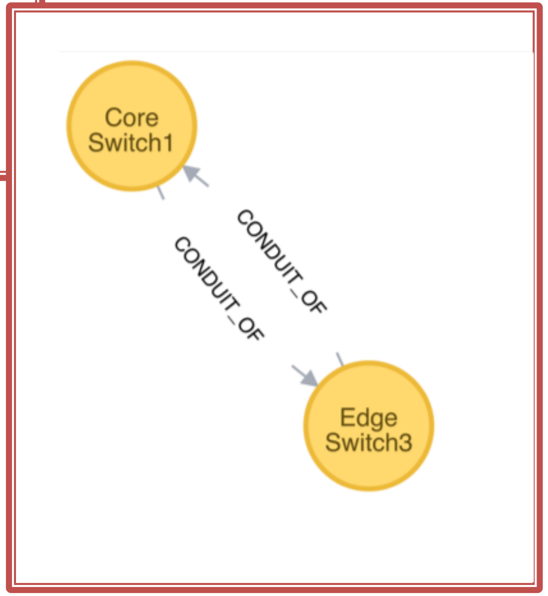
- Designed when the systems are much simpler
  - Node-centric models
  - SLURM: bitmaps to represent a set of compute nodes
  - PBSPro: a linked-list of nodes
- HPC has become far more complex
  - Evolutionary approach to cope with the increased complexity
  - E.g., add auxiliary data structures on top of the node-centric data model
- Can be quickly unwieldy
  - Every new resource type requires new a user-defined type
  - A new relationship requires a complex set of pointers cross-referencing different types.

Flux uses a graph-based resource data model to represent schedulable resources and their relationships.

- A graph consists of a set of vertices and edges
  - Vertex: a resource
  - Edge: a relationship between two resources
- Highly composable to support a graph with arbitrary complexity
- The scheduler remains to be a highly generic graph code.



Containment subsystem



Network connectivity subsystem

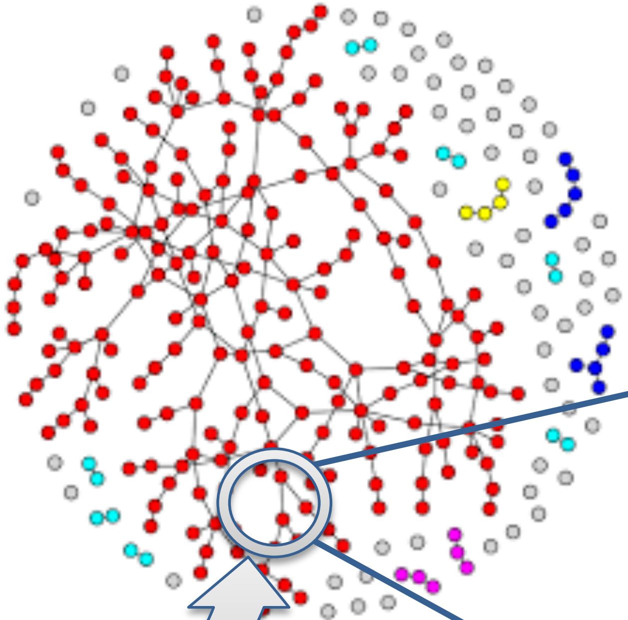
Flux's graph-oriented canonical job-spec allows for a highly expressive resource requests specification.

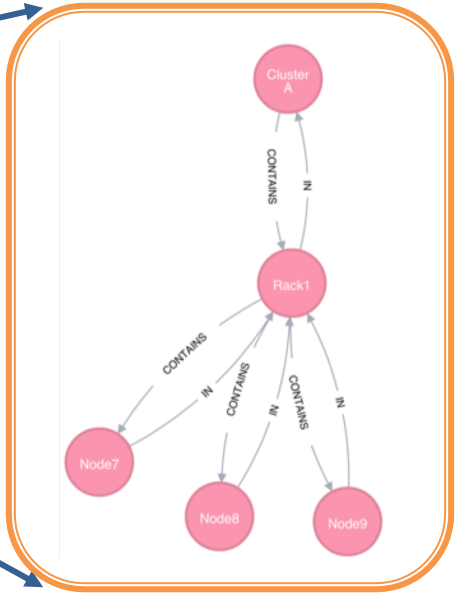
- Graph-oriented resource requirements
  - Express the resource requirements of a program to the scheduler
  - Express program attributes such as arguments, run time, and task layout, to be considered by the execution service
- cluster->racks[2]->slot[3]->node[1]->sockets[2]->core[18]
- ***slot*** is 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
```

Flux maps our complex scheduling problems into graph matching problems.

```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

Flux significantly addresses emerging workflow and resource challenges on high-end HPC systems.

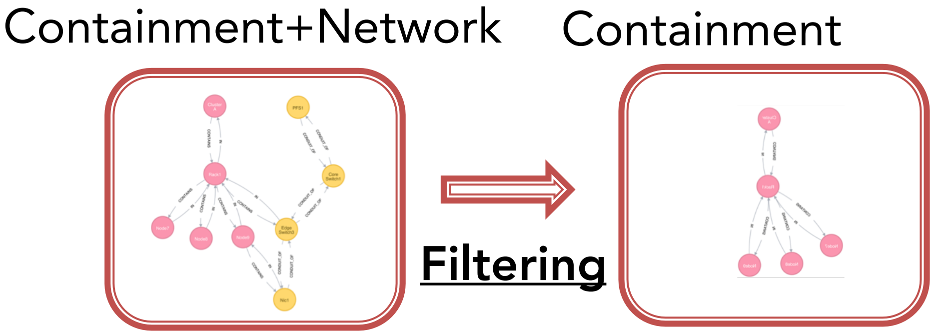
- Scheduling today's HPC centers are hampered by two broad categories of technical challenges: the workflow and resource challenges
- Flux's fully hierarchical scheduling comprehensively addresses workflow challenges.
- Flux is powering up the production runs of the major science runs on LLNL's Sierra, pre-exascale system.
- Flux's graph-based resource model and jobspec lays the foundation for addressing the resource challenge.

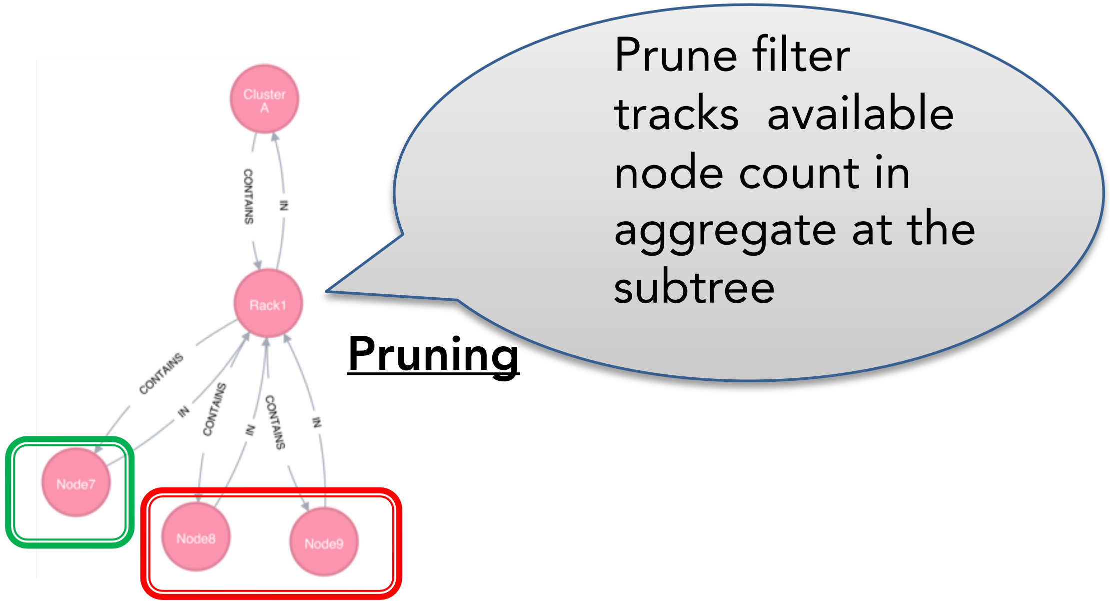
Resources

- flux-core:https://github.com/flux-framework/flux-core
- flux-sched: https://github.com/flux-framework/flux-sched
- Fully hierarchical scheduler: https://github.com/flux-framework/flux-hierarchy
- Workflow examples: https://github.com/flux-framework/flux-workflow-examples
- Quick guide: https://github.com/flux-framework/flux-framework.github.io

We use graph filtering and pruned searching to manage the graph complexity and optimize our 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





Lawrence Livermore National Laboratory