The Flux framework: overcoming scheduling and management challenges of exascale workflows

ECP Tutorial, Module 1:

Feb 10, 2023, 2-3:30p CST

Al Chu, James Corbett, Ryan Day, Jim Garlick, Giorgis Georgakoudis, Mark Grondona, **Dan Milroy**, Zeke Morton, Chris Moussa, **Tapasya Patki,** Barry Rountree, Abhik Sarkar, **Tom Scogland,** Vanessa Sochat, Becky Springmeyer, **Jae-Seung Yeom**

LLNL-PRES-844974

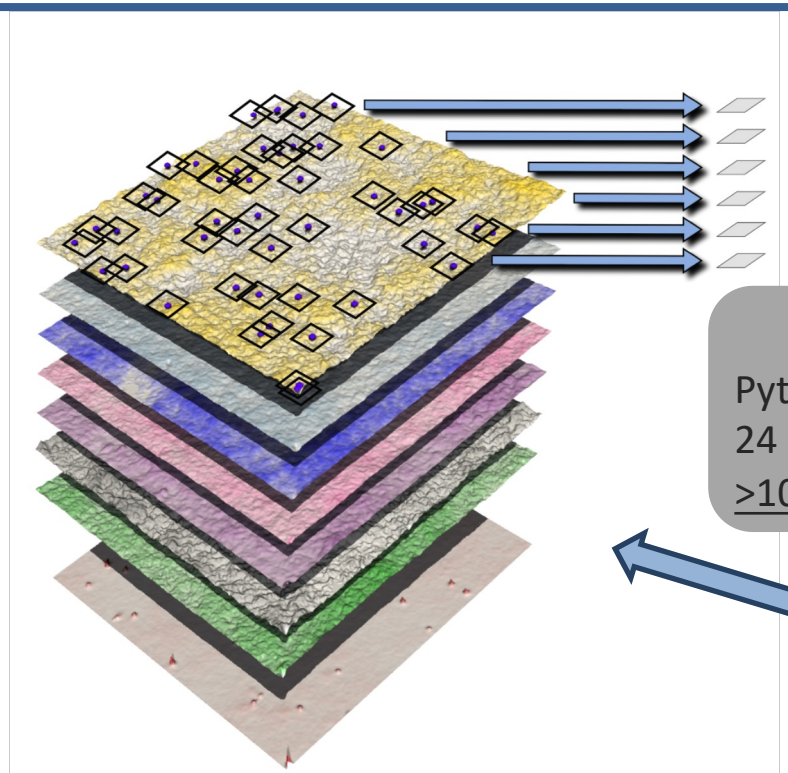
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

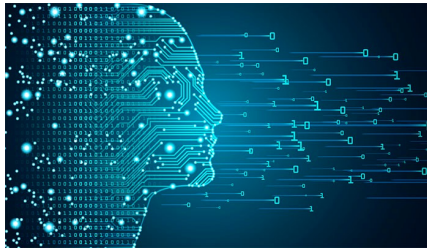
Lawrence Livermore National Laboratory

Agenda for Module 1

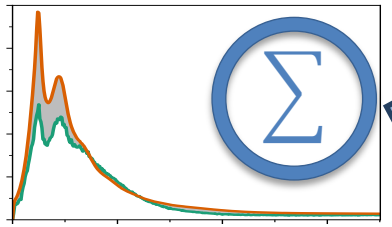
Introduction to Flux • Overview and Roadmap • Workflow and Resource Management Challenges at Exascale • Hierarchical and Graph-based Scheduling • Building and using Flux APIs	Tom Scogland	30 min
Hands-on exercises with AWS and Jupyter • Getting to know Flux and launching jobs • Process and job utilities • Bulk job submission • Flux Python API • Deeper dive	Dan Milroy, Flux team	60 min

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

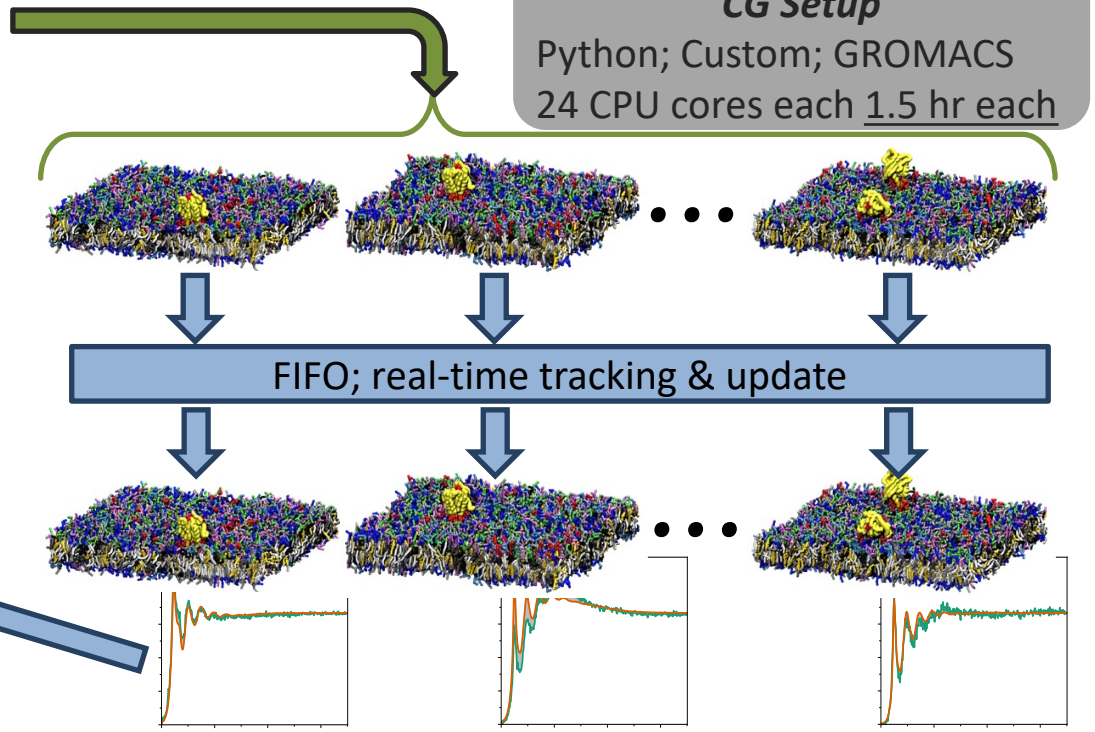




ML-based Selection
Python; ML frameworks; FAISS
24 CPU cores
>1000 decisions per minute



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



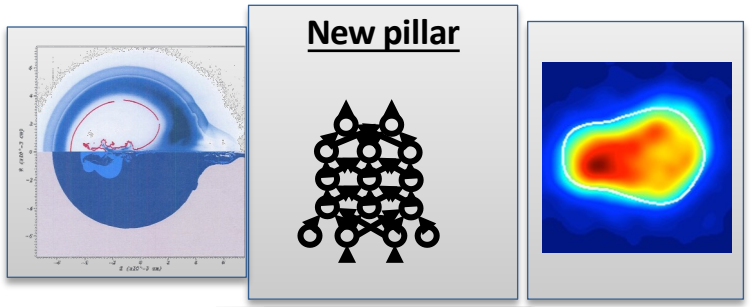
Single Macro Simulation
C++ (with MPI); MOOSE; ddcMD
24 CPU cores/node; 2400 MPI tasks
242 GB per day

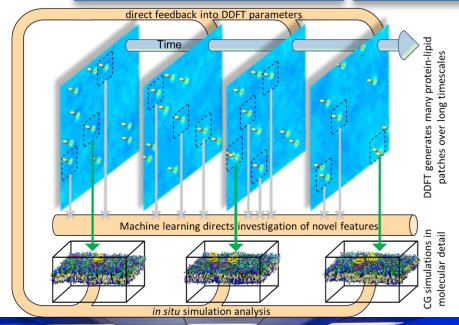
Analysis Aggreg. & Feedback
Python; Custom
24 CPU cores
120,000 reads per cycle

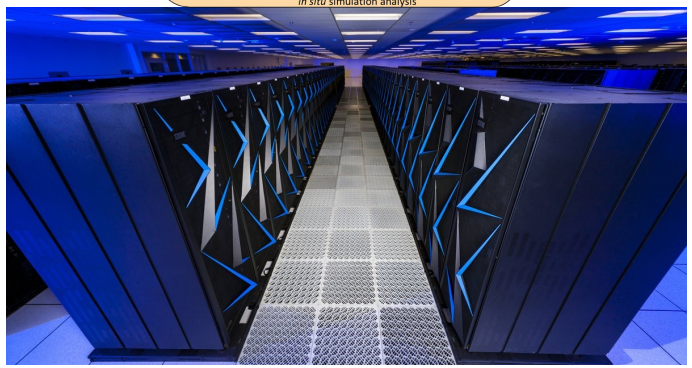
CG Simulation
C++ (with CUDA); ddcMD
1 GPU + 1 CPU core each
1.04 μs and ~6.5 GB per day

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.

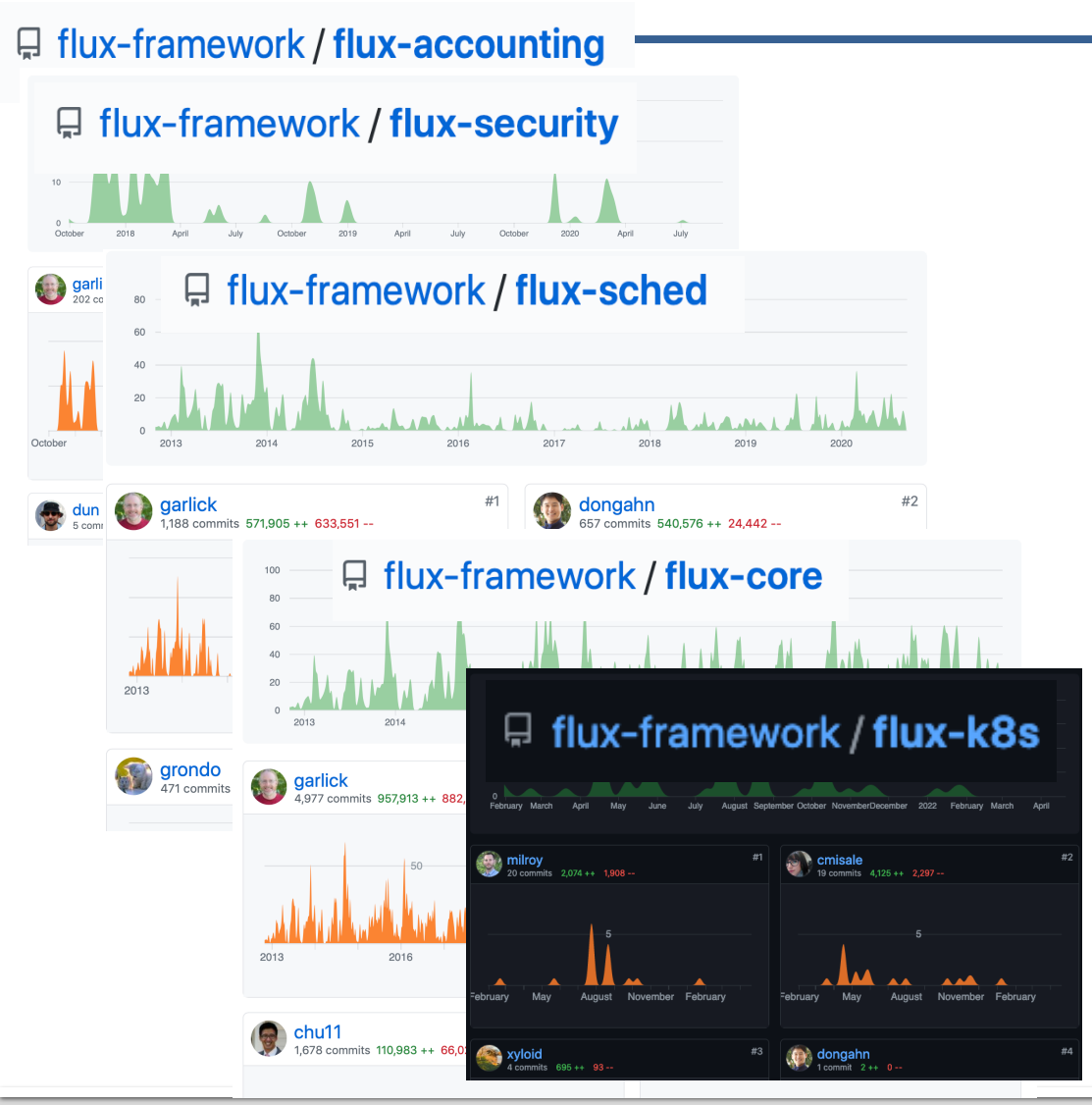






- Co-scheduling
- Job throughput
- Job communication/coordination
- Portability
- Extremely heterogenous resources

Flux solves key technical problems that emerge from these trends.

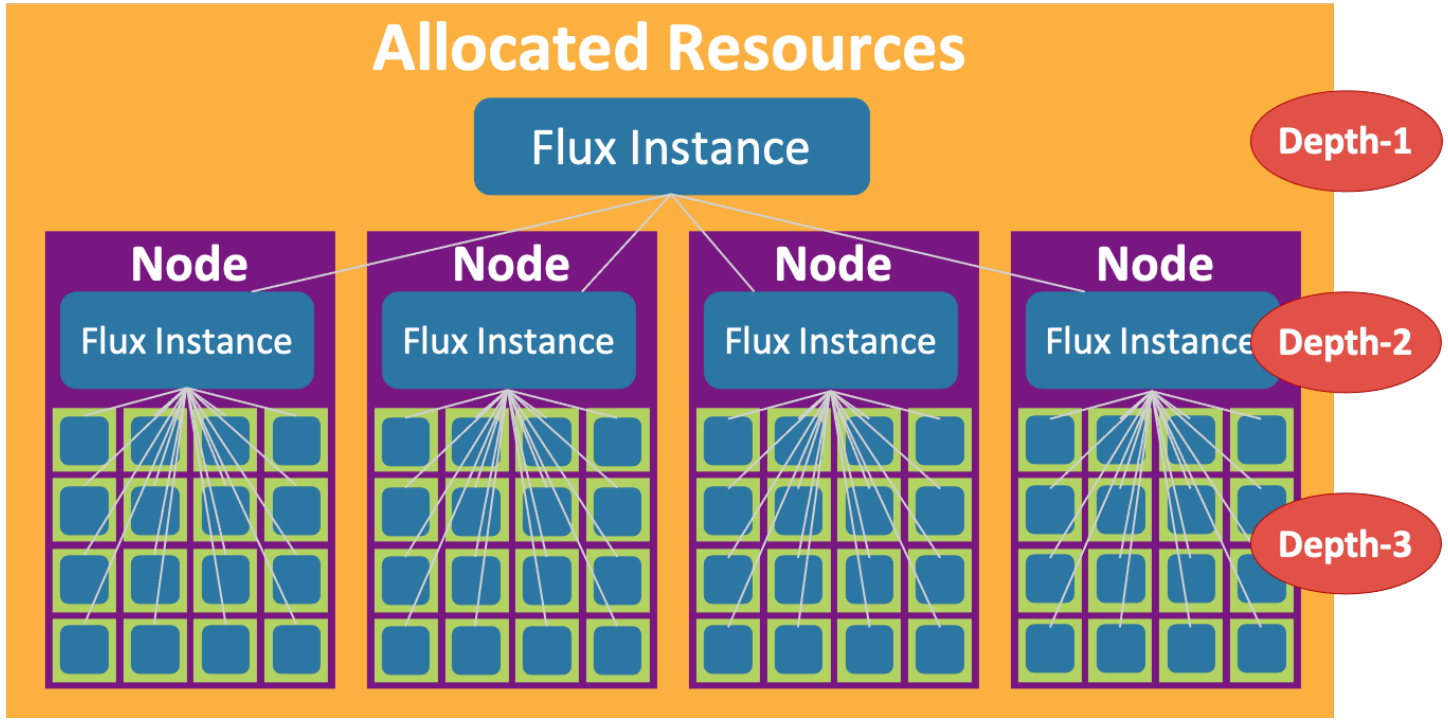


- Open-source project in active development at flux-framework GitHub organization
  - Multiple projects: flux-core, -sched, -security, -accounting, -k8s etc.
  - Over 15 contributors including some principal engineers behind Slurm
- Single-user and System instance modes
  - Single-user mode in production for about 4 years
  - Multi-user mode debuting on LLNL Linux clusters
- Plan of record for **LLNL El Capitan** exascale system

flux

2021 R&D 100 WINNER

Flux's fully hierarchical resource management solves primary deficiencies of the conventional approach.



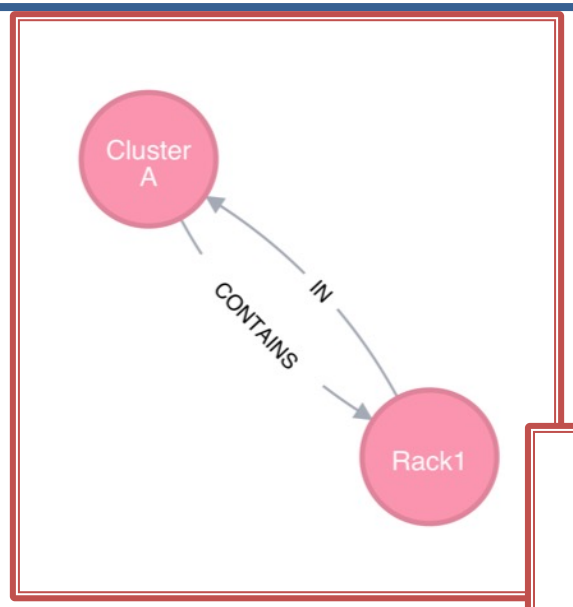
"Fractal scheduling" mitigates centralized scheduler bottleneck

- handles high throughput
- job steps needn't hit central sched

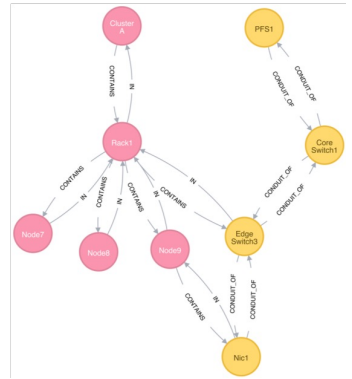
- Full workflow-enablement support
  - Via hierarchical resource subdivision
  - Sub-resource manager per subdivision with service specialization
  - E.g., at LLNL: MuMMI, AHA MoleS, UQP
- Capable of managing resources from almost anywhere
  - Bare metal resources, virtual machines in the cloud, HPC resources allocated by another workload manager
  - Workflows only need to program to Flux
- Provide rich and well-defined interfaces
  - Facilitate communications and coordination among tasks within a workflow
  - CLI, Python, C

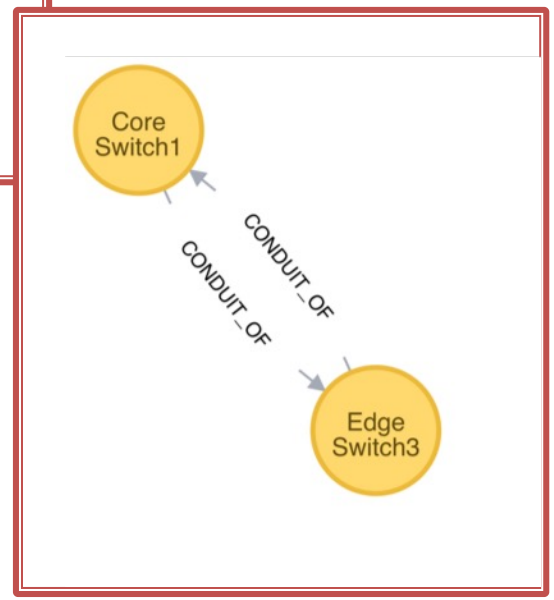
Flux pioneers and uses graph-based scheduling to manage complex combinations of extremely heterogenous resources.

- Traditional resource data models are largely ineffective for resource heterogeneity
  - designed when systems were simpler
  - node-centric models
- Elevate resource relationships (edges) to an equal footing with resources (vertices)
- Complex scheduling can be expressed without changing the scheduler code
- Rich and well-defined C and C++ API for graph traversal and allocation



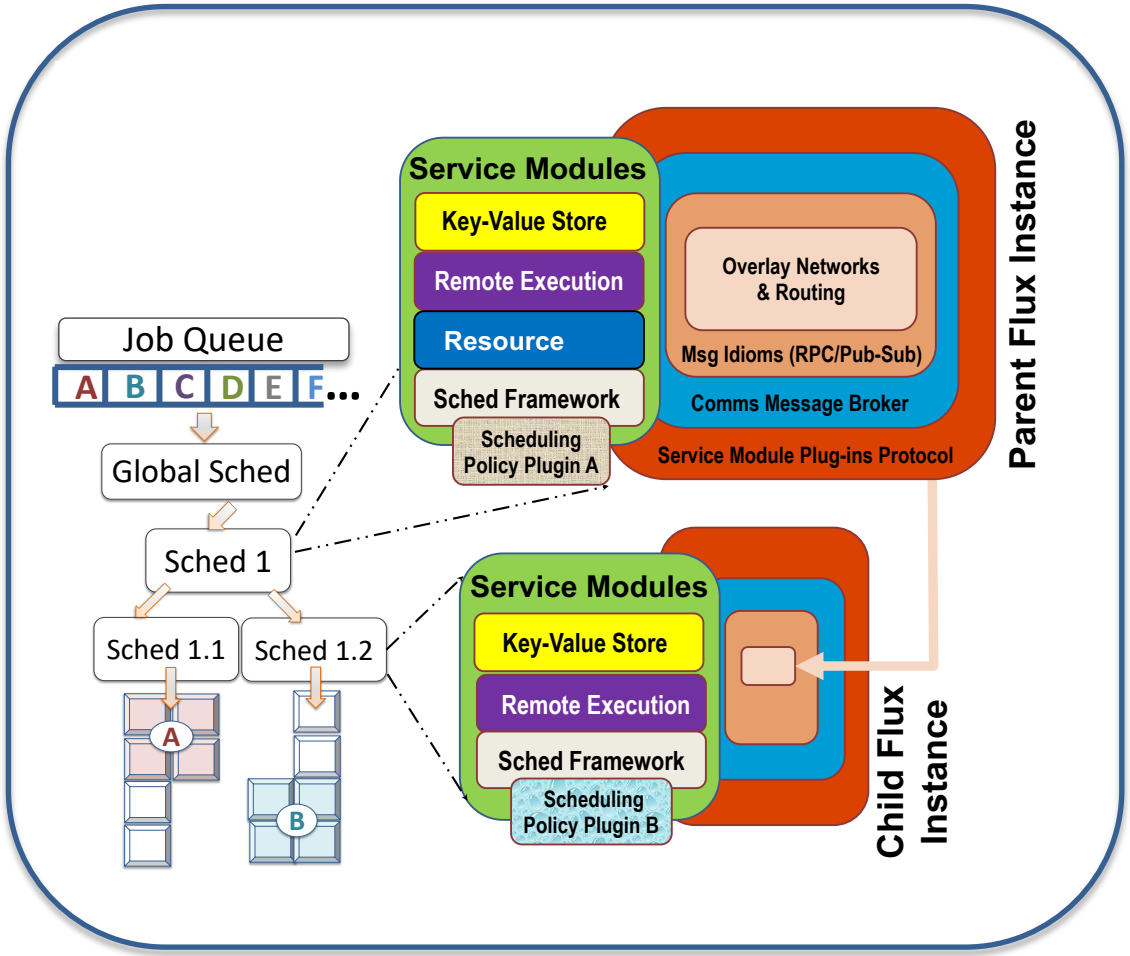
Containment subsystem





Network connectivity subsystem

Flux is architected to embody our fully hierarchical scheduling model.



Flux framework

Techniques

Scheduler Specialization

Scheduler Parallelism

Rich API set

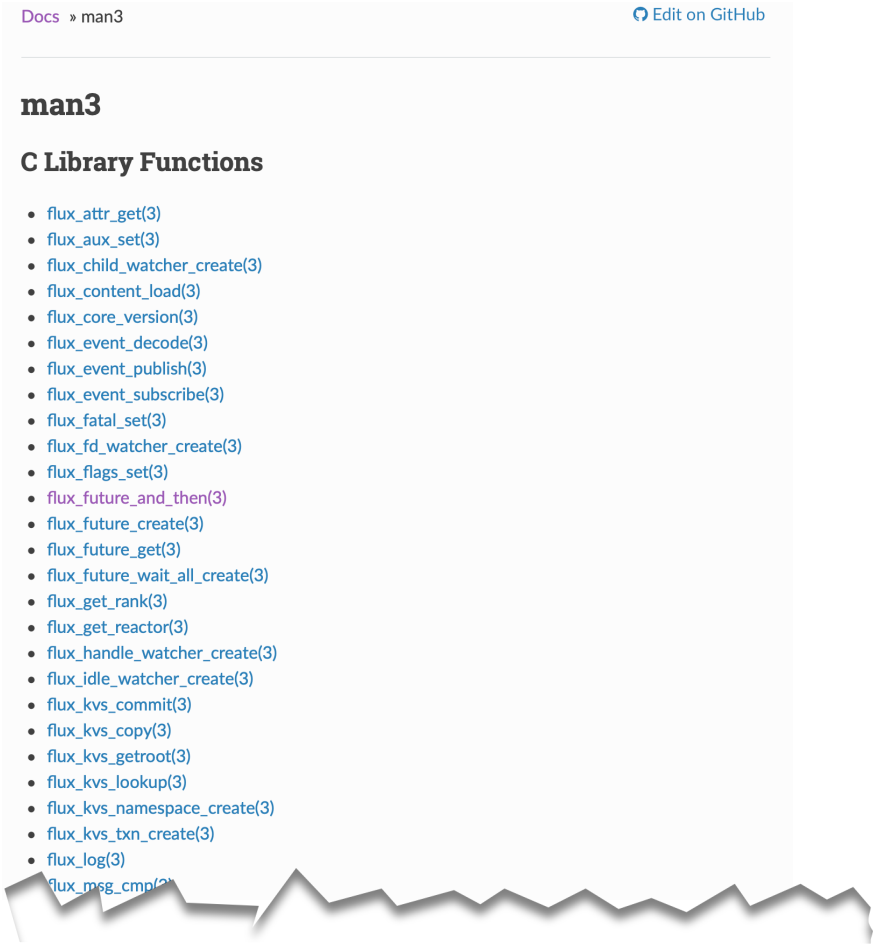
Consistent API set

Challenges

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

A rich set of well-defined APIs enables easy job coordination and communication.

