Agenda

Session	Time	Presenter
Overcoming Scheduling Challenges for Exascale Workflows	8:30AM – 8:50AM	Dong H. Ahn
Flux APIs for Scientific Workflows	8:50AM – 9:20AM	Stephen Herbein
Real-World Practices and Examples	9:20AM – 10:20AM	Joe Koning (30 mins) and Stephen Herbein (20 mins)
Break	10:20AM – 10:30AM	
Future topics: Resource data models and performance variation-aware scheduling	10:30AM – 11:00AM	Dong H. Ahn and Tapasya Patki
Practical Considerations of using Flux and hands-on	11:00AM – 12:00PM	Tapasya Patki + Stephen Herbein hands-on workflow examples

Flux: Using Next-Generation Resource Management and Scheduling Infrastructure for Exascale Workflows

Tutorial for the 2019 ECP Annual Meeting, Jan 16, 2019

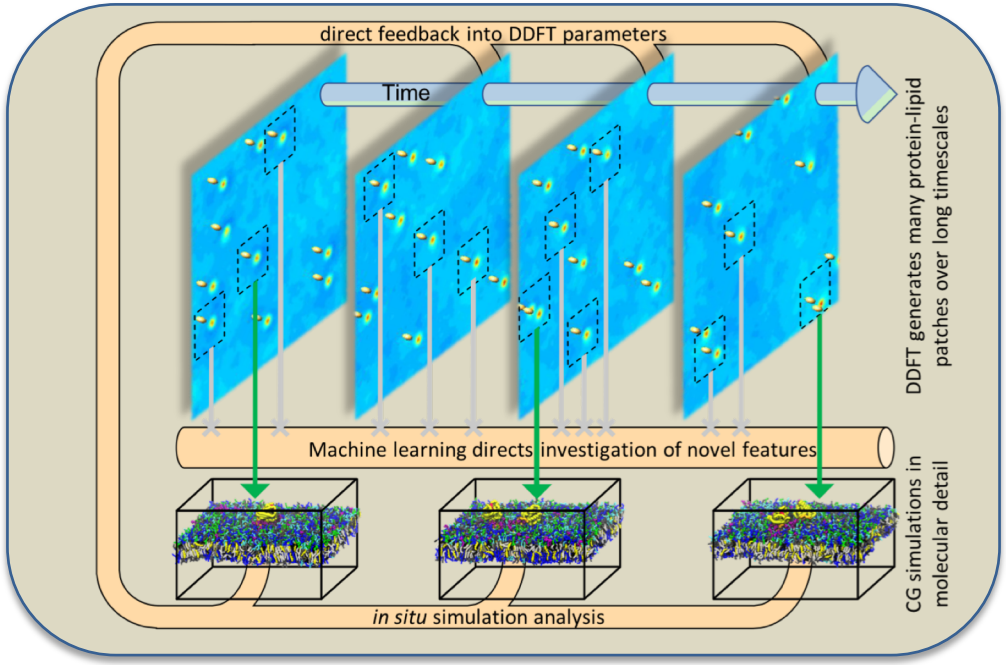
Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, Tapasya Patki, Thomas R. W. Scogland, Becky Springmeyer, and Michela Taufer

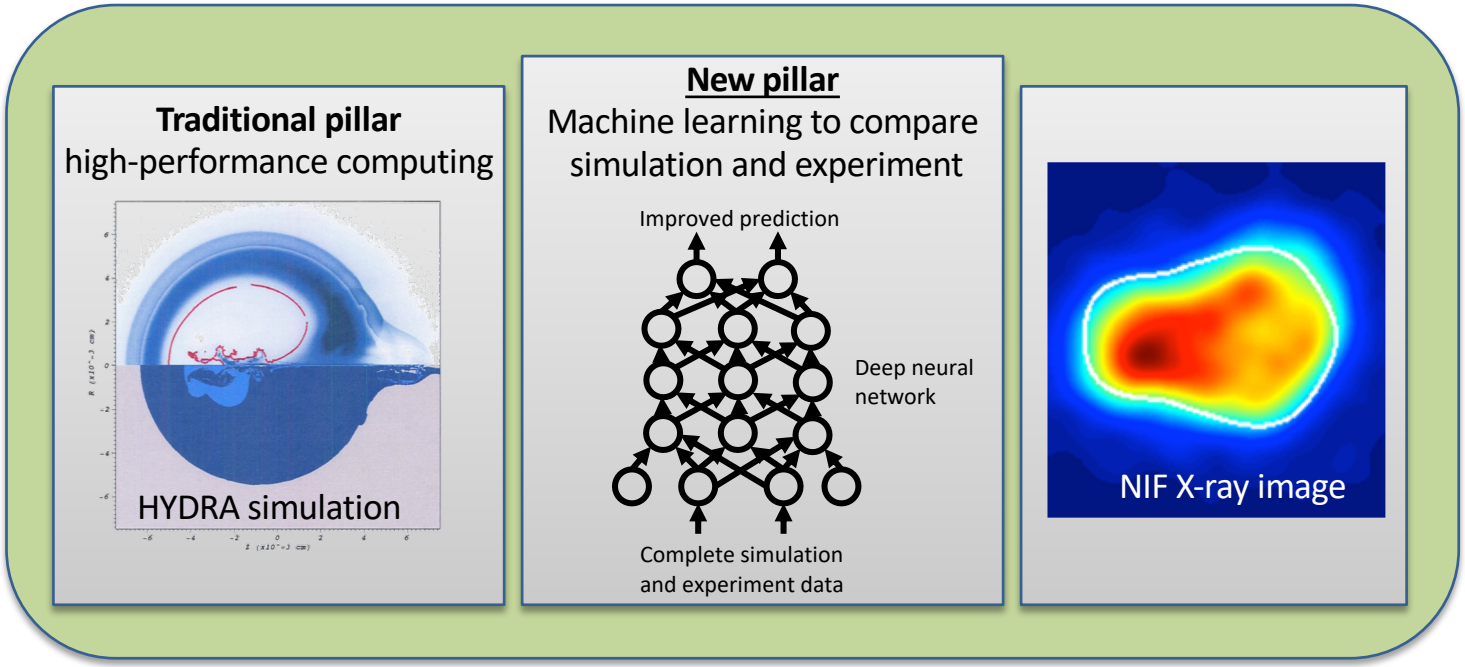
LLNL-PRES- 765450

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.

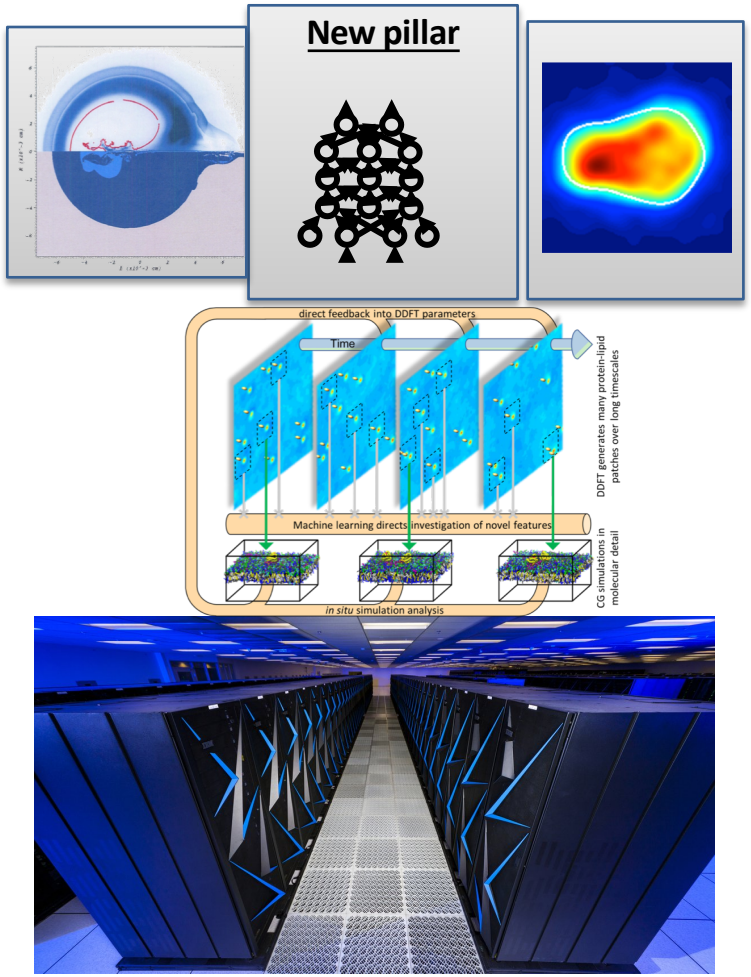




- Cancer Moonshot Pilot2 – co-schedule many elements and ML continuously schedules, de-schedules and executes MD jobs.
- In-situ analytics modules
- ~7,500 jobs simultaneously running

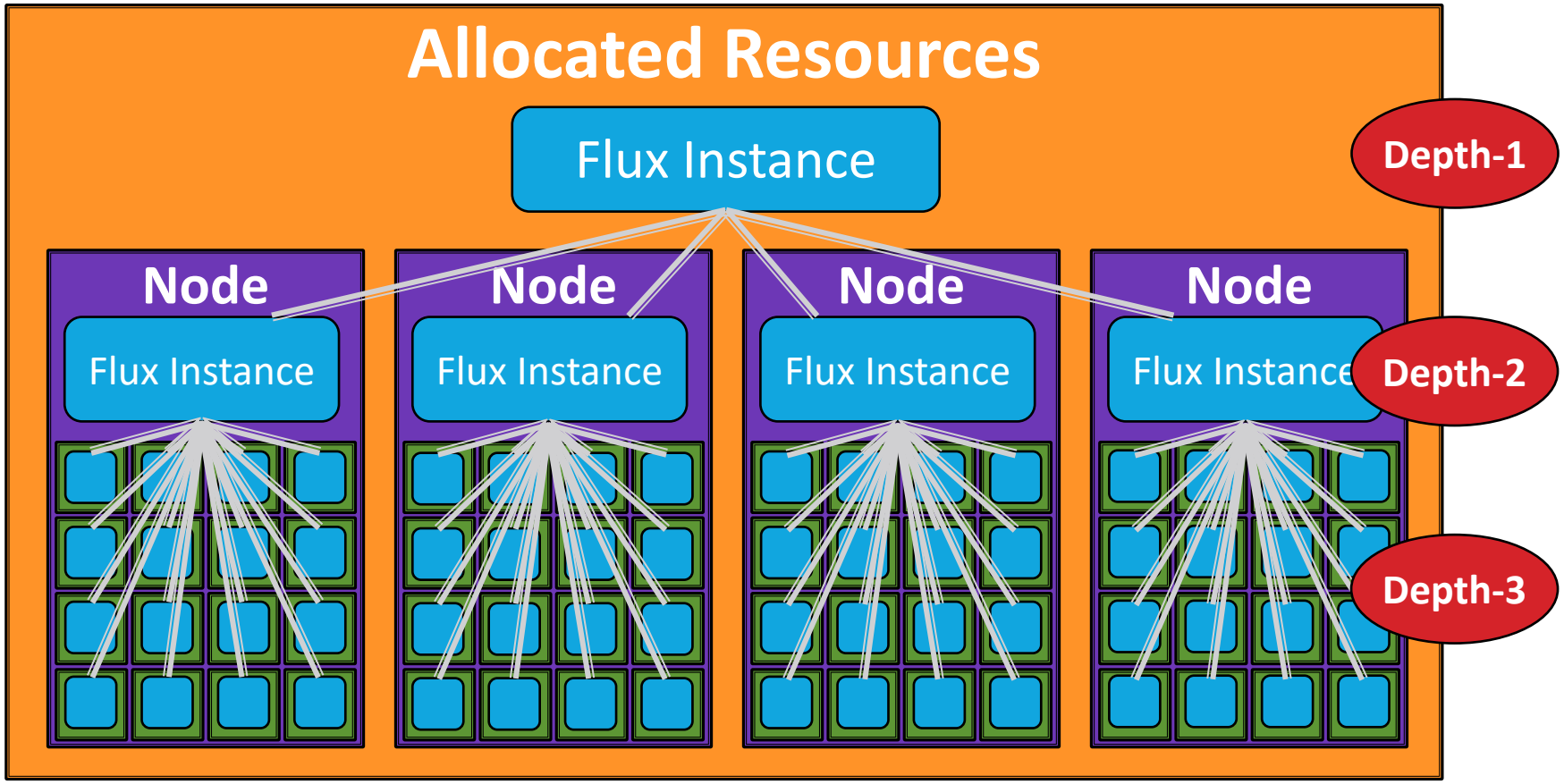
- Machine Learning Strategic Initiative (MLSI) – 1 billion short-running jobs!
- Similar needs for co-scheduling heterogenous components

Key challenges in emerging workflow scheduling include…



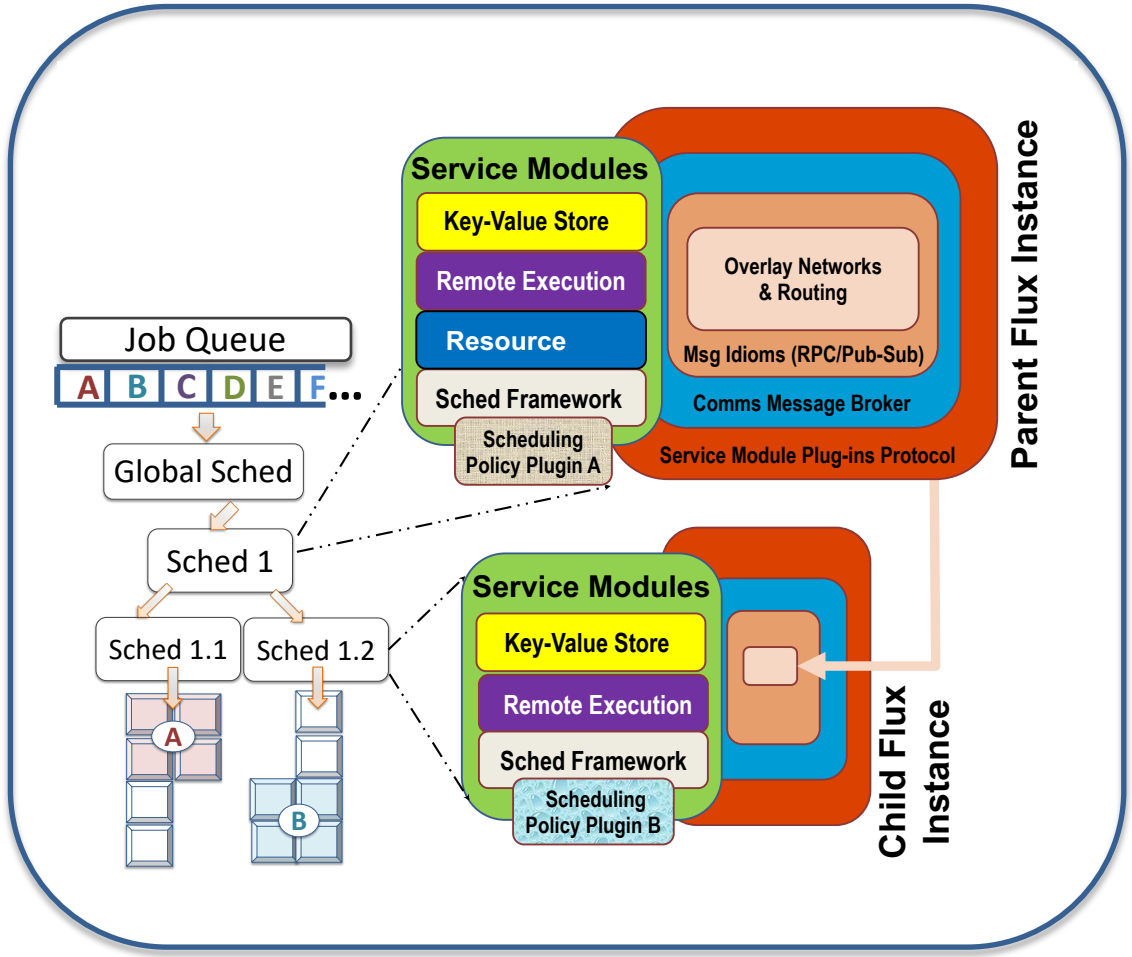
- Co-scheduling challenge
- Job throughput challenge
- Job communication/coordination challenge
- Portability challenge

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.



Techniques

Scheduler Specialization

Scheduler Parallelism

Rich API set

Consistent API set

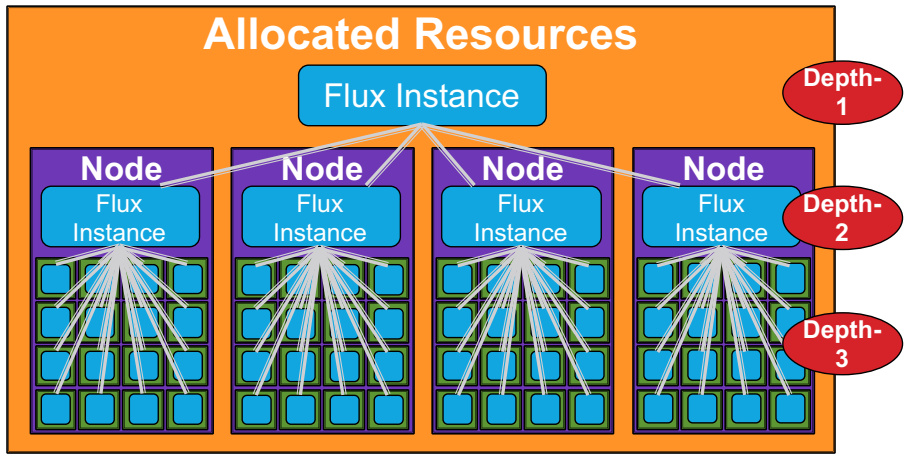
Challenges

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

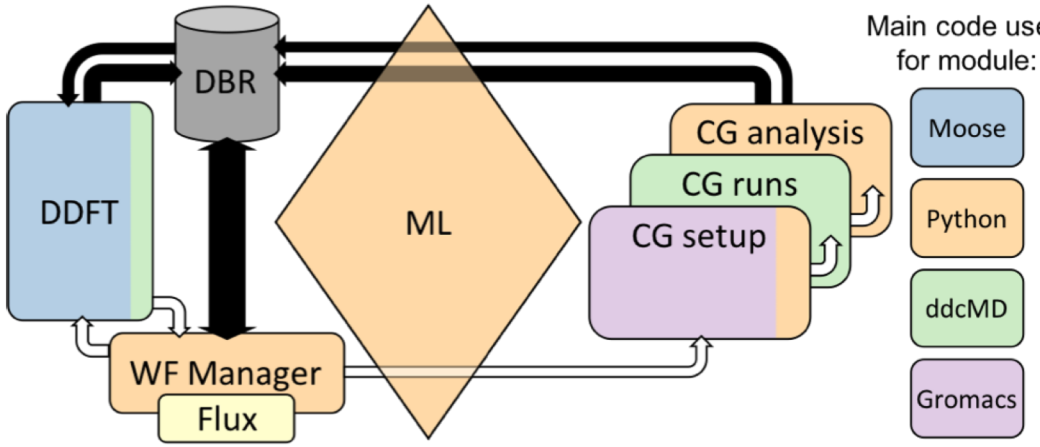
A rich API set enables easy job coordination and communication.

- 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

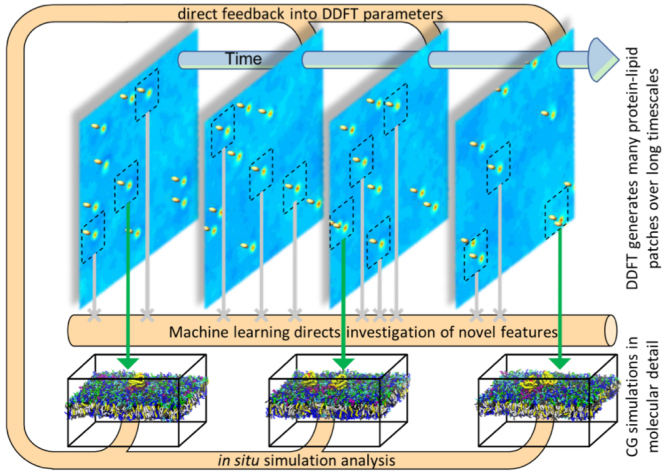
A consistent API set facilitates high portability.

- 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

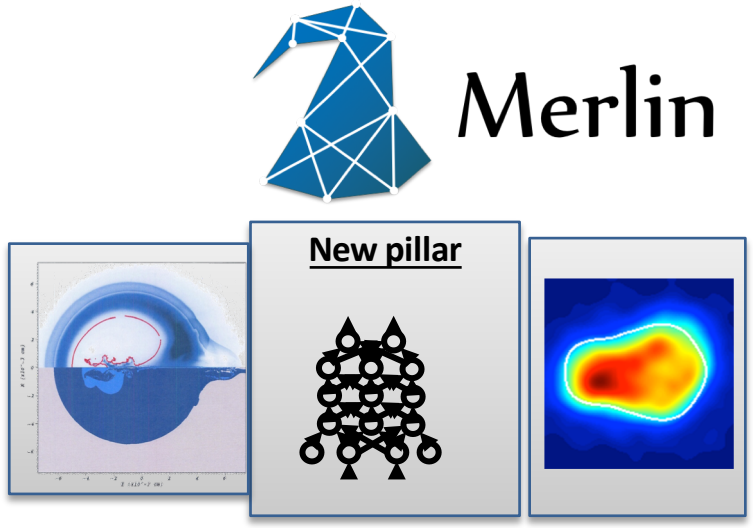


Pilot2 workflow architecture



- 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 handles the volume of jobs and co-scheduling
  - At least 5 different logically separate jobs, each with CPU and/or GPU requirements on every node
- Scheduling policy specialized to fastest FCFS policy (queue-depth=1) for throughput.
- Use of a large single Flux instance

Flux allows MLSI's Merlin workflow manager for easy, scalable interaction with the scheduler.



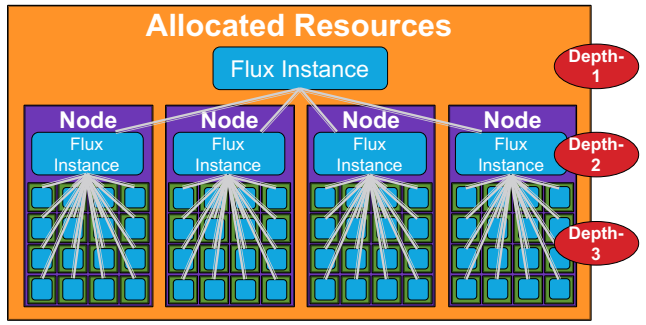
Machine Learning Strategic Initiative

- Merlin manages the running of an ensemble of simulations with in-situ machine learning.
- Divide up the large sample space into chunks to run a subset of simulations on each allocation.
- Issues encountered in Merlin
  - Merlin worker runs on every node of the batch job allocation.
  - A simple python-based subprocess maps the number of tasks and resources onto the different system launch commands: portability and maintenance issues.
  - LSF jsrun does not allow Merlin brokers to call jsrun for simulations.
  - co-scheduling with heterogeneous CPUs and GPUs requirements

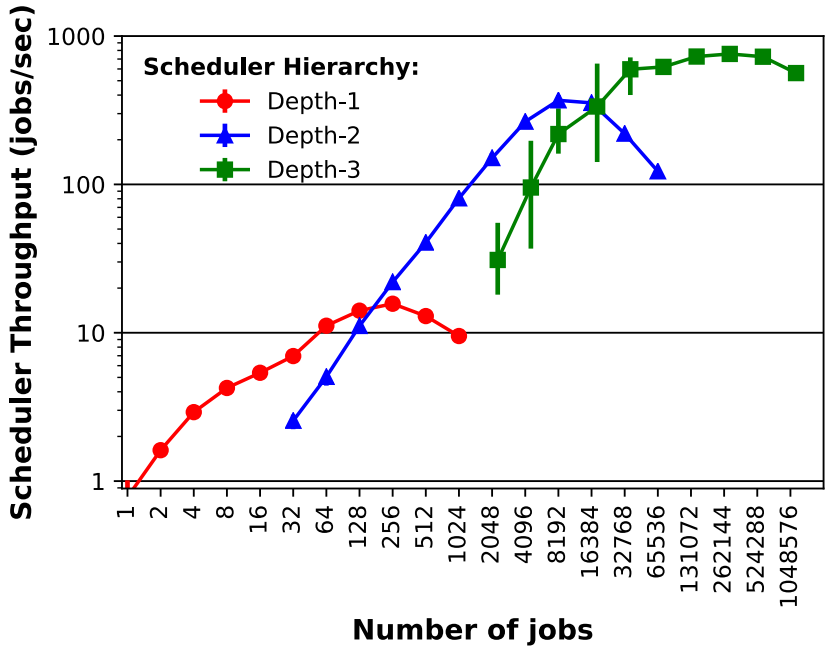
Flux helps Merlin with all of the key challenges.

- A Flux instance is instantiated on each batch job allocation.
- Replace python subprocess call with Flux's python binding (**rpc_send** with a **job.submit**) to submit the job into the containing Flux instance.
- The workflow manager can be informed on all stages of the job submission by registering a JSC callback on each status change of the submitted job.
- Use of many small to medium-sized Flux instances for higher job throughput and resiliency.

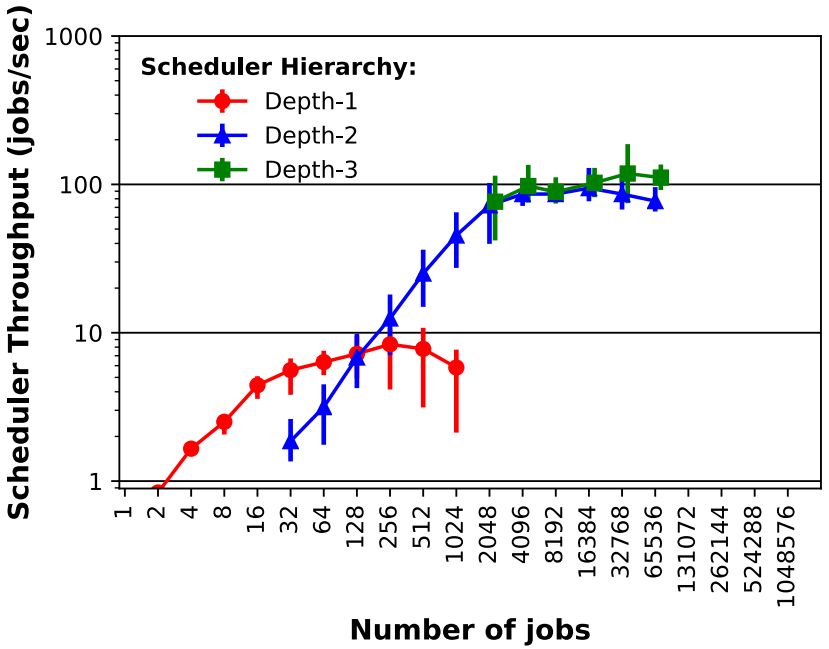
Deepening scheduler hierarchy can significantly improve job throughput.



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



Stress test



UQ workflow

Flux significantly enables emerging workflows on high-end HPC systems.

- Ensemble-, coupling-based workflows are increasingly becoming a "norm" on high-end HPC systems and traditional approaches are hard-pressed.
- Four key challenges have been identified.
- Flux provides a new scheduling model and an implementation and API set that embody this model.
- Our case studies and performance evaluations suggest that our model can significantly address all of the challenges.
- Flux is powering up the production runs of the major science runs on LLNL's Sierra, pre-exascale system.

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

Flux APIs for Scientific Workflows

Dong H. Ahn, **Stephen Herbein**, Joe Koning, Tapasya Patki, Tom Scogland

January 16, 2019

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

What is Flux?

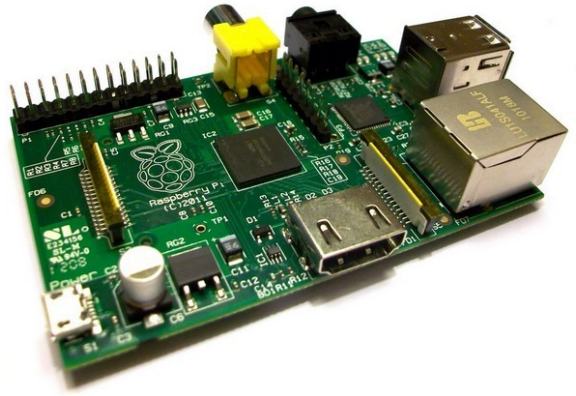
- New Resource and Job Management Software (RJMS) developed at LLNL
- A way to manage remote resources and execute tasks on them

What is Flux?

- New Resource and Job Management Software (RJMS) developed at LLNL
- A way to manage **remote resources** and execute tasks on them

amazon web services™

Google Cloud



flickr: dannychamoro

What is Flux?

- New Resource and Job Management Software (RJMS) developed at LLNL
- A way to manage remote resources and **execute tasks** on them

Why Flux?

- Extensibility
  - Open source
  - Modular design with support for user plugins
- Scalability
  - Designed from the ground up for exascale and beyond
  - Already tested at 1000s of nodes & millions of jobs
- Usability
  - C, Lua, and Python bindings that expose 100% of Flux's functionality
  - Can be used as a single-user tool or a system scheduler
- Portability
  - Optimized for HPC and runs in Cloud and Grid settings too
  - Runs on any set of Linux machines: only requires a list of IP addresses or PMI

Flux is designed to make **hard** scheduling problems **easy**

Portability: Running Flux

- Already installed on TOSS systems (including Sierra)
  - `spack install flux-sched` for everywhere else
- Flux can run anywhere that MPI can run, (via PMI – Process Management Interface)
  - Inside a resource allocation from: itself (hierarchical Flux), Slurm, Moab, PBS, LSF, etc
  - `flux start` **OR** `srun flux start`
- Flux can run anywhere that supports TCP and you have the IP addresses
  - `flux broker -Sboot.method=config -Sboot.config_file=boot.conf`
  - `boot.conf:`

```
session-id = "mycluster"
tbon-endpoints = [
  "tcp://192.168.1.1:8020",
  "tcp://192.168.1.2:8020",
  "tcp://192.168.1.3:8020"]
```

Why Flux?

- Extensibility
  - Open source
  - Modular design with support for user plugins
- Scalability
  - Designed from the ground up for exascale and beyond
  - Already tested at 1000s of nodes & millions of jobs
- Usability
  - C, Lua, and Python bindings that expose 100% of Flux's functionality
  - Can be used as a single-user tool or a system scheduler
- Portability
  - Optimized for HPC and runs in Cloud and Grid settings too
  - Runs on any set of Linux machines: only requires a list of IP addresses or PMI

Usability: Submitting a Batch Job

- Slurm
  - `sbatch -N2 -n4 -t 2:00 sleep 120`
- Flux CLI
  - `flux submit -N2 -n4 -t 2m sleep 120`
- Flux API:

```
import json, flux

jobreq = {
    'nnodes'   : 2,
    'ntasks'   : 4,
    'walltime' : 120,
    'cmdline'  : ["sleep", "120"]}

f = flux.Flux ()
resp = f.rpc_send ("job.submit", json.dumps(jobreq))
```

Usability: Running an Interactive Job

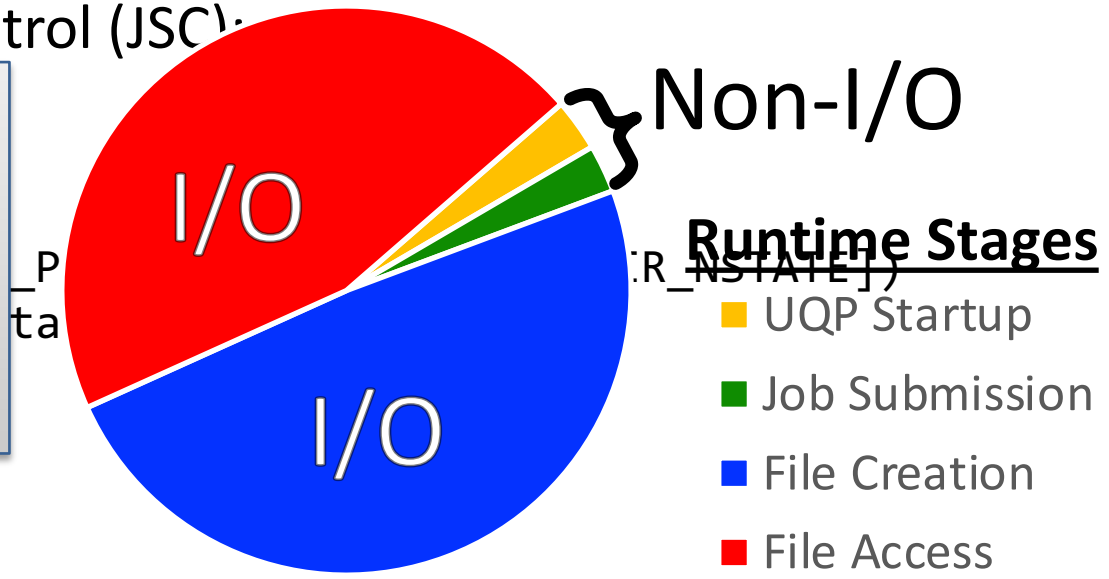
- Slurm
  - `srun –N2 –n4 –t 2:00 sleep 120`
- Flux CLI
  - `flux wreckrun –N2 –n4 –t 2m sleep 120`
- Flux API:

```
import sys
from flux import kz

resp = f.rpc_send ("job.submit", json.dumps(jobreq))
kvs_dir = resp['kvs_dir']

for task_id in range(jobreq['ntasks']):
    kz.attach (f, "{}.{}.stdout".format(kvs_dir, task_id), sys.stdout)

f.reactor_run (f.get_reactor (), 0)
```

Usability: Tracking Job Status

- CLI: slow, non-programmatic, inconvenient to parse
  - `watch squeue –j JOBID`
  - `watch flux wreck ls JOBID`
- Tracking via the filesystem
  - `date > $JOBID.start; srun myApp; date > $JOBID.stop`
- Push notification via Flux's Job Status and Control (JSC):

```
→ quota -vf ~/quota.conf
Disk quotas for herbein1:
Filesystem     used   quota  limit   files
/p/lscratchrza 760.3G n/a    n/a     8.6M
```



Why Flux?

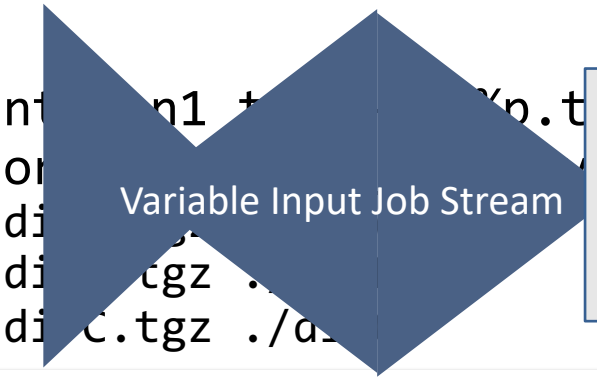
- Extensibility
  - Open source
  - Modular design with support for user plugins
- Scalability
  - Designed from the ground up for exascale and beyond
  - Already tested at 1000s of nodes & millions of jobs
- Usability
  - C, Lua, and Python bindings that expose 100% of Flux's functionality
  - Can be used as a single-user tool or a system scheduler
- Portability
  - Optimized for HPC and runs in Cloud and Grid settings too
  - Runs on any set of Linux machines: only requires a list of IP addresses or PMI

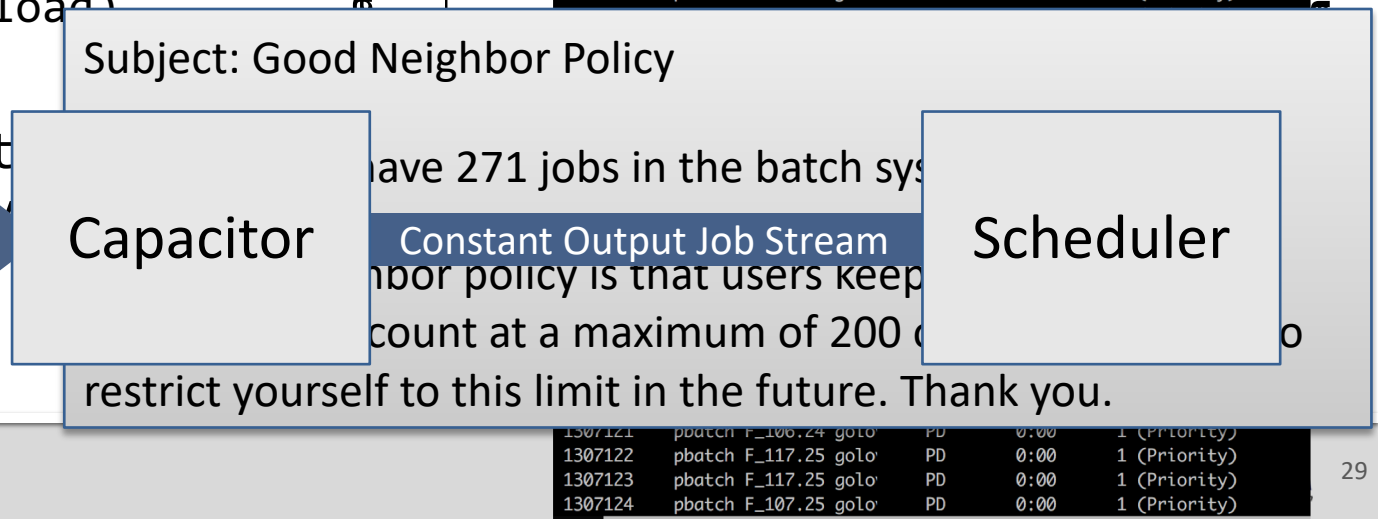
Scalability: Running Many Jobs

- Slurm
  - `find ./ -exec sbatch –N1 tar –cf {}.tgz {}\;`
    - Slow: requires acquiring a lock in Slurm, can timeout causing failures
    - Inefficient: uses 1 node for each task
  - `find ./ -exec srun –n1 tar –cf {}.tgz {}\;`
    - Slow: spawns a process for every submission
    - Inefficient: is not a true scheduler – can overlap tasks on cores
- Flux API:

```
for f in os.listdir('.'):
    payload['command'] = ["tar", "-cf", "{}.tgz".forma
    resp = f.rpc_send ("job.submit", payload)
```

- Flux Capacitor
  - `find ./ -print  n1 t  p.t`
  - `flux-capacitor`
    - `-n1 tar -cf di`
    - `-n1 tar -cf di tgz .`
    - `-n1 tar -cf di c.tgz ./di`





Scalability: Running Many Heterogeneous Jobs

- Slurm
  - No support for heterogeneous job steps in versic
  - Limited support in versions after 17.11
- Flux Capacitor
  - `flux-capacitor --command_file my_com`
    - `-n1 tar -cf dirA.tgz ./dirA`
    - `-n32 make –j 32`
    - `-N4 my_mpi_app`
    - `...`

https://slurm.schedmd.com/heterogeneous_jobs.html#limitations

Limitations

The backfill scheduler has limiations in how it tracks usage of CPUs and memory in the future. This typically requires the backfill scheduler be able to allocate each component of a heterogeneous job on a different node in order to begin its resource allocation, even if multiple compoents of the job do actually get allocated resources on the same node.

In a federation of clusters, a heterogeneous job will execute entirely on the cluster from which the job is submitted. The heterogeneous job will not be eligible to migrate between clusters or to have different components of the job execute on different clusters in the federation.

Job arrays of heterogeneous jobs are not supported.

The srun command's --no-allocate option is not supported for heterogeneous jobs.

Only one job step per heterogeneous job component can be launched by a single srun command (e.g. "srun --pack-group=0 alpha : --pack-group=0 beta" is not supported).

The sattach command can only be used to attach to a single component of a heterogeneous job at a time.

Heterogeneous jobs are only scheduled by the backfill scheduler plugin. The more frequently executed scheduling logic only starts jobs on a first-in first-out (FIFO) basis and lacks logic for concurrently scheduling all components of a heterogeneous job.

Heterogeneous jobs are not supported with Slurm's select/serial plugin.

Heterogeneous jobs are not supported on Cray ALPS systems.

Heterogeneous jobs are not supported on IBM PE systems.

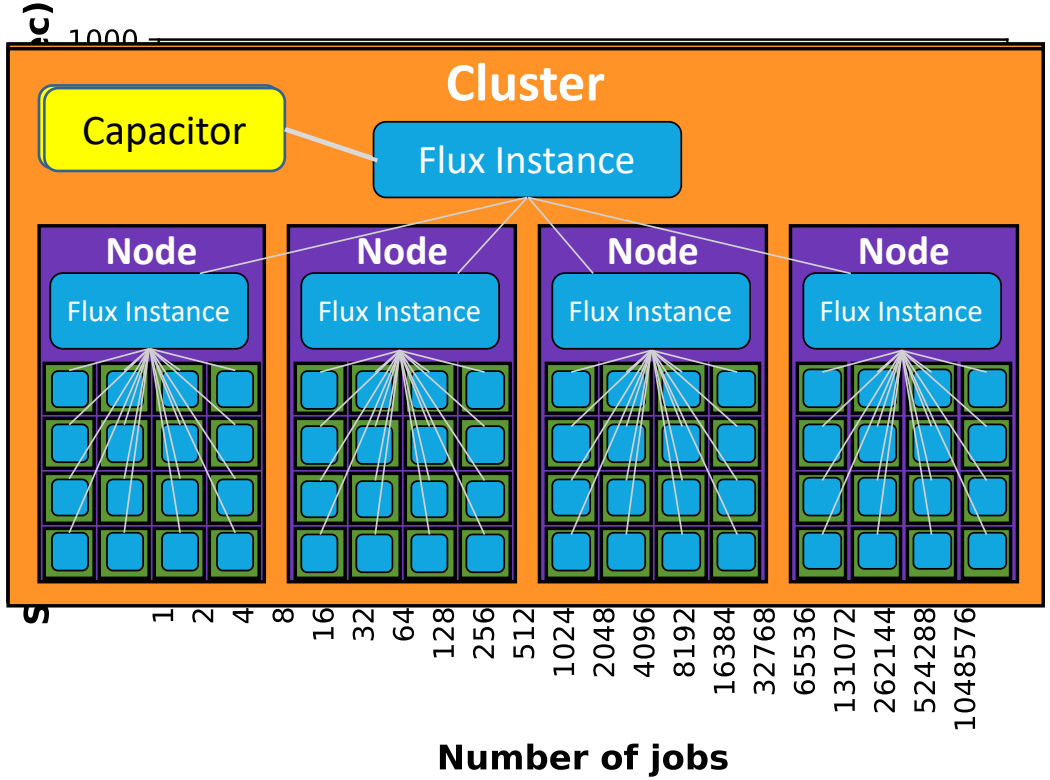
Slurm's PERL APIs currently do not support heterogeneous jobs.

The srun --multi-prog option can not be used to span more than one heterogeneous job component.

The srun --open-mode option is by default set to "append".

Scalability: Running Millions of Jobs

- Flux Capacitor (Depth-1)
  - `flux-capacitor --command_file my_command_file`
- Hierarchical Flux Capacitor (Depth-2)
  -
```
for x in ./*.commands; do
      flux submit -N1 flux start  \
           flux-capacitor --command_file $x
done
```
- Flux Hierarchy (Depth-3+)
  - `flux-hierarchy --config=config.json --command_file my_command_file`

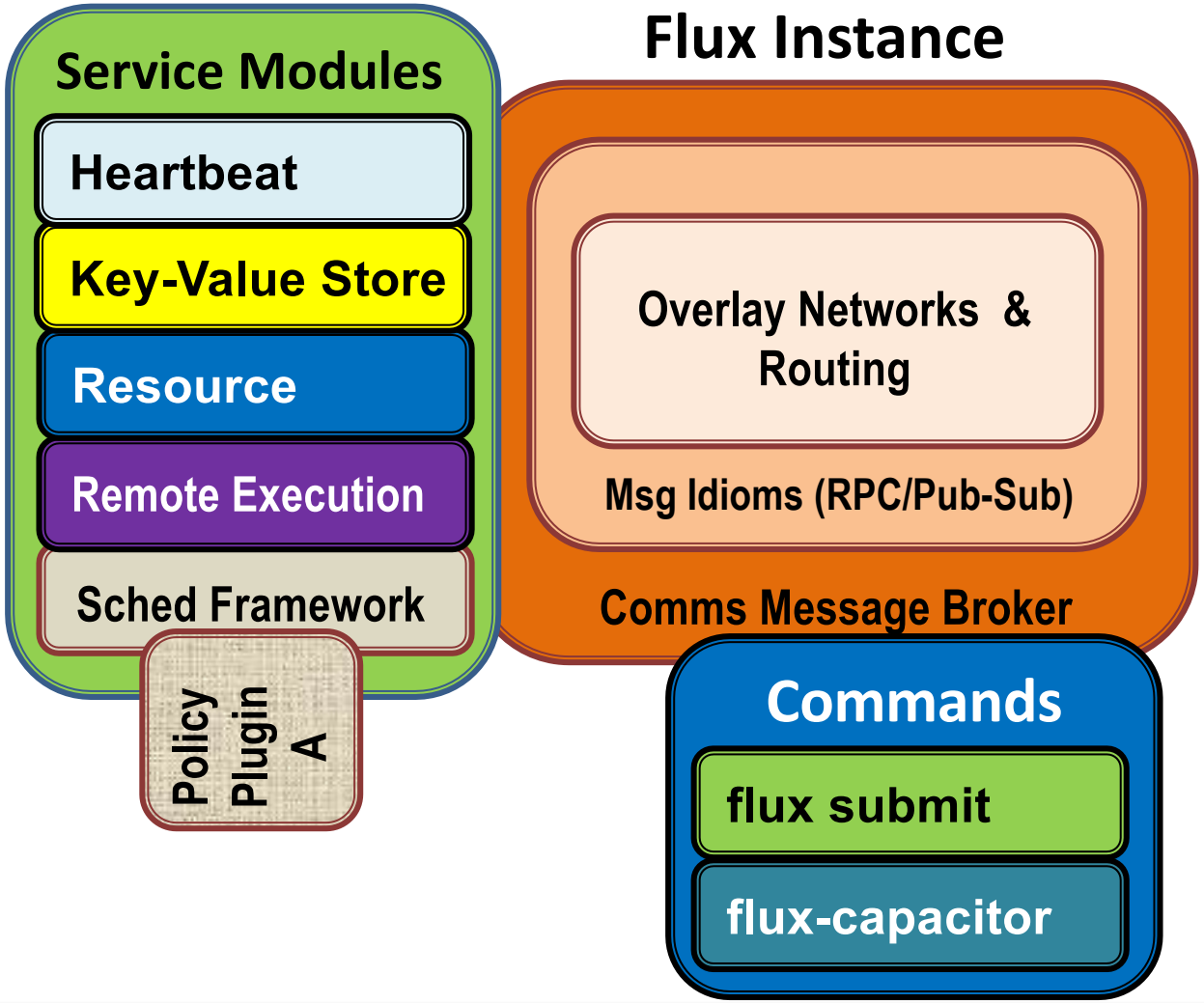


Why Flux?

- Extensibility
  - Open source
  - Modular design with support for user plugins
- Scalability
  - Designed from the ground up for exascale and beyond
  - Already tested at 1000s of nodes & millions of jobs
- Usability
  - C, Lua, and Python bindings that expose 100% of Flux's functionality
  - Can be used as a single-user tool or a system scheduler
- Portability
  - Optimized for HPC and runs in Cloud and Grid settings too
  - Runs on any set of Linux machines: only requires a list of IP addresses or PMI

Extensibility: Modular Design

- At the core of Flux is an overlay network
  - Built on top of ZeroMQ
  - Supports RPCs, Pub/Sub, Push/Pull, etc
- Modules provide extended functionality (i.e., services)
  - User-built modules are loadable too
  - Some modules also support plugins
- External tools and commands can access services
  - User authentication and roles supported



Extensibility: Creating Your Own Module

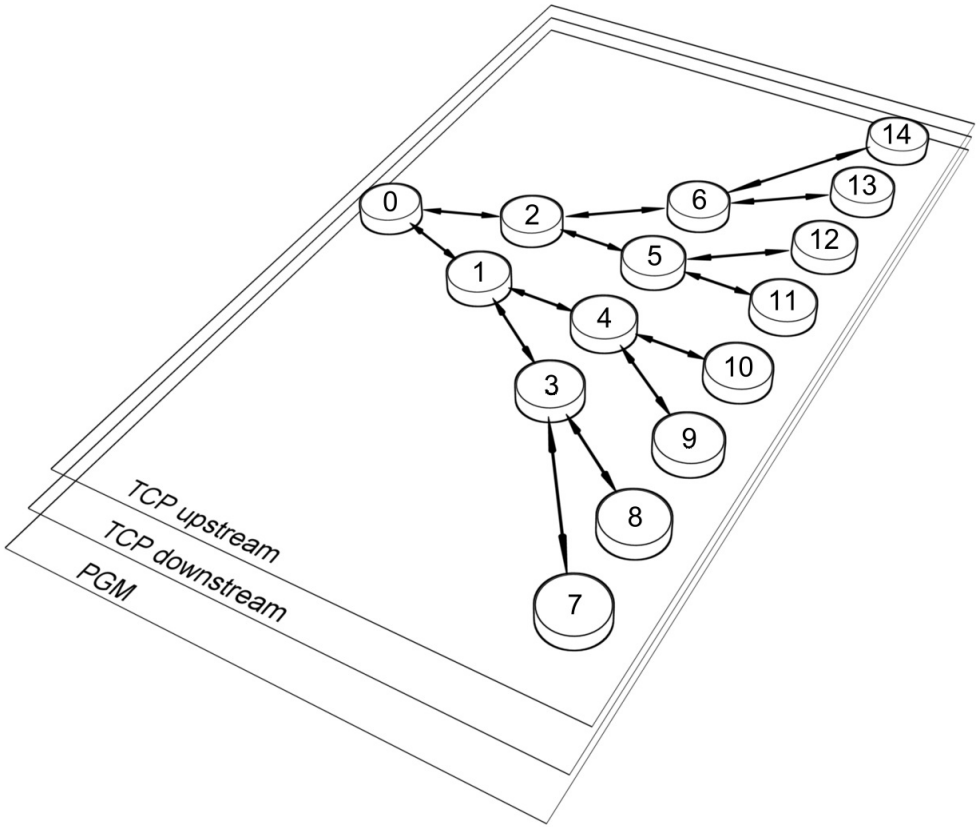
- Register a new service “pymod.new_job” that ingests jobs and responds with a Job ID

```
import itertools, json, flux

def handle_new_job(f, typemask, message, arg):
    job_queue, job_ids = arg
    job_queue.append(message.payload)
    response = {‘jobid’ : job_ids.next()}
    f.respond(message, 0, json.dumps(response))

def mod_main(f, *argv):
    f.msg_watcher_create(flux.FLUX_MSGTYPE_REQUEST,
                         handle_new_job,"pymod.new_job",
                         args=([], itertools.count(0))).start()

    f.reactor_run(f.get_reactor(), 0)
```

- Load using `flux module load pymod --module=path/to/file.py`

Extensibility: Flux's Communication Overlay

- Connect to a running flux instance
  - `f = flux.Flux()`
- Send an RPC to a service and receive a response
  - `resp = f.rpc_send (”pymod.new_job", payload)`
    `jobid = json.loads(resp)[‘jobid’]`
- Subscribe to and publish an event
  - `f.event_subscribe(“node_down”)`
    `f.msg_watcher_create(node_down_cb,`
    `raw.FLUX_MSGTYPE_EVENT,`
    `“node_down”).start()`
  - `f.event_send(“node_down”)`



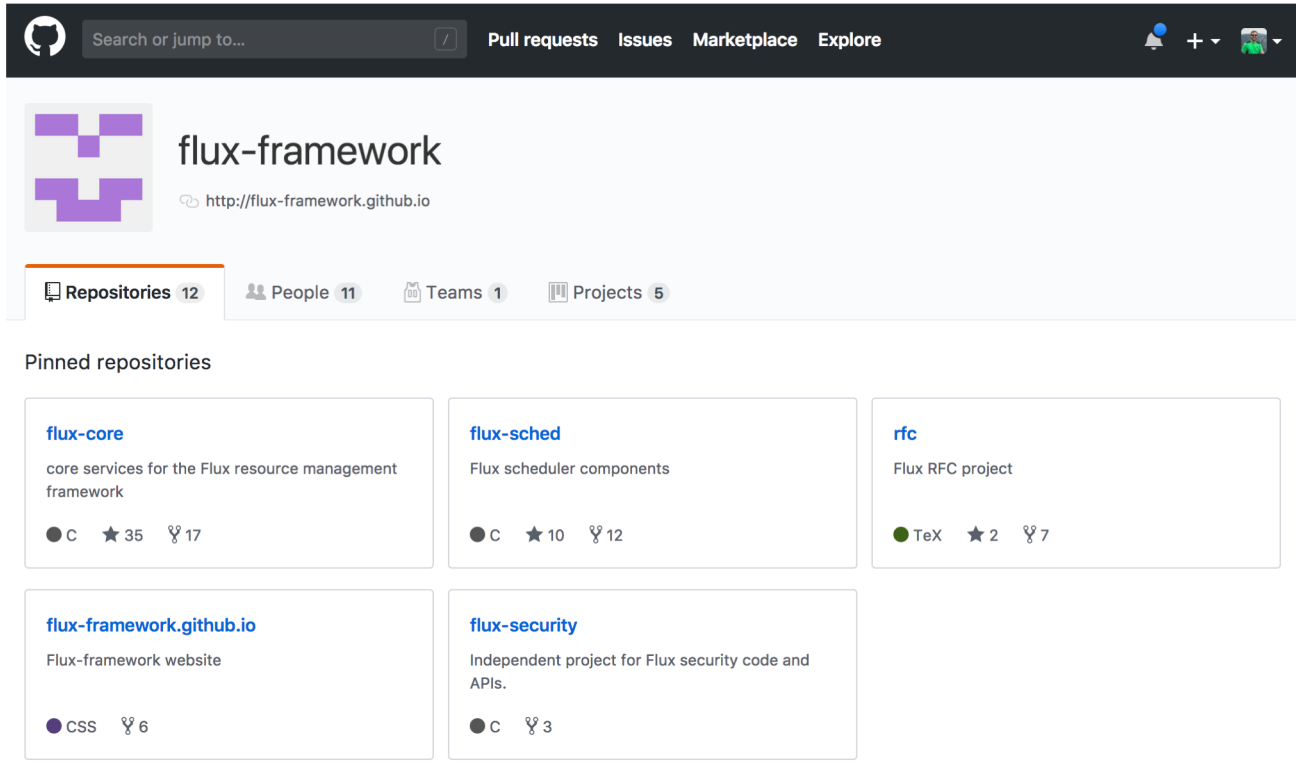
Extensibility: Scheduler Plugins

- Common, built-in scheduler plugins:
  - First-come First-Served (FCFS)
  - Backfilling
    - Conservative
    - EASY
    - Hybrid
- Various, advanced scheduler plugins:
  - I/O-aware
  - CPU performance variability aware
  - Network-aware
- Create your own!
- Loading the plugins
  - `flux module load sched.io-aware`
  - `FLUX_SCHED_OPTS="plugin=sched.fcfs" flux start`

Extensibility: Open Source

- Flux-Framework code is available on GitHub
- Most project discussions happen in GitHub issues
- PRs and collaboration welcome!

GitHub



Thank You!

Merlin Workflow Tutorial
Featuring FLUX scheduling

ECP Meeting, Houston, TX

Jan 2019

Joseph Koning

LLNL-PRES- 765450

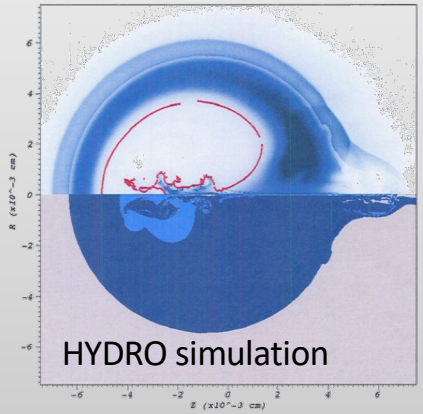
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

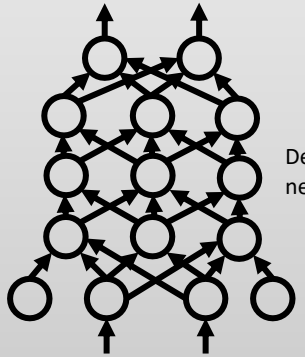
The LLNL Machine Learning Strategic Initiative LDRD aims to add a third pillar to scientific discovery

- Add a machine learning pillar to bridge the gap between simulation and experimental data
  - Brian Spears, LDRD PI
  - Luc Peterson, Workflow PI
  - Merlin is developed by: Luc Peterson, Peter Robinson, Jessica Semler, Benjamin Bay, John Field, Joe Koning

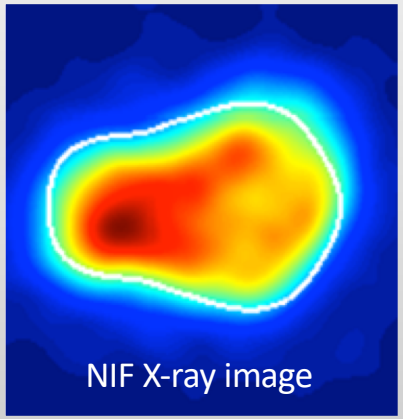
Traditional pillar
high-performance computing



New pillar
Machine learning to compare simulation and experiment



Traditional pillar
Experimental Data



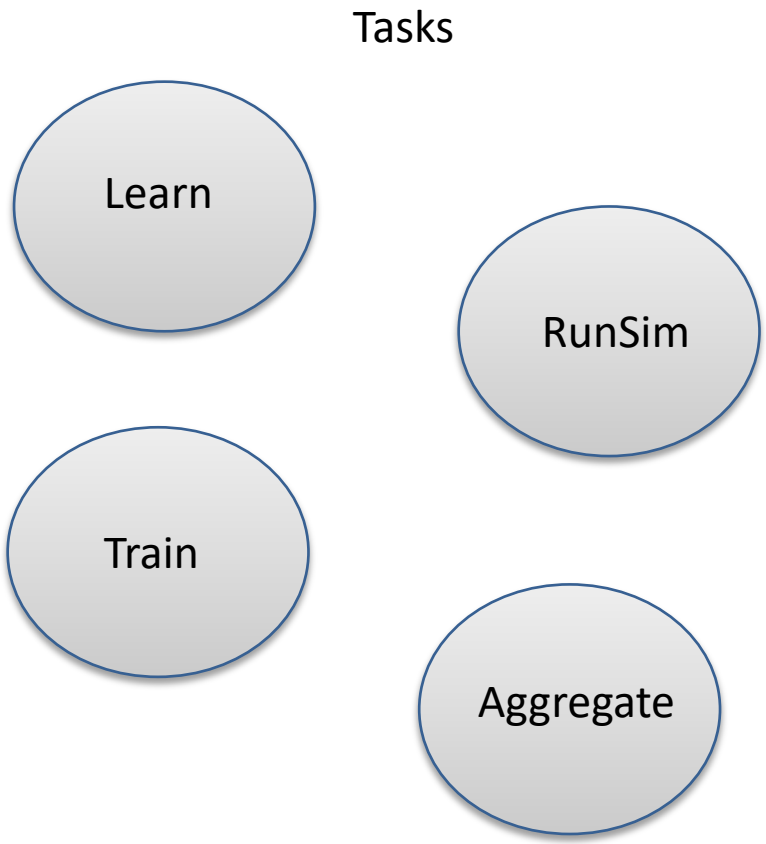
Merlin was created to handle the full workflow of an Ensemble of simulations

- Provide for the ability to run an ensemble of data while learning and speculating with the learned model
- Run a large ensemble of simulations from python to large multi-physics simulation codes
- Machine Learning needs large amounts of data
- Use machine learning to create a model of the data from the ensemble

The previous ensemble method was too rigid and did not allow for arbitrary tasks

Problem

- An ensemble of simulations may consist of various components that do not run at the same time or on the same resources
- This ensemble may consist of a large number of tasks that are intractable for a user to orchestrate themselves
- The ensemble may change as it is being run



Runtime issues and finite allocations can cripple productivity in large ensembles

Problems

- Large Ensembles are more prone to issues with resources
  - Compute resource availability
  - Task crashes
  - Compute resource sharing (timeouts)

- Data state is more difficult to assess

Implementing a scheduler to handle all the arbitrary tasks is challenging

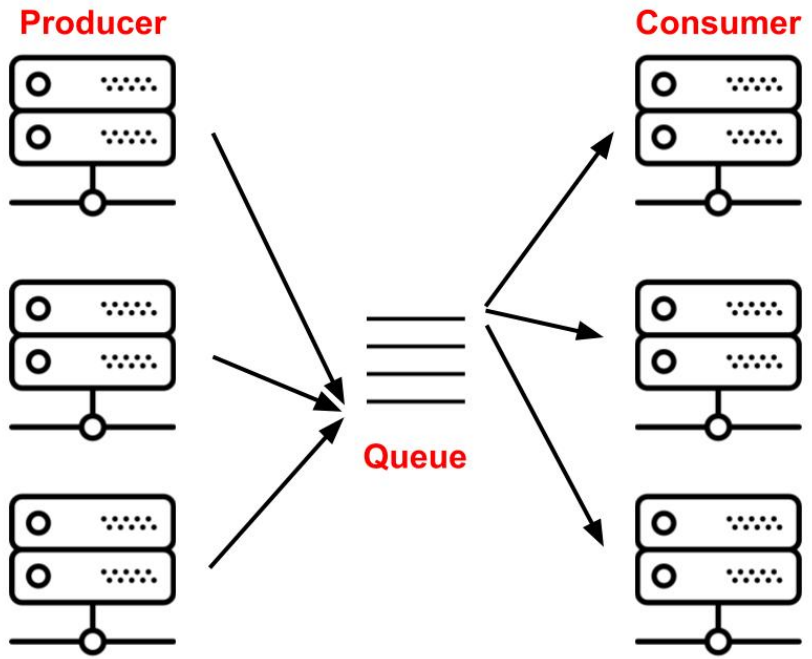
Problem

- Different tasks may need different resources
  - May need to write our own scheduler
- The task based nature of the workflow means there will be nested launches (workers launched with jsrun calling jsrun)
- Utilizing all the resources on the allocation is imperative

The celery system allows for arbitrary task definition

Solution

- Celery provides a producer/consumer environment
- A broker server will contain the tasks to be run
- Workers (cluster compute resources) will request tasks , run them and return state and results
- A backend server will collect the results



A server with state saving capability can alleviate issues with finite resources

Solution

- The redis in memory key-value store can provide the broker and backend functionality while not requiring a dedicated server.
- State can be written to files for compute resource restart
- The server can be run on a dedicated machine or as part of the allocation

Flux allows for scheduling on any cluster

Solution

- Provides a consistent API to launch jobs on all of the LLNL platforms (toss3, BlueOS, etc.)
- Provides job state introspection to notify when each stage of job is reached
- Caches run information in a node master key value store (kvs)
  - Can be queried while flux is live
- Allows for scheduling/launch on heterogenous architectures. Some jobs on CPU and some on GPU for Sierra.

The Merlin Ensemble combines the executable, environment and configuration into celery tasks

- The Merlin ensemble contains:
  - Configuration to run each simulation:
    - Environment
    - Resource request
    - Arguments
  - A set of variables and ranges combined into the sample set for the ensemble
  - Methods to convert the state into all necessary tasks including running, combining and saving the data

- Several methods can be used to run tasks including python, shell, MPI (slurm, lsf) or Flux launched

Merlin Demo: Configuration definition

- The celery app, ExexEnsemble and task definitions are imported from merlin and demo modules
- The mpi configuration for a single task is then defined

```
 1 #!/usr/bin/env python
 2 import os
 3 import sys
 4 import numpy
 5 from merlin.celery import app
 6 from merlin.common.ensemble.ensemble import
ExeEnsemble
 7 from merlin.common.ensemble.sample_interface
import FileSampleDescriptor, FileSampleGenerator
 8 from hello_tasks import HelloTask
 9 env = {}
10 flux = True
11 mpi_exe='hello.exe`'
12 mpi_exe_args={}
13 mpi=dict(mpi_tasks=1, mpi_nodes=1)
14 cfg = dict(
15     debug=True,
16     flux=flux,
17     env=env
18 )
```

Merlin Demo Task definition

- The Task definition for the Hello example will run the Hello executable with a set of parameters

```
"""The Demo exe Tasks and executable."""
from __future__ import absolute_import
from merlin.common.ensemble.bundled_tasks import
BundledTask
from merlin.common.executable import Executable

def runexe(**kwargs):
    """ Get the exe executable, set the output_path
for this
    sample set and run. This is the function merlin
will run.
    """
    mexe = kwargs.pop('exe', None)
    ret = mexe.start(**kwargs)
    return ret

class HelloTask(BundledTask):
    def __init__(self):
        super(HelloTask, self).__init__(runHello,
"hello")
```

Merlin Demo : Executable definition

- The configuration for a single Executable is is created, this excutable will be given a sample set to run by the Ensemble

```
21 exe = Executable(mpi,mpi_exe,mpi_exe_args,**cfg)
22 params = dict(
23     parm1=[1.3, 1.3],
24     foo=[3.3, 3.3],
25     bar=[5.5, 5.5],
26 )
27
28 OUTPUT_DIR = os.path.join(runpath, 'EXE_OUTPUT_D')
29 DEFAULT_SAMPLES = 10
30 BUNDLE_SIZE = 1
31 DIRECTORY_SIZES = [4, 2]
```

Merlin Demo: Samples and Ensemble setup

- The samples are written to a file from use by the Emsemble
- The Emsemble and Executable are configured using the setup dict

```
33 # The number of samples to run with.
34 try:
35     print("Input {0}".format(sys.argv[1]))
36     nsamples = int(sys.argv[1])
37 except IndexError:
38     nsamples = DEFAULT_SAMPLES
39 filename = "exesamples.npy"
40 naxes = 8
41 numpy.save(filename, numpy.random.uniform(0., 1.,
(nsamples, naxes)))
42 my_sample_descriptor = FileSampleDescriptor(filename)
43 my_samples =
FileSampleGenerator(my_sample_descriptor).generate()
44 # Workflow parameters and setup configurations.
45 setup = {
46     'samples': my_samples,
47     'outdir': OUTPUT_DIR,
48     'bundle_size': BUNDLE_SIZE,
49     'directory_sizes': DIRECTORY_SIZES,
50     'archive_path': ARCHIVE_DIR,
51     'exe': mpi_exe,
52     'parameters': params
53 }
```

Merlin Demo: Ensemble definition and run

- The ExeEmsemble contains the methods to launch the job and the specific task that will be run and the methods for creating the sample list
- The Ensemble is then run and waits for all tasks to complete before archiving the results

```
 54 ensemble = ExeEnsemble('Demo
Ensemble','Demo', HelloTask, **setup)
 55 print ("Queueing tasks...")
 56 ensemble.setup()
 57 print ("waiting for tasks (worker
needs to be live for this to return)\n"
 58        "This test assumes the worker
has access to same file system as\n"
 59        "this test")
 60 ensemble.run()
 61 ensemble.waitAll()
 62 ensemble.printStats()
 63
 64 print ("archiving")
 65 ensemble.archive()
```

FluxExecutable: payload definition

- The payload for the flux rpc request defines the variables necessary to launch the parallel job

```
105     def get_payload(self, eargs):
106         """Create the flux process payload which is the
job description for a
107         single MPI instance of the executable."""
108
109         compute_jobreq = {
110             'cmdline': args,
111             'ntasks': self.tasks,
112             'cwd': cwd,
113             'environ': cenv,
114             'walltime': self.walltime,
115             'ngpus': self.ngpus,
116         }
117
118         if self.nodes:
119             compute_jobreq['nnodes'] = self.nodes
120
121         if self.cores:
122             compute_jobreq['ncores'] = self.cores
123
124         return json.dumps(compute_jobreq)
```

FluxExecutable: Running the simulation with rpc calls

- This run method will get the payload and submit the request then wait for one of the states defined in the jsc_cb to occur

```
172     def run(self, eargs):
173         """Launch the Flux process of the exe and wait
174         for the return message.
175         """
176         # The flux module is provided by the flux environment.
177         # This environment isn't available until the task is run
178         # by the worker which is run under flux.
179         import flux
180         import flux.kvs as kvs  # key,value store
181         from flux import jsc, kz  # Job status and control and kvs zio interface
182         result = {}
183         # Create the flux handle, this is unpickleable
184         fh = flux.Flux()
185         # Register the callback in the local flux handle (fh)
186         # Send self through so the jobid can be read from self.jobid
187         # once it is known after the submission
188         jsc.notify_status(fh, FluxExecutable.jsc_cb, (fh, self))
189         payload = self.get_payload(eargs)
190         # Submit the job, this is async.
191         resp = fh.rpc_send("job.submit", payload)
192         self.jobid = resp['jobid']
193         kvs_path = resp['kvs_path']
194         rootstdout = StringIO()
195         kz.attach(fh, kvs_path + ".0.stdout", rootstdout)
196         # Call the reactor , the reactor will block until
197         # the callback stops it.
198         fh.reactor_run(fh.get_reactor(), 0)
199         # Get the job lwj from the flux kvs.
200         jobkvs = kvs.get_dir(fh, kvs_path)
201         jobkvskeys = jobkvs.keys()
202         if jobkvs['state'] == 'complete':
203             pass
204         return result
```

FluxExecutable: job status and control callback

- The job status and control (jsc) callback method will inform the celery task when the simulation enters the complete, cancelled or failed states

```
83      @staticmethod
84      def jsc_cb(jcbstr, args, errnum):
85          """ The job status and control callback.
86          This function will be called everytime a job known by
87          self.fh changes state.
88
89          Modified from flux-framework/flux-workflow-examples/example7,
90          by Dong Ahn.
91          """
92          from flux import jsc  # Job status and control
93          f, the_self = args
94          jcb = json.loads(jcbstr)
95          jobid = jcb['jobid']
96          state = jsc.job_num2state(
97              jcb[jsc.JSC_STATE_PAIR][jsc.JSC_STATE_PAIR_NSTATE])
98          if the_self.debug:
99              print("flux.jsc: job", jobid, "changed its state to ", state)
100             print("flux.jsc: monitoring ", the_self.jobid)
101         states = ["complete", "cancelled", "failed"]
102         if jobid == the_self.jobid and state in states:
103             f.reactor_stop(f.get_reactor())
```

Running the Demo:

- Create a redis server to provide the broker and backend service
- Start flux and then celery workers using flux
- Run the executable ensemble through the celery workers

Flux: Real-World Practices and Examples – Part 2

Dong H. Ahn, **Stephen Herbein**, Joe Koning, Tapasya Patki, Tom Scogland

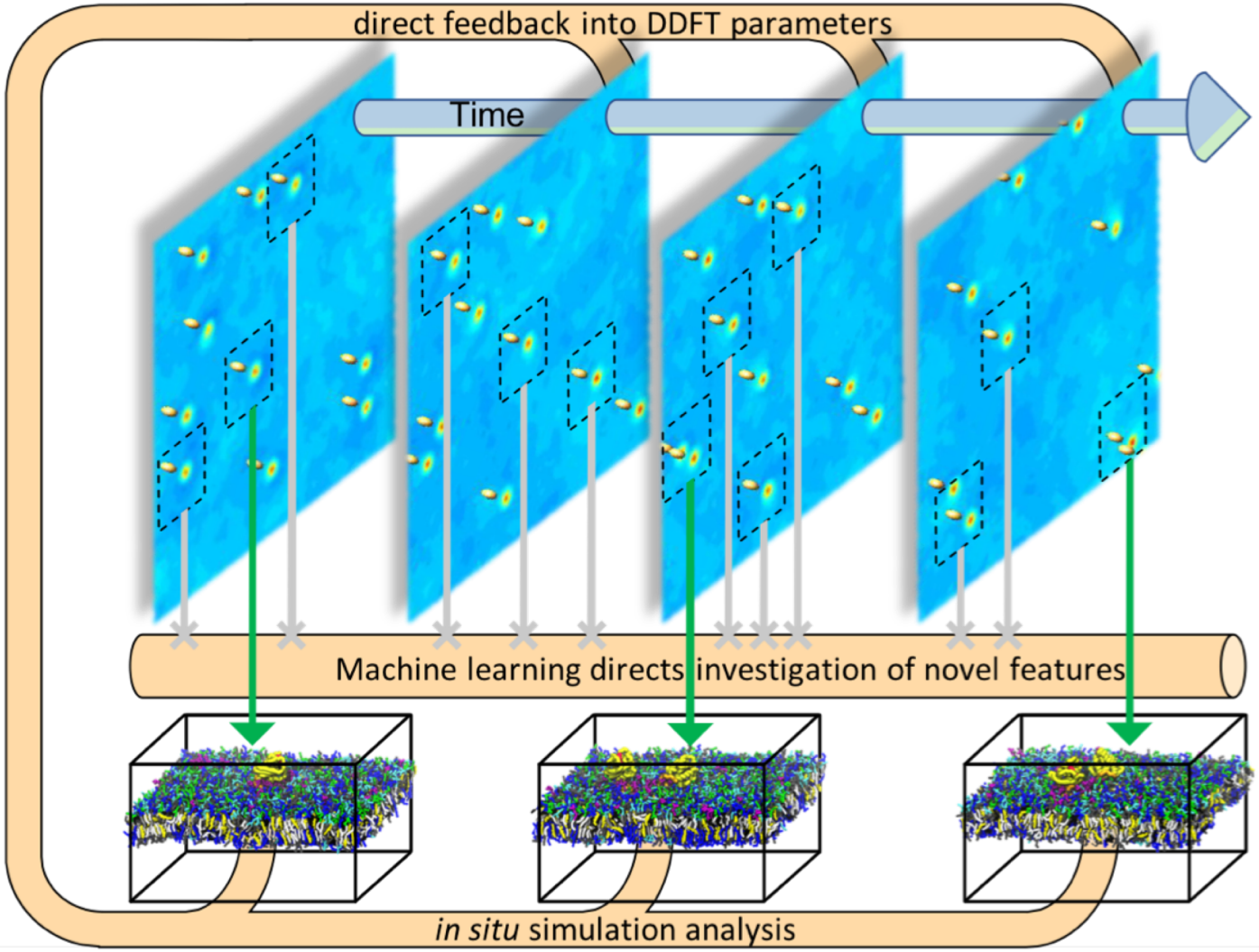
January 16, 2019

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

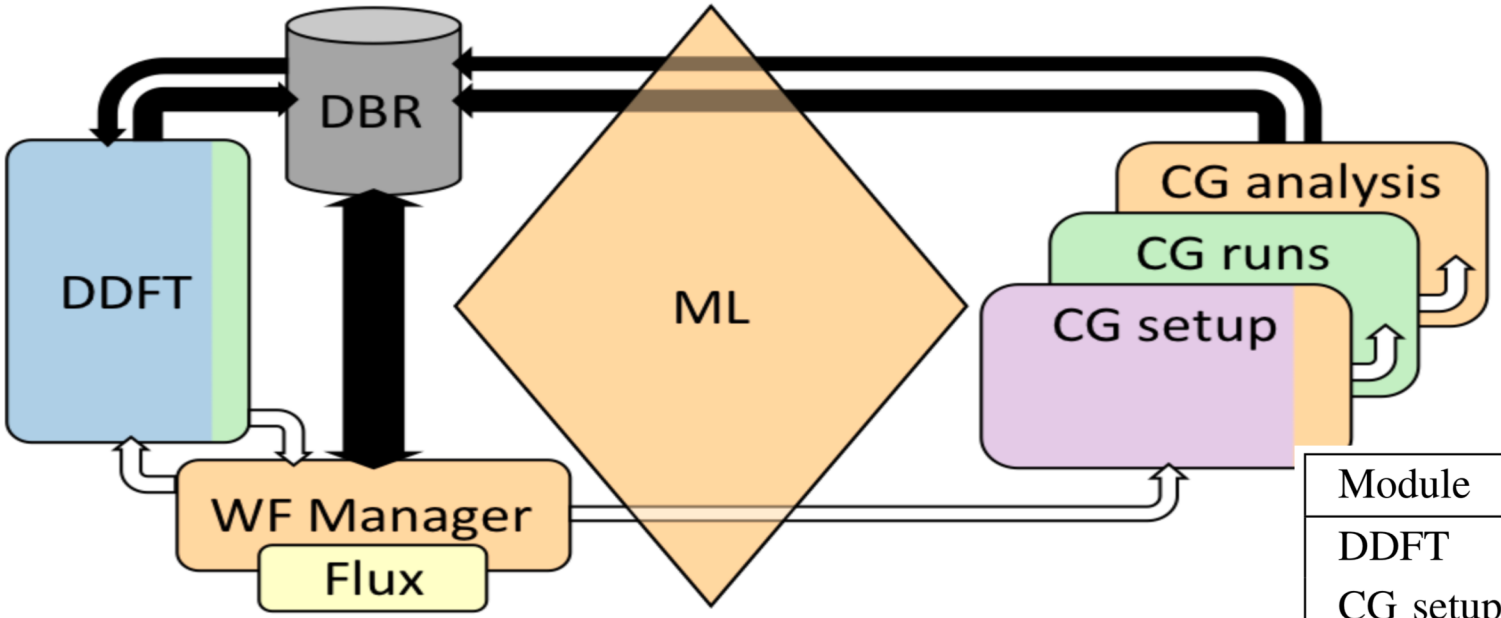
Lawrence Livermore National Laboratory

Pilot2 Workflow Overview



- Cellular-scale continuum model simulates large spatial and temporal scales
- Machine learning model selects novel patches to simulate in molecular detail
- Microscale results fed back into macroscale

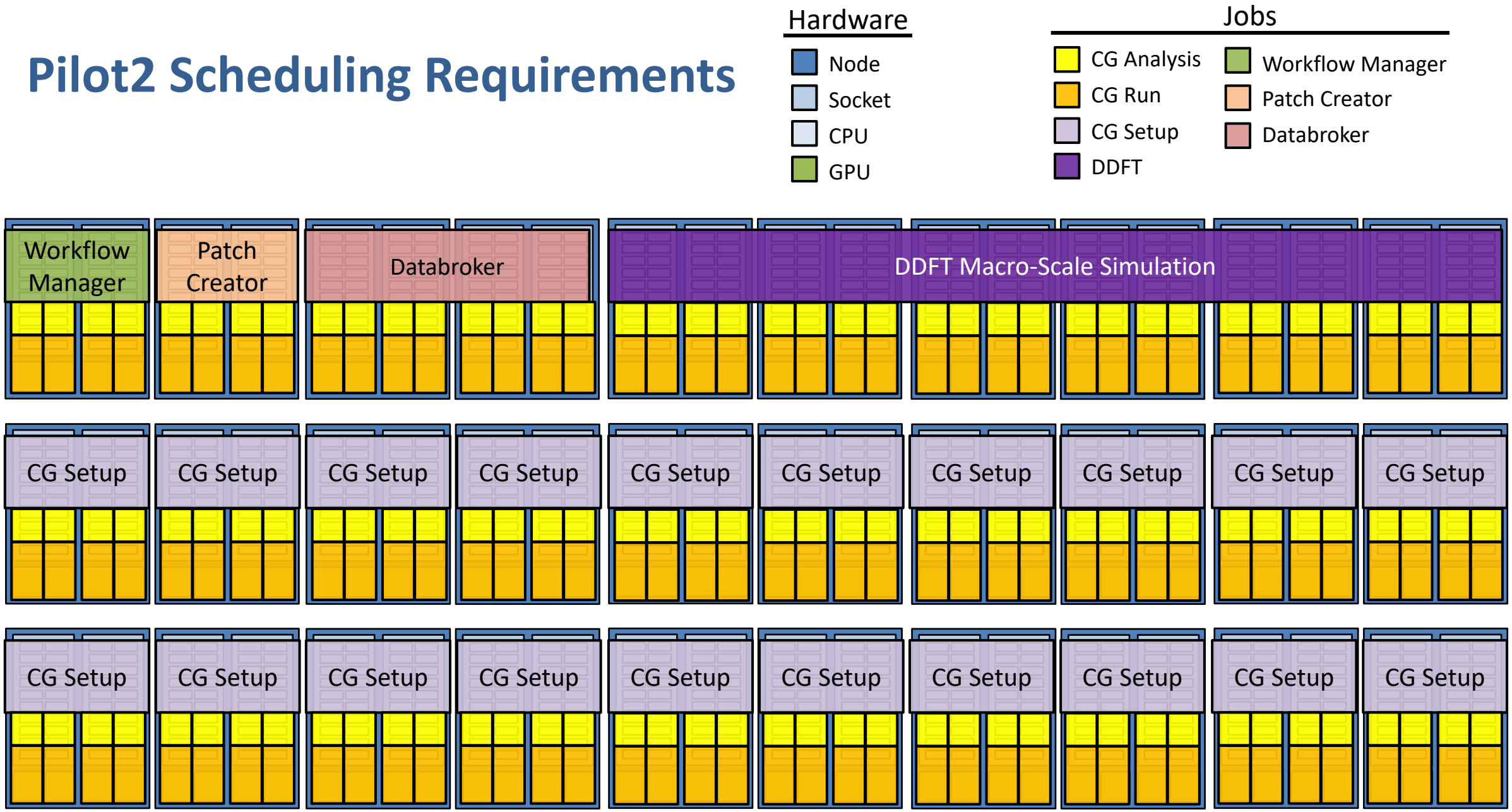
Pilot2 Scheduling Requirements



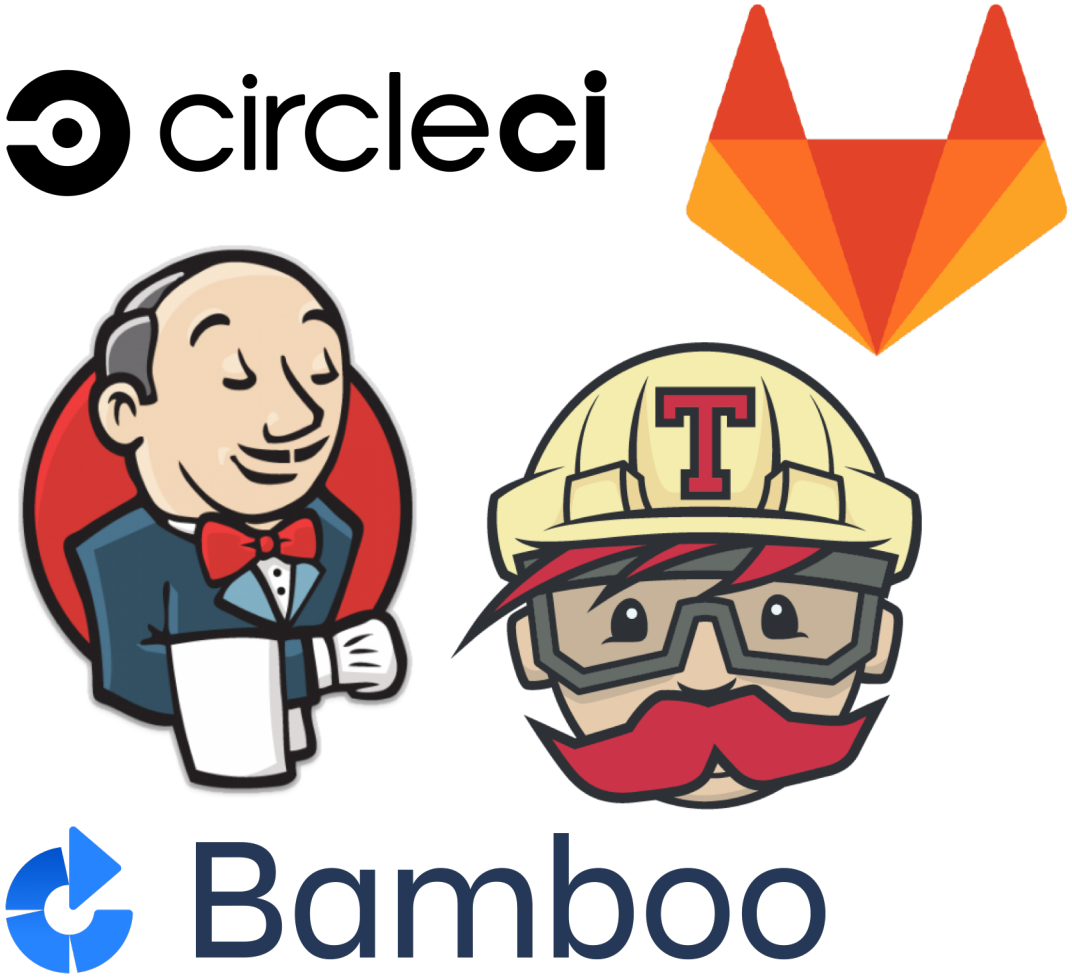
- Many co-scheduled components, some are co-located
- Each with their own unique resource requirements

Module	# Jobs	# Nodes	# CPU cores	# GPUs
DDFT	1	1000	1000×24	0
CG setup	2528	2528	2528×24	0
CG runs	3540×4	3540	3540×8	3540×4
CG analysis	3540×4	3540	3540×12	0
DBR	1	10	10×24	0
PatchCreator	1	1	1×24	0
WFManager	1	1	1×24	0
Total	30852	3540	3540×44	3540×4

Pilot2 Scheduling Requirements

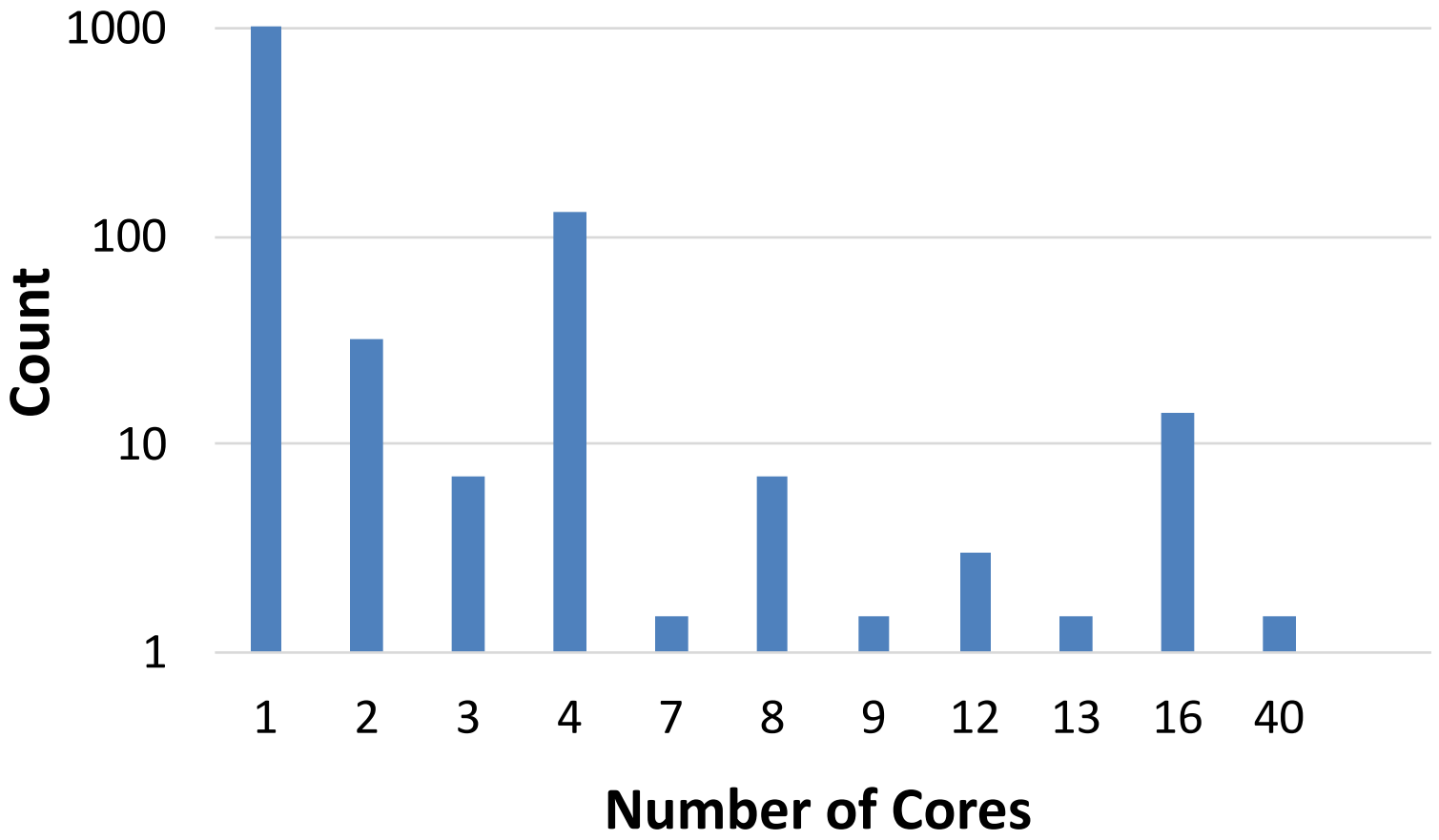


Automated Testing System (ATS) Overview



- "CI for HPC"
- Run regression and performance tests of MPI application that may require multiple cores
- Test in the same environment (hardware and software) that is used for production runs
- Generate nightly reports for developers
- Useful for comparing performance when compilers or system software change

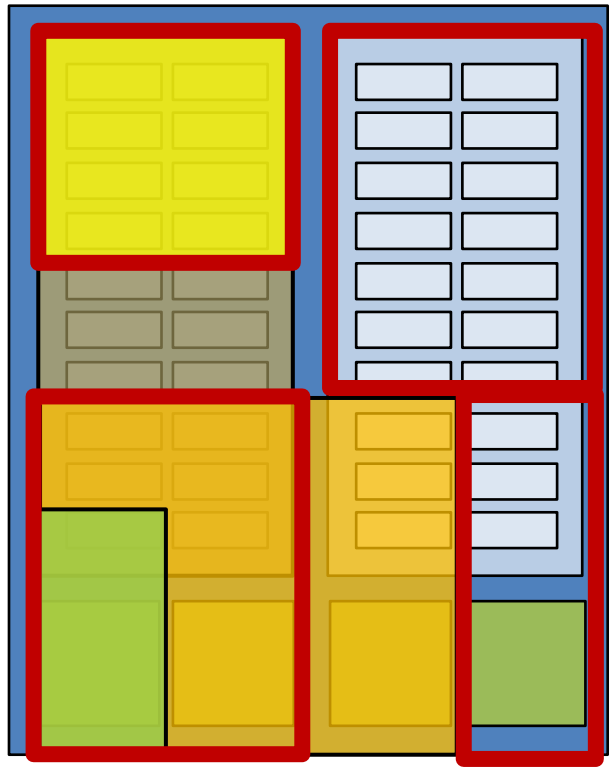
ATS Scheduling Requirements



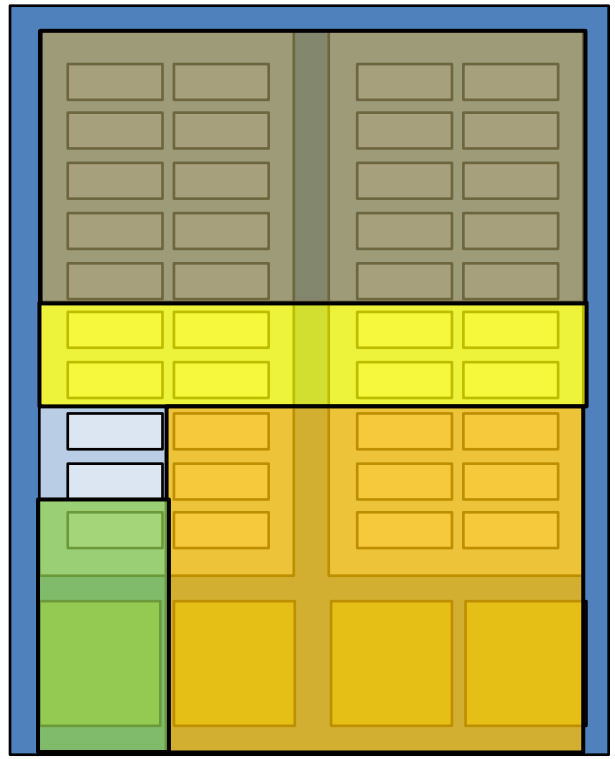
- Launch all tasks without oversubscribing cores
- Minimize wall time of testing suite
- Maximize resource utilization
- Portable across multiple schedulers and resource managers

ATS Scheduling Requirements

Traditional HPC Scheduler's
Job Step Packing



Flux's "Job Step" Packing



We saw **3x-10x speedups** when using Flux

Flux's Resource Data Model and Performance Variation-Aware Scheduling

ECP Annual Meeting, Jan, 2018

Dong H. Ahn and Tapasya Patki

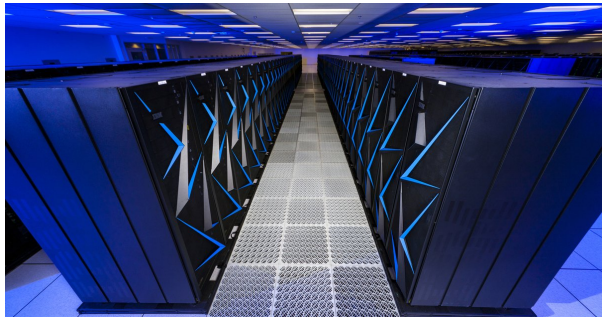
LLNL-PRES- 765450

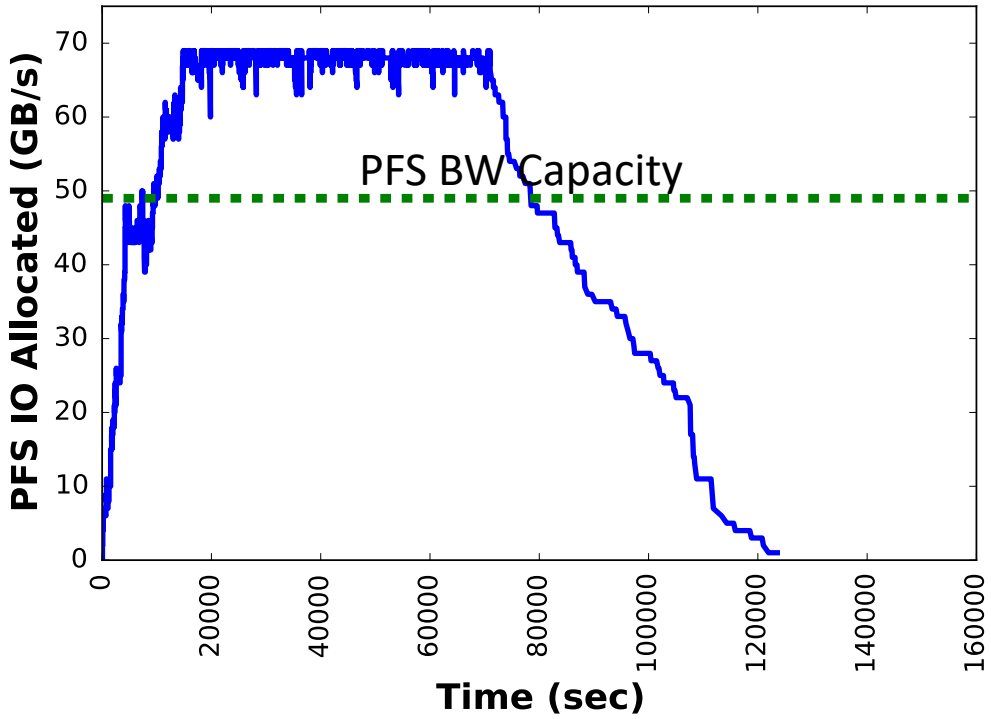
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

The changes in resource types are equally challenging.

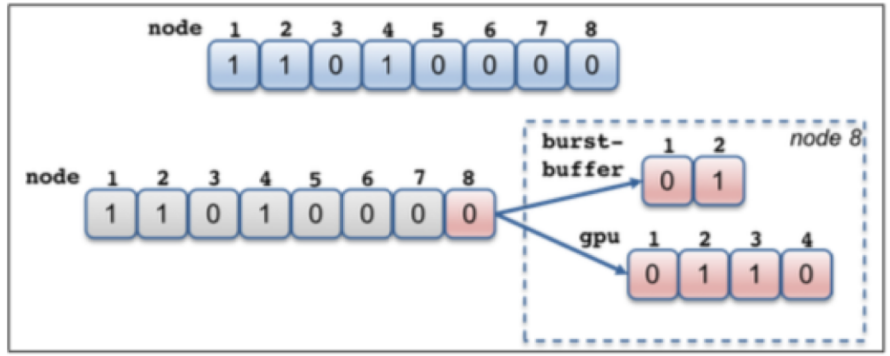
- Problems are not just confined to the workload/workflow challenge.
- Resource types and their relationships are also becoming increasingly complex.
- Much beyond compute nodes and cores…
  - GPGPUs
  - Burst buffers
  - I/O and network bandwidth
  - Network locality
  - Power





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

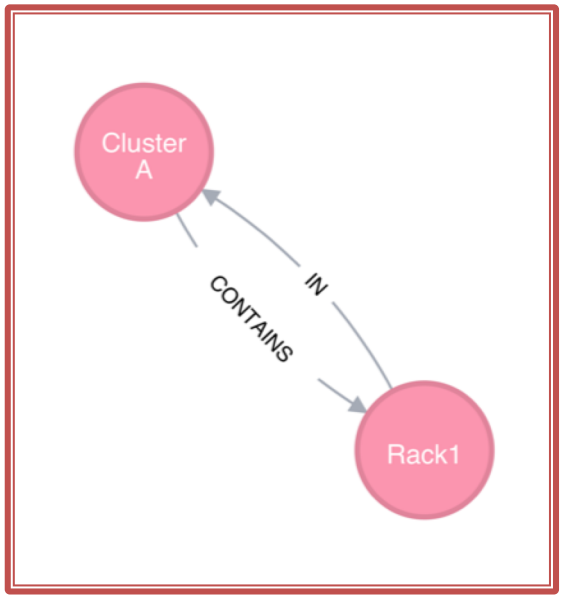
- Designed when the systems are much simpler
  - Node-centric models
  - SLURM: bitmaps to represent a set of compute nodes
  - PBSPro: a linked-list of nodes



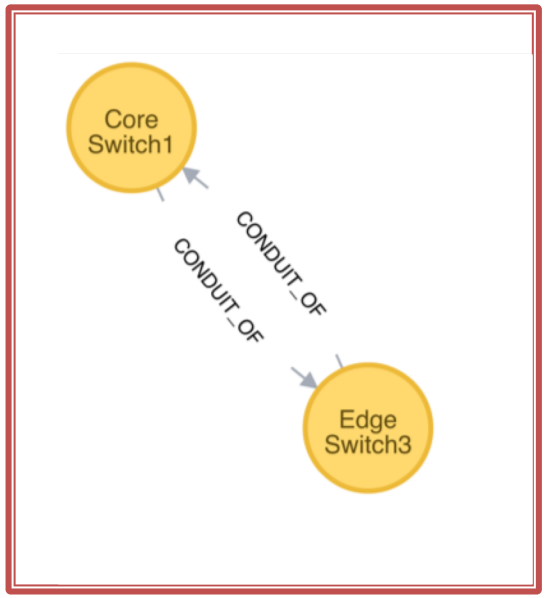
- HPC has become far more complex
  - Evolutionary approach to cope with the increased complexity
  - E.g., add auxiliary data structures on top of the node-centric data model
- Can be quickly unwieldy
  - Every new resource type requires new a user-defined type
  - A new relationship requires a complex 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



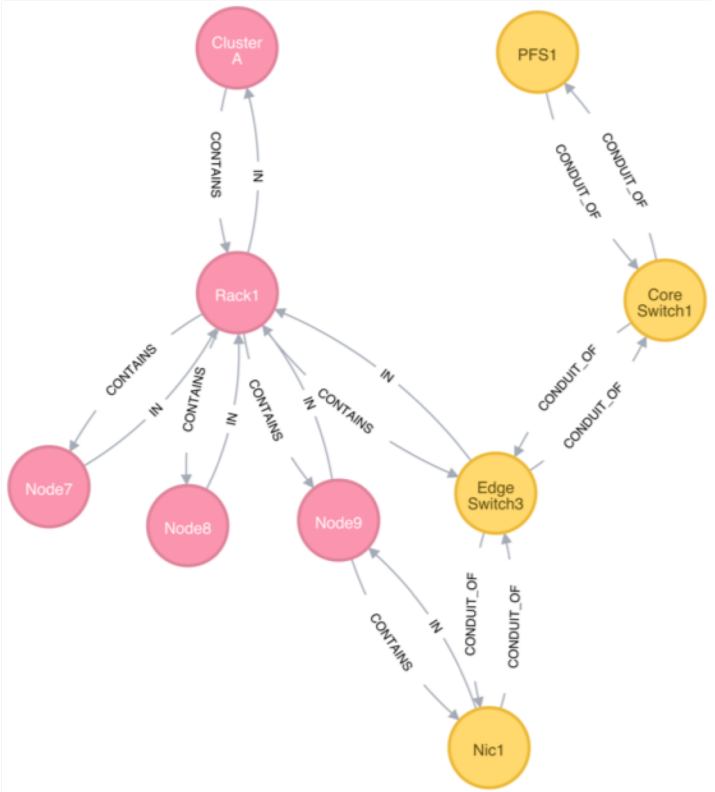
Containment subsystem



Network connectivity subsystem

Our graph-based resource data model reduces the complexity of our scheduler.

- Highly composable to support a graph with arbitrary complexity
  - Simply add new vertices and connectivity to compose
- The scheduler remains to be a highly generic graph code.
  - No code change is required on new resource types and relationships.



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

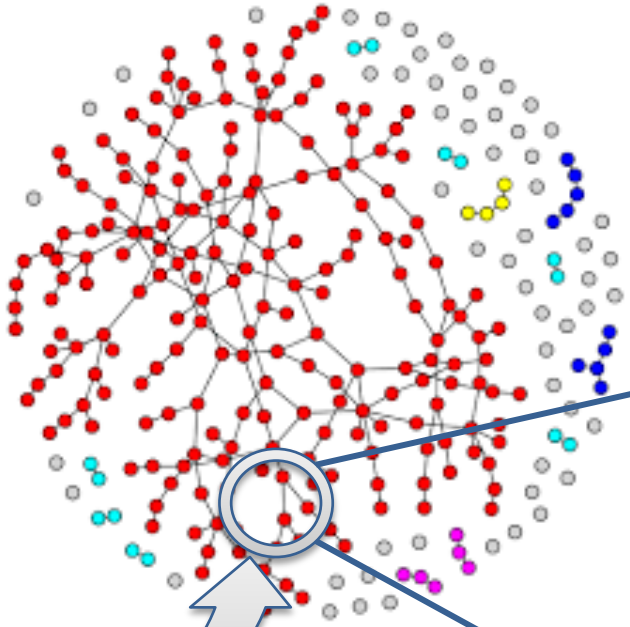
A jobspec example

```
1  version: 1
2  resources:
3    - type: cluster
4      count: 1
5      with:
6        - type: rack
7          count: 2
8          with:
9            - type: slot
10             label: myslot
11             count: 3
12             with:
13               - type: node
14                 count: 1
15                 with:
16                   - type: socket
17                     count: 2
18                     with:
19                       - type: core
20                         count: 18
21
22 # a comment
23 attributes:
24   system:
25     duration: 3600
26 tasks:
27   - command: app
28     slot: myslot
29     count:
30       per_slot: 1
```

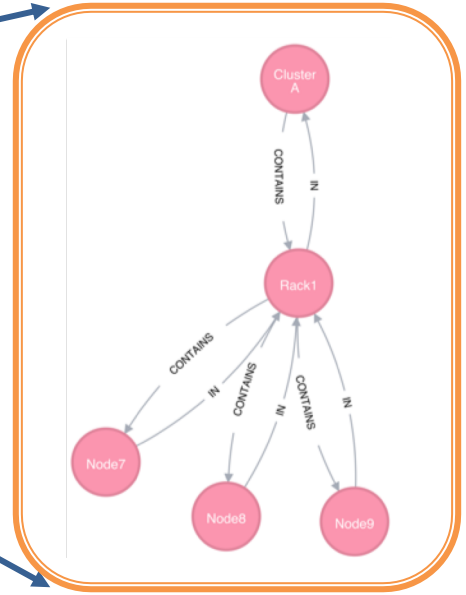
- YAML for human readability
- End users will mostly use a more user-friendly notation
- cluster->racks[2]->slot[3]->node[1]->sockets[2]->core[18]
- ***slot*** is the only non-physical resource type
  - Represent a schedulable place where program process or processes will be spawned and contained
- Referenced from the tasks section

Flux maps our complex scheduling problems into graph matching problems.

```
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
```

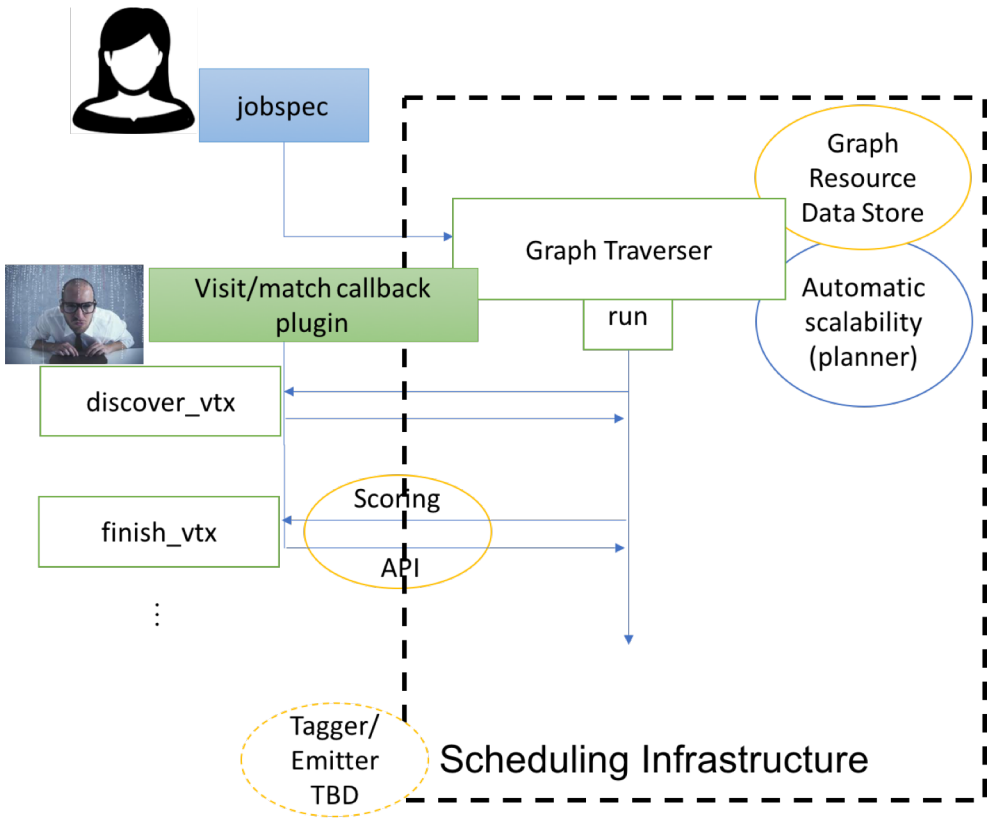


Traverse, match and score



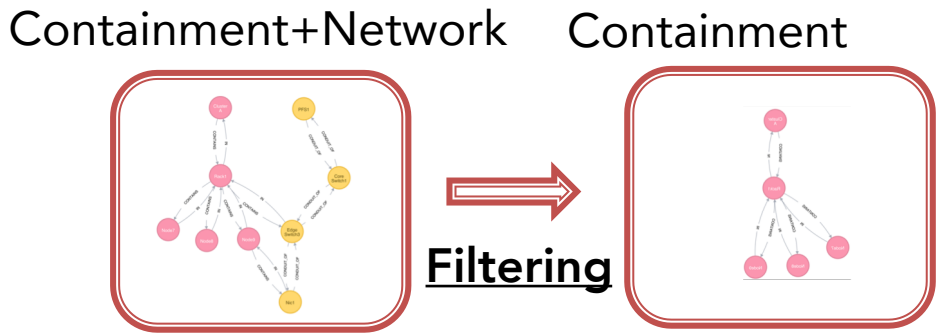
Our architecture embodies our graph matching scheme with good separation of concerns and extensibility.

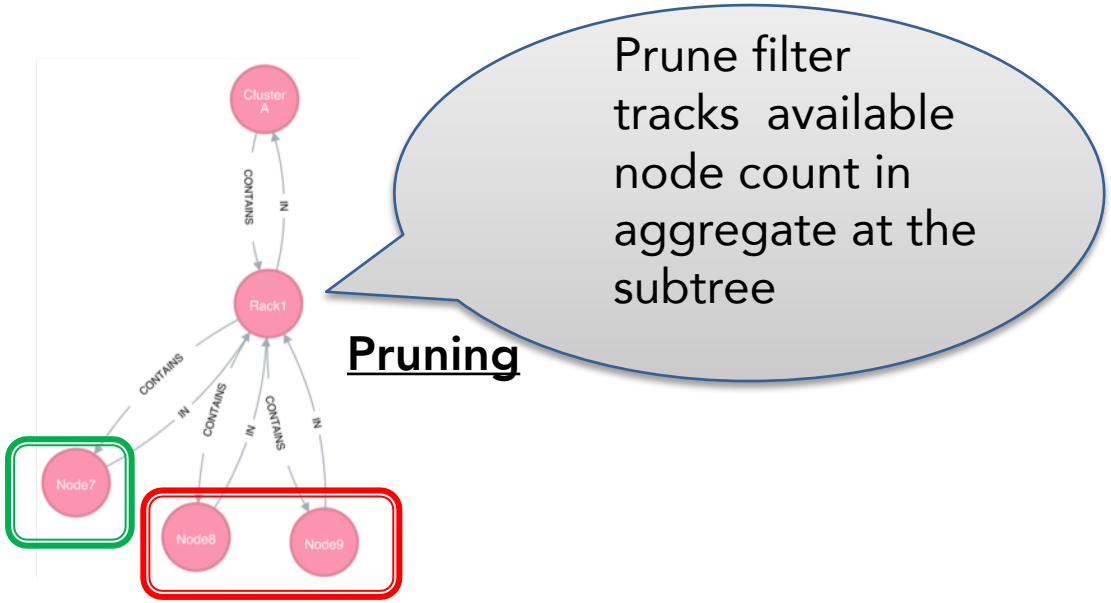
- Allow our resource module to populate and maintain the total graph of the instance.
  - Graph resource data store
- On receiving each job-spec object:
  - Start to traverse the graph with a known visit type (currently DFV and DFU).
  - On each visit event (e.g., post-order visit)
    - a policy match callback is invoked to score the visiting vertex
    - high- or low-id first, locality-aware, and performance variation-aware policies
- Best matching resources are selected at the end of the traversal.



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





Overview

- Problem:
  - Performance variation in applications
  - Worsening as we scale out
- Potential solutions:
  - Advanced scheduling
  - Job-level runtime systems
  - Application-level load balancing
- Our approach:
  - Use Flux's flexible resource data model to design scheduling policies

Sources of variation

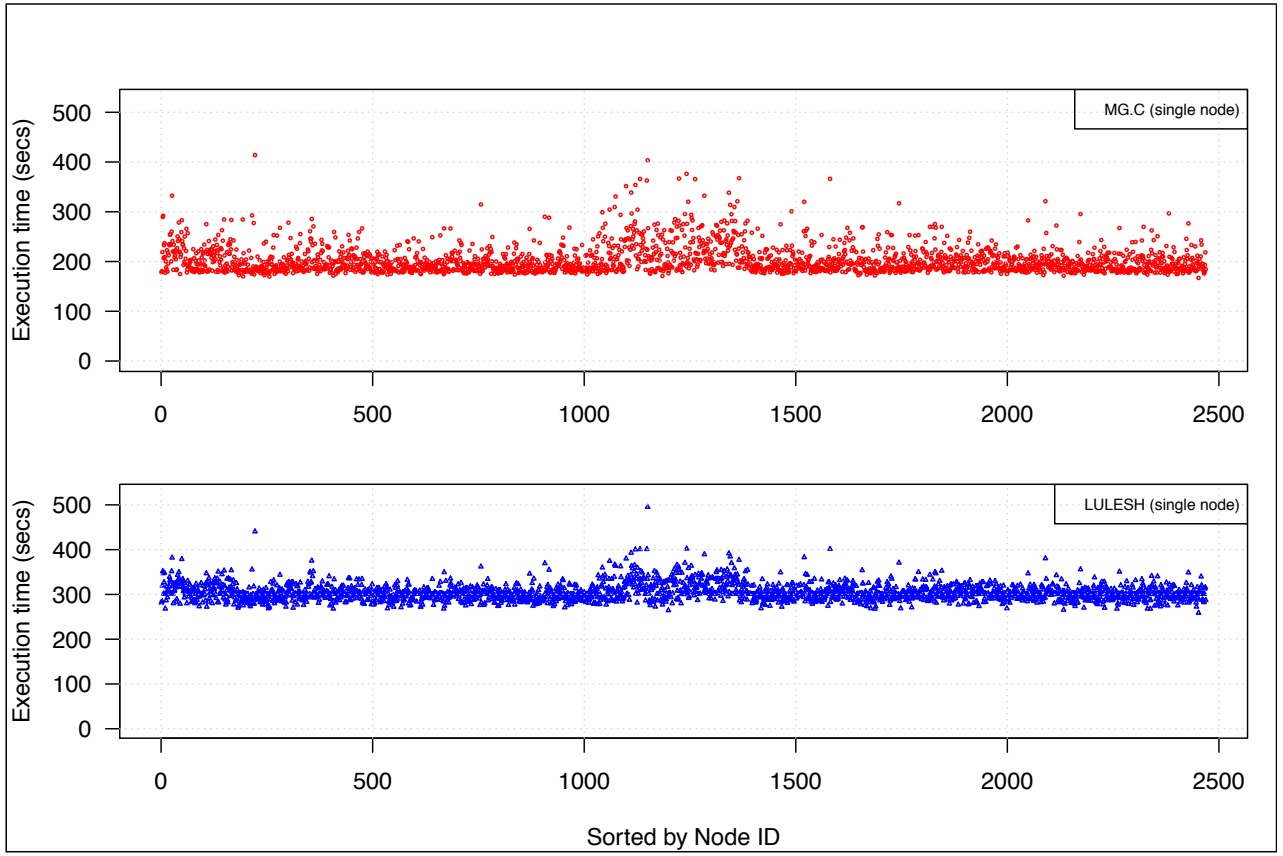
- Deterministic sources
  - Based on hardware
  - Can be understood statically, are reproducible, can be predicted
  - Offline analysis is possible
  - Do not depend on dynamic user environments or job mix
  - *Examples*: component-level heterogeneity, processor manufacturing or aging

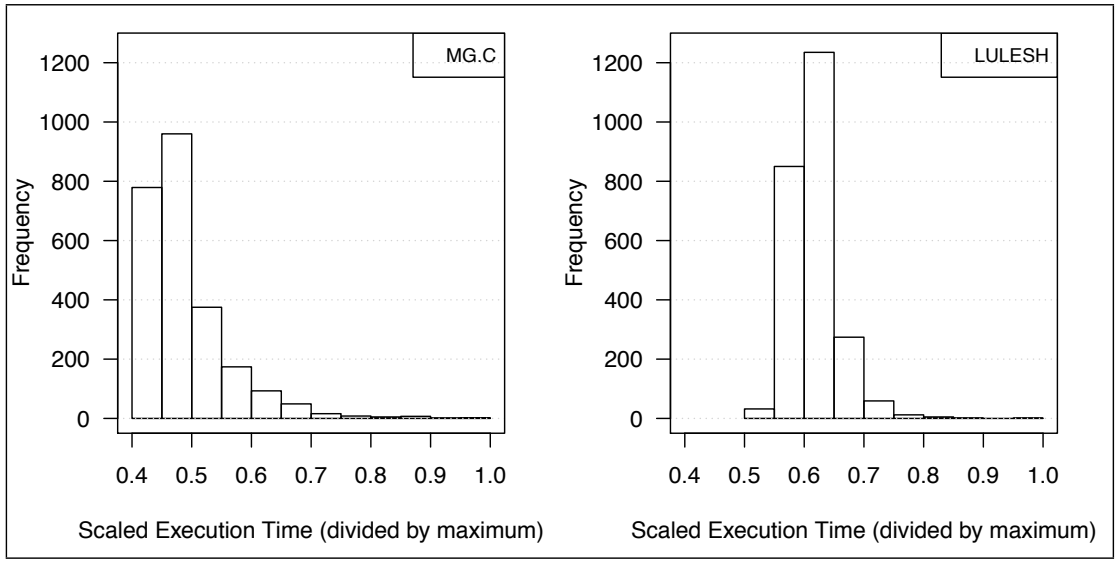
- Non-deterministic sources
  - Not reproducible and cannot be understood statically
  - Depend on specific workloads, performance of neighboring jobs, current job mix, network or IO congestion, and user or system parameters
  - Online monitoring and runtime adjustment is required

Types of variation

- Rank-to-rank variation
  - Occurs within the application resulting in unforeseen slowdowns and load imbalance
  - Application performance bound by slowest task
- Run-to-run variation
  - Subsequent executions of the same application get vastly distinct allocations, lack reproducibility
- First cut: Deterministic, rank-to-rank variation resulting from processor manufacturing variability in power-limited scenarios
- Can be extended easily to other deterministic sources and for run-to-run variation (with or without power caps)

Real world example of variation: Quartz cluster, 2469 nodes, 50 W CPU power per socket





- 2.47x difference between the slowest and the fastest node for MG
- 1.91x difference for LULESH.

Statically determining node performance classes

- Ranking every processor is not feasible from point of view of accounting as well as application differences

- 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 power cap

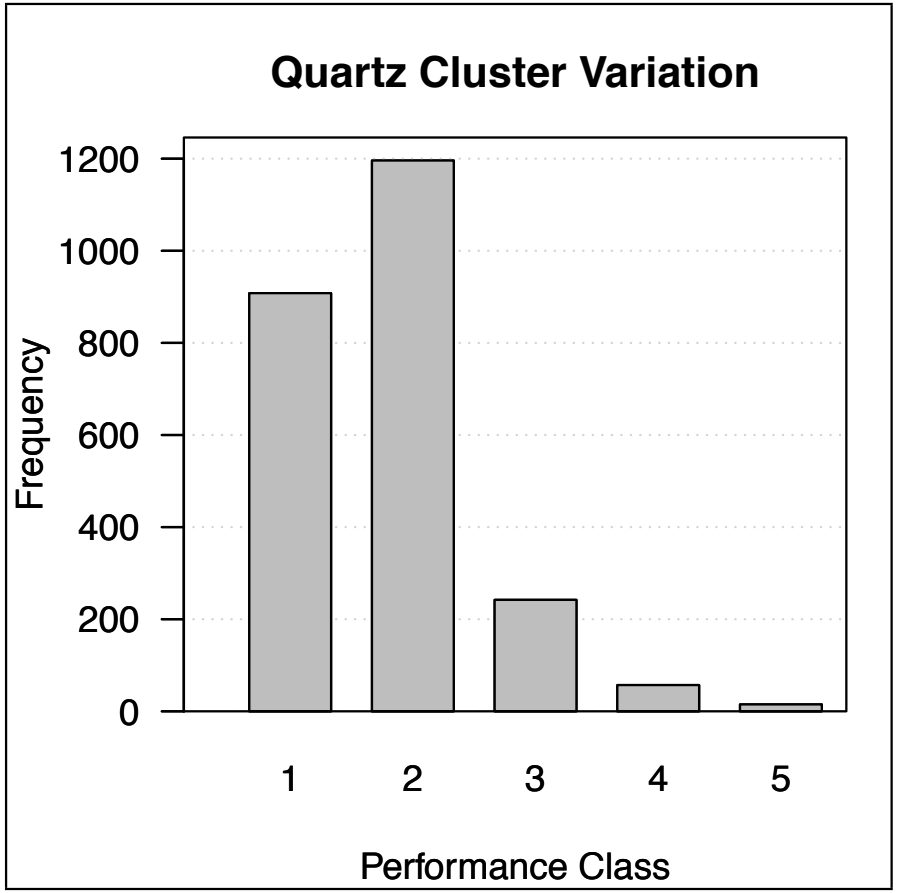
- Mitigation
  - Rank-to-rank: minimize spreading application across performance classes
  - Run-to-run: allocate nodes from same set performance classes to similar applications

Statically determining node performance classes: 2469 nodes of Quartz

$$t_{combined_i} = \frac{\frac{t_{MG_i}}{median(t_{MG_{1:n}})} + \frac{t_{LULESH_i}}{median(t_{LULESH_{1:n}})}}{2}$$

$$t_{norm_j} = \frac{t_{combined_j} - min(t_{combined_j})}{max(t_{combined_j}) - min(t_{combined_j})}$$

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

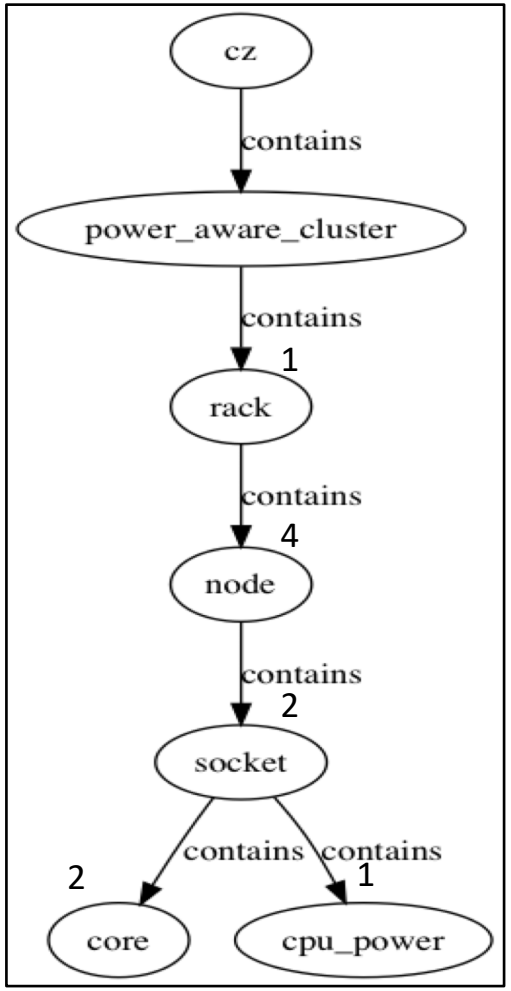


Measuring impact of variation-aware scheduling

$$P_j := \{p_a | a \in n \wedge allocated(a, j)\}$$
$$fom_j = max(P_j) - min(P_j)$$

- *allocated(a,j)* returns true if node *a* has been allocated to job *j*
- P_j is the set of performance classes of the nodes allocated to job *j*
- Figure of merit, fom_j, is a measure of how widely the job is spread across different performance classes
- For a job trace, we will look for number of jobs with low figure of merit

Implementation with Flux's resource model: Resource graph, the layout of the cluster



Input format: graphml (GRUG)

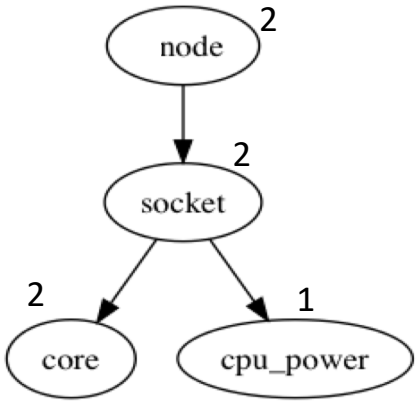
Defined once when the cluster is brought up

Example resource graph with 4 nodes per rack, 2 sockets per node, 2 cores per socket and 1 unit of cpu_power (e.g. 100 W)

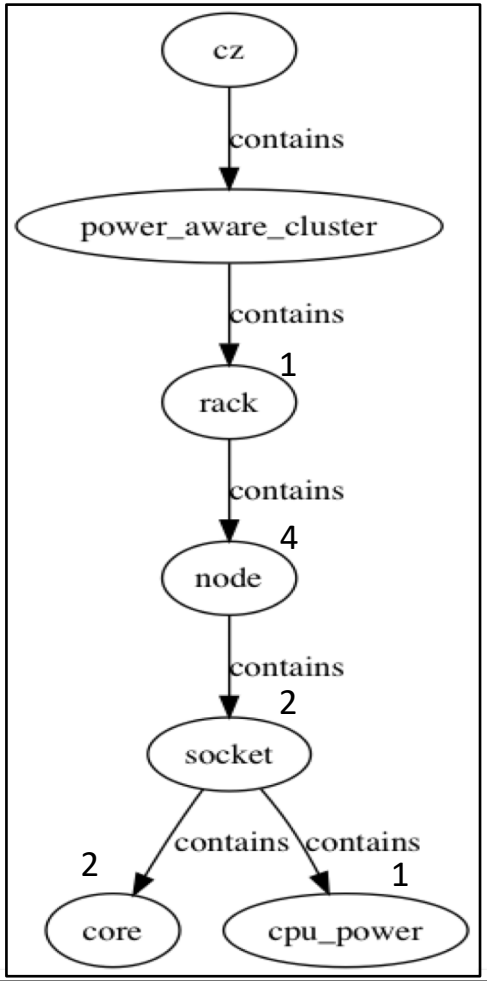
Flux's resource-service:
Users submit job requests, depth first traversal for matching

Example job request

```
resources:
  - type: node
    count: 2
    with:
    - type: slot
      label: default
      count: 2
      with:
        - type: socket
          count: 1
          with:
            - type: core
              count: 2
            - type: cpu_power
              count: 1
# a comment
attributes:
  system:
    duration: 100000
tasks:
  - command: default
    slot: socketlevel
    count:
      per_slot: 1
```



Resource Graph



- Match root of job request with vertex in resource graph
- Depth first traversal to allocate resources
- Backtrack if resources are unavailable, and reserve for future

```
INFO: Loading a matcher: CA
      ------------------core0[1:x]
      ------------------core1[1:x]
      ------------------cpu_power0[130:x]
      --------------socket0[1:x]
      ------------------core2[1:x]
      ------------------core3[1:x]
      ------------------cpu_power1[130:x]
      --------------socket1[1:x]
      ------------node0[1:s]
      ------------------core0[1:x]
      ------------------core1[1:x]
      ------------------cpu_power0[130:x]
      --------------socket0[1:x]
      ------------------core2[1:x]
      ------------------core3[1:x]
      ------------------cpu_power1[130:x]
      --------------socket1[1:x]
      ------------node2[1:s]
      ---------rack0[1:s]
      ------power_aware_cluster0[1:s]
      ---cz0[1:s]
INFO: ============================
INFO: JOBID=1
INFO: RESOURCES=ALLOCATED
INFO: SCHEDULED AT=Now
INFO: ELAPSE=0.001321
INFO: ============================
```

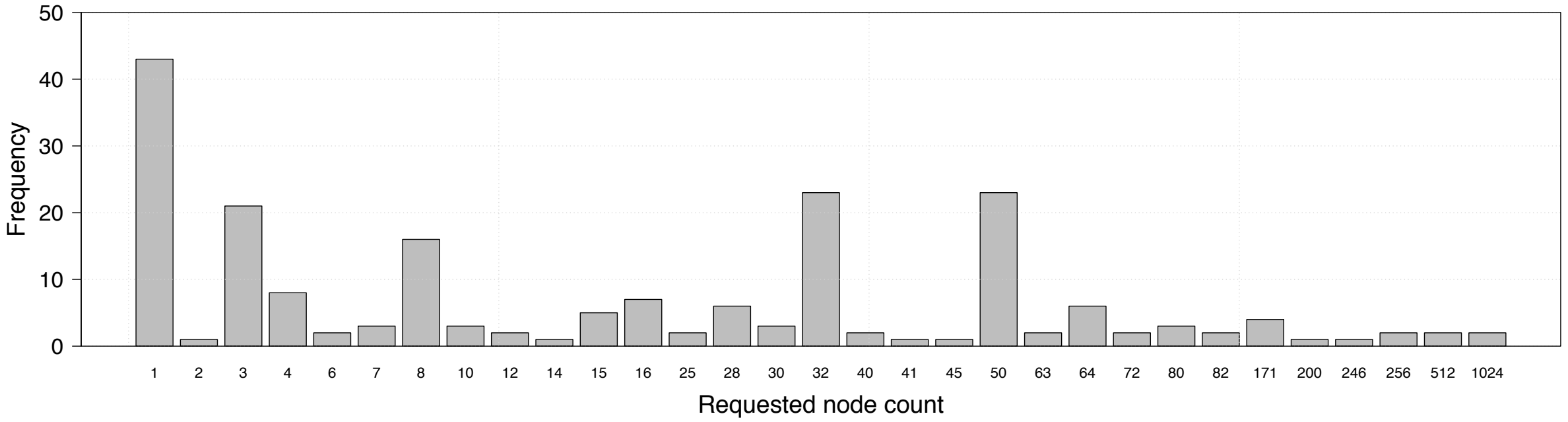
Example output

Adding a new scheduling policy is simple, <250 LOC

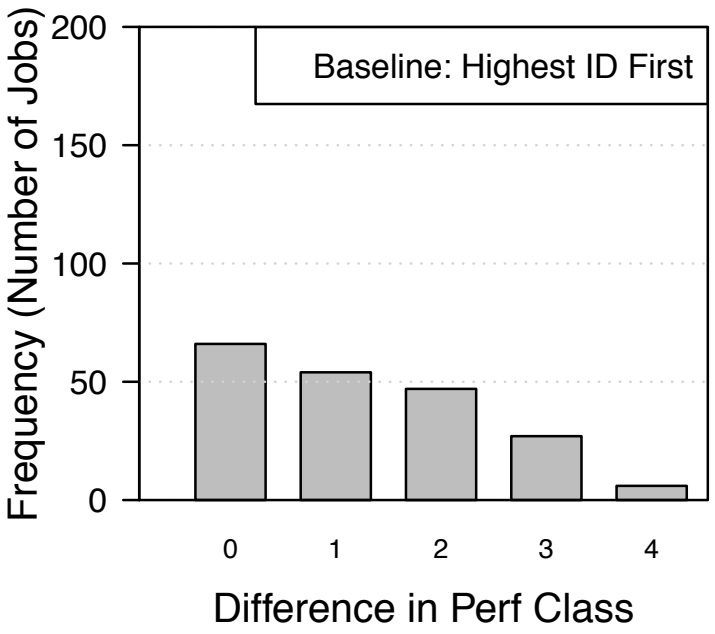
- Depth-first upwalk for graph matching
- Implement by deriving from the base matcher callback class (`dfu_match_cb_t`) and by overriding one or more of its virtual methods.
- The DFU traverser's `run()` method calls back these methods on well-defined graph vertex visit events. Supported events:
  - Preorder, postorder, slot, and finishgraph events on the selected dominant subsystem
  - Preorder and postorder events on one or more selected auxiliary subsystems
- Performance class information was captured through other headers
  - Can be specified as input CSV

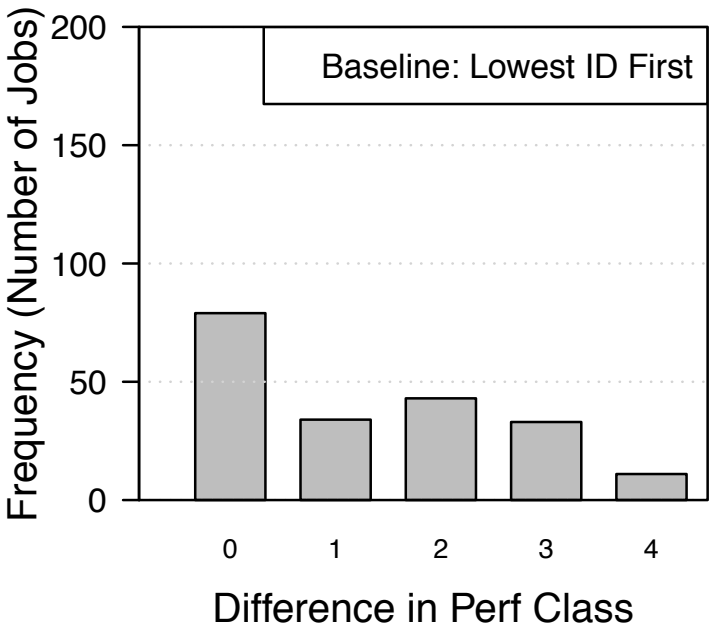
Evaluation: Real job trace from Quartz cluster at LLNL

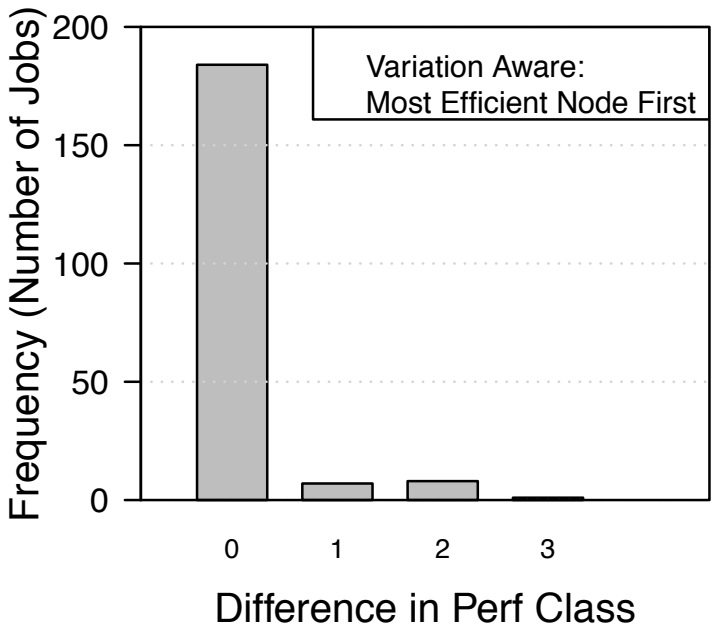
- 200 jobs captured as a trace from quartz cluster



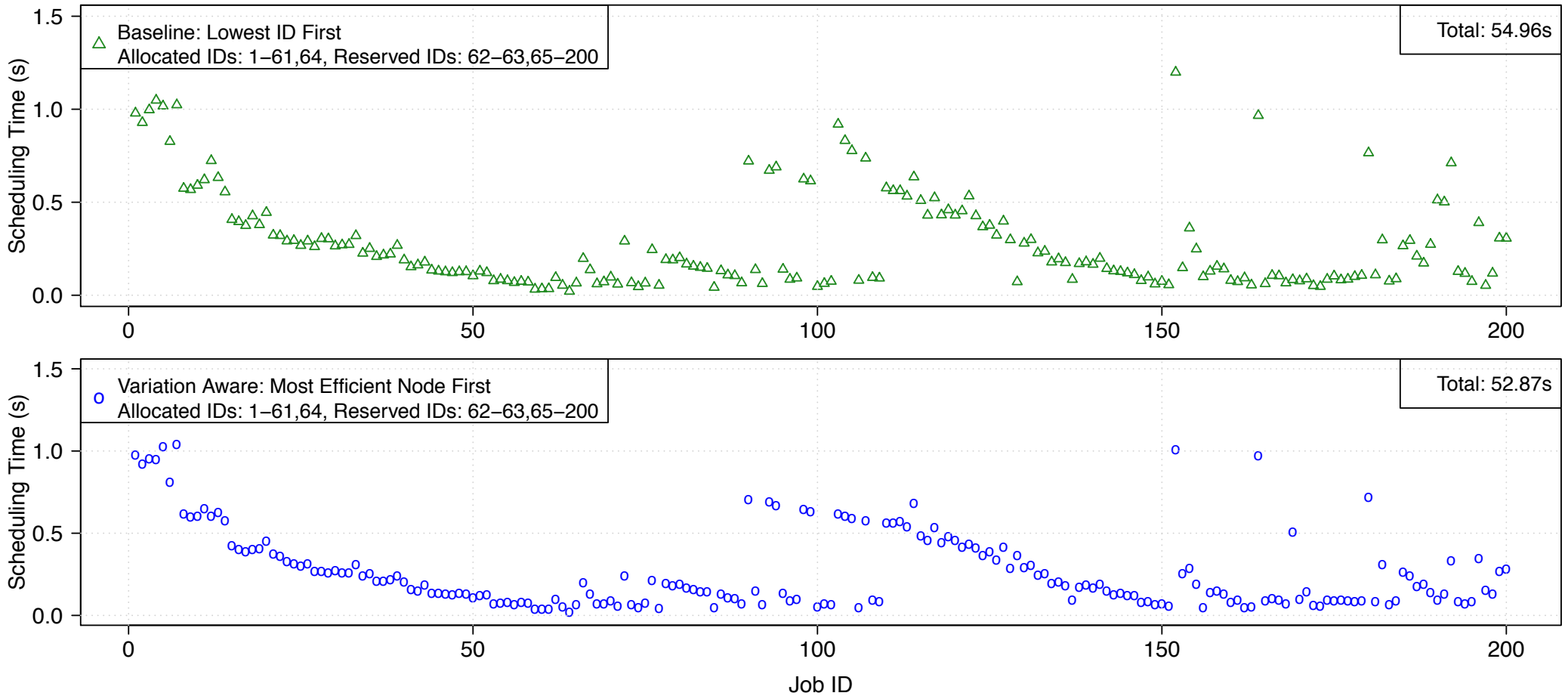
Variation-aware scheduling results in 2.4x reduction in rank-to-rank variation in applications







Pruning intelligently can lead to good scaling and low overhead for the scheduling policy



Flux: Hands-on Workflow Examples

Dong H. Ahn, Stephen Herbein, Joe Koning, Tapasya Patki, Tom Scogland

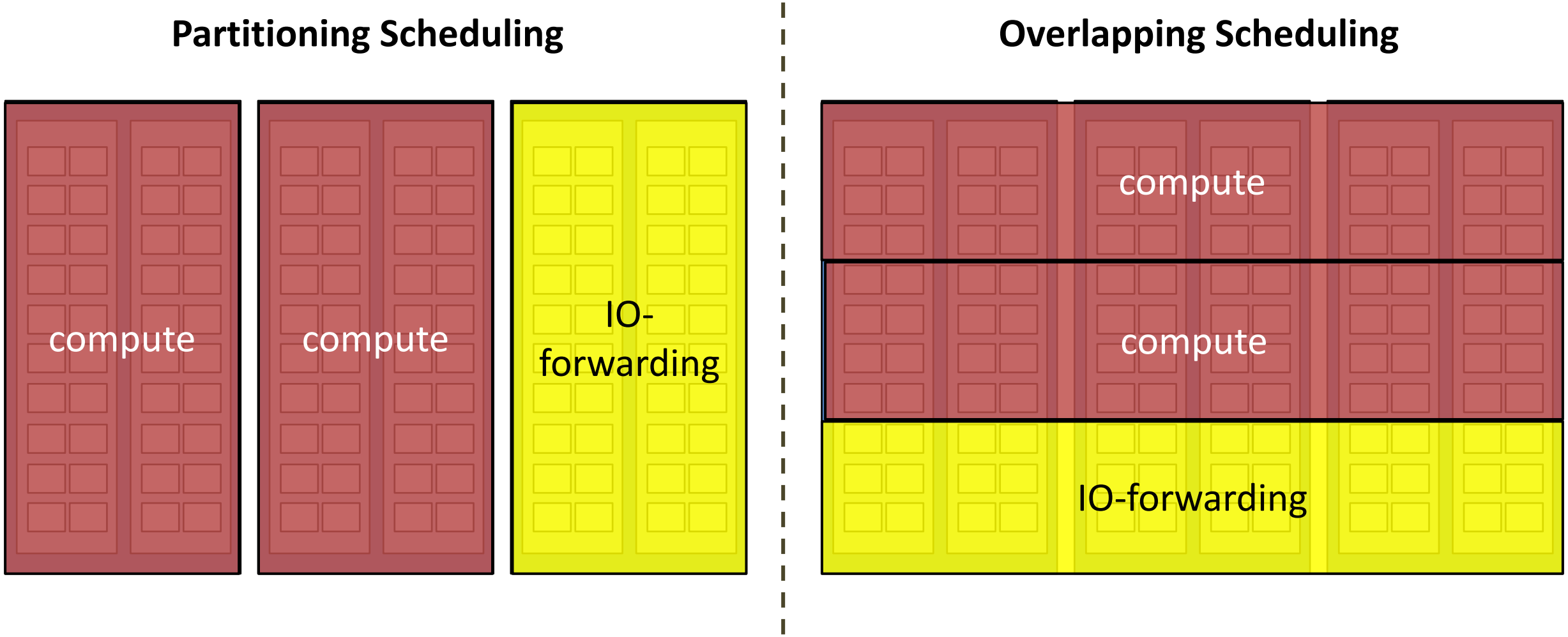
January 16, 2019

LLNL-PRES- 765450

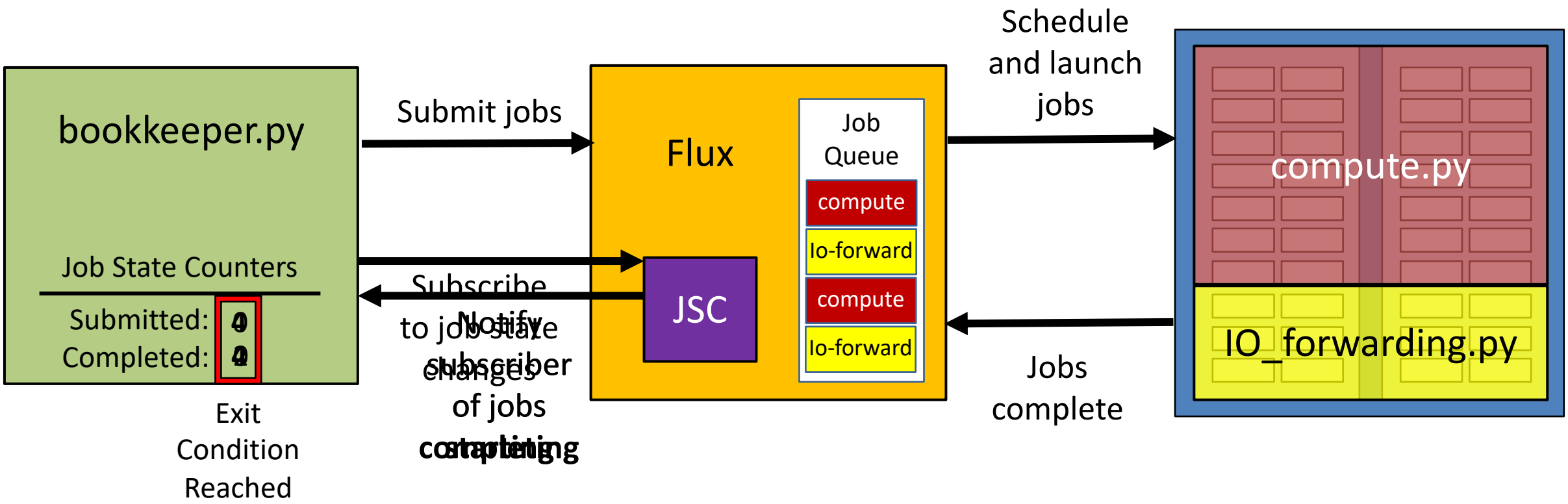
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Example 1: Partitioning vs Overlapping Scheduling



Example 7: Job Status & Control (JSC) API



Example 10: Data Conduit