Raythena: a Vertically Integrated Scheduler for ATLAS Applications on Heterogeneous Distributed Resources

Charles Leggett, Illya Shapoval, Miha Muškinja,
Paolo Calafiura, Vakho Tsulaia
obo ATLAS Experiment

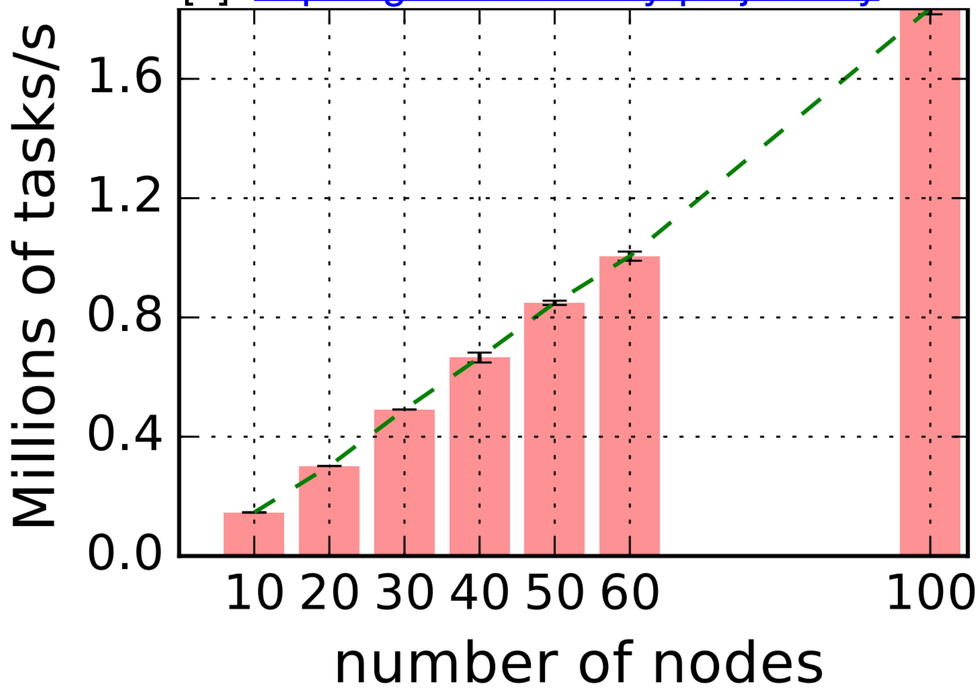
CHEP in Adelaide
Tuesday 05 November 2019

BERKELEY LAB
Lawrence Berkeley National Laboratory

Introduction

- We are exploring the applicability of a modern *distributed execution framework* for ATLAS workflows— Ray[1],
- **Distributed execution frameworks** allows the user to seamlessly transform a single-node application to run efficiently on a cluster of nodes or on a (heterogeneous) HPC,
- Ray has a simple python API, supports stateless and stateful operations and allows us to express data dependencies in the application.
- As a proof-of-concept, we present a Ray-based prototype of the ATLAS Event Service:
  - This is a workflow used for offline production jobs at HPCs.

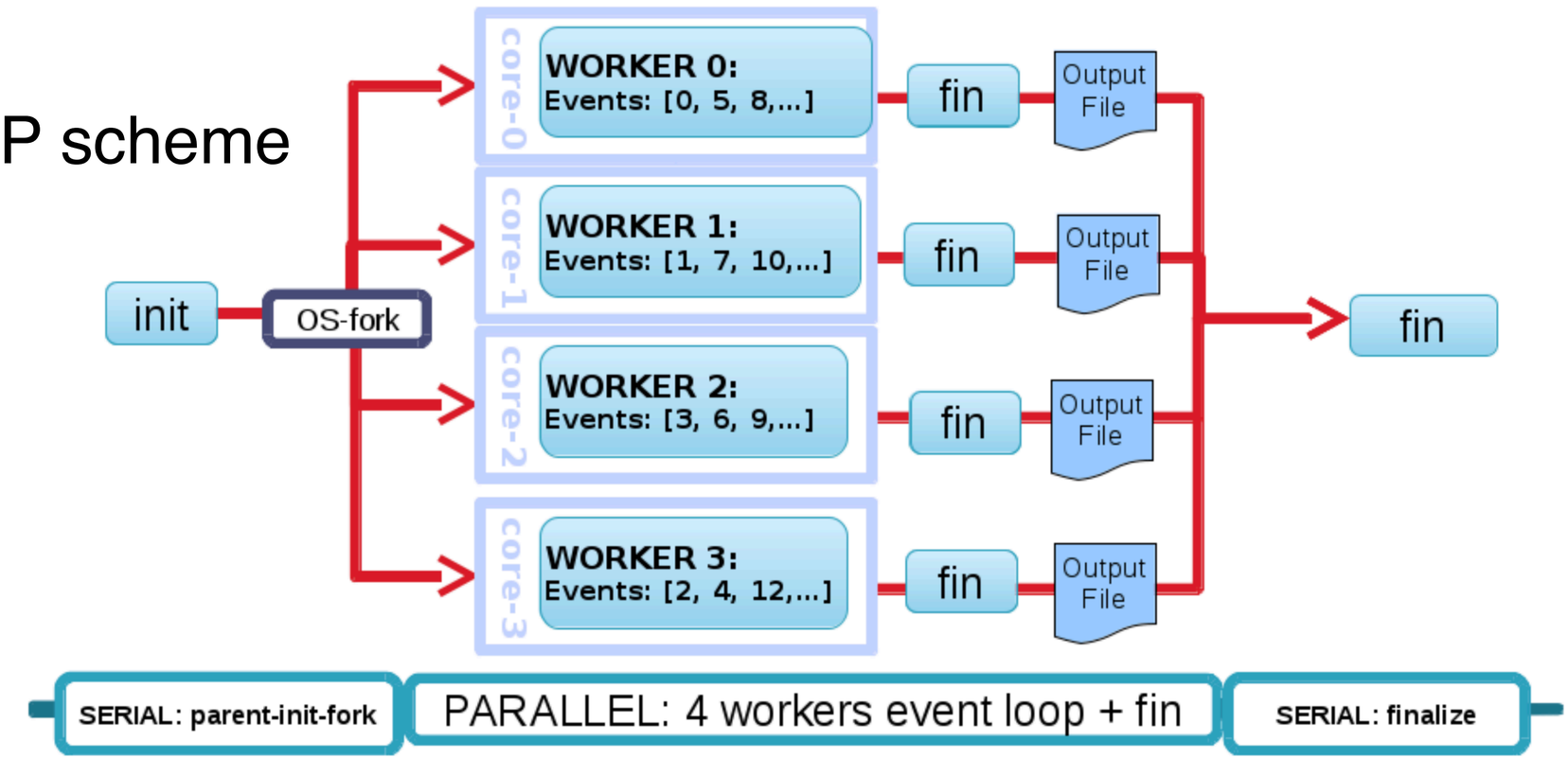
[1]: https://github.com/ray-project/ray



Athena framework in ATLAS

- Athena is the main software framework in ATLAS used for all data analysis steps,
- In this application we are using ‘AthenaMP’, the multi-process version of Athena,
- In the Event Service mode, input events are provided on demand by an external application. The number of input events does not need to be known in advance.

AthenaMP scheme



https://iopscience.iop.org/article/10.1088/1742-6596/664/7/072050/pdf

Current scheduling on HPCs

- ATLAS production workflows are currently comprised of many separate layers that communicate through different (ad-hoc) interfaces,

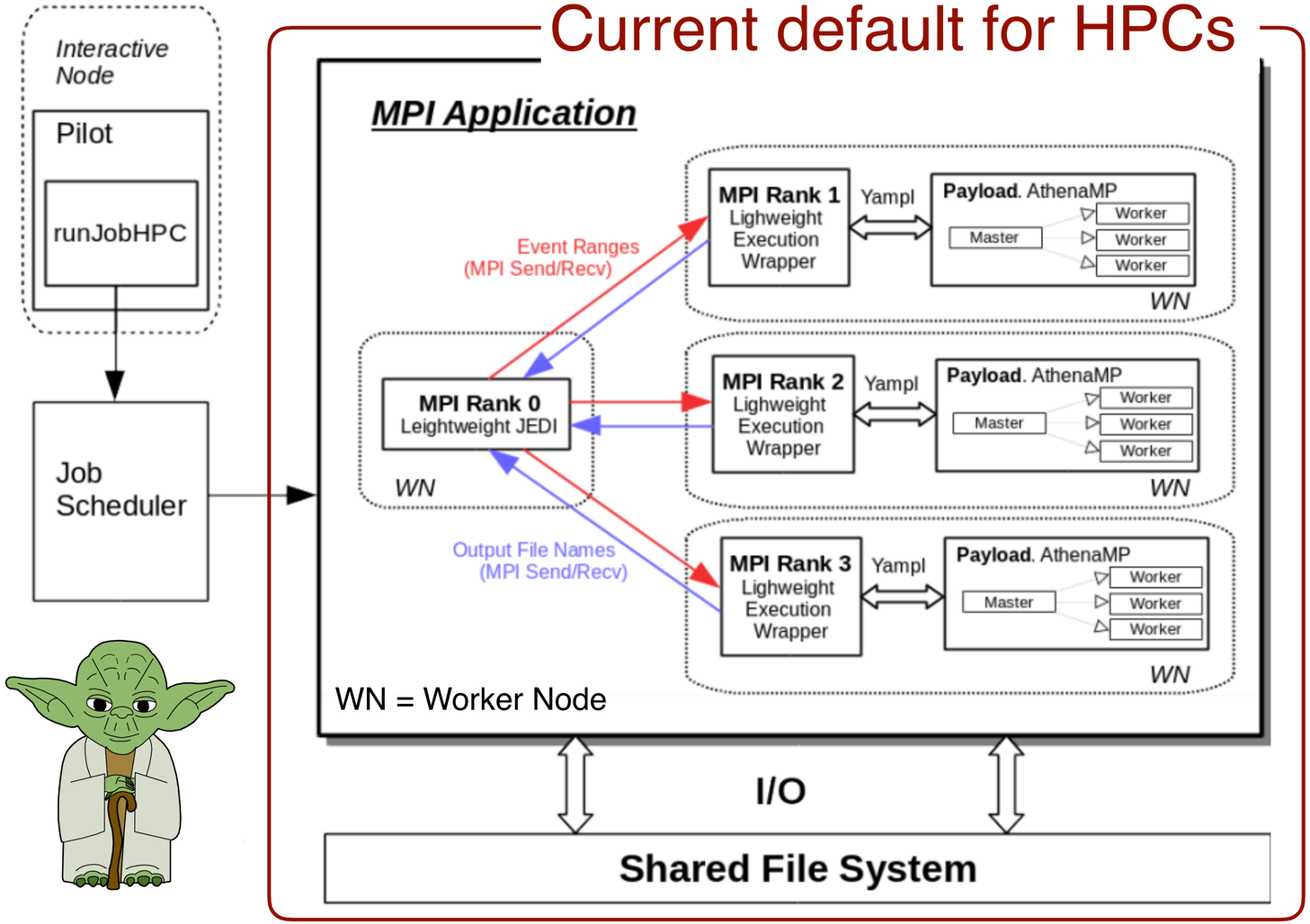


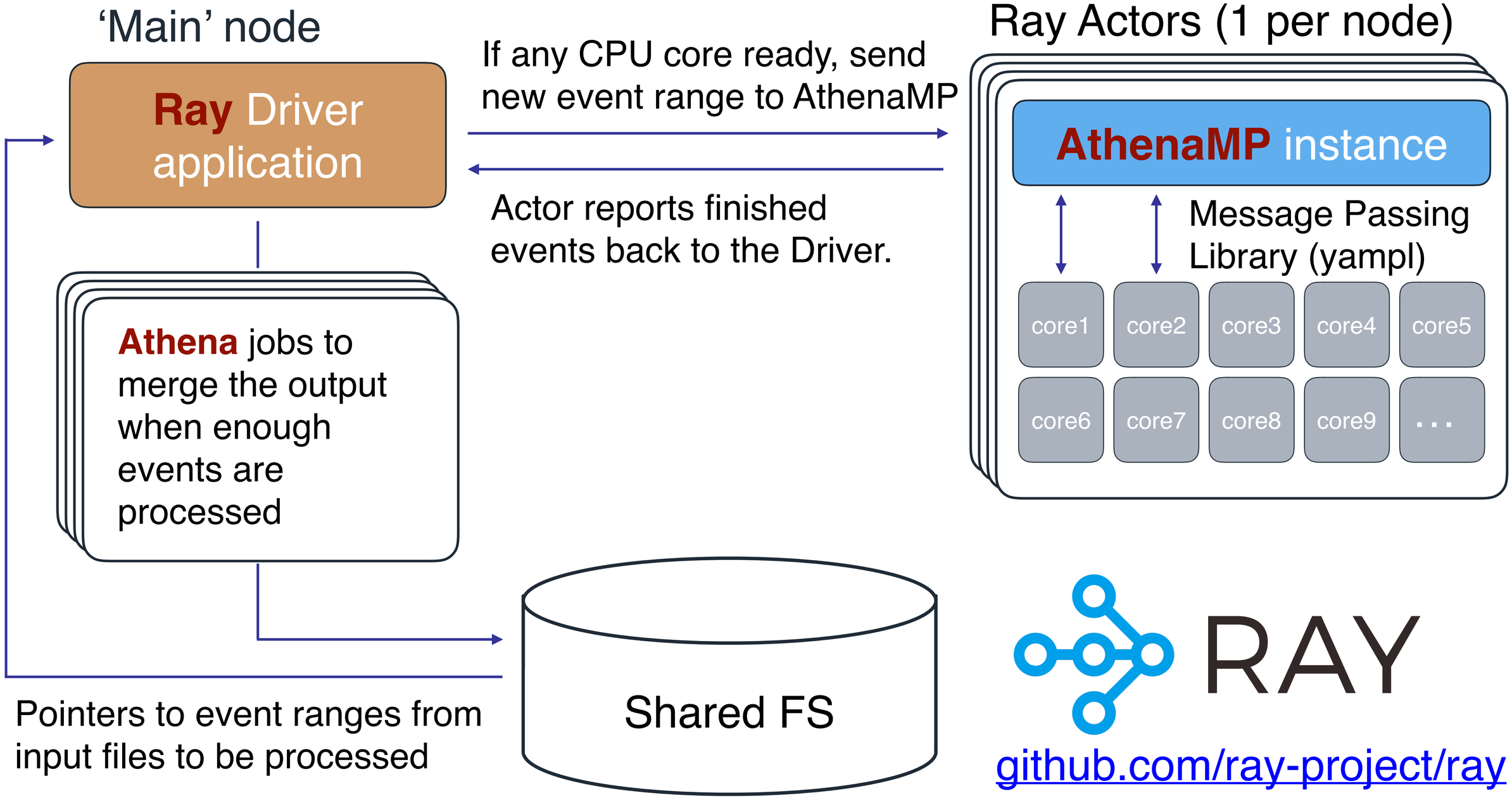
Figure 1. Schematic view of Yoda

iopscience/10.1088/1742-6596/664/9/092025/pdf

We used Ray in place of the current 'Yoda' scheme:

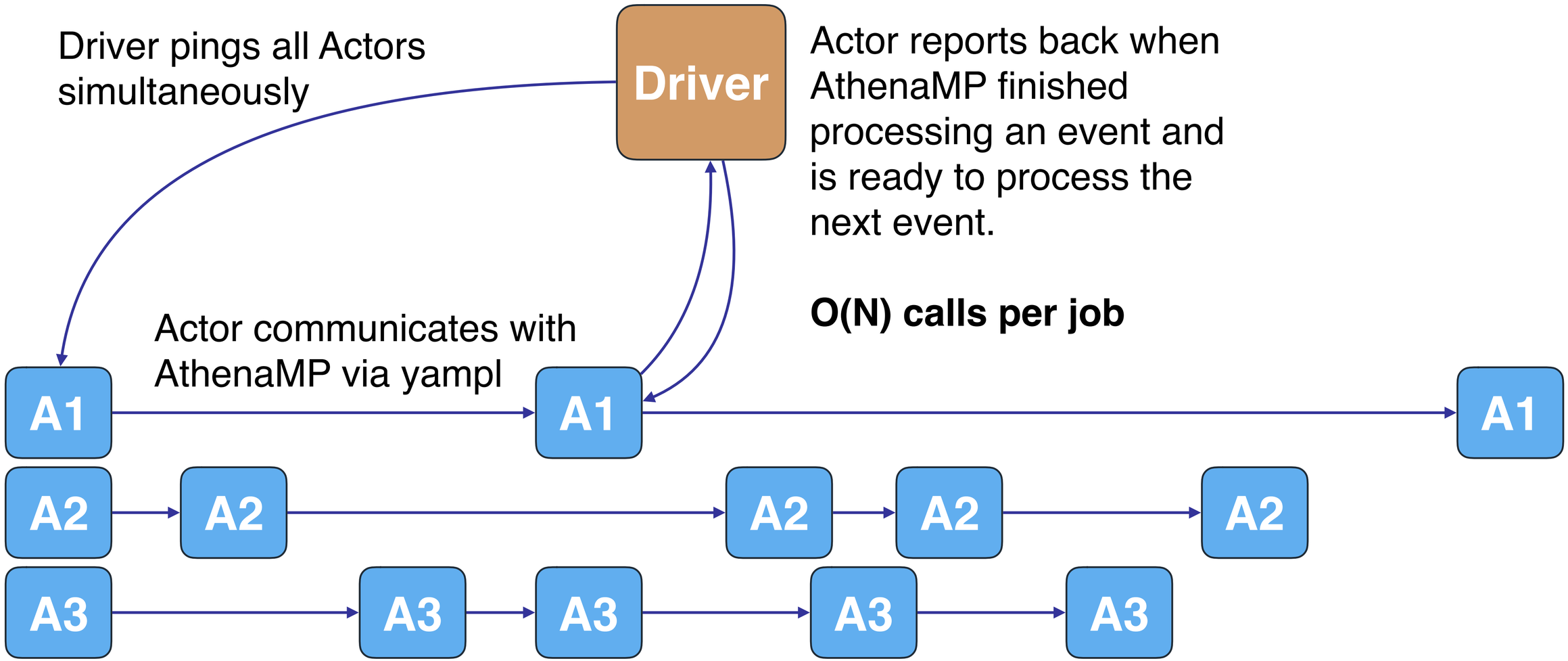
1. Launch AthenaMP processes on all allocated compute nodes,
2. Collect output from AthenaMP sub-processes and give them new input to process when needed,
3. Merge output and store it on the shared FS.

Raythena: Ray-based ATLAS Event Service



Asynchronous communication between the Driver and Actors

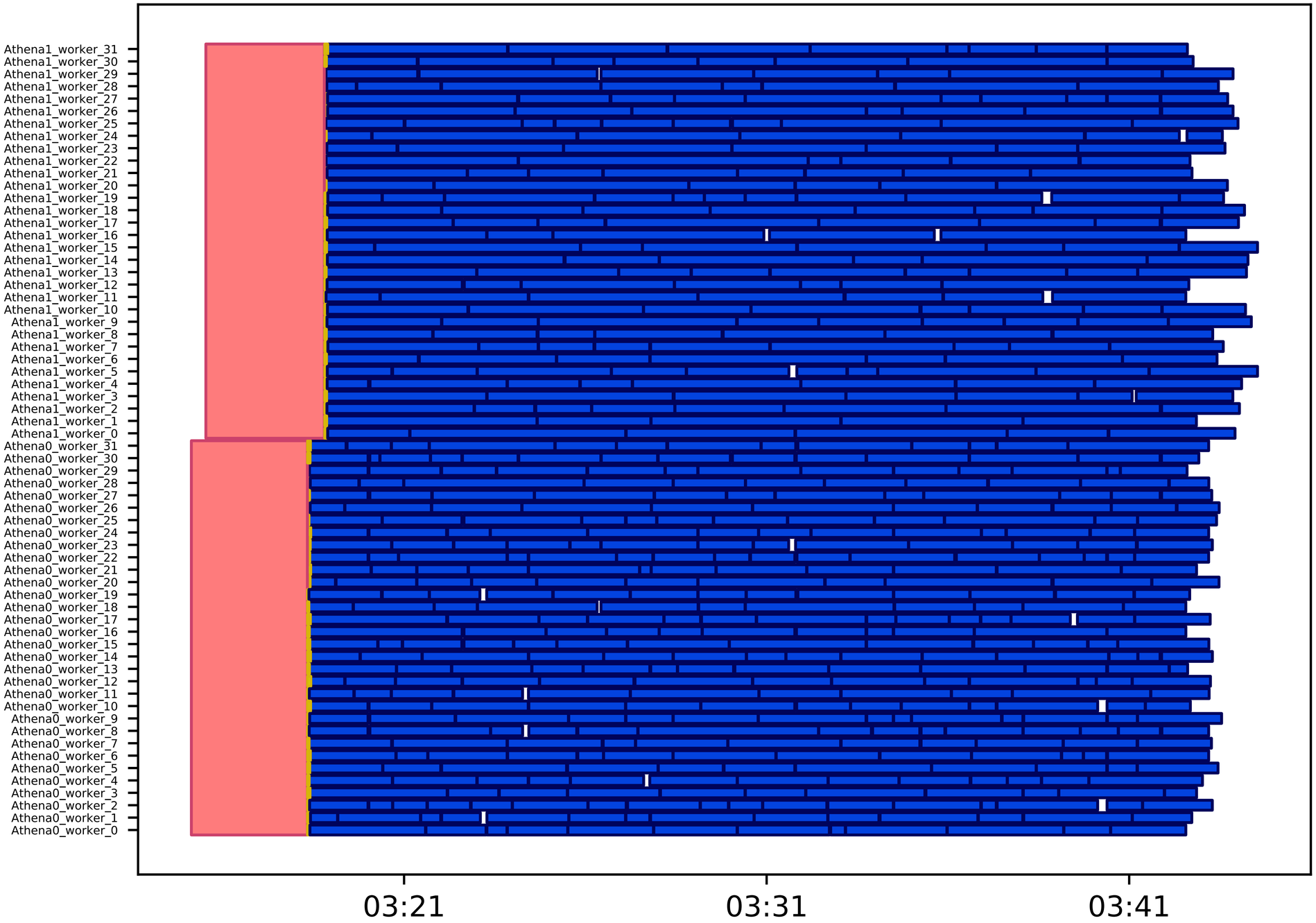
- Asynchronous communication is implemented in a few 100 python lines using Ray explicit parallelism expressions,
- Actors independently communicate with the AthenaMP instance and report back to the driver only when new input events are needed or when an event was processed.



Working example on Cori at NERSC

- Successfully tested the Raythena workflow on Cori **Haswell** and **KNL** nodes at NERSC,
- Athena merge jobs are spawned on-the-fly when enough events are processed,
- Largest test that we tried so far:
  - 60 Haswell nodes with 32 cores each,
  - Processed 100k events in total and spawned merge jobs every 100 events to form 1000 merged output files.
- No bottlenecks found so far in Ray.

Close-up — two AthenaMP instances on Haswell nodes



AthenaMP Initialization

Worker

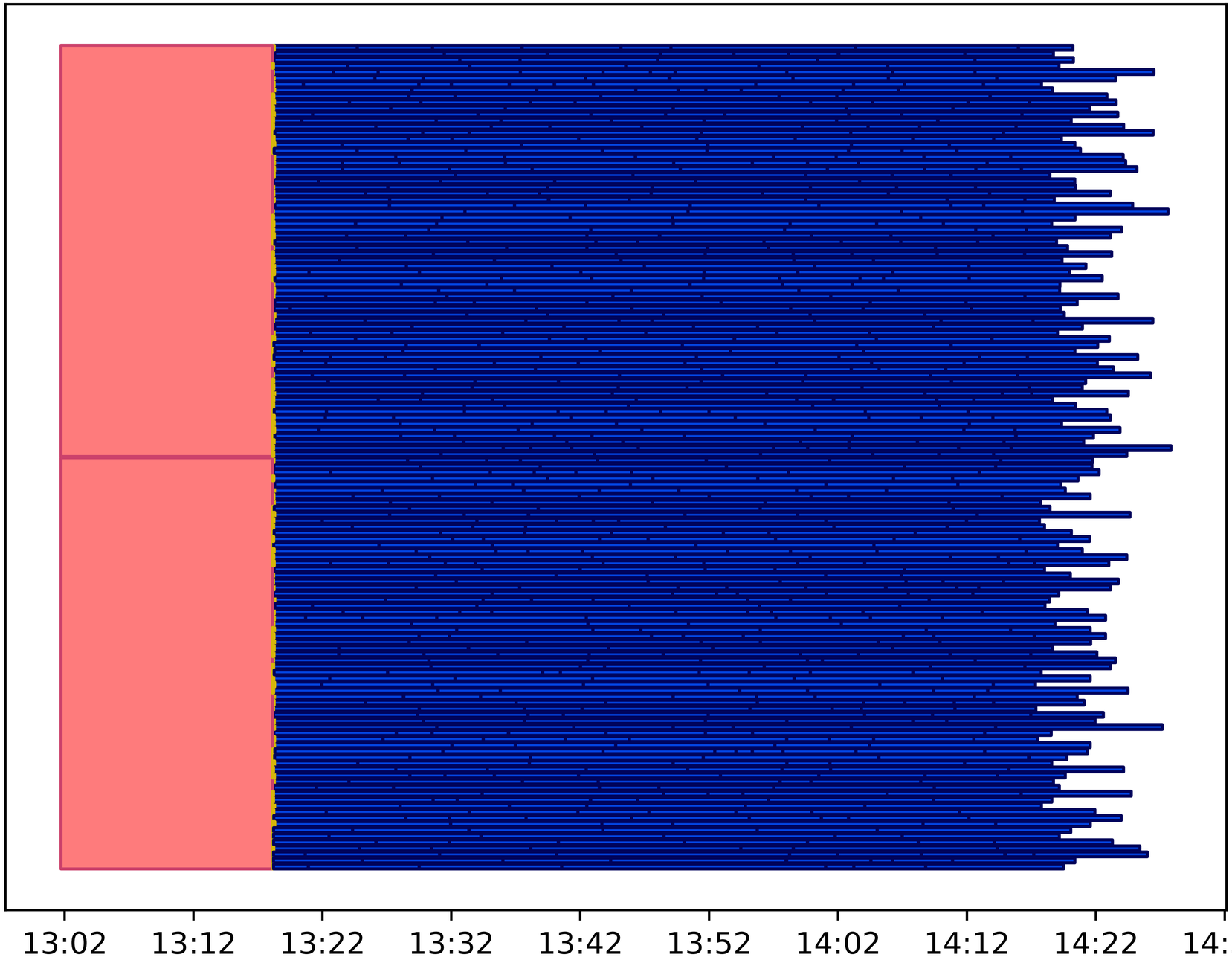
Event Processing

Worker Initialization

[Cori Haswell Nodes](#)
- 32 cores / node

wall time

Close-up — two AthenaMP instances on KNL nodes



AthenaMP Initialization

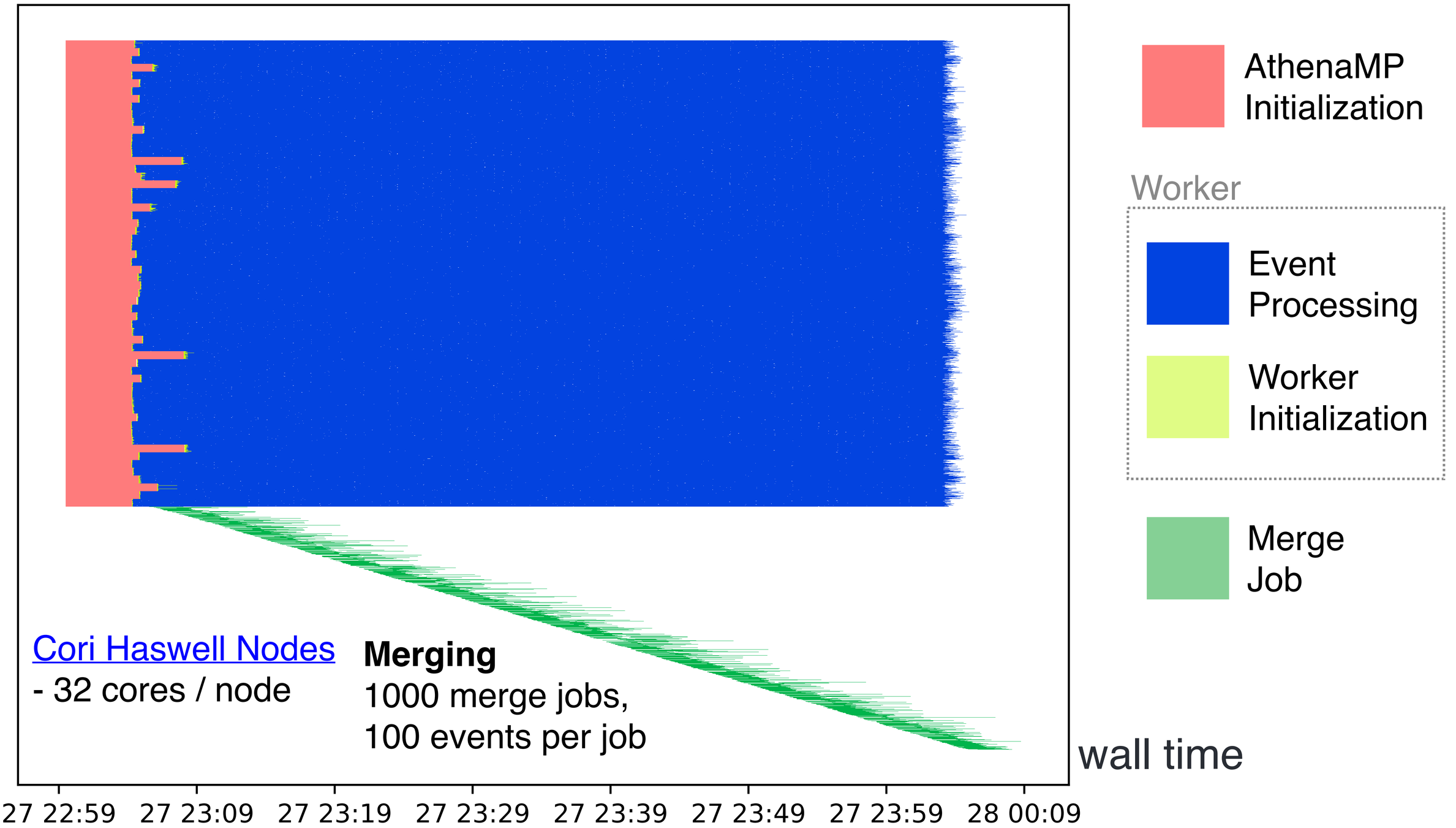
Worker

Event Processing

Worker Initialization

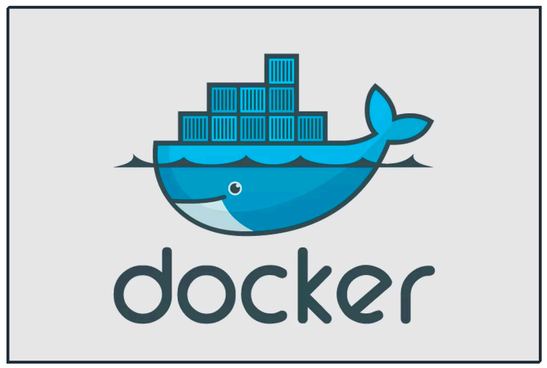
[Cori KNL Nodes]
- 68 cores / node

Large Haswell job example (60 AthenaMP instances, 100k events)



Raythena running scheme

```
$ sbatch –image mmuskinj/centos7-atlasos-ray:1.0.0 –module=cvmfs
```



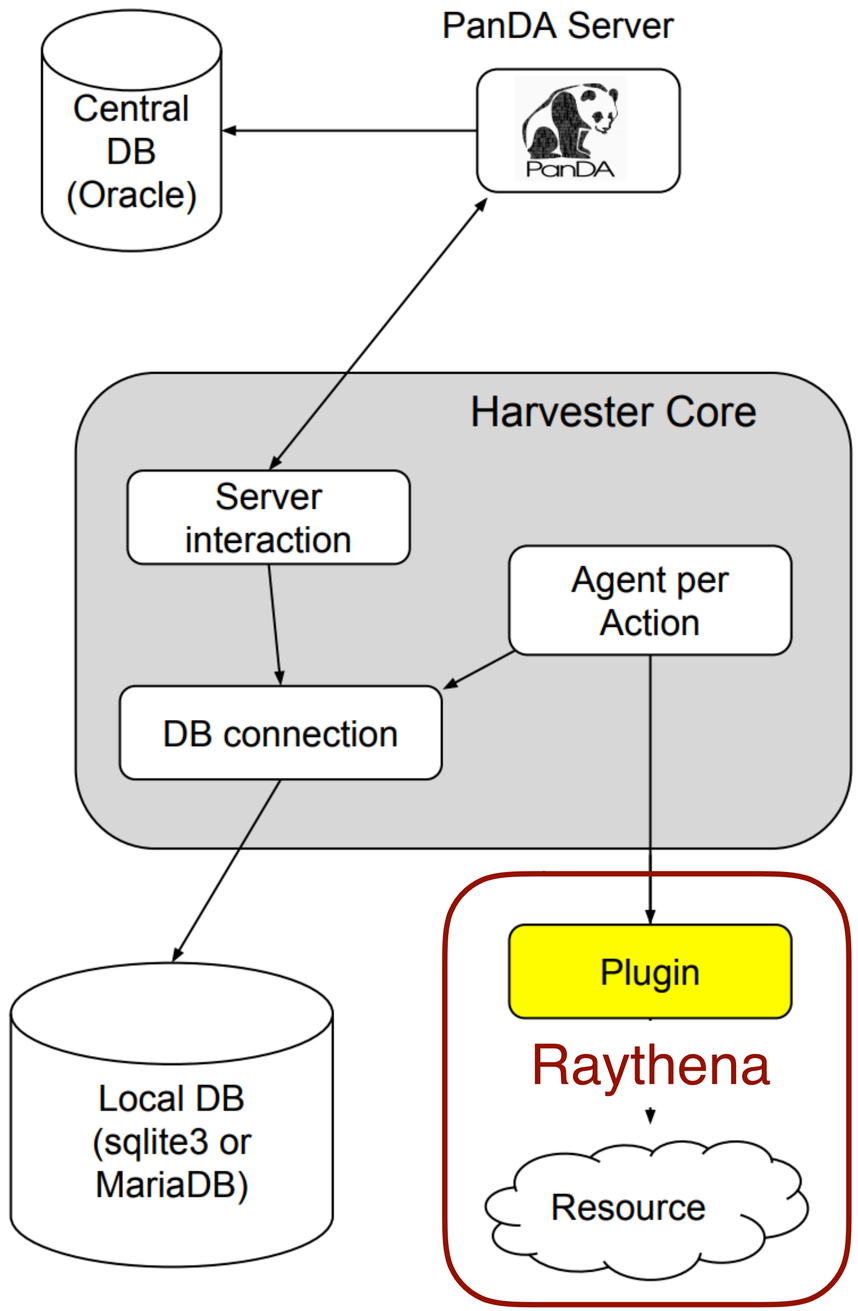

```
$ shifter ./ray_start_head.sh
$ srun shifter ./ray_start_other.sh &
$ shifter ./run_raythena.sh
```

- Ray, Raythena, and Athena are all running in a container on all nodes,
- At NERSC we are using Shifter containers which are built from Docker images,
- Can be ported to other HPCs without too much effort.

Raythena plans for Run 3

- We are working towards using Raythena as the default job orchestrating application on HPCs in Run 3,
- Raythena batch jobs will be spawned by Harvester— an application connected to the PanDA server.

- Ray is widely used by the broader community and centrally maintained. Using Ray would eliminate the need of supporting some of the ATLAS home-built software,
- Proven to be scalable on HPCs, lightweight and easy to install (e.g. as a module),
- Establishes a modular scheme for Run 3 job scheduling. Ray could be replaced by other frameworks (e.g. Dask, Spark).



https://cds.cern.ch/record/2625435/

Digging deeper into Athena / Gaudi

- The long-term project is to interface Athena/Gaudi algorithms directly to Ray for a much finer control over scheduling the workload,
- This would replace the current event loop with Ray and enable scheduling of a single event across several nodes,
- Data needed by the algorithms is provided by Ray's Global Control Store (GCS),
- Maximize throughput by more efficient/tailored scheduling of algorithms to computing resources (e.g., CPU vs GPU).

PyAthenaEventLoopMgr.py

```
for alg in AlgSequence:
    result = alg.sysExecute( theEventContext )
    if result.isFailure():
        return result.getCode()
```

AlgSequence
- Algorithm1 →
- Algorithm2 →
- … →
- AlgorithmN →