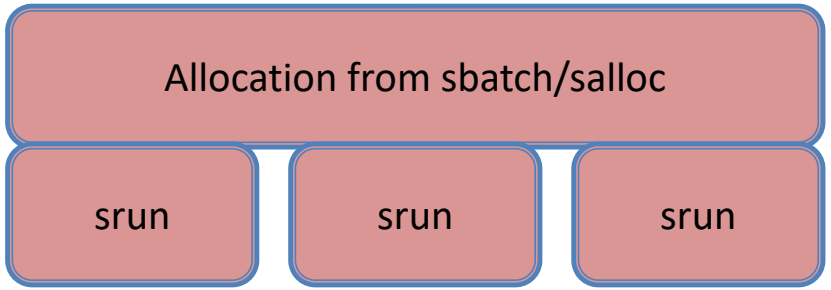
- Jobs in ensemble-based simulations often require close coordination and communication with the scheduler as well as among them.
  - Traditional CLI-based approach can be slow and cumbersome.
  - Ad hoc approaches (e.g., many empty files) can lead to many side effects.
- Flux provides well-known communication primitives.
  - Pub/sub, request-response, and send-recv patterns
- High-level services
  - Key-value store (KVS) API
  - Job API (submit, wait, state change notification, etc)
- Flux's APIs are consistent across different platforms



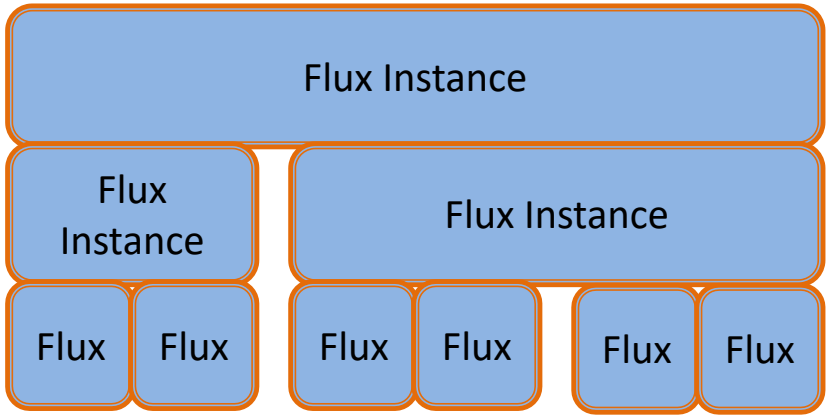
https://flux-framework.readthedocs.io/projects/flux-core/en/latest/index.html

Flux's hierarchical scheduling makes managing and scheduling complex allocations easier.

- Significant start-up cost for running on new resource manager (portability)
  - Learn to use the manager
  - Translate batch script logic
- Traditional resource managers can't subdivide allocations (hierarchy)
  - Slurm has two levels: **`sbatch`** creates allocation, **`srun`** launches parallel task
  - **`sbatch`** from inside an allocation starts a separate allocation
- Flux's hierarchical scheduling provides a natural solution
  - Request one large allocation and partition schedulable resource subgroups



The Slurm hierarchy

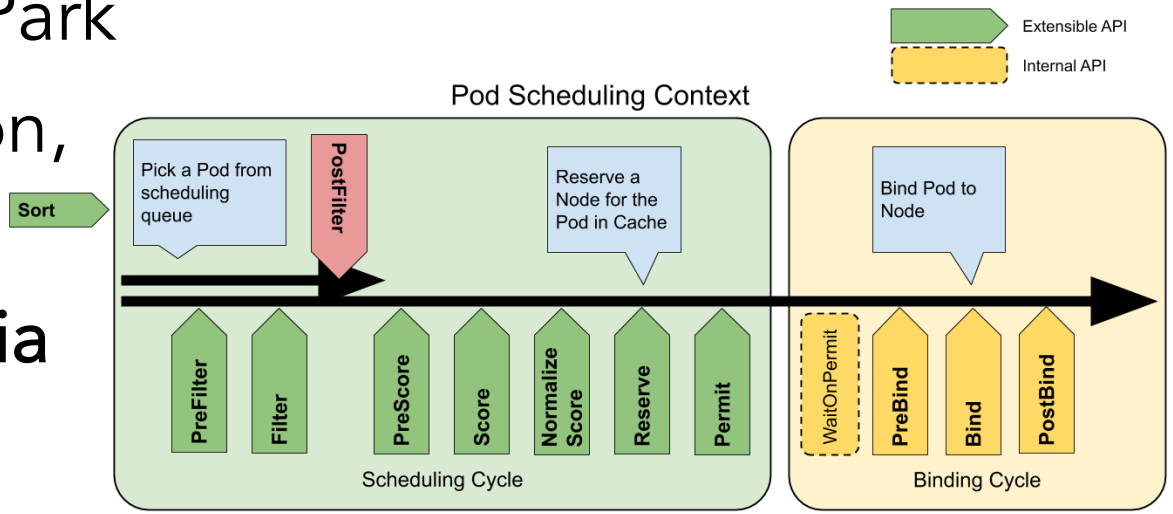


One possible Flux hierarchy

Team formed industry collaborations to tackle converged computing challenges and contribute to community.

We bring complementary backgrounds in HPC, cloud, and performance-oriented orchestration

- LLNL LDRD team
- **IBM T.J. Watson Research Center**: T. Elengikal, M. Drocco, C. Misale, Y. Park
- **AWS**: E. Bollig, M. Hugues, H. Poxon, L. Wofford
- Integrate Fluxion into Kubernetes via Fluence



Accessing the hands-on tutorial

- Using our EC2 instance at **https://tutorial.flux-framework.org**
  - Choose a unique username (if not, you'll be sharing with someone else)
  - Password: **ecp-flux-tutorial2023**
  - Double-click **ecp23-flux.ipynb** to start
    - To execute a cell in JupyterLab: **Shift+Enter**
  - It's a shared instance- please don't run computationally demanding tasks in your container
  - The instance will disappear shortly after the tutorial
- Running locally with Docker:
  - **git clone https://github.com/flux-framework/Tutorials.git**; README in the 2023-ECP JupyterNotebook directory

The Flux framework: overcoming scheduling and management challenges of exascale workflows

ECP Tutorial, Module 2:

Feb 10, 2023, 3:45 – 5:15p CST

Al Chu, James Corbett, Ryan Day, Jim Garlick, Giorgis Georgakoudis, Mark Grondona, **Dan Milroy**, Zeke Morton, Chris Moussa, **Tapasya Patki,** Barry Rountree, Abhik Sarkar, **Tom Scogland,** Vanessa Sochat, Becky Springmeyer, **Jae-Seung Yeom**

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

Agenda for Module 2

Fluxion and Graph-Based Scheduling • I/O and Variation-aware scheduling	Tapasya Patki	15 min
Converged Computing and Fluence	Daniel Milroy	15 min
I/O Awareness and Data Management • DYAD • Hands on exercises with DYAD	Jae-Seung Yeom and Ian Lumsden	30 min
AHA Moles: Example of Integrated Workflow Management with Flux	Xiaohua Zhang	20min
Summary and Q&A	Tom Scogland, Flux team	10 min

Fluxion: Flux's Resource & Job Model

ECP Tutorial, Feb 10, 2023

Tapasya Patki

Acknowledgment: Dong H. Ahn, Ned Bass, Subhendu Behera, Albert Chu, Kyle Chard, Brandon Cook, James Corbett, Ryan Day, Franc Di Natalie, David Domyancic, Phil Eckert, Stephen Herbein, Jim Garlick, Elsa Gonsiorowski, Mark Grondona, Helgi Ingolfsson, Shantenu Jha, Zvonko Kaiser, Dan Laney, Carlos Eduardo Gutierrez, Edgar Leon, Don Lipari, Daniel Milroy, Joseph Koning, Claudia Misale, Chris Moussa, Frank Muller, Yoonho Park, Luc Peterson, Barry Rountree, Thomas R. W. Scogland, Becky Springmeyer, Michela Taufer, Jae-Seung Yeom, Veronica Vergara and Xiaohua Zhang

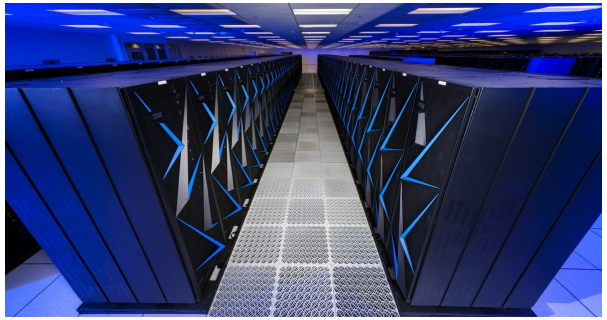
LLNL-PRES-844974

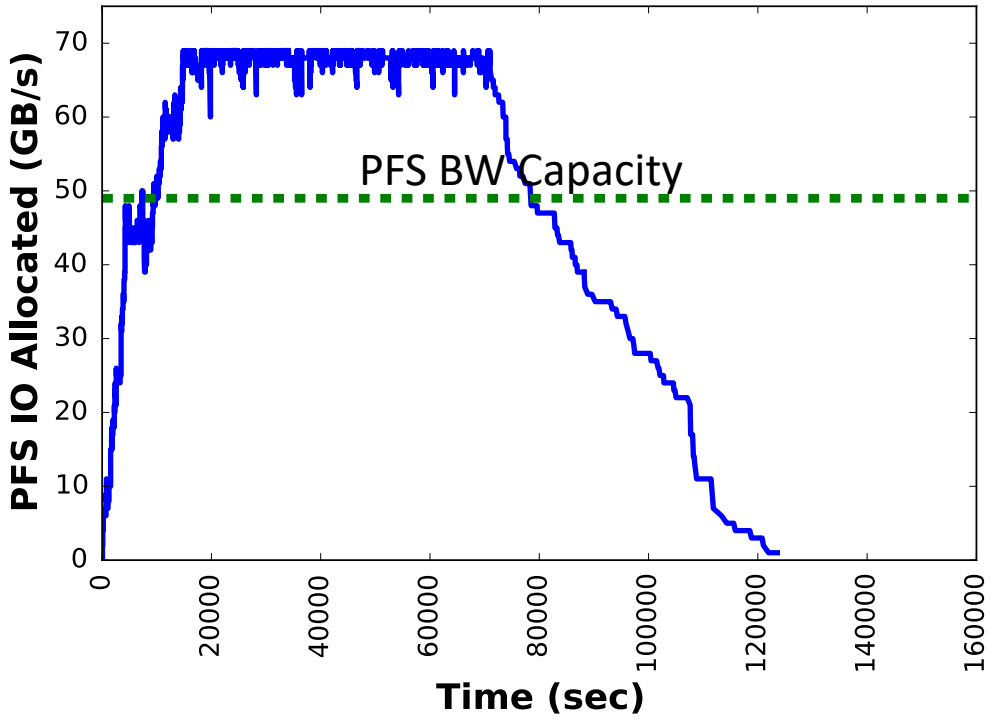
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

The changes in resource types are challenging.

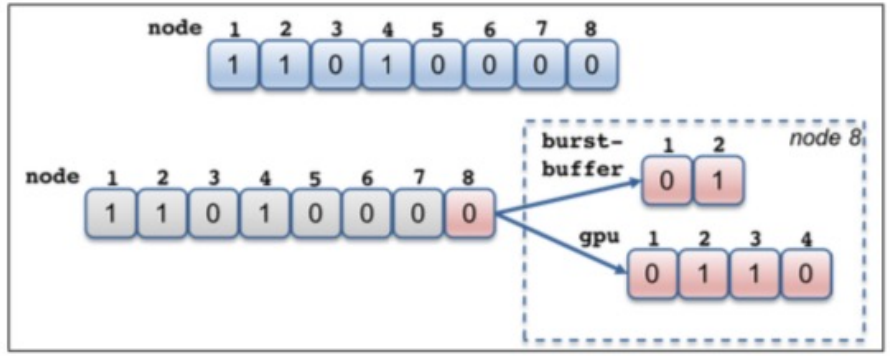
- Problems are not just confined to the workload/workflow challenge.
- Resource types and their relationships are also becoming increasingly complex.
- Much beyond compute nodes and cores requiring partial occupancy and accounting…
  - GPGPUs
  - Burst buffers
  - I/O and network bandwidth
  - Network locality
  - Power
  - Variation





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

- Designed when the systems are much simpler
  - Node- or core-centric models
  - SLURM: bitmaps to represent a set of compute nodes
  - PBSPro: a linked-list of nodes

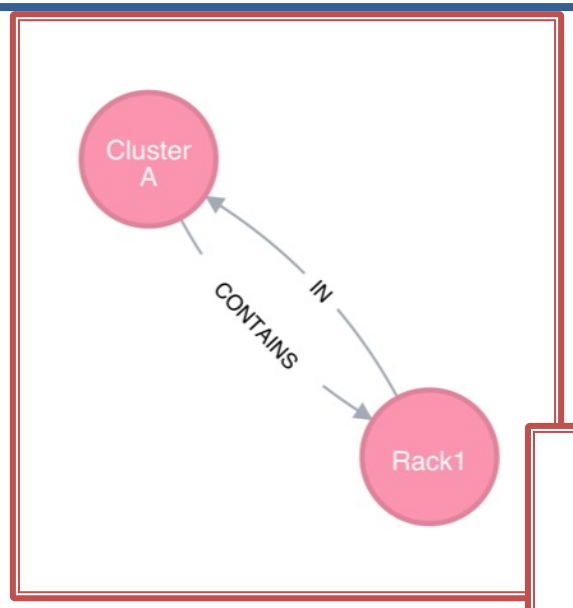


- HPC has become far more complex
  - Evolutionary approach to cope with the increased complexity
  - E.g., add auxiliary data structures on top of the node-centric data model

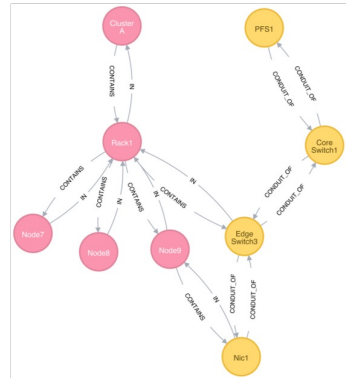
- Can be quickly unwieldy
  - Every new resource type requires new a user-defined type
  - A new relationship requires a complex set of pointers cross-referencing different types.
  - Dynamic updating of flow resources is not supported

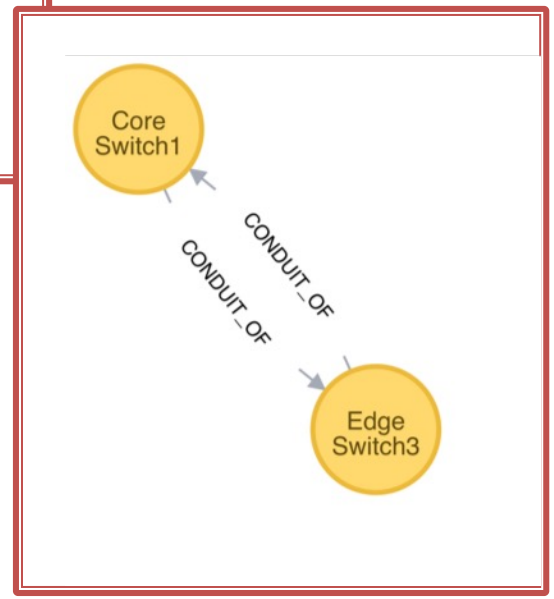
Flux pioneers and uses graph-based scheduling to manage complex combinations of extremely heterogenous resources.

- Traditional resource data models are largely ineffective for resource heterogeneity
- Elevate resource relationships (edges) to an equal footing with resources (vertices)
- Complex scheduling can be expressed without changing the scheduler code
- Rich and well-defined C and C++ API for graph traversal and allocation
- High expressibility and level of detail control



Containment subsystem





Network connectivity subsystem

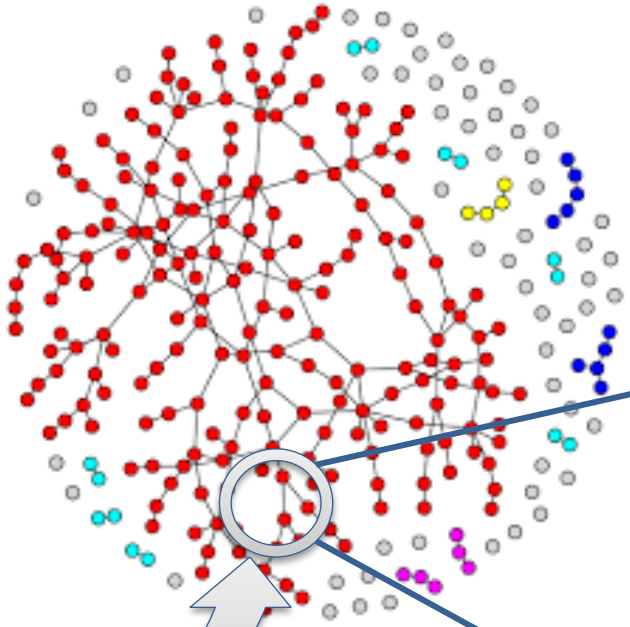
Flux's graph-oriented canonical job-spec allows for a highly expressive resource requests specification.

- 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

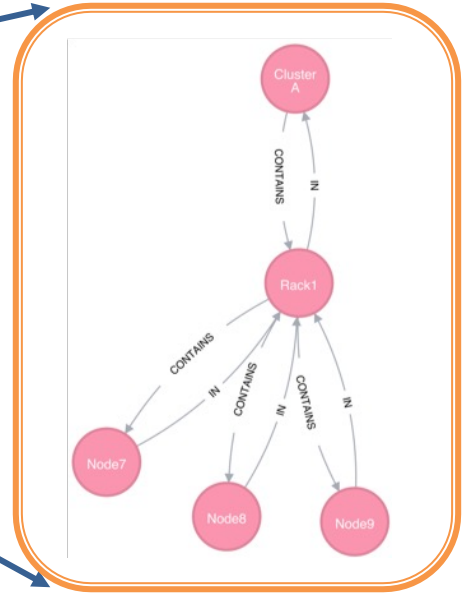
```
version: 1
resources:
  - type: cluster
    count: 1
    with:
      - type: rack
        count: 2
        with:
          - type: slot
            label: myslot
            count: 3
            with:
              - type: node
                count: 1
                with:
                  - type: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

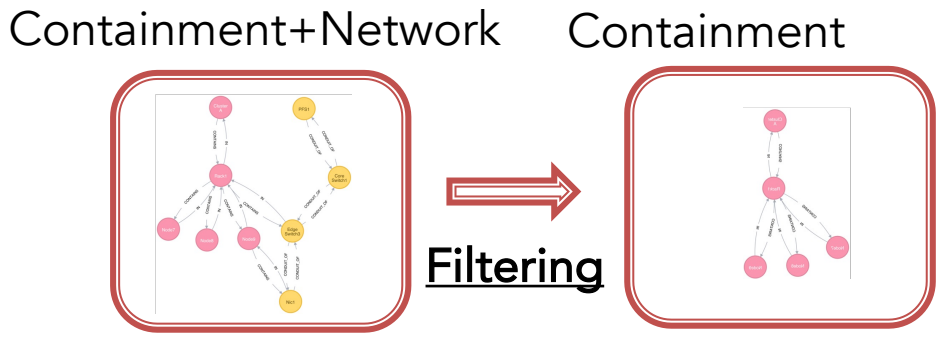


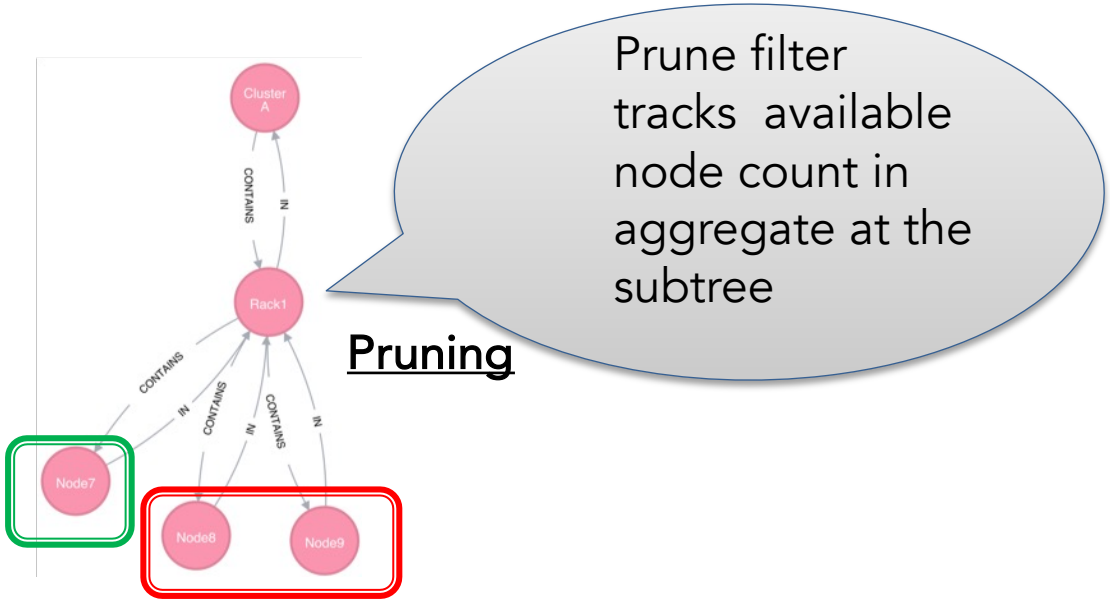
Traverse, match and score



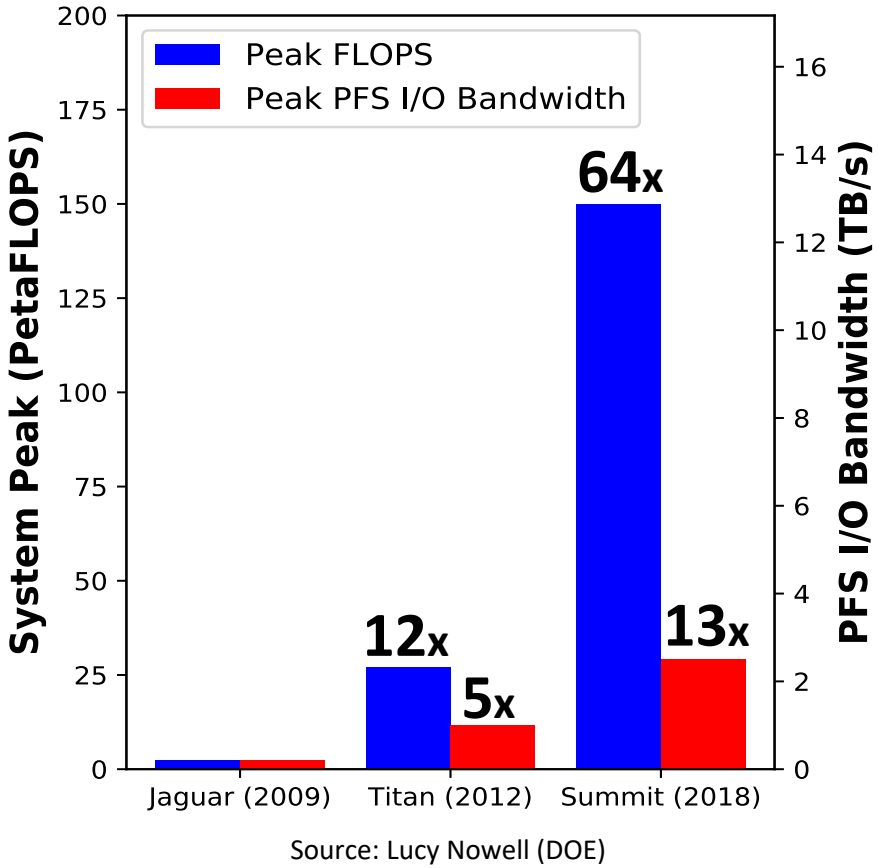
We use graph filtering and pruned searching to manage the graph complexity and optimize our graph search.

- 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

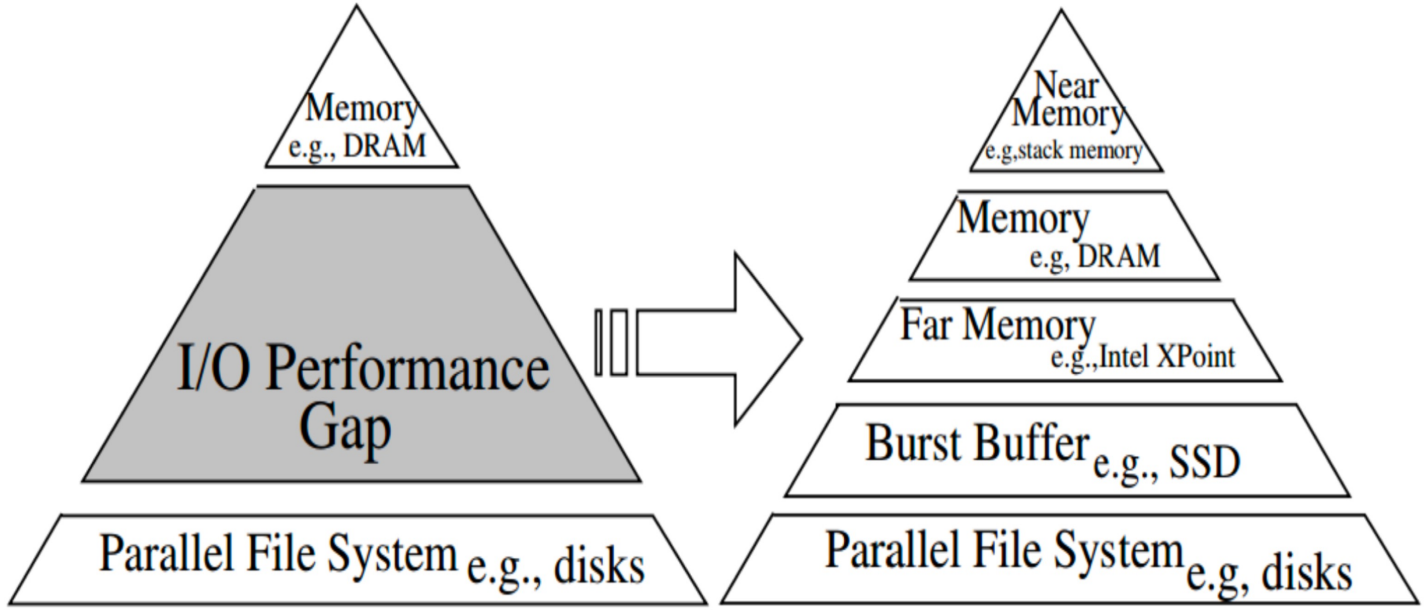




Tiered Storage in HPC

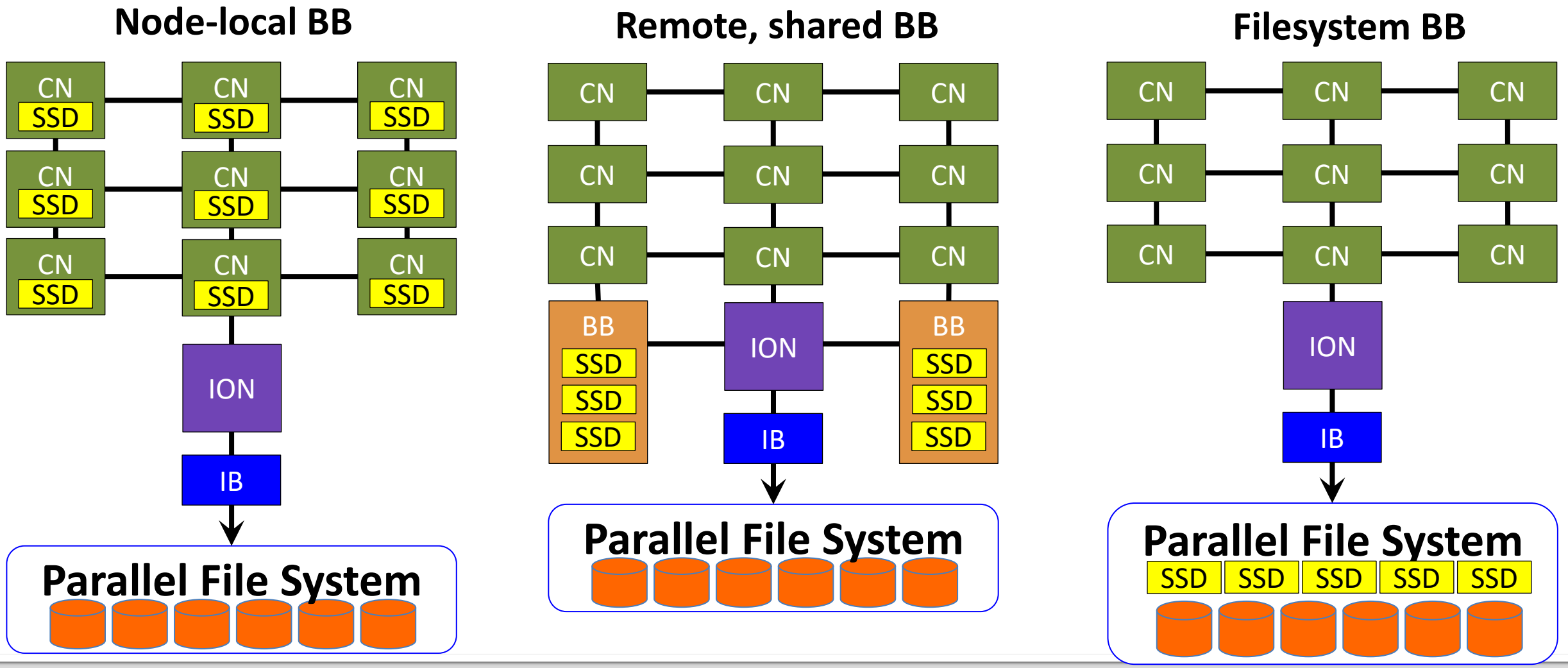


Source: Lucy Nowell (DOE)



How can we use the Flux graph-based scheduler to allocate these new storage tiers?

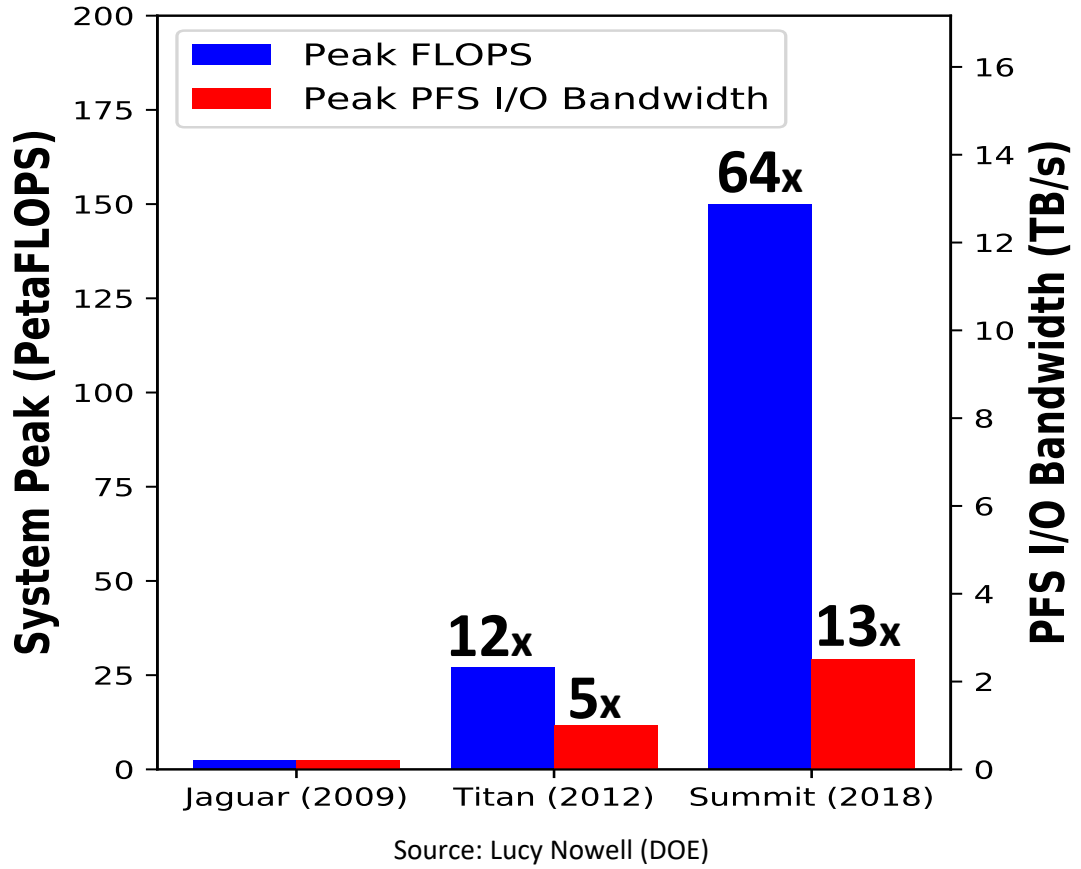
Burst Buffer Architectures



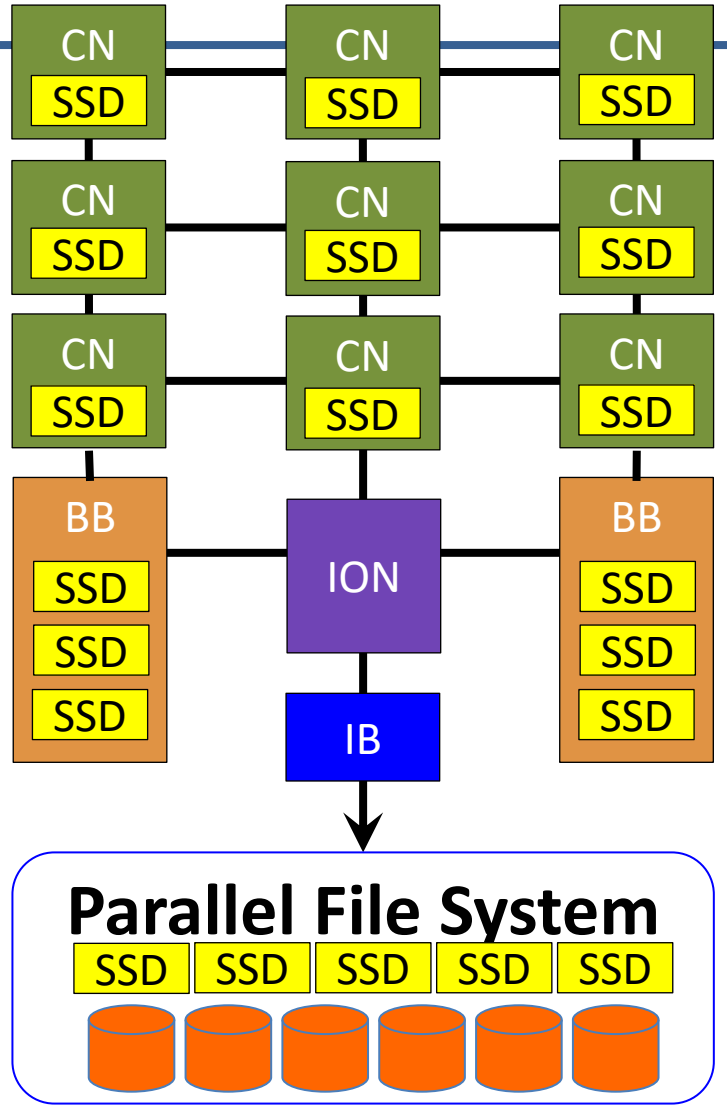
Burst Buffers Functionality "Classes"

- Cache
  - acts like an extended OS page cache for a given job or parallel filesystem
  - all data movement between the BB and the PFS occurs implicitly
- Scratch
  - functions like a /tmp file system for the given node, job, or workflow
- Staging
  - all data movement occurs explicitly for a given job
  - data movement can occur before and after the job is the running

Tiered Storage in HPC



Source: Lucy Nowell (DOE)

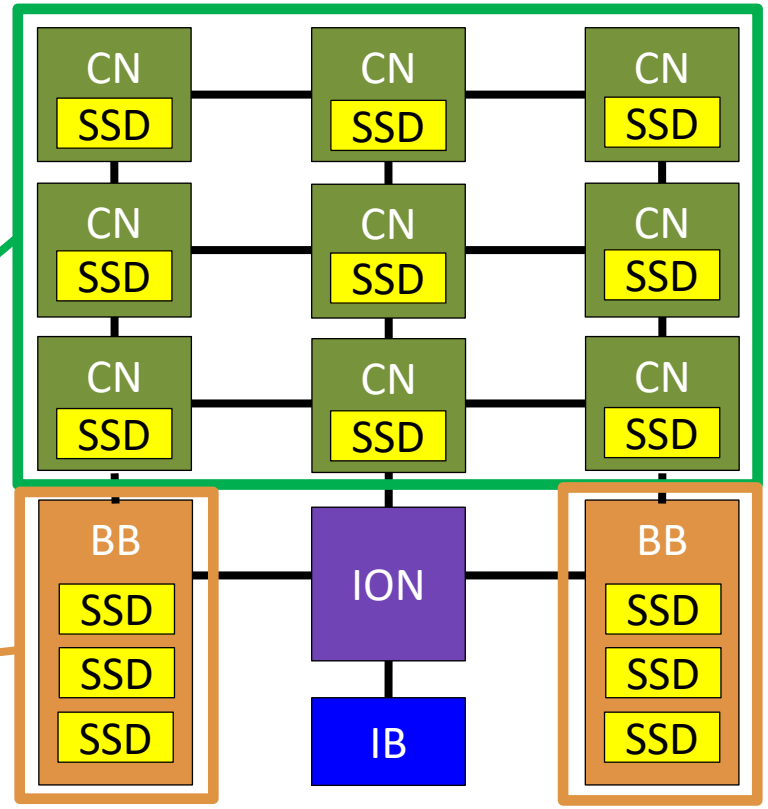


We can use the Flux graph scheduler to allocate these new storage tiers with 0 code changes

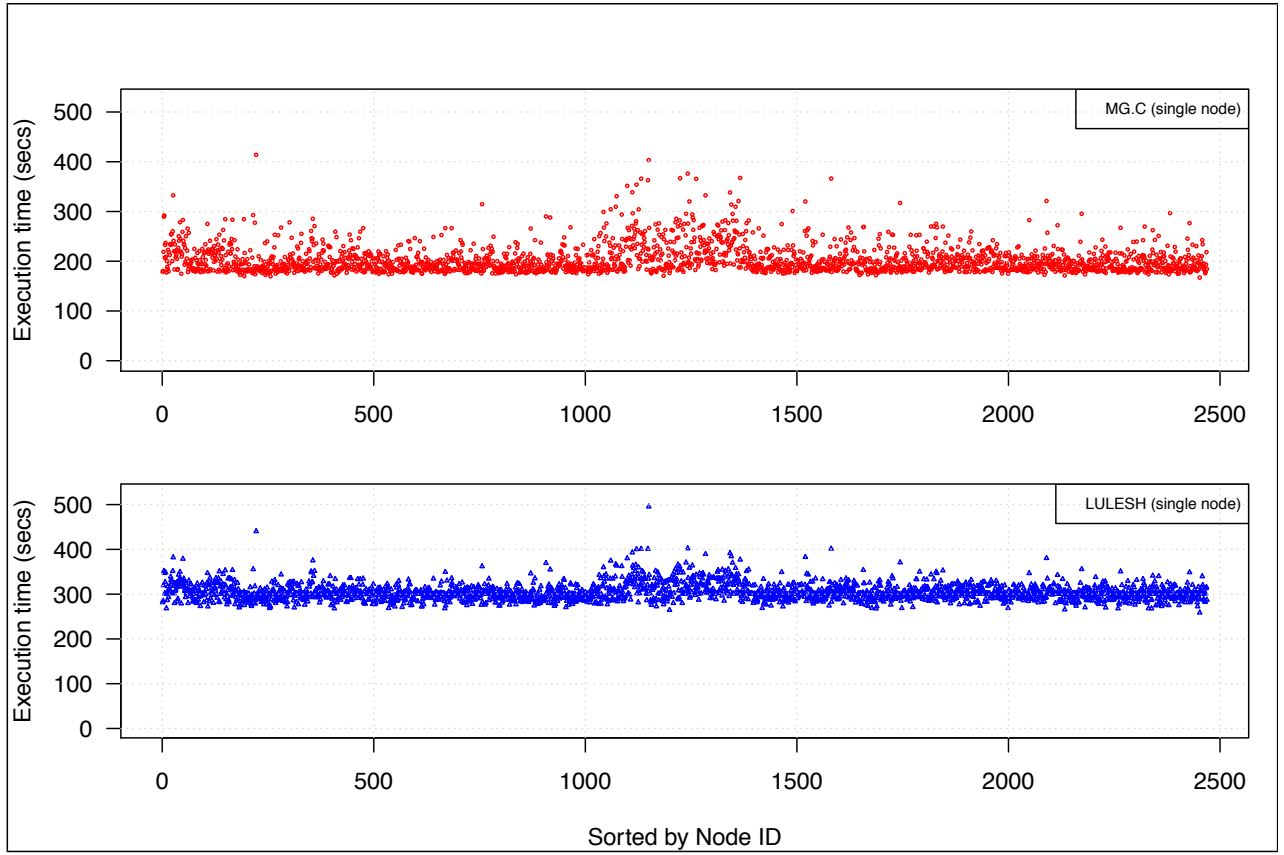
Example Tiered Storage Request in Flux Jobspec

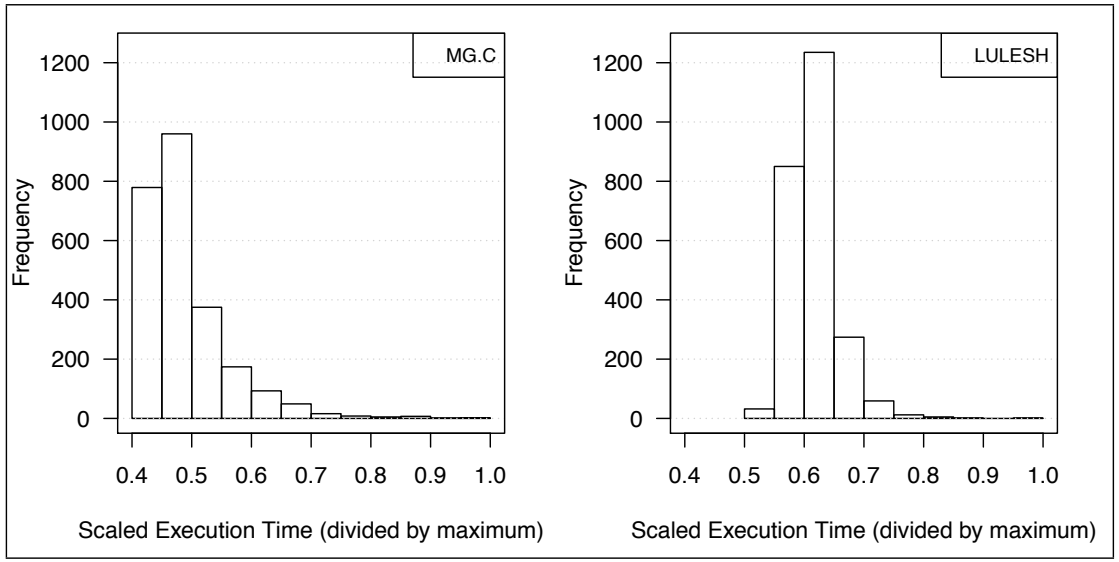
```
resources:
  - type: node
    count: 9
    with:
      - type: slot
        count: 1
        label: default
        with:
          - type: core
            count: 2
      - type: storage
        count: 1
        unit: terabytes
        label: node-local-scratch
  - type: storage
    count: 4
  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



Variation-aware scheduling with Flux: Addressing Manufacturing Variability, Processor Aging, and inherent heterogeneity



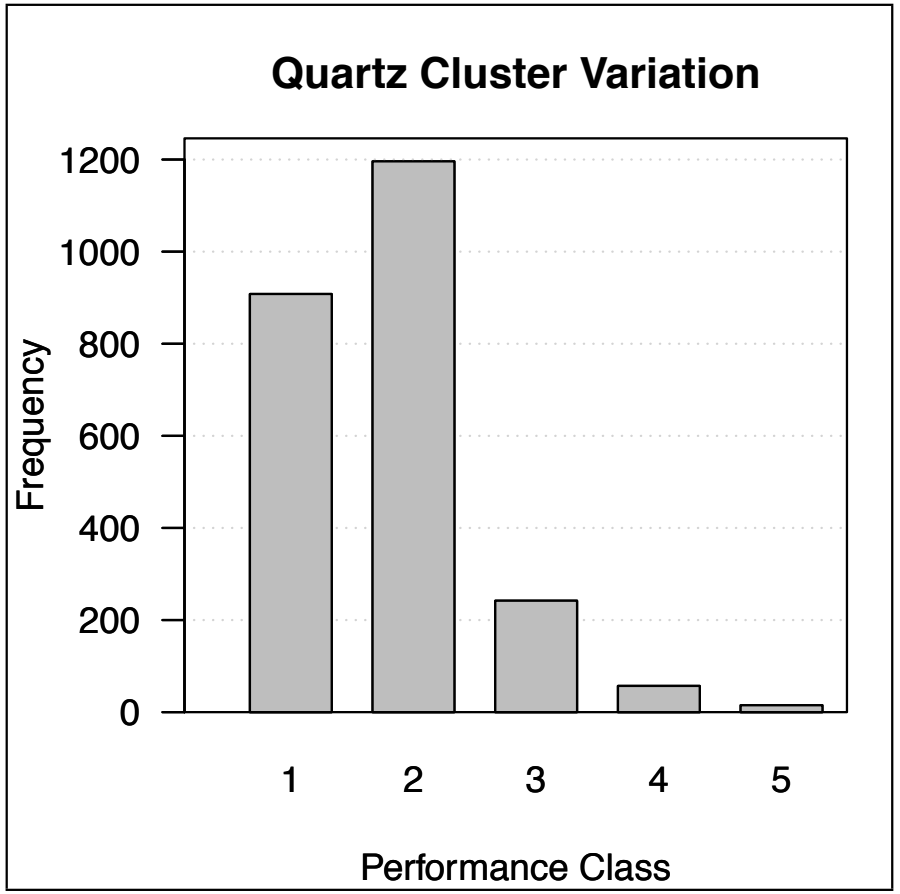


- Real world example under power constraints: Quartz cluster, 2469 nodes, 50 W CPU cap
- 2.47x difference between the slowest and the fastest node for MG
- 1.91x difference for LULESH.

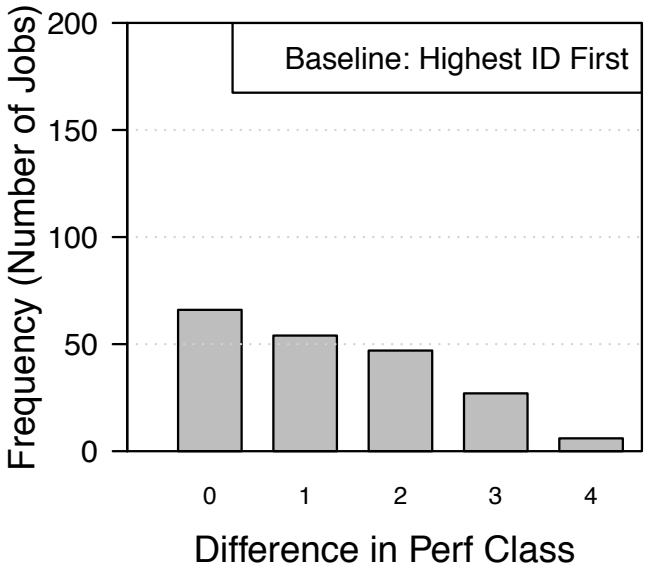
https://github.com/flux-framework/flux-sched/tree/master/resource/policies

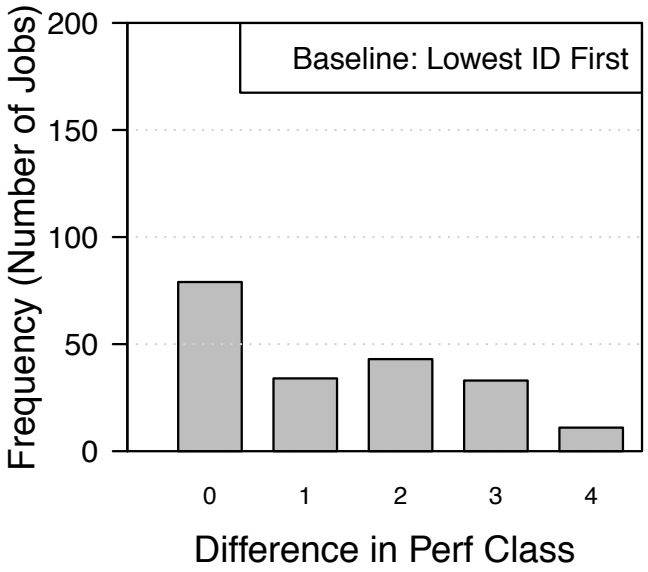
Example: Statically determining node performance classes

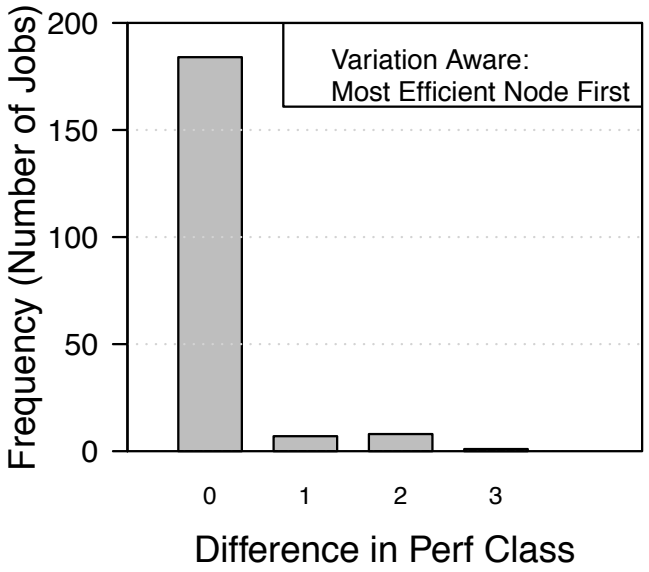
- Ranking every processor is not feasible
- Statically create bins of processors with similar performance instead
  - Techniques for this can be simple or complex
  - How many classes to create, which benchmarks to use, which parameters to tweak
  - Our choice: 5 classes, LULESH and MG, 50 W cap
- Mitigation
  - 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

The Flux Framework and HPC+cloud Converged Computing

Dan Milroy

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

Pre-exascale scientific workflows strain the capabilities of traditional HPC resource managers and schedulers.

Co-scheduling:

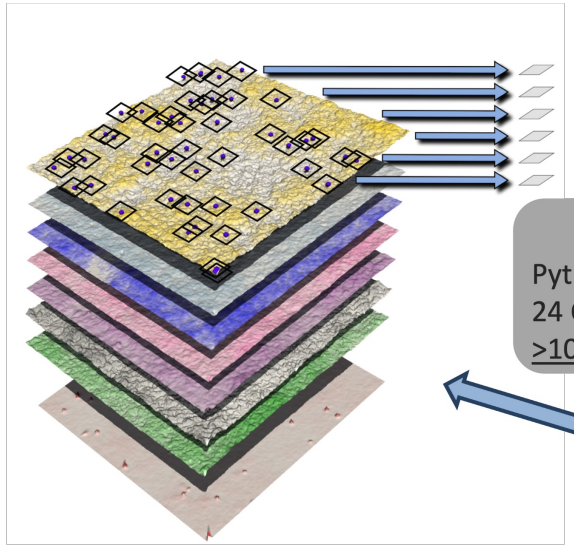
CG, analysis bound to cores nearest PCIe buses

Job comms/coordination:

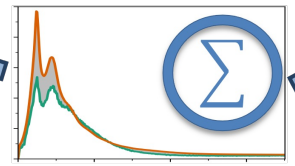
36,000 concurrent tasks;
176,000 cores, 16,000 GPUs

Portability:

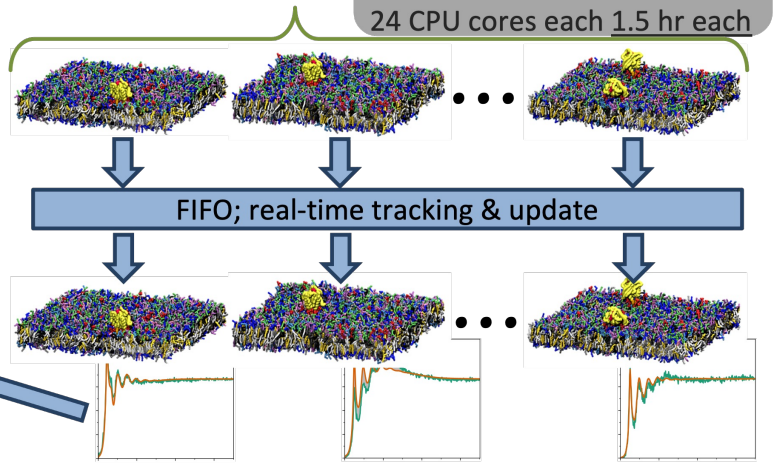
adapt tasks to different schedulers/managers



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



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

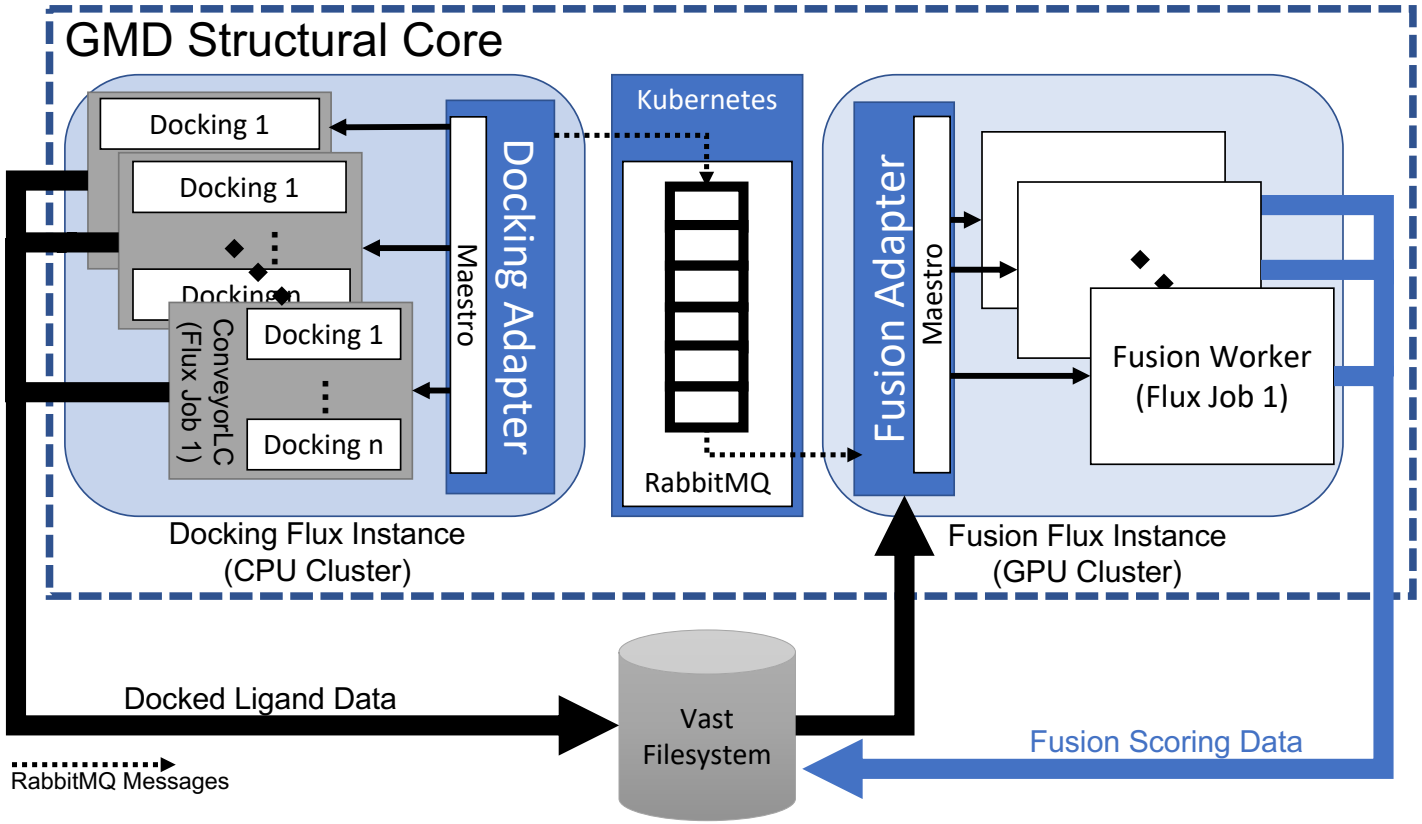
CG Simulation
C++ (with CUDA); ddcMD
1 GPU + 1 CPU core each
1.04 μs and ~6.5 GB per day

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

MuMMI: SC'19 best paper, SC'21 paper

MPI-based simulation with in-situ analysis plus ML

Next-generation, cross-cluster scientific workflows are demanding portability and cloud integration.



AHA MoleS: eScience'22 best paper

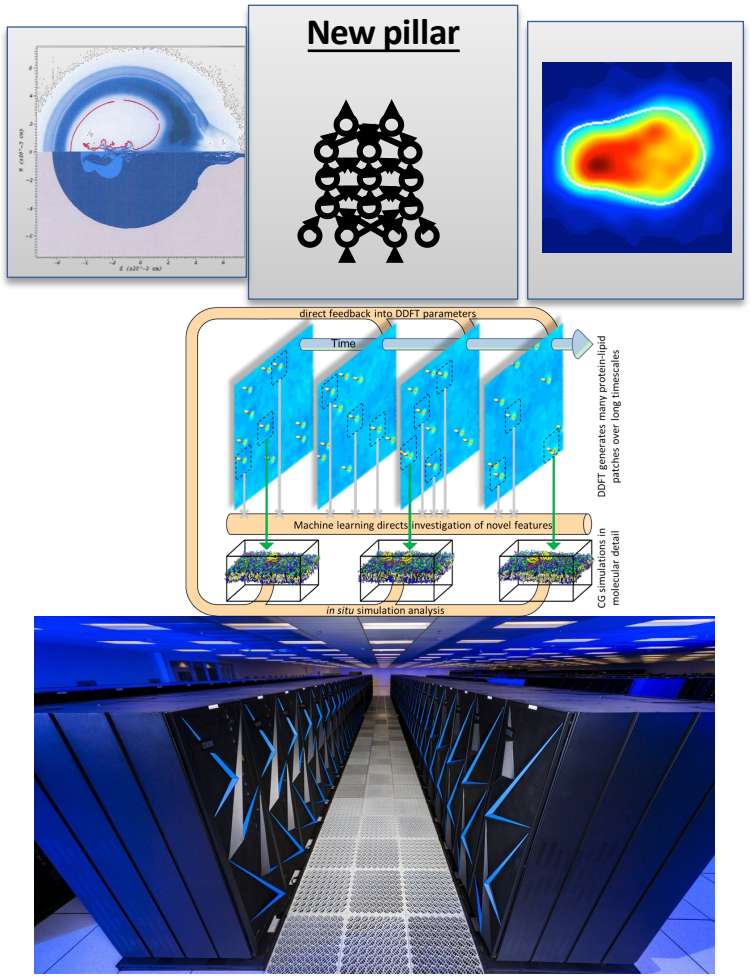
Complex workflows integrating cloud technologies at LLNL and beyond

- Scalable message broker couples MPI-based tasks, analysis, workflow runs anywhere (**AHA MoleS**)
- HPC simulation with AI/ML surrogates, orchestrated databases (AMS)
- Many other examples: ATOM, AMPL, GMD, etc.

2020 lab survey found that 73% of LLNL workflows interested in cloud integration

MPI-based simulation with analysis, ML, and containerized components

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

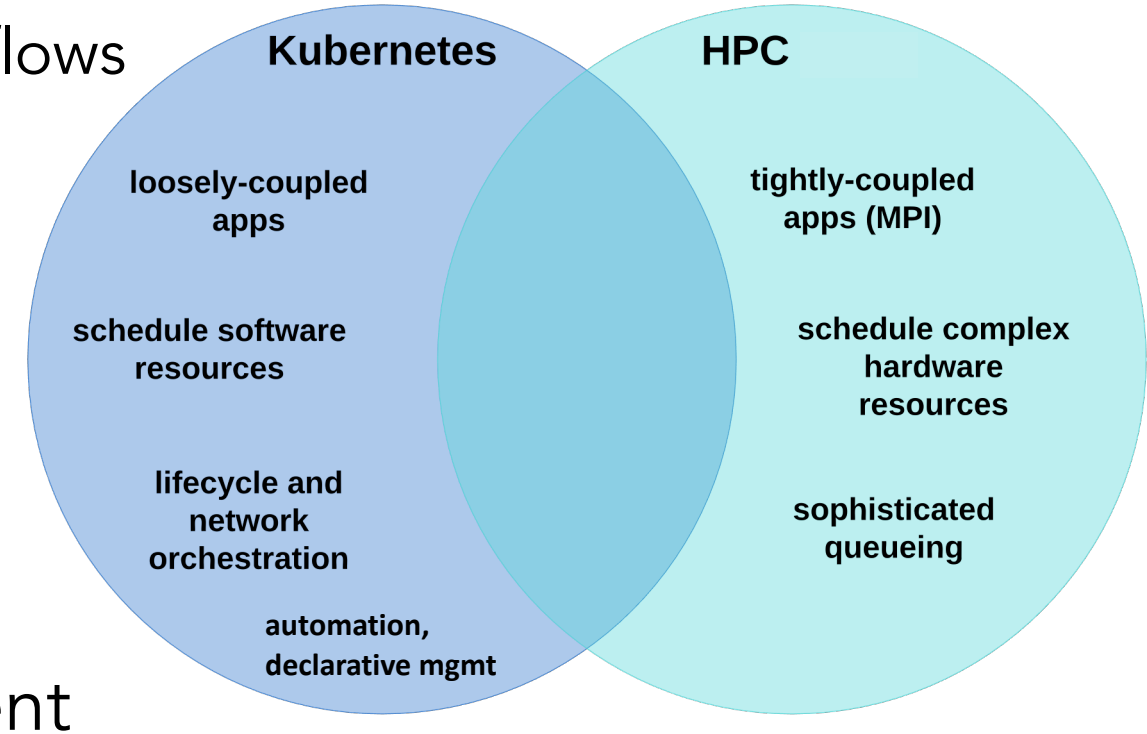


- Co-scheduling
- Job throughput
- Job communication/coordination
- Portability
- Extremely heterogenous resources

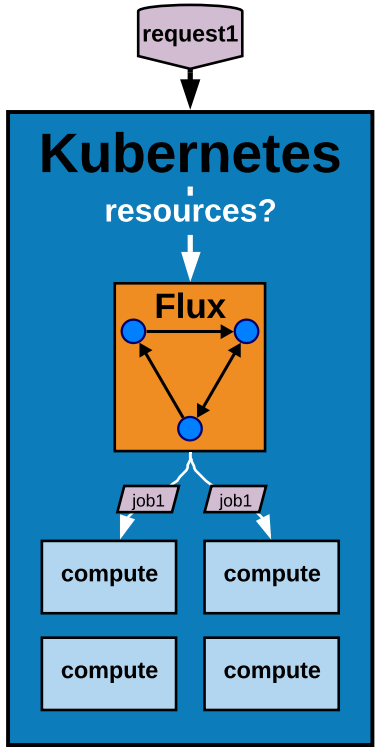
HPC Management and Scheduling and Kubernetes (K8s) are complementary technologies

- HPC limitations:
  - becoming very difficult to manage workflows
  - not designed for dynamism/elasticity
- K8s limitations:
  - designed for loosely coupled apps
  - not focused on performance (scheduling limitations, resource expression, throughput…)

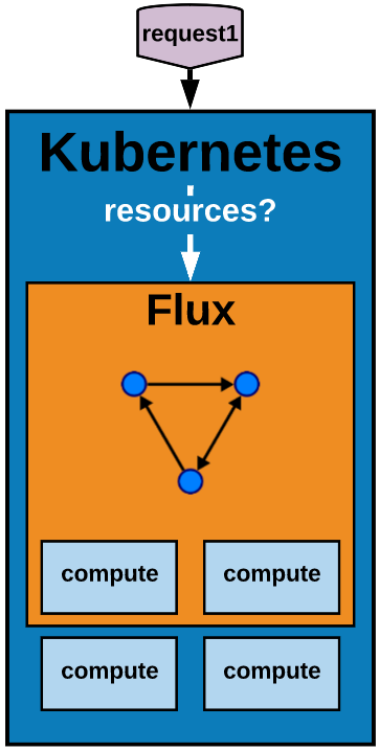
How to create a converged environment that combines the best of both worlds?



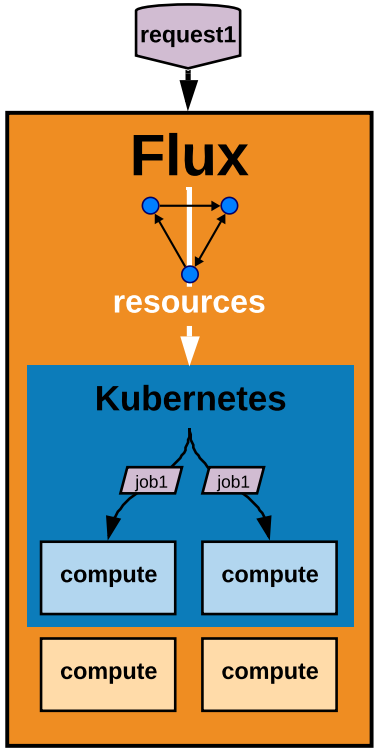
We are contributing to convergence by implementing Flux-based models.



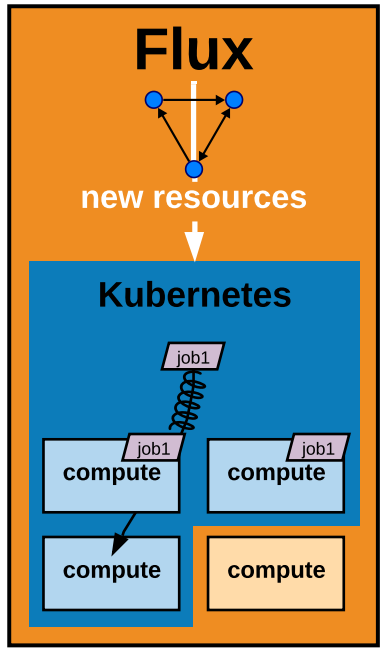
L1: standard APIs, resource representations, sched plugin. **HPC nesting model enables portable converged workflows**



L1.5: Flux manages, schedules cloud. **(portable converged workflows)**



L2: resource co-management (requires L1). **Cloud nesting model reduces software complexity, increases automation, performance**

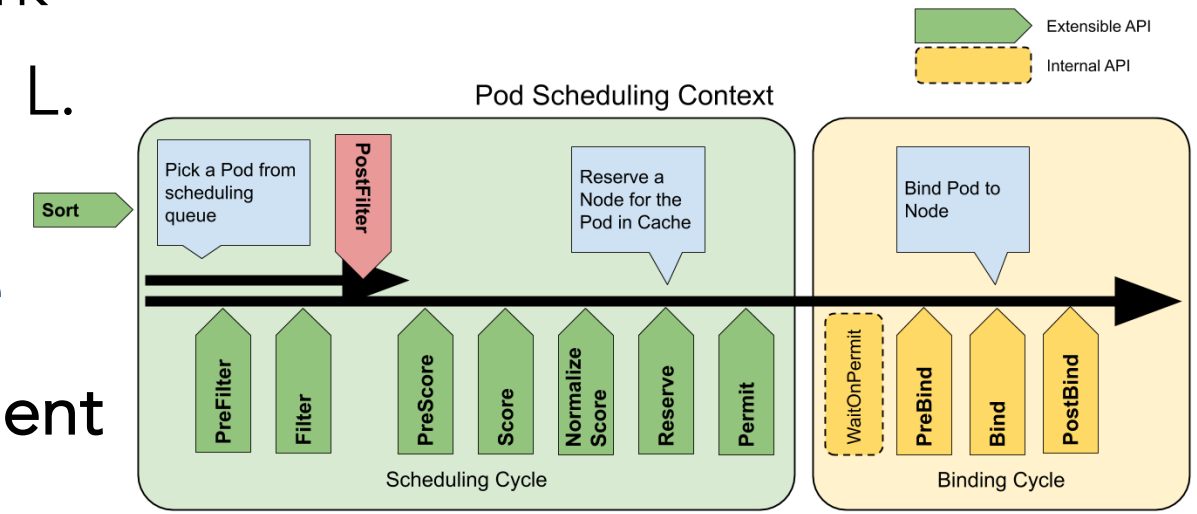


L3: elastic resource co-management (L1+L2) **Elastic cloud nesting model enables autoscaling and dynamism**

Team formed industry collaborations to tackle converged computing challenges and contribute to community.

We bring complementary backgrounds in HPC, cloud, and performance-oriented orchestration

- LLNL LDRD team
- **IBM T.J. Watson Research Center**: T. Elengikal, M. Drocco, C. Misale, Y. Park
- **AWS**: E. Bollig, M. Hugues, H. Poxon, L. Wofford
- Integrate Fluxion into K8s via Fluence
- Declarative, automated Flux deployment in K8s via Flux Operator



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

More placement control and functionality

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

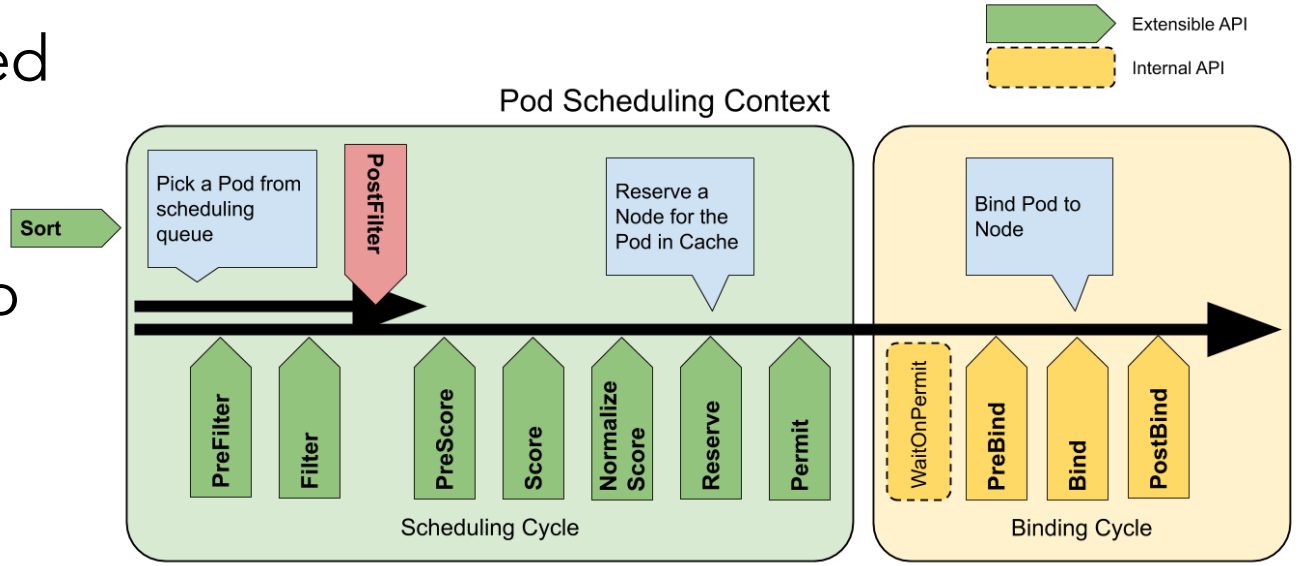
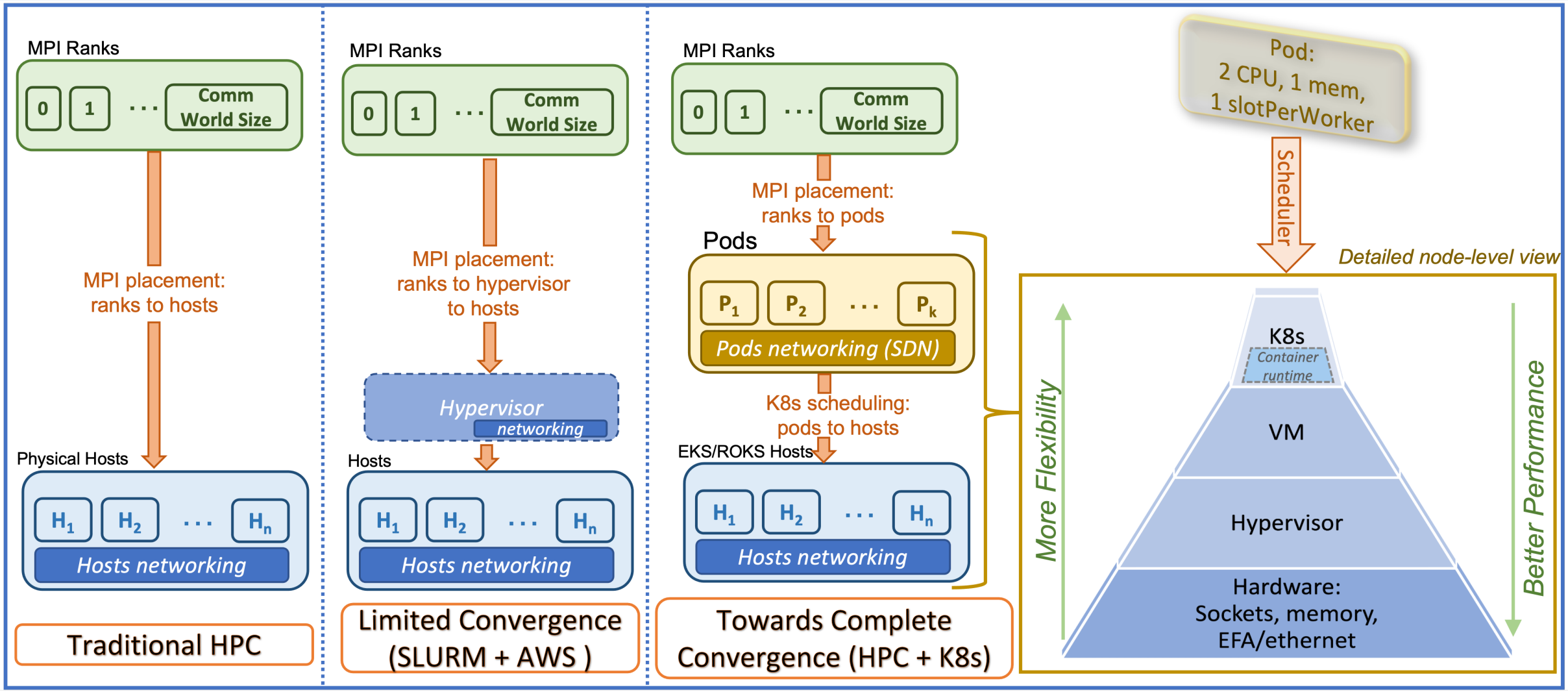


image: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

Easier deployment

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

L1-1.5: creating HPC cloud "slice" is complicated and requires scalable MPI bootstrap and pod placement.



We enabled cloud-native, declarative MPI to scale three orders of magnitude higher.

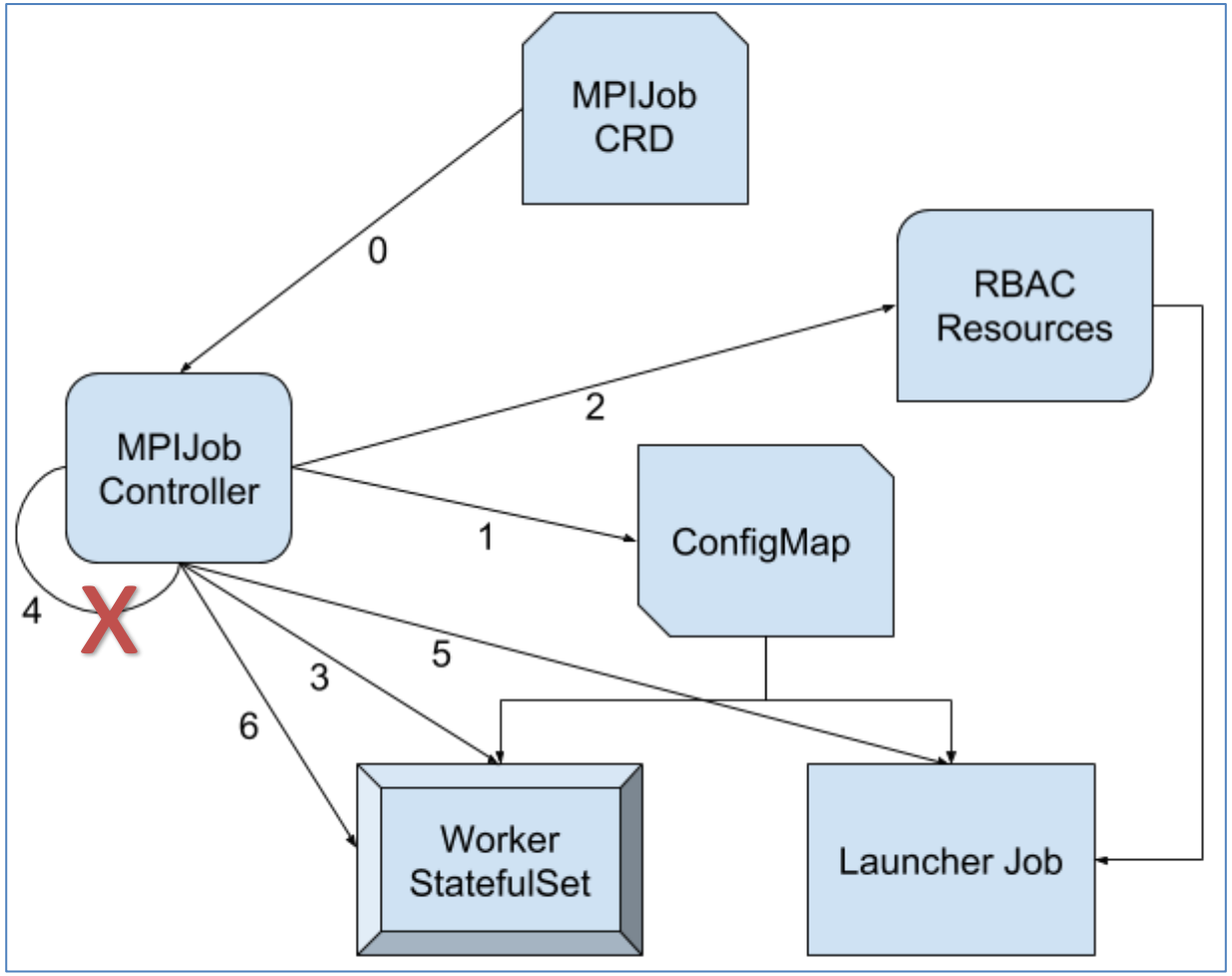


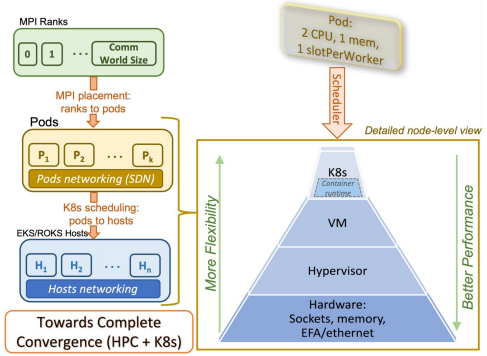
image source Kubeflow: https://medium.com/kubeflow/introduction-to-kubeflow-mpi-operator-and-industry-adoption-296d5f2e6edc

KubeFlow MPI Operator created to manage MPI Jobs; only scaled to around 80 ranks

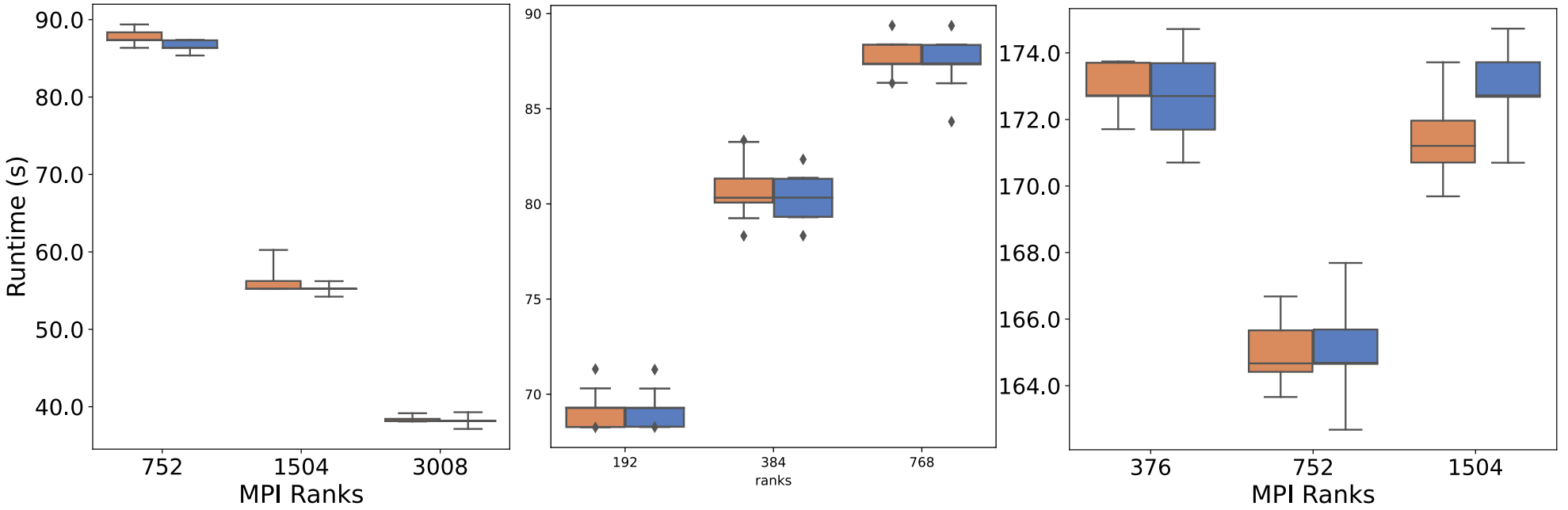
- Race conditions in Controller (steps 1-4): Launcher doesn't wait for Workers, Workers report **`READY`** before **ssh** available
- Fixed DNS flood

Tested fixed MPI Operator with *Hello World* at 16,384 ranks on AWS EKS

We ran CORAL-2 benchmarks on AWS Elastic Kubernetes Service with the improved MPI Operator.



Control pod shape via resource request, MPI shape via ***slotsPerWorker***, ***mpirun*** options



LAMMPS AMG QMCPACK

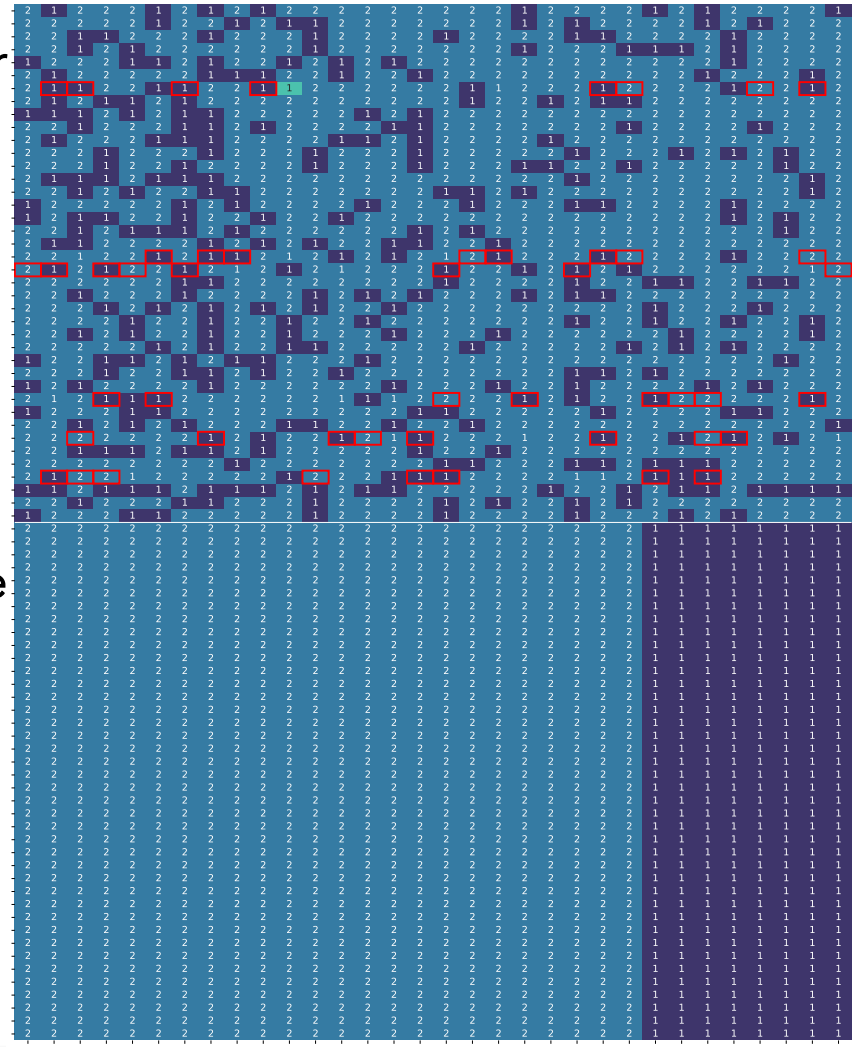
32 Hpc6a instances with 94 usable CPUs, 360 GB memory, Elastic Fabric Adapter (EFA)

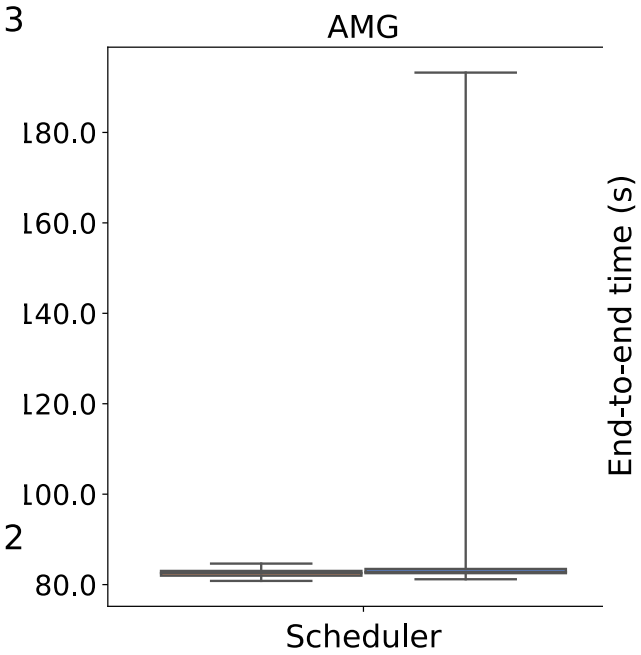
- LAMMPS strong scales 94 ranks/node with EFA up to 3008 ranks
- AMG exhibits decent weak scaling 24 ranks/node with and without EFA
- QMCPACK exhibits inverse (!) scalability after 8 nodes

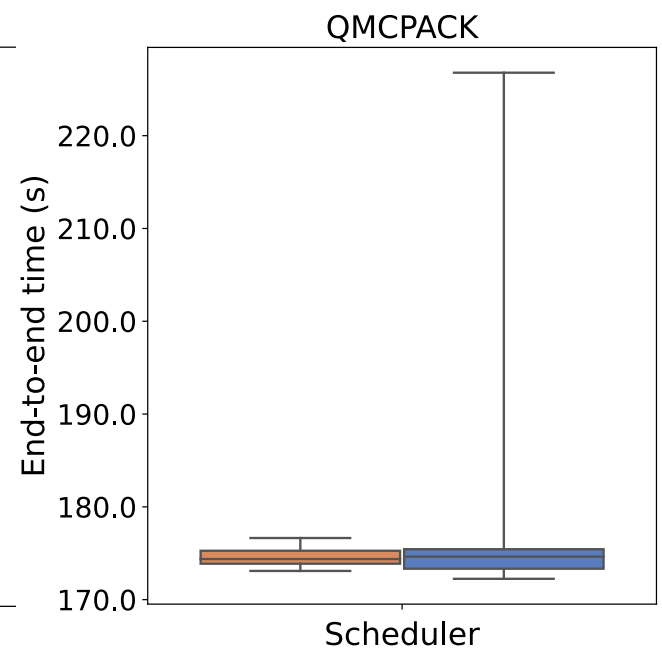
We compared multi-app simulated workflow performance scheduled by Fluence vs. Kube-scheduler.

Kube-scheduler

Fluence

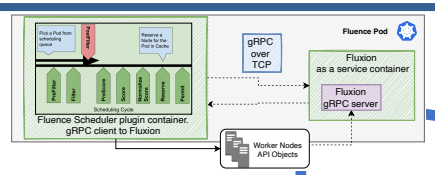




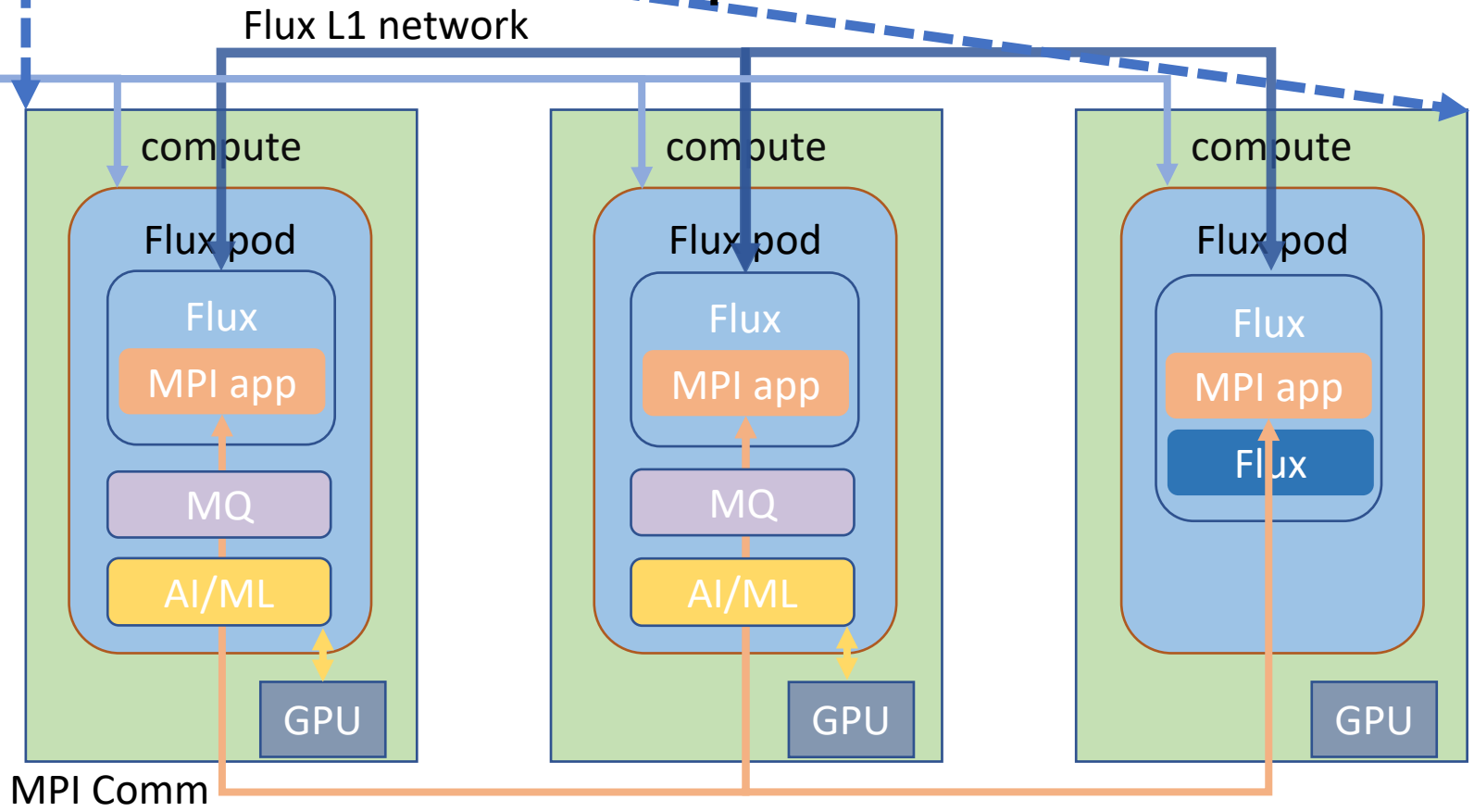


Fluence-scheduled workflows run up to 3x faster with low variability, deterministic placement

L1.5: Fluence + Flux operator will enable hierarchical management with HPC-grade pod scheduling.



Flux Operator replaces MPI Operator for our workflows



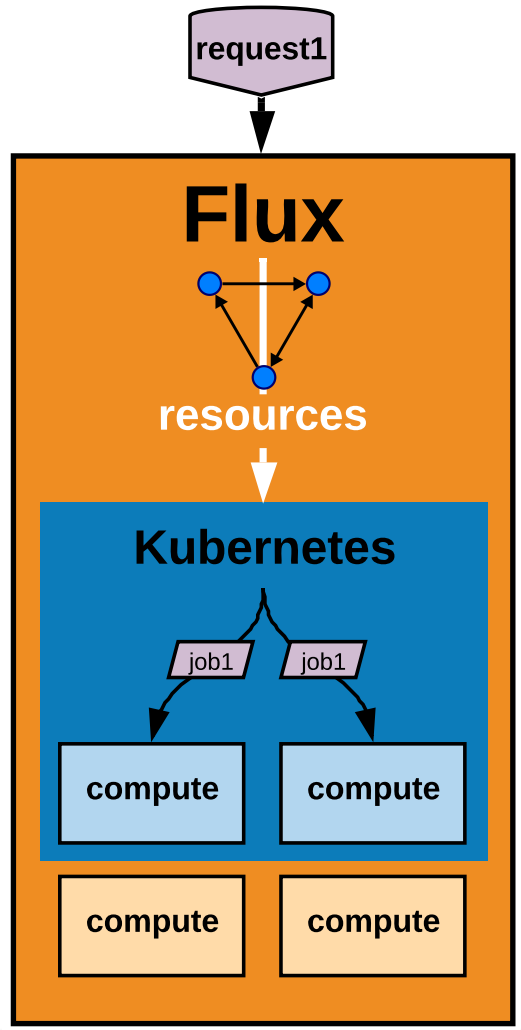
Next up: port **AHA MoleS, etc.** to K8s

L2: Develop an effective and efficient resource co-management framework

Challenges:

- which manager has authority over resources; how?
- how to minimize/mitigate contention?
- consistency model? (potentially millions of vertices, edges, and thousands of jobs)
- how to deliver high performance?

Benefit: enables tighter coupling between HPC and K8s components, greater automation and reduced complexity via, e.g. Operators



There is still much research to be done to make resource dynamism and elasticity useful as embodied by Flux.

- Flux needs capability to reshape/rebalance its ranks and network
- Automatically resize Fluence resource graph when new K8s resources added; grow and shrink allocations
- Algorithmic enhancements to Fluxion to support elastic jobs
  - Scheduler driven grow/shrink: malleablility
  - Application driven grow/shrink

Thank you!
Questions?

Lawrence Livermore
National Laboratory

DYAD (DYnamic and Asynchronous Data-streamliner)

Jae-Seung Yeom[*], Dong H. Ahn[†], Ian Lumsden[‡], Jakob Luettgau[‡], Silvina Caino-Lores[‡], Michela Taufer[‡]

[*]Lawrence Livermore National Laboratory,
[†]NVIDIA Corporation,
[‡]Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville

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

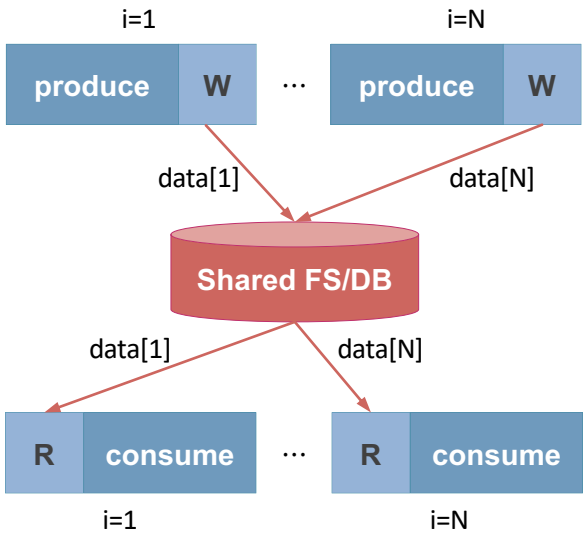
Resolving inter-task data dependence in a workflow

- Existing models
  - *in situ*, *in-transit*, shared file system and so on
  - Varying degree of performance, user productivity, portability and flexibility
- Challenges
  - Lack of synchronization support at the file/data object level
  - Conflict with code change requirements
  - Poor temporal/spatial locality
  - Low file metadata-operation performance

Benefits we aim to bring in

- No/little code change
  - Productivity (Fast construction of workflows with less effort)
  - Portability
    - Data sharing based on universal file abstraction
    - Open-sourced and proprietary applications
    - widely used POSIX IO instead of specific API
- Performance
  - Use of local storage enable faster accesses
  - Fine-grain file level synchronization exposes further parallelism

Producer-consumer patterns are common in modern workflows




```
void producer() {
    for (i=1; I <= N; i++) {
        produce(data[i])
        write(data[i]);
    }
}
```

```
void consumer() {
    for (i=1; I <= N;i++) {
        read(data[i]);
        consume(data[i])
    }
}
```

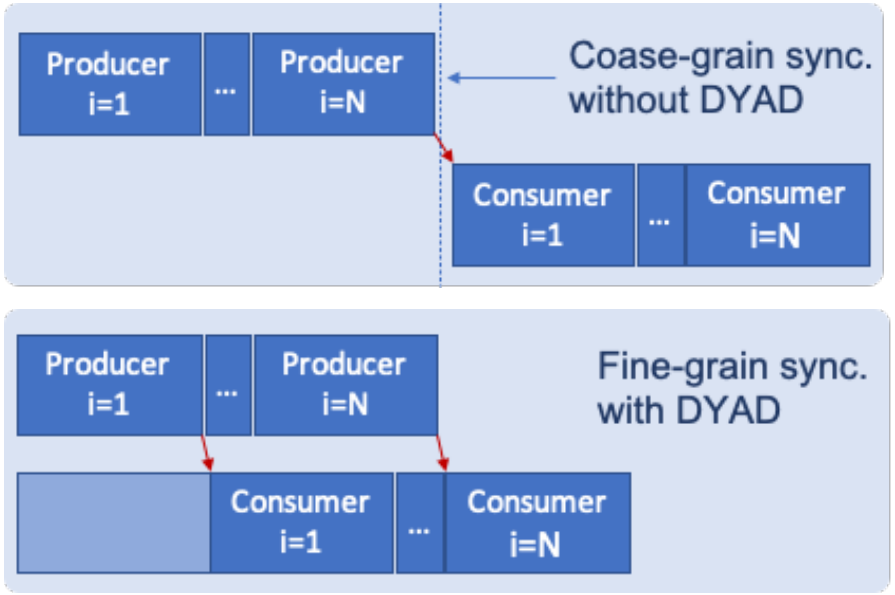
(a) Producer-consumer example

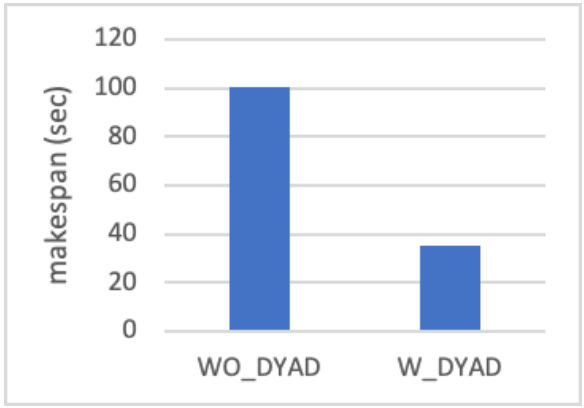
```
- description: producer task
  name: run-producer
  run:
      cmd: producer

- description: consumer task
  name: run-consumer
  run:
      cmd: consumer
      depends: [run-producer]
```

(b) Maestro Specification in YAML

Fine-grain synchronization with DYAD

```
producer() {
    for (i=1; i<=N; i++) {
        produce(data[i])
        write(data[i]);
    }
}
consumer() {
    for (i=1; i<=N;i++) {
        read(data[i]);
        consume(data[i])
    }
}
```



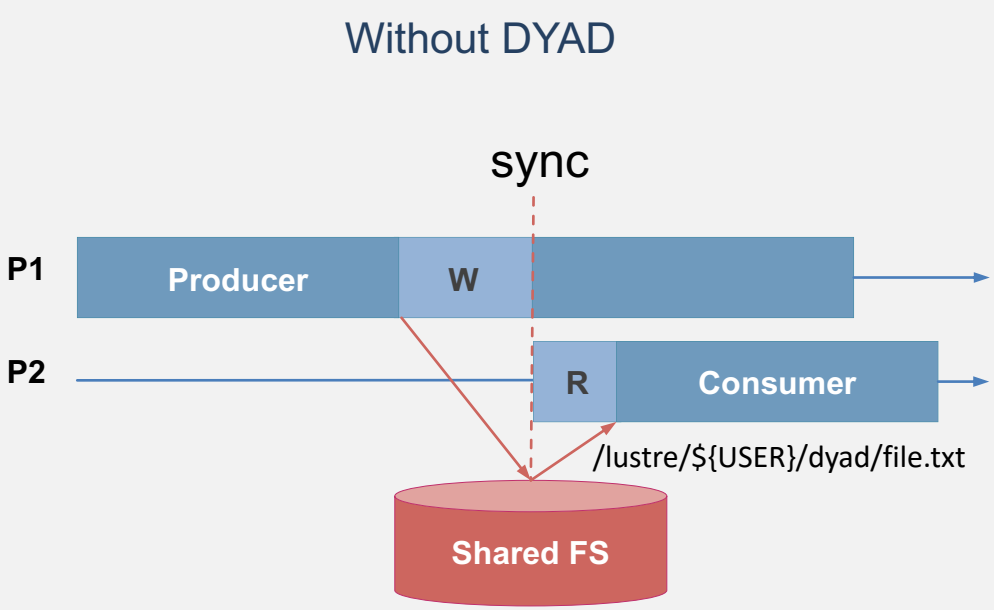


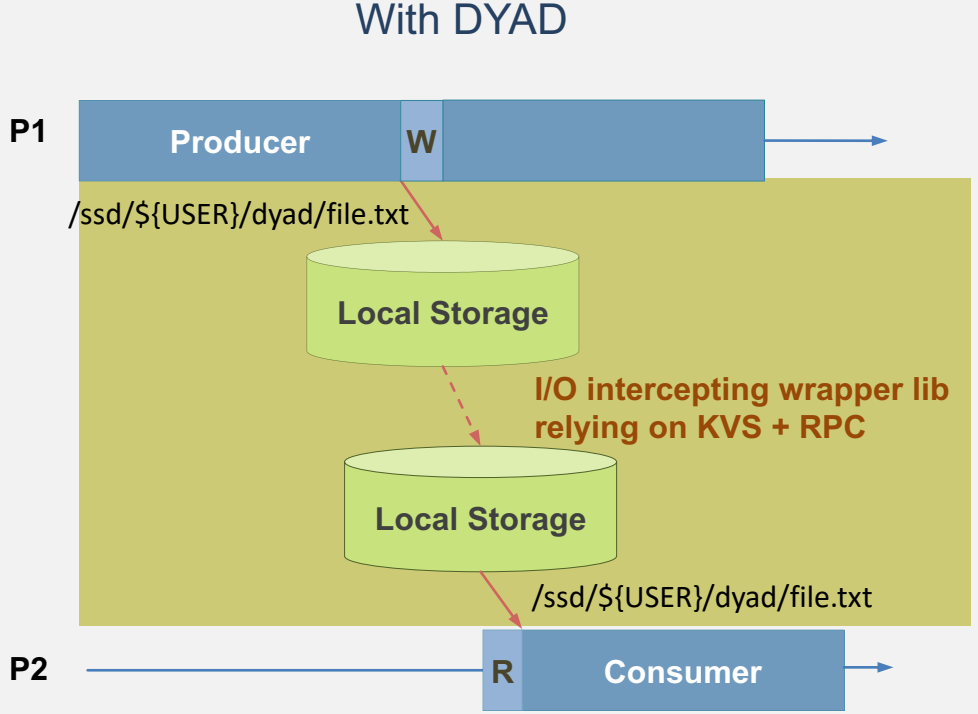
- Explore the effectiveness of the scheduling to minimize the temporal and the spatial distance between producers and consumers

DYAD operation

Explicit synchronization between read and write
- e.g., dependent jobs, API, or polling

- Relies on local storages
- Transparent data transfer between storages and synchronization per shared file





Minimal example

- Running DYAD service with FLUX

  flux exec -r all flux module load dyad.so /ssd/${USER}/dyad

```
flux exec -r all flux module load dyad.so /ssd/${USER}/dyad
```

- Configure DYAD via env variables

```
DYAD_KVS_NAMESPACE
DYAD_PATH_PRODUCER
DYAD_PATH_CONSUMER
DYAD_SHARED_STORAGE
```

- Running user app written in C with DYAD

```
LD_PRELOAD=libdyad_sync.so:${LD_PRELOAD} ./app
```

- Using DYAD stream library in C++

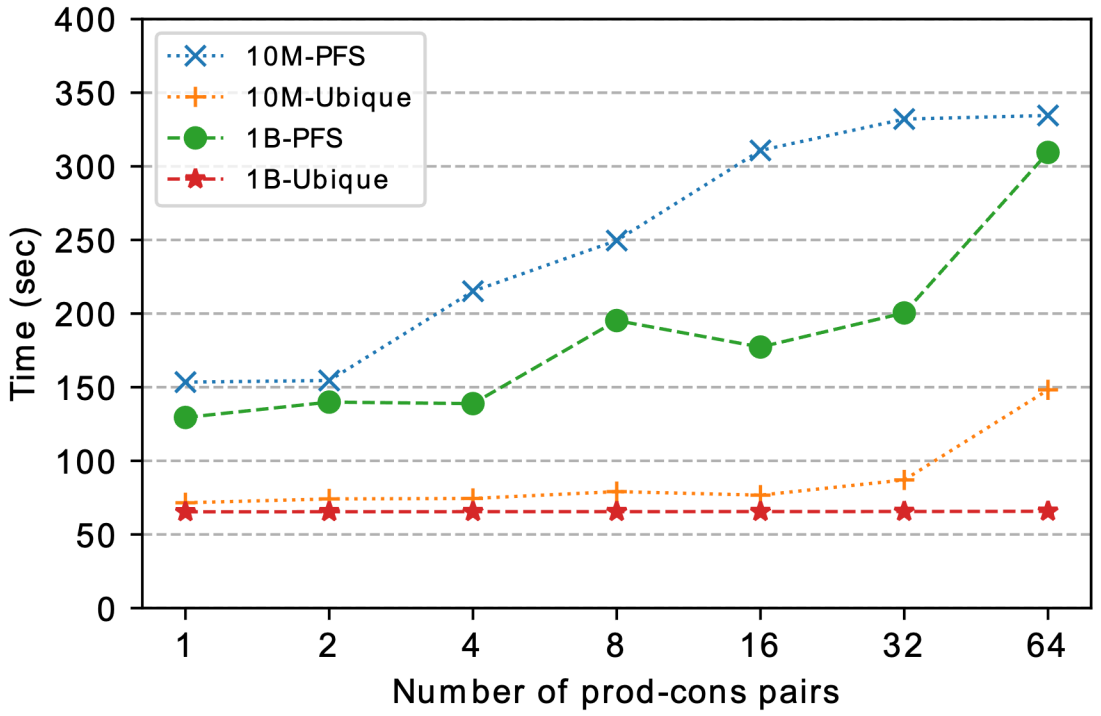
```
#include <dyad_stream_api.hpp>

void producer(dyad::dyad_stream_core& dyad)
{
   dyad::ofstream_dyad ofs_dyad;
   ofs_dyad.init(dyad);
   ofs_dyad.open("test.txt");
   std::ofstream& ofs = ofs_dyad.get_stream();
   ofs << "test" << std::endl;
   ofs_dyad.close();
}

void consumer(dyad::dyad_stream_core& dyad)
{
   dyad::ifstream_dyad ifs_dyad;
   ifs_dyad.init(dyad);
   ifs_dyad.open("test.txt");
   std::ifstream& ifs = ifs_dyad.get_stream();
   std::string line;
   ifs >> line;
   std::cout << line << std::endl;
   ifs_dyad.close();
}
```

```
std::ofstream ofs;
ofs.open("test.txt");
```

DYAD: DYnamic and Asynchronous Data-streamliner

- Handles producer-consumer I/O pattern in a workflow
- Take advantage of local storage
  - Transfer data transfer directly from producers to consumers
- Fine-grain synchronization
  - At file open/close granularity
- Enable dynamic sharing (dynamic dependency)
- Flux API-based tool (KVS, RPC)

Performance of producer-consumer benchmark



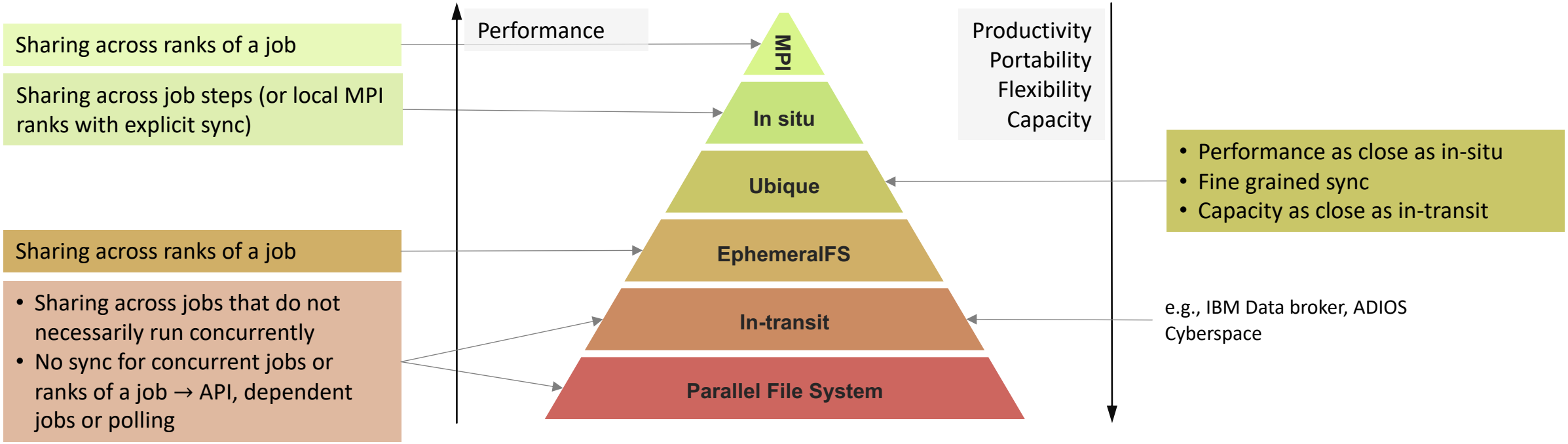

```
void producer() {
    for (i=1; I <= N; i++) {
        produce(data[i])
        write(data[i]);
    }
}
```

```
void consumer() {
    for (i=1; I <= N;i++) {
        read(data[i]);
        consume(data[i])
    }
}
```

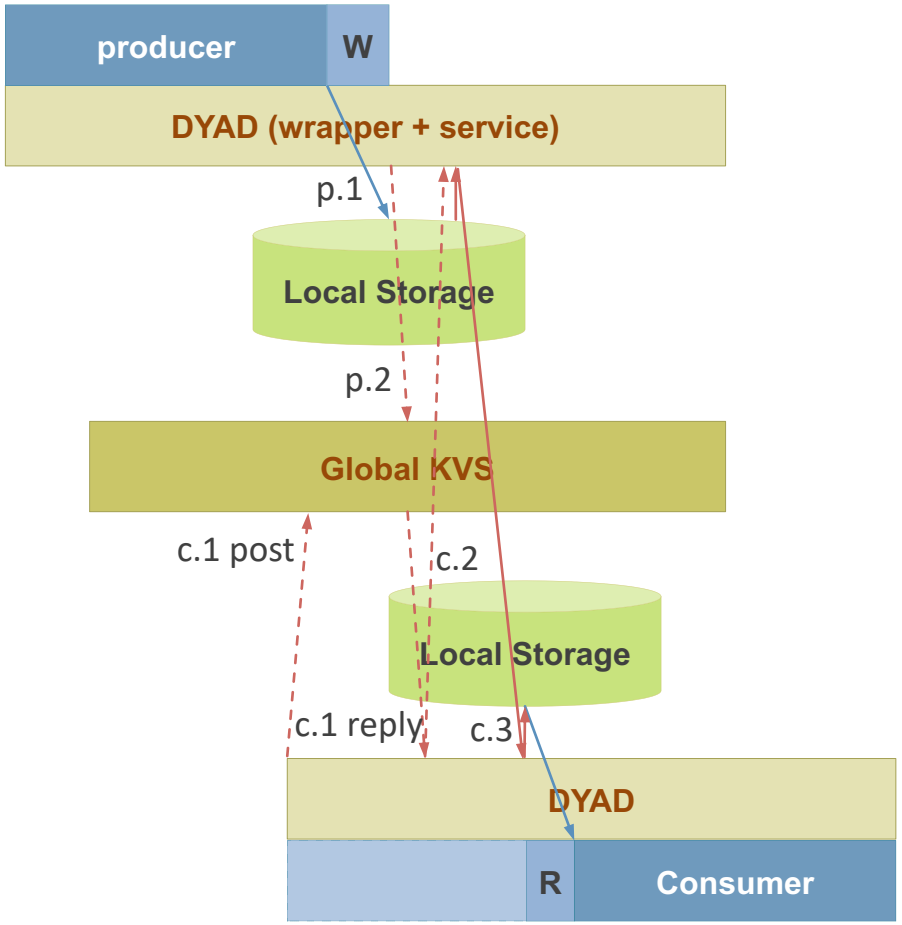
File sharing by DYAD vs parallel file system (Lustre/quartz)

Design trade-off space for data transfer model

DYAD is an embodiment of Ubique model based on FLUX API



Synchronization and file transfer



DYAD

- DYAD module (or service) runs on each node.
- DYAD wrapper only intercepts I/O for files on the directory it manages
  - If managed dir is on local storage (LS), sync accesses and transfer files
  - If it is on shared storage, only sync accesses.

Producer

- p.1 *write(manged_dir/filename)*
- p.2 *publish(<filename, prod_rank>)*
  - If a file is written into ***managed_dir*** or its subdirectory, DYAD registers the filename into the global key-value-store (KVS) managed by FLUX
  - KVS entry is a pair of ***filename*** and ***rank***, where rank is the FLUX rank which owns the file and sees it locally

Consumer

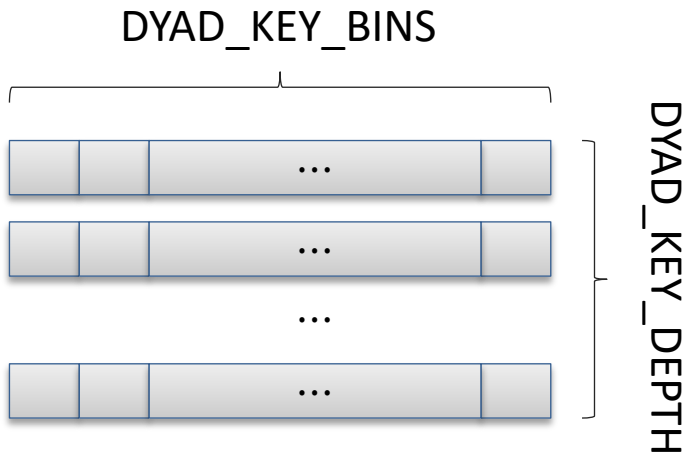
- c.1 *query(filename)→ prod_rank*
  - Consumer queries KVS to obtain the rank of the file owner (producer). Then, blocking wait.
- c.2 *rpc_get(prod_rank, filename)*
  - Ask the owner rank to transfer the file. Once received, store it on LS
- c.3 *read(managed_dir/filename)*

Scaling the performance of key value store (KVS)

- KVS
  - FLUX module (service)
  - The larger the number of keys, the slower it gets to complete a search/update
- Namespace
  - Within a namespace, users can get/put KVS values completely independent of other KVS namespaces.
  - DYAD generates queries under the namespace given by DYAD_KVS_NAMESPACE env variable
  - Needs to execute `flux kvs namespace create ${DYAD_KVS_NAMESPACE}` in advance
- Hierarchy
  - DYAD creates a hierarchical key string for a given filename
  - env variables DYAD_KEY_DEPTH and DYAD_KEY_BINS to control
  - Murmur3 hash vectorized for 128 bit
- Remove keys that are no longer used
  - `flux kvs namespace remove ${DYAD_KVS_NAMESPACE}`

$$key \leftarrow bin_1.bin_2.\cdots.bin_d.filename$$

$$bin_1 \leftarrow hash(filename, seed_1)$$
$$bin_2 \leftarrow hash(filename, seed_2)$$
$$\cdots$$
$$bin_d \leftarrow hash(filename, seed_d)$$



Future work

- Generalize the supporting environment
  - Leveraging external data transport layers
    - for HPC and Cloud
  - Leveraging external meta-data management
- Taking advantage of deep storage hierarchy
- Data-locality-aware scheduling
- Data lifespan and ownership management

AHA MoleS: Accelerate Drug Discovery by American Heart Association Molecular Screening Workflow

Xiaohua Zhang
BBTD, PLS, LLNL

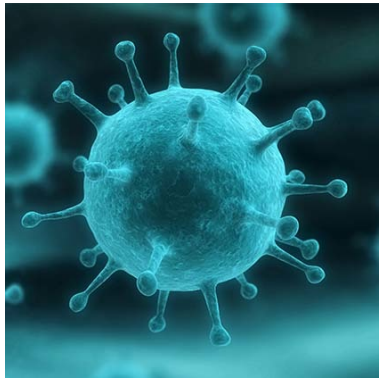
LLNL-PRES-844974

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

We need to be prepared for new emerging threats

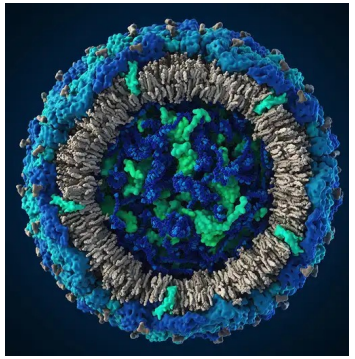
HIV



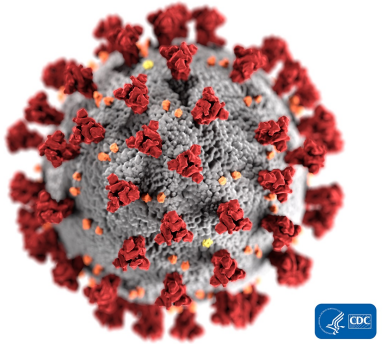
Superbug

Ebola

Zika



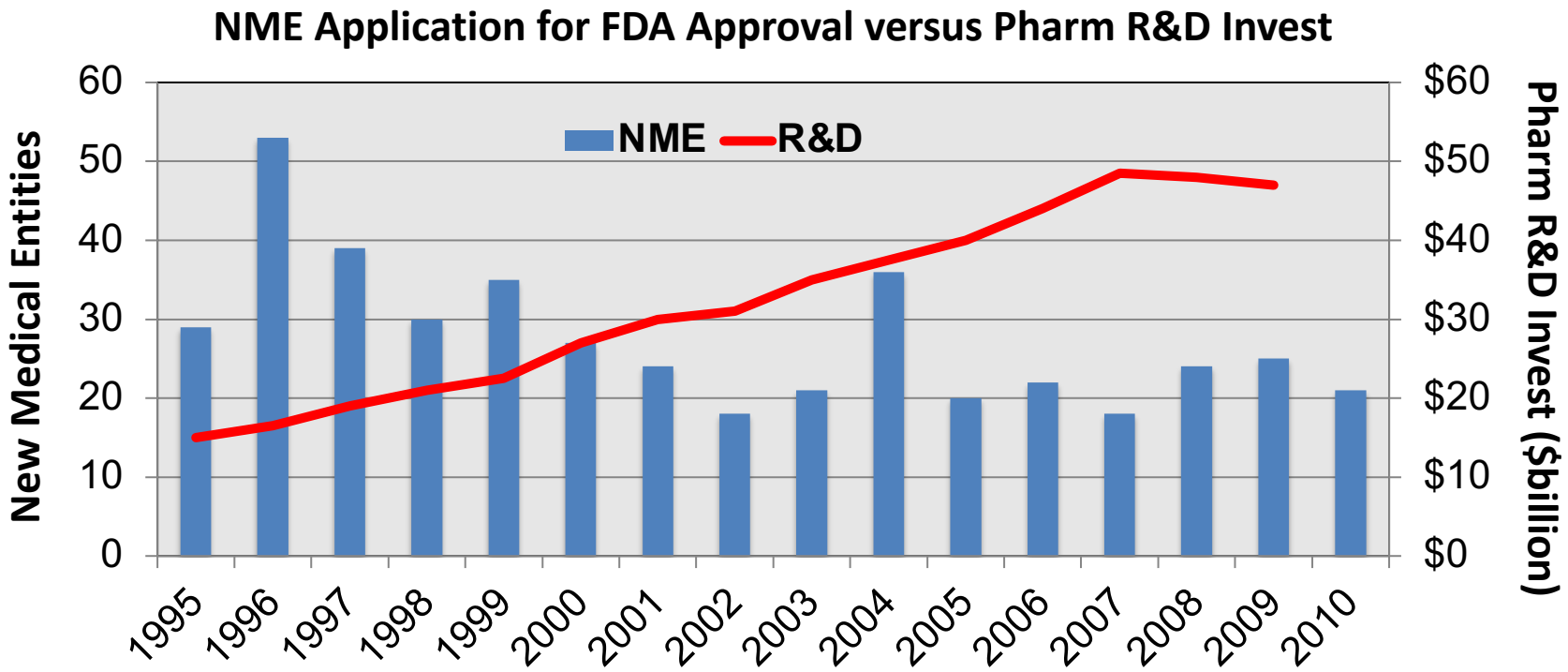
Covid-19



Next?

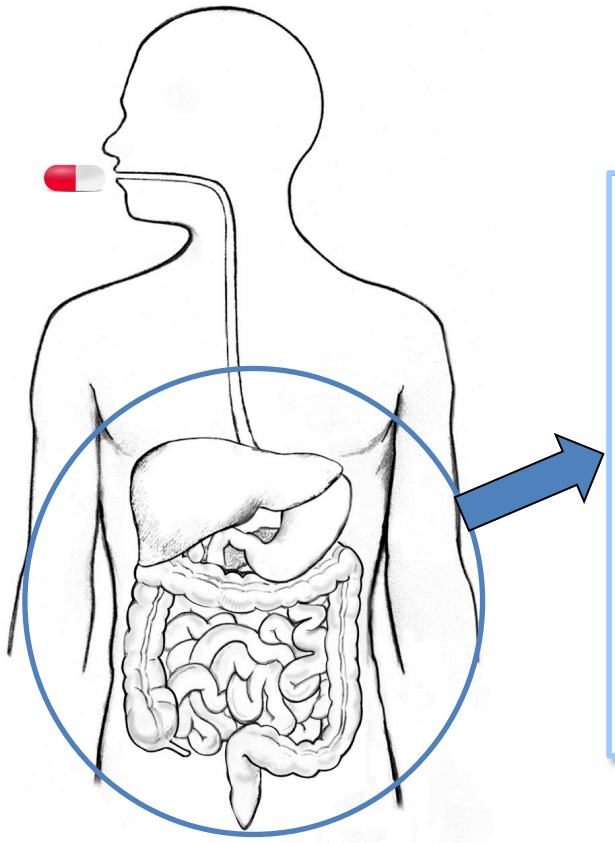
Urgent need for better *in-silico* prediction tools to reduce the cost and failure rate of drug discovery

- 7-15 years and > 1 Billion USD per drug
- >90% failure at late stage. 42% of approved drugs (25/59) in 2018 were based on a single pivotal trial.
- Post-market failure due to adverse drug effect (e.g. Vioxx, Lipobay).

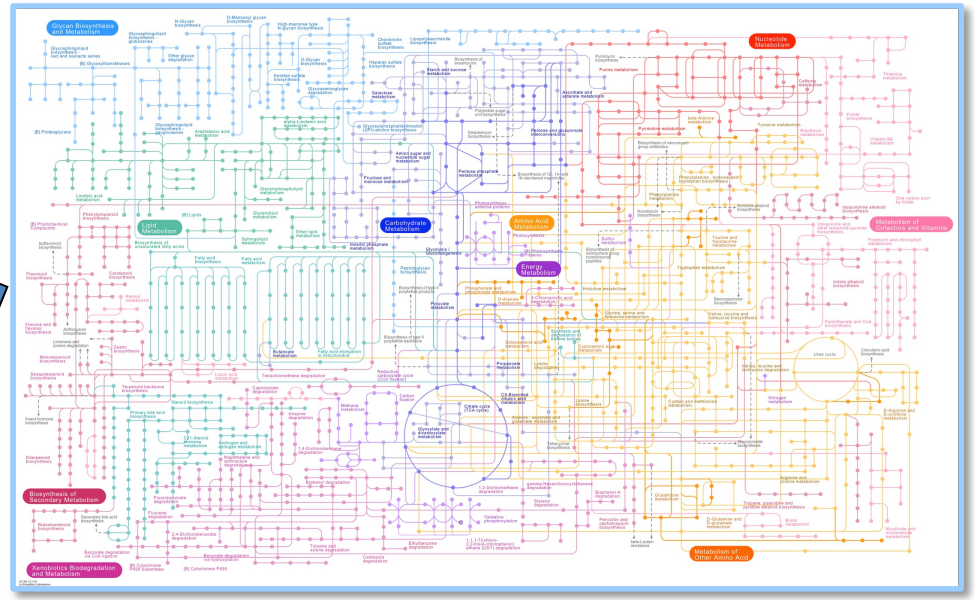


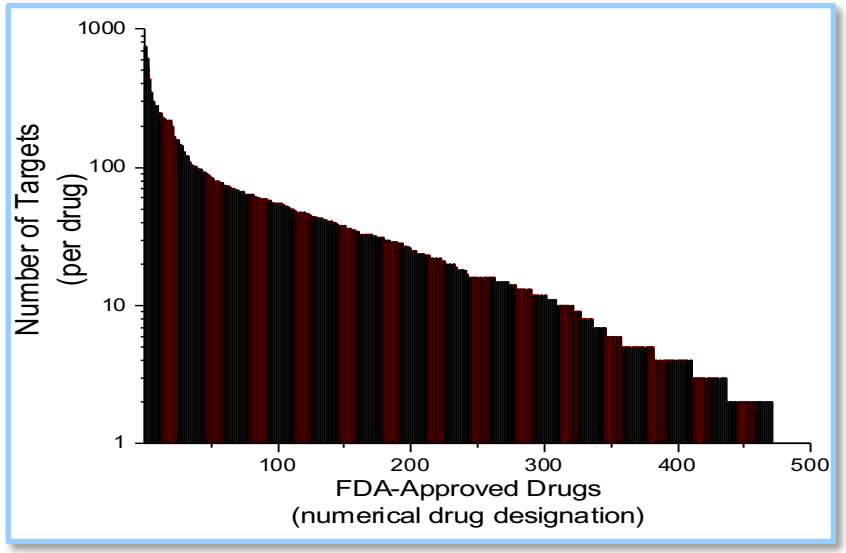
Update: 2010 – 2021 NME per year. 21, 30, 39, 27, 41, 45, 22, 46, 59, 48, 53, 50

Drug on and off target binding along its metabolic pathway



Metabolic Pathways



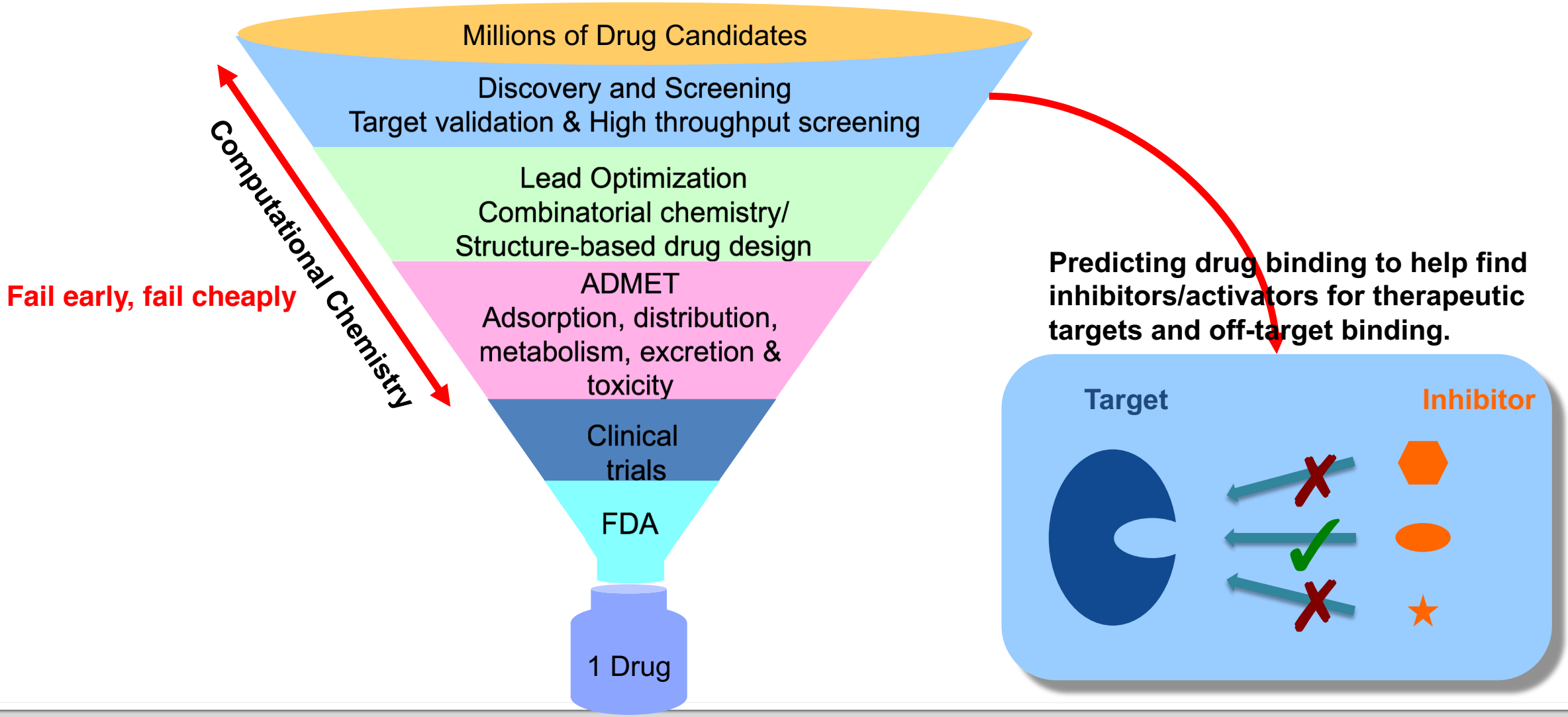


Drug could hit unexpected off-targets and cause severe side effects

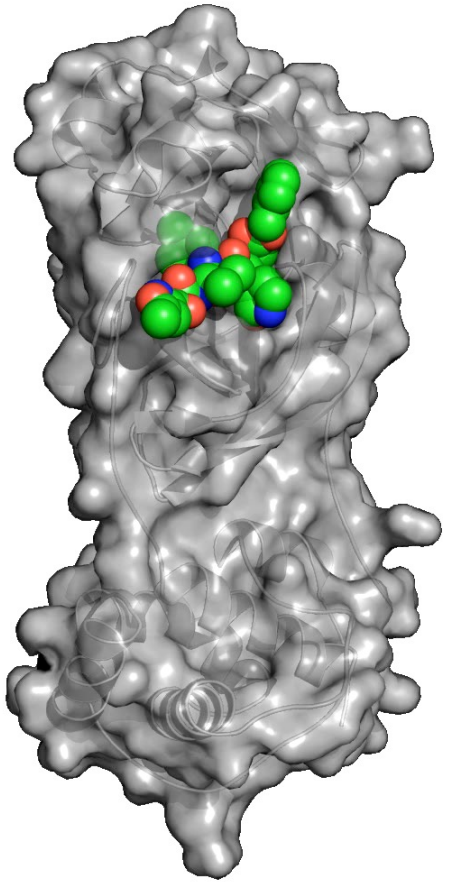
Average: 32.4 targets/drug

Chartier et al. *BMC Pharmacol. Toxicol.* 2017, 18, 18

What can computational chemistry do for drug design?



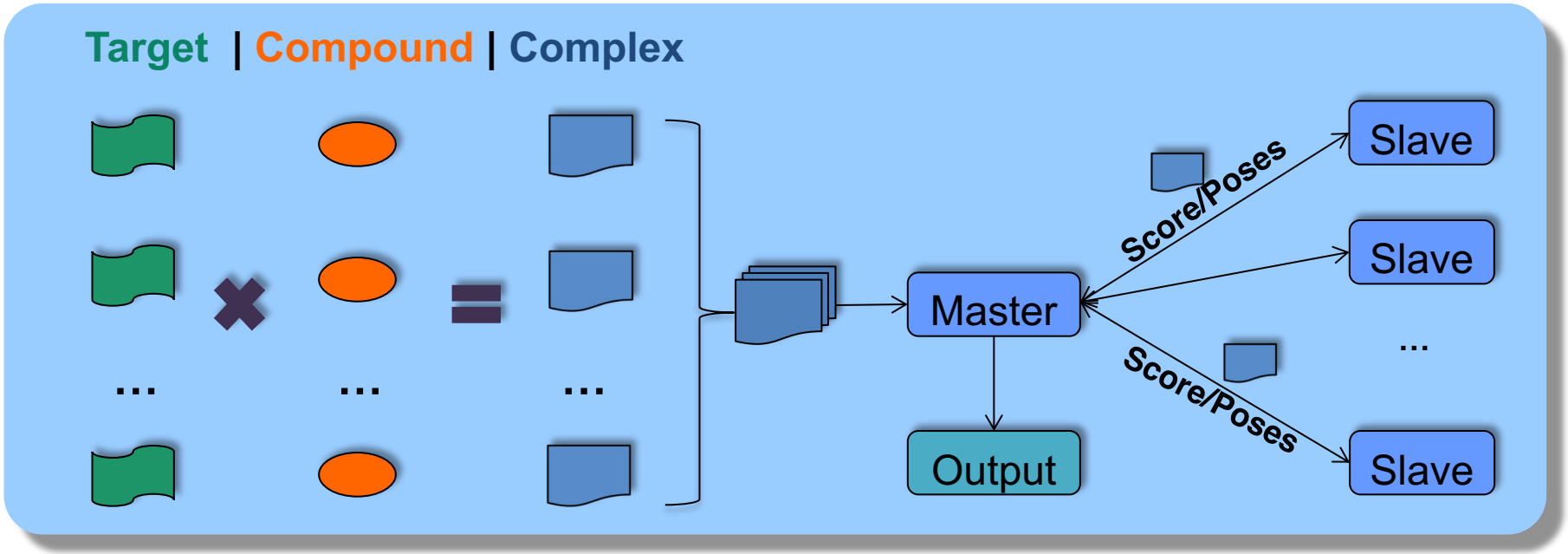
All drugs must bind their protein target



Predict the protein and small molecule binding affinity

- Physical based prediction
  - Molecular Docking
  - Free energy (MM/GBSA) rescoring
- Machine learning based prediction
  - Fusion model
- Docking is required to obtain the complex structure, which will be used in MM/GBSA rescoring and fusion calculation

ConveyorLC pipeline for physical-based binding prediction

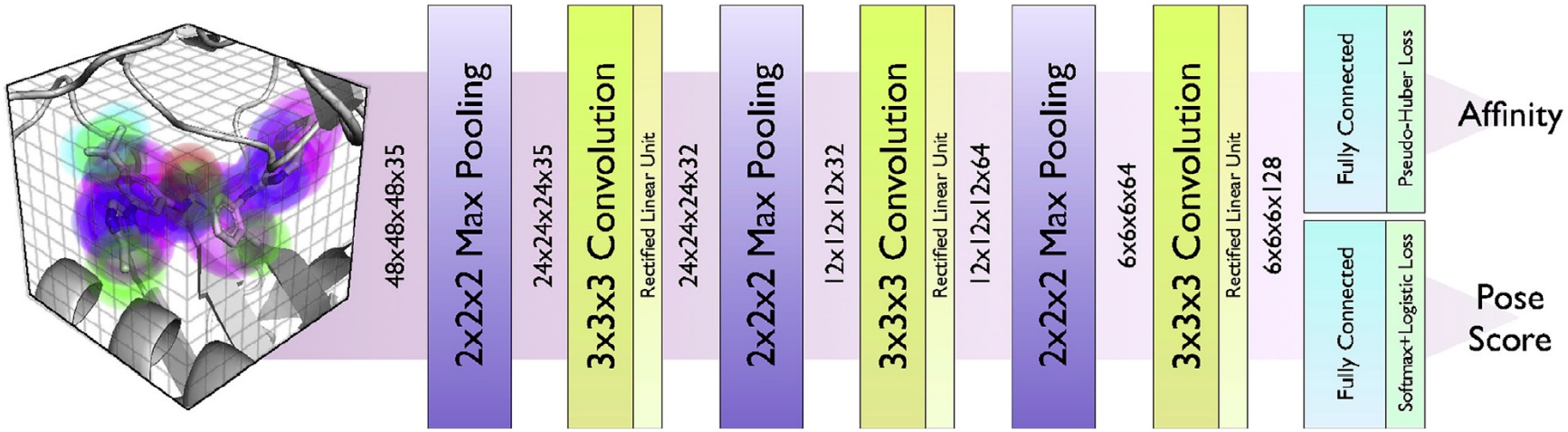


HPC:	~10 min @ 1 CPU	~1 hr @ 100 CPU	~1 hr @ 700 CPU	~5 hr @ 15K CPU
PC:	~10 min	~4 day	~1 month	~10 year

An example for a target (~4K atoms) with ~40K ligands

Binding affinity prediction by 3D convolutional neural network

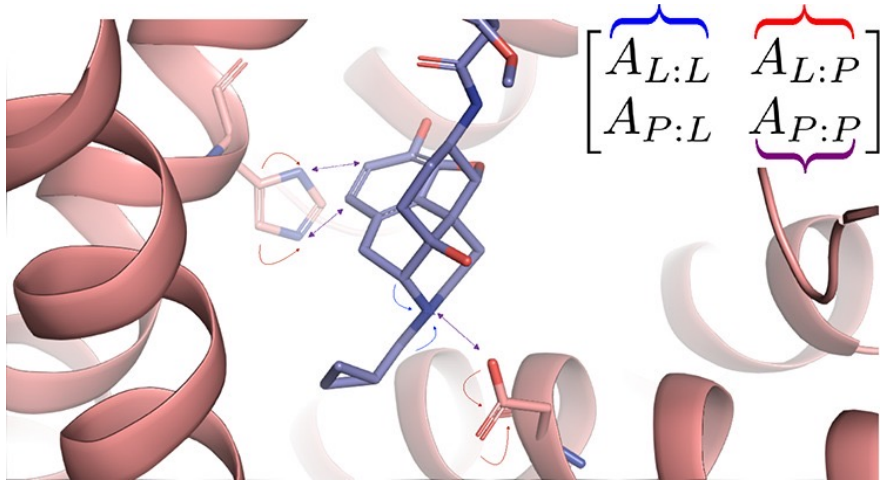
- 3D-CNN discretizes 3D space into 3D-voxels, apply convolution operations to progressively down sample feature maps
- Select features including element type, hybridization, number of heavy atom bonds, partial charges, van der Waals, hydrophobic, ring, donor/acceptor, etc.
- Training on the PDBBind general set and use the Ki and KD values only (13K crystal structures)



Hochuli, J. et. al. *J. Mol. Graph. & Model.* 2018 *84*, 96-108.

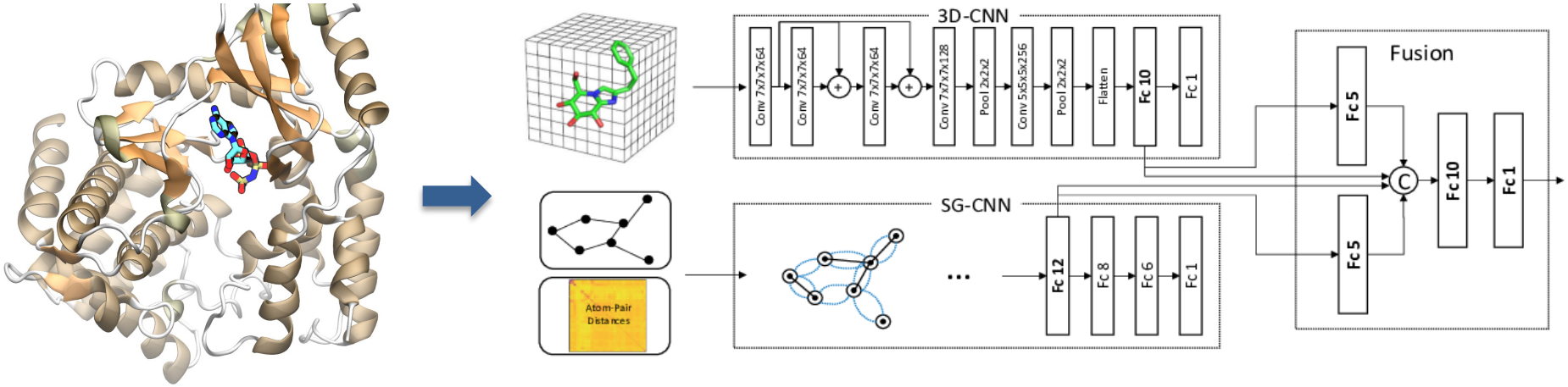
Binding affinity prediction by spatial graphs convolutional neural network

- SG-CNN computes distance matrices for atoms in ligand (L:L), protein (P:P), and complex (P:L,:L:P)
- Form graph where nodes are atoms and edges are the covalent/nonbonded interactions, label edges with distances



Feinberg EN, et al. *ACS Cent Sci*. 2018, 4, 1520-1530

Combine 3D-CNN and SG-CNN for better prediction



Performance on PDBBind Core Set — Docking Poses (Not Xray Structures)

Model	Pearson r	Spearman r	MAE	RMSE
3D-CNN	0.523	0.503	1.843	2.358
SG-CNN	0.656	0.635	1.343	**1.649**
Vina	0.616	0.618	-	-
MM/GBSA	0.629	0.641	-	-
Mid Fusion	0.677	0.647	1.351	1.715
Fusion	**0.685**	**0.668**	**1.340**	1.701

Use the **second-** and **fourth-last** layer's output activations from 3D-CNN and SG-CNN, respectively

Jones, D. et. al. *J. Chem. Inf. & Model.* 2021, 61, 1583-1592

We are creating a queryable protein-small molecule atlas

A composite science workflow (AHA MoleS) is developed for the docking and fusion calculations

Human Proteins: 28,658 models for 14,813 proteins

Experimental structures (>35%): 8,283

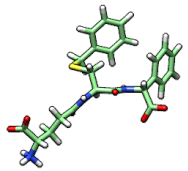
Experimental structures (>10%): 12,894

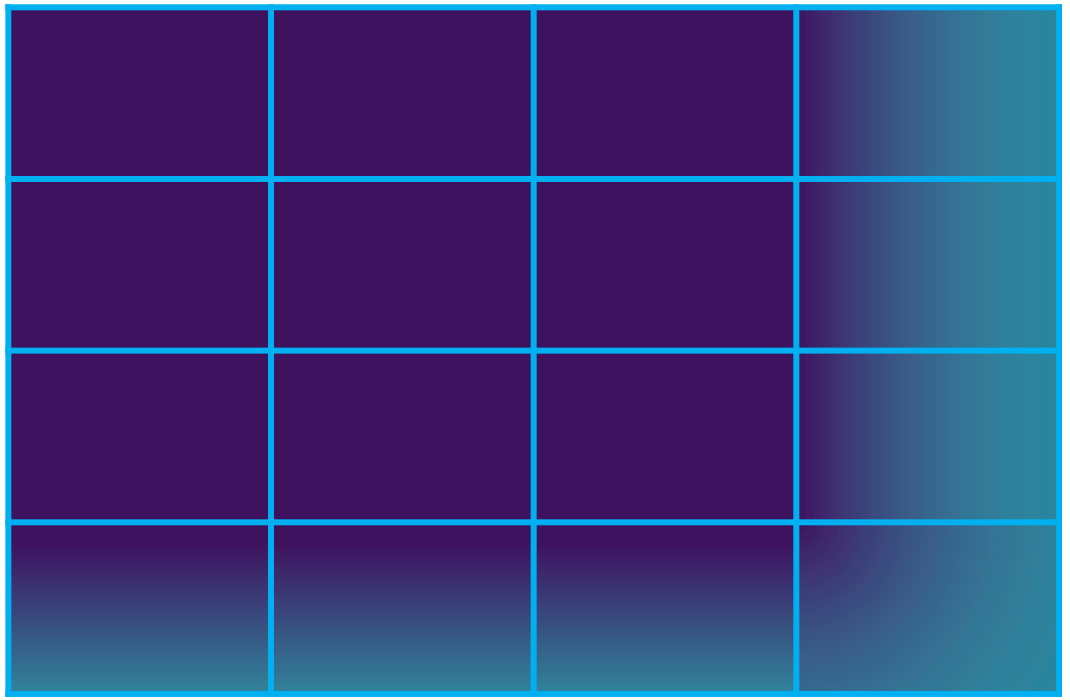
M

Small Molecule Compounds: ~2.1M

ChEMBL28:	**~1.9M**
Broad Drugs:	**6550**
Foodome:	**24,114**
NCI60:	**244,800**
PDBBind:	**14,744**

N

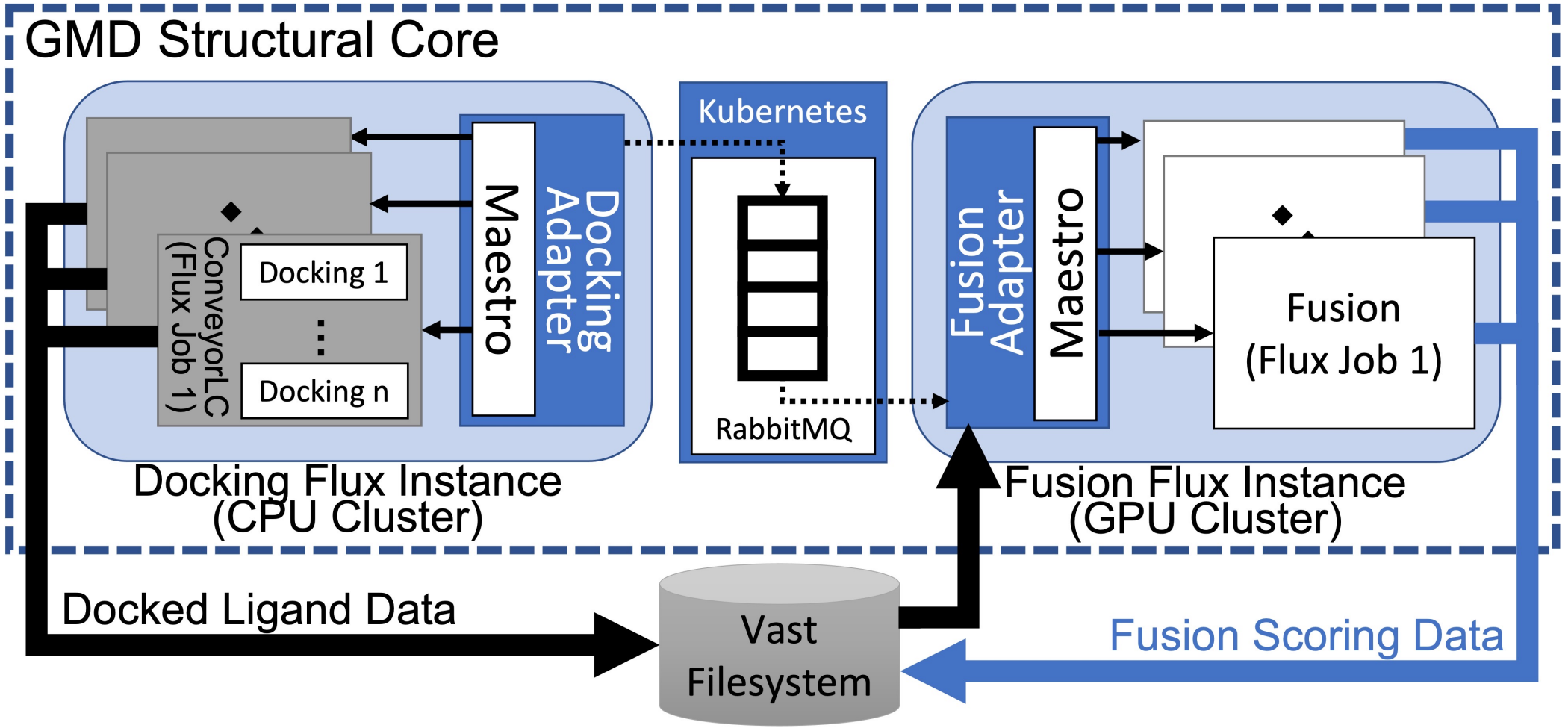




Total docking calculations: ~30 B

Total fusion calculstions: ~300 B

AHA MoleS workflow architecture



Demo of CPU only workflow

Beginning of workflow

```
(cpu-workflow-r1) ahashare@ruby964$ fluxj
      JOBID USER     NAME                ST URG NTASKS NNODES  R
    fAx1yUX9 ahashare main                 R 16      1      1  9
    fAJKDjrX ahashare batch_completion     R 16      1      1  1
    f9XghMpT ahashare docking_adapter      R 16      1      1  1
    f8eQiCWj ahashare fusion_adapter       R 16      1      1  1
```

- Running on 24 Ruby nodes.
- Compound batch size: ~1500
- 1 docking adapter
- Each docking job uses 3 nodes running 21 MPI tasks
- 36 pre-fetch docking jobs
- 1 fusion adapter

Launching of docking jobs

```
(cpu-workflow-r1) ahashare@ruby964$ fluxj
      JOBID USER     NAME                ST URG NTASKS NNODES  RUNTIME NODEl
    fGVtBfjd ahashare docking            PD 16     21      -        - -
    fGdomq6X ahashare docking            PD 16     21      -        - -
    fGkMJfEo ahashare docking            PD 16     21      -        - -
    fGtrzY5d ahashare docking            PD 16     21      -        - -
    fH47g46T ahashare docking            PD 16     21      -        - -
    fH6ppjy1 ahashare docking            PD 16     21      -        - -
    fHHjsTJF ahashare docking            PD 16     21      -        - -
    fHHyDLqM ahashare docking            PD 16     21      -        - -
    fHJ2BKQ3 ahashare docking            PD 16     21      -        - -
    fHX2o1qZ ahashare docking            PD 16     21      -        - -
    fHdKVyAP ahashare docking            PD 16     21      -        - -
    fHdgFo7D ahashare docking            PD 16     21      -        - -
    fHsJTCK1 ahashare docking            PD 16     21      -        - -
    fHwPfDQB ahashare docking            PD 16     21      -        - -
    fJ1iD82T ahashare docking            PD 16     21      -        - -
    fJ5BqSMM ahashare docking            PD 16     21      -        - -
    fJBaULoZ ahashare docking            PD 16     21      -        - -
    fJBbxL5u ahashare docking            PD 16     21      -        - -
    fJJtUoQB ahashare docking            PD 16     21      -        - -
    fJQ8QGAs ahashare docking            PD 16     21      -        - -
    fJSKuCQX ahashare docking            PD 16     21      -        - -
    fJSdh3nf ahashare docking            PD 16     21      -        - -
    fJVHsm6X ahashare docking            PD 16     21      -        - -
    fJWYX9uR ahashare docking            PD 16     21      -        - -
    fJcbtDh9 ahashare docking            PD 16     21      -        - -
    fJdKPsao ahashare docking            PD 16     21      -        - -
    fJesq7mq ahashare docking            PD 16     21      -        - -
    fJicHJvB ahashare docking            PD 16     21      -        - -
    fJjMGx6B ahashare docking            PD 16     21      -        - -
    fJmehqTD ahashare docking            PD 16     21      -        - -
    fGRmVfN7 ahashare docking             R 16     21      3   7.746s ruby[
    fGGZn7uy ahashare docking             R 16     21      3   8.106s ruby[
    fG5uZHQC ahashare docking             R 16     21      3   8.500s ruby[
    fG5uZHQB ahashare docking             R 16     21      3   8.502s ruby[
    fG5t5J7r ahashare docking             R 16     21      3   8.504s ruby[
    fG5t5J7q ahashare docking             R 16     21      3   8.506s ruby[
    fAx1yUX9 ahashare main                R 16      1      1   20.19s ruby1
    fAJKDjrX ahashare batch_completion    R 16      1      1   21.66s ruby1
    f9XghMpT ahashare docking_adapter     R 16      1      1   23.41s ruby1
    f8eQiCWj ahashare fusion_adapter      R 16      1      1   25.41s ruby1
```

Balance of docking and fusion jobs

```
(cpu-workflow-r1) ahashare@ruby964$ fluxj
      JOBID USER     NAME                ST URG NTASKS NNODES  RUNTIME NODELIST
    fJdKPsao ahashare docking            PD 16     21      -        - -
    fJesq7mq ahashare docking            PD 16     21      -        - -
    fJicHJvB ahashare docking            PD 16     21      -        - -
    fJjMGx6B ahashare docking            PD 16     21      -        - -
    fJmehqTD ahashare docking            PD 16     21      -        - -
    f2tLcX2ym ahashare docking           PD 16     21      -        - -
    f3vQADVXu ahashare docking           PD 16     21      -        - -
    f6J5TKAA3 ahashare docking           PD 16     21      -        - -
    f7GSxBJrb ahashare docking           PD 16     21      -        - -
    fAUNhoK2o ahashare docking           PD 16     21      -        - -
    fBzXmkNoh ahashare docking           PD 16     21      -        - -
    fC78gGtHD ahashare docking           PD 16     21      -        - -
    fGWzoRrK9 ahashare docking           PD 16     21      -        - -
    fLdF1uryD ahashare docking           PD 16     21      -        - -
    fP9zHLjLs ahashare docking           PD 16     21      -        - -
    fQ6ivUqFm ahashare docking           PD 16     21      -        - -
    fQXKx1Xzo ahashare docking           PD 16     21      -        - -
    fQgSXZUVV ahashare docking           PD 16     21      -        - -
    fRBDD57SX ahashare docking           PD 16     21      -        - -
    fUBULEtto ahashare docking           PD 16     21      -        - -
    fVv7LqG1u ahashare docking           PD 16     21      -        - -
    fWZKbXjRR ahashare docking           PD 16     21      -        - -
    fYJQ6DkWK ahashare docking           PD 16     21      -        - -
    fbqmuXHQT ahashare docking           PD 16     21      -        - -
    ffhTcCQRm ahashare docking           PD 16     21      -        - -
    fg7VqczcP ahashare docking           PD 16     21      -        - -
    fgFQQUrEb ahashare docking           PD 16     21      -        - -
    fhqL87zb9 ahashare docking           PD 16     21      -        - -
    finXkq26B ahashare docking           PD 16     21      -        - -
    fmYGYCrs1 ahashare docking           PD 16     21      -        - -
    fmYP8hfsd ahashare fusion             R 31      1      1  4.794m ruby1453
     fJcbtDh9 ahashare docking            R 16     21      3  5.112m ruby[1239,1269,1286]
     fJWYX9uR ahashare docking            R 16     21      3  11.14m ruby[1182,1190,1213]
    fhq6rgRdq ahashare fusion             R 31      1      1  12.94m ruby1481
     fJVHsm6X ahashare docking            R 16     21      3  13.28m ruby[227,407,491]
    fgFaJE4bh ahashare fusion             R 31      1      1   16.4m ruby1455
     fJSdh3nf ahashare docking            R 16     21      3  16.75m ruby[944,999,1048]
     fJJtUoQB ahashare docking            R 16     21      3  26.36m ruby[1300,1352,1383]
    fYJNdiTkK ahashare fusion             R 31      1      1  33.84m ruby1391
     fJBaULoZ ahashare docking            R 16     21      3  37.97m ruby[520,757,819]
    fUBdmJLBh ahashare fusion             R 31      1      1  42.87m ruby1453
    fQgB1ezjq ahashare fusion             R 31      1      1  50.56m ruby1481
     fAx1yUX9 ahashare main               R 16      1      1   1.702h ruby1391
     fAJKDjrX ahashare batch_completion   R 16      1      1   1.702h ruby1453
     f9XghMpT ahashare docking_adapter    R 16      1      1   1.703h ruby1455
     f8eQiCWj ahashare fusion_adapter     R 16      1      1   1.703h ruby1481
```

Scaling on hybrid CPU and GPU clusters

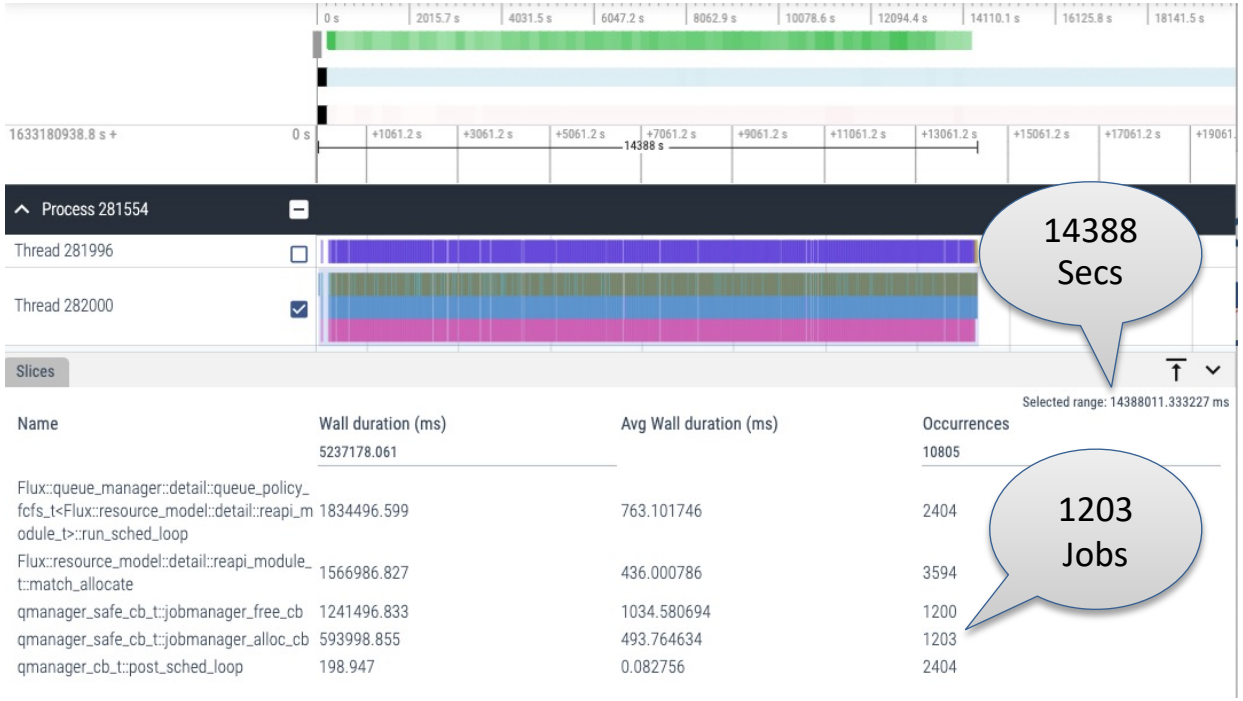
- Workflow consists of just Docking and Fusion currently scaling to ~1000- Ruby nodes and 140 or 220 Lassen nodes respectively (uses Flux)
- **Compound batch size**: ~1500
- 2 Docking adapters process each batch via maestro/flux/MPI sub-workflow:
  - **per batch: Use 145 nodes or 5 nodes**.
  - Pre-fetch set to 36 batches for each adapter (2 adapters * 36 prefetch * 11 nodes = **792 nodes)**
- Each Fusion worker processes each batch on lassen with:
  - **per batch: 1 task using 1 GPU and 10 cores-per-task** (~30 minutes per batch).
  - Each node runs 4 fusion works (**560 or 880 workers**)
- **hybrid cluster workflow** requires DAT to acquire resources simultaneously on both clusters

Performance of different configurations using PerfFlowApect

Flux Version	Docking Nodes/Job	Fusion Total Nodes	Makespan (second)	Time Per Mol. (ms)
0.26.0	145	220	15208	7.67
0.26.0	5	220	7841	3.95
0.26.0	145	140	15257	7.69
0.26.0	5	140	9717	4.9
0.29.0	145	220	19049	9.6
0.29.0	5	220	8041	4.05
0.29.0	145	140	19236	9.7
0.29.0	5	140	9666	4.87

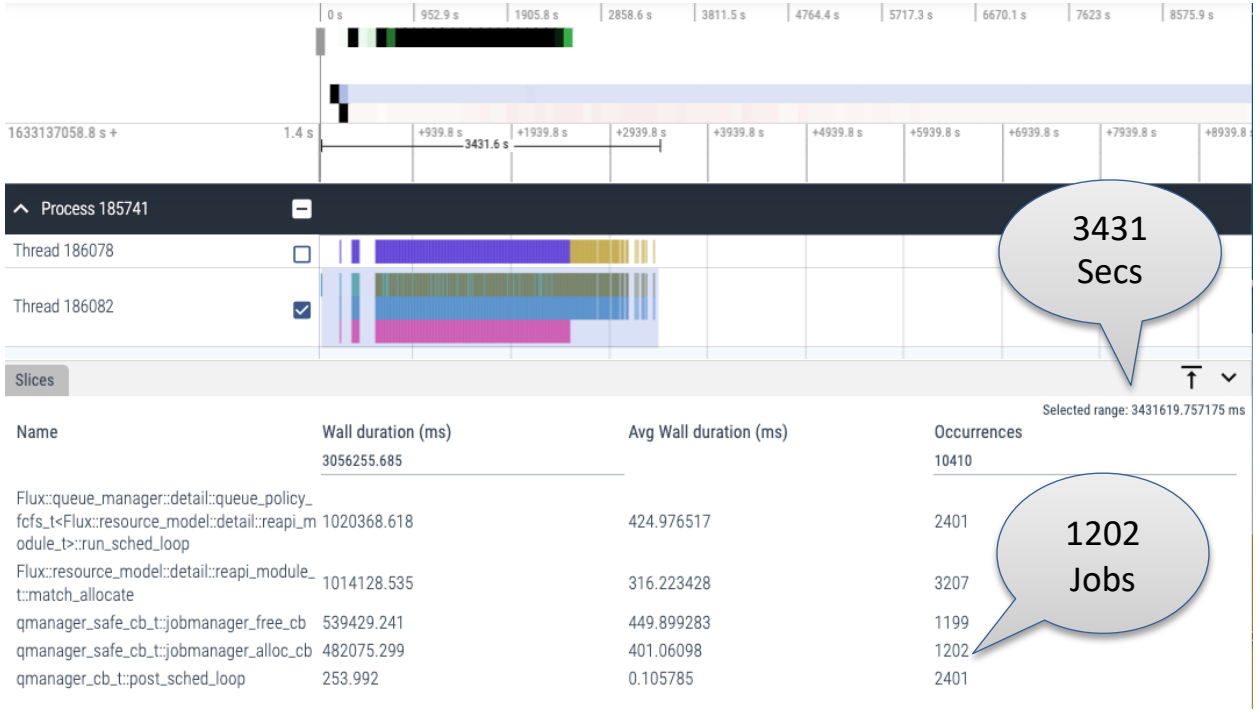
Note: Use the PerfFlowApect data at the top-level Flux instance on Ruby

Performance impacted by docking job size



Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	5237178.061		10805
Flux::queue_manager::detail::queue_policy_fcfs_t<Flux::resource_model::detail::reapi_module_t>::run_sched_loop	1834496.599	763.101746	2404
Flux::resource_model::detail::reapi_module_t::match_allocate	1566986.827	436.000786	3594
qmanager_safe_cb_t::jobmanager_free_cb	1241496.833	1034.580694	1200
qmanager_safe_cb_t::jobmanager_alloc_cb	593998.855	493.764634	1203
qmanager_cb_t::post_sched_loop	198.947	0.082756	2404

Flux=0.26.0 Dock=145 nodes/Job Fusion=220

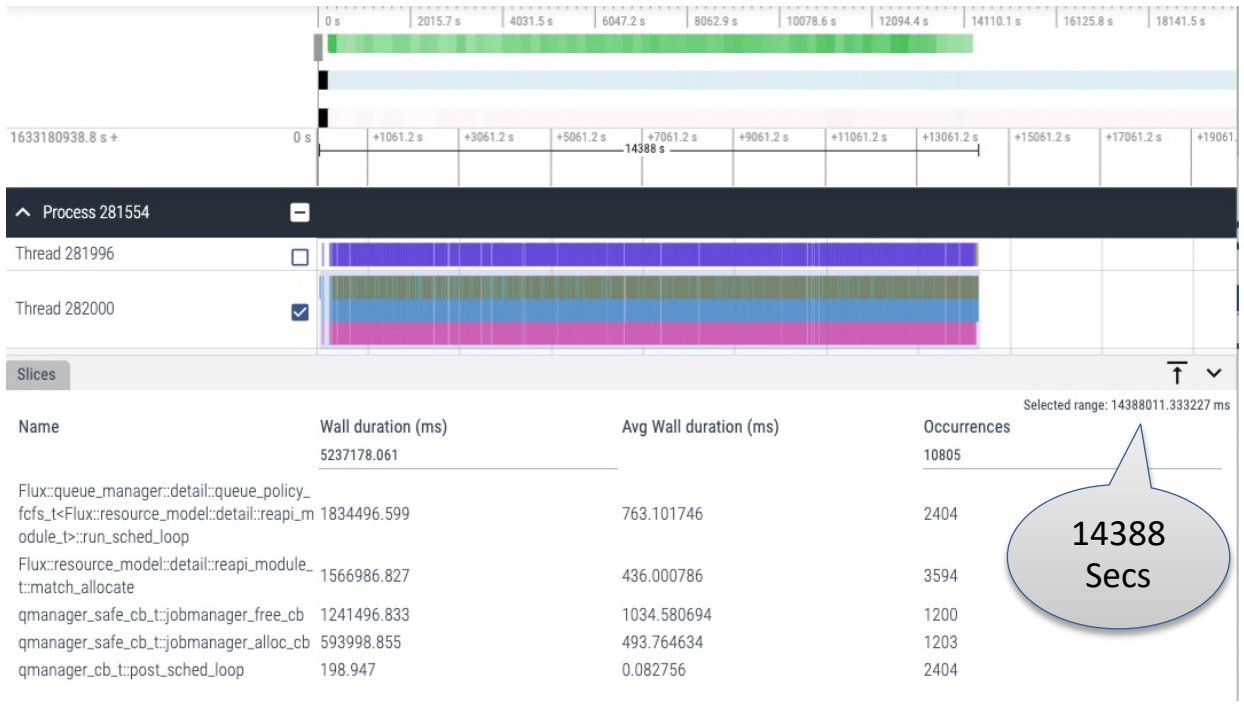


Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	3056255.685		10410
Flux::queue_manager::detail::queue_policy_fcfs_t<Flux::resource_model::detail::reapi_module_t>::run_sched_loop	1020368.618	424.976517	2401
Flux::resource_model::detail::reapi_module_t::match_allocate	1014128.535	316.223428	3207
qmanager_safe_cb_t::jobmanager_free_cb	539429.241	449.899283	1199
qmanager_safe_cb_t::jobmanager_alloc_cb	482075.299	401.06098	1202
qmanager_cb_t::post_sched_loop	253.992	0.105785	2401

Flux=0.26.0 Dock=5 nodes/Job Fusion=220

Using larger number of nodes per docking job slow down the calculation by ***1.93x***

Performance affected by flux overhead



Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	5237178.061		10805
Flux::queue_manager::detail::queue_policy_fcfs_t<Flux::resource_model::detail::reapi_module_t>::run_sched_loop	1834496.599	763.101746	2404
Flux::resource_model::detail::reapi_module_t::match_allocate	1566986.827	436.000786	3594
qmanager_safe_cb_t::jobmanager_free_cb	1241496.833	1034.580694	1200
qmanager_safe_cb_t::jobmanager_alloc_cb	593998.855	493.764634	1203
qmanager_cb_t::post_sched_loop	198.947	0.082756	2404

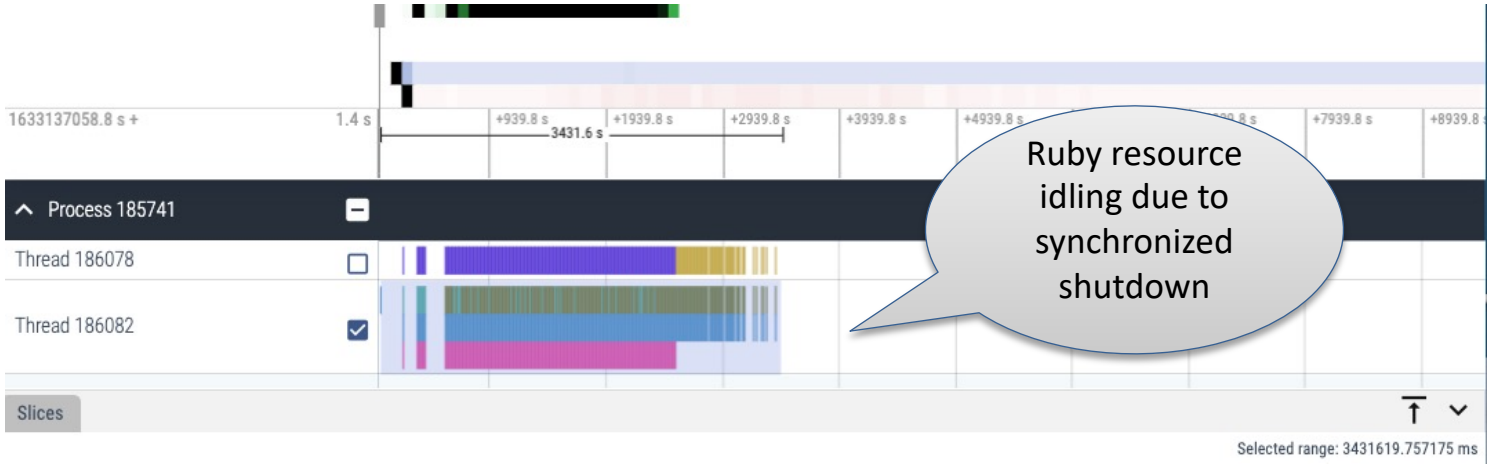
Flux=0.26.0 Dock=145 nodes/job Fusion=220 nodes

Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	265255.141		15509
Flux::queue_manager::detail::queue_policy_fcfs_t<Flux::resource_model::detail::reapi_module_t>::run_sched_loop	132645.126	55.803586	2377
Flux::resource_model::detail::reapi_module_t::cancel	130696.249	109.004377	1199
Flux::resource_model::detail::reapi_module_t::match_allocate_multi	958.425	0.403717	2374
qmanager_safe_cb_t::jobmanager_alloc_cb	547.092	0.454773	1203
qmanager_cb_t::post_sched_loop	183.663	0.051302	3580
Flux::resource_model::detail::match_allocate_multi_cont	131.052	0.036637	3577
qmanager_safe_cb_t::jobmanager_free_cb	93.534	0.07801	1199

Flux=0.29.0 Dock=145 noes/job Fusion=220 nodes

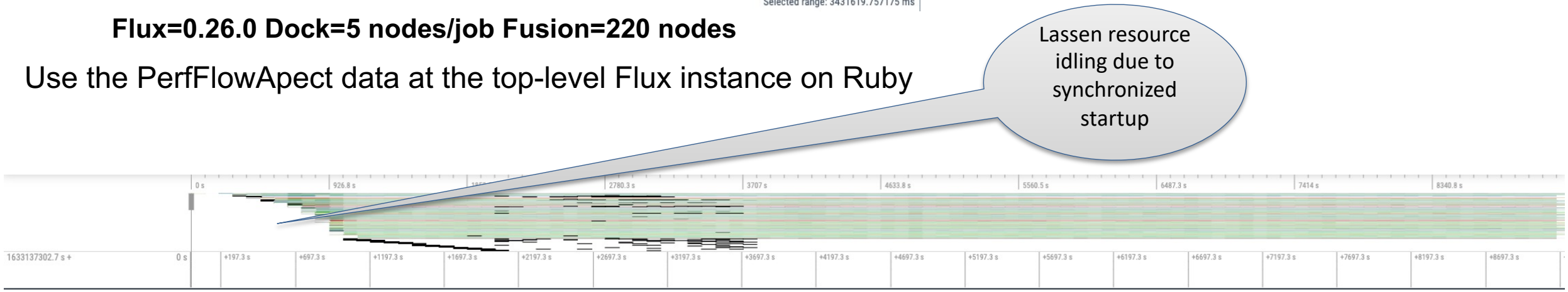
The higher scheduling overhead with Flux Version 0.29.0 slows down the calculations by **1.27x**.

How PerfFlow analysis can help understand other future choices of performance variables



Flux=0.26.0 Dock=5 nodes/job Fusion=220 nodes

Use the PerfFlowApect data at the top-level Flux instance on Ruby



Use the PerfFlowApect data at the top-level Flux instance on Lassen

Conclusions

- AHA MoleS workflow supports scaling from a small number of nodes to 1000+ nodes, potentially spanning multiple heterogeneous clusters depending on the application and workload resource requirements
- Computer resource is idling due to synchronization of docking and fusion
- For Ruby resources, the more imbalanced Docking and fusion throughputs are the worse
- For Lassen resources, the larger the batch size is the more lead time idling. It cannot be determined by flux-level perfflow, Fusion adapter and maestro level perfflows are required.
- In the future, workflow needs to loose/remove synchronization between docking and fusion for the hybrid computer resource.

Acknowledgements

- LLNL PLS/BBTD
  - Felice Lightstone
  - Drew Bennett
  - Brian Bennion
  - Dan Kirshner
  - Edmond Lau
  - Ali Navid
  - Sergio Wong
  - Fangqiang Zhu
- LLNL GS
  - Jonathan Allen
  - Aidan Epstein
  - Stewart He
  - Derek Jones
  - Jeff Mast
  - Garrett Stevenson
  - Marisa Torres
  - Adam Zemla
- LLNL Computing
  - Dong Ahn
  - Bronis de Supinski
  - Francesco Di Natale
  - Stephen Herbein
  - Sam Ade Jacobs
  - Ian Karlin
  - Hyojin Kim
  - Daniel Milroy
  - Brian Van Essen
- Funding
  - American Heart Association CRADA

Thank you!

Questions?

https://github.com/flux-framework/Tutorials.git

Lawrence Livermore
National Laboratory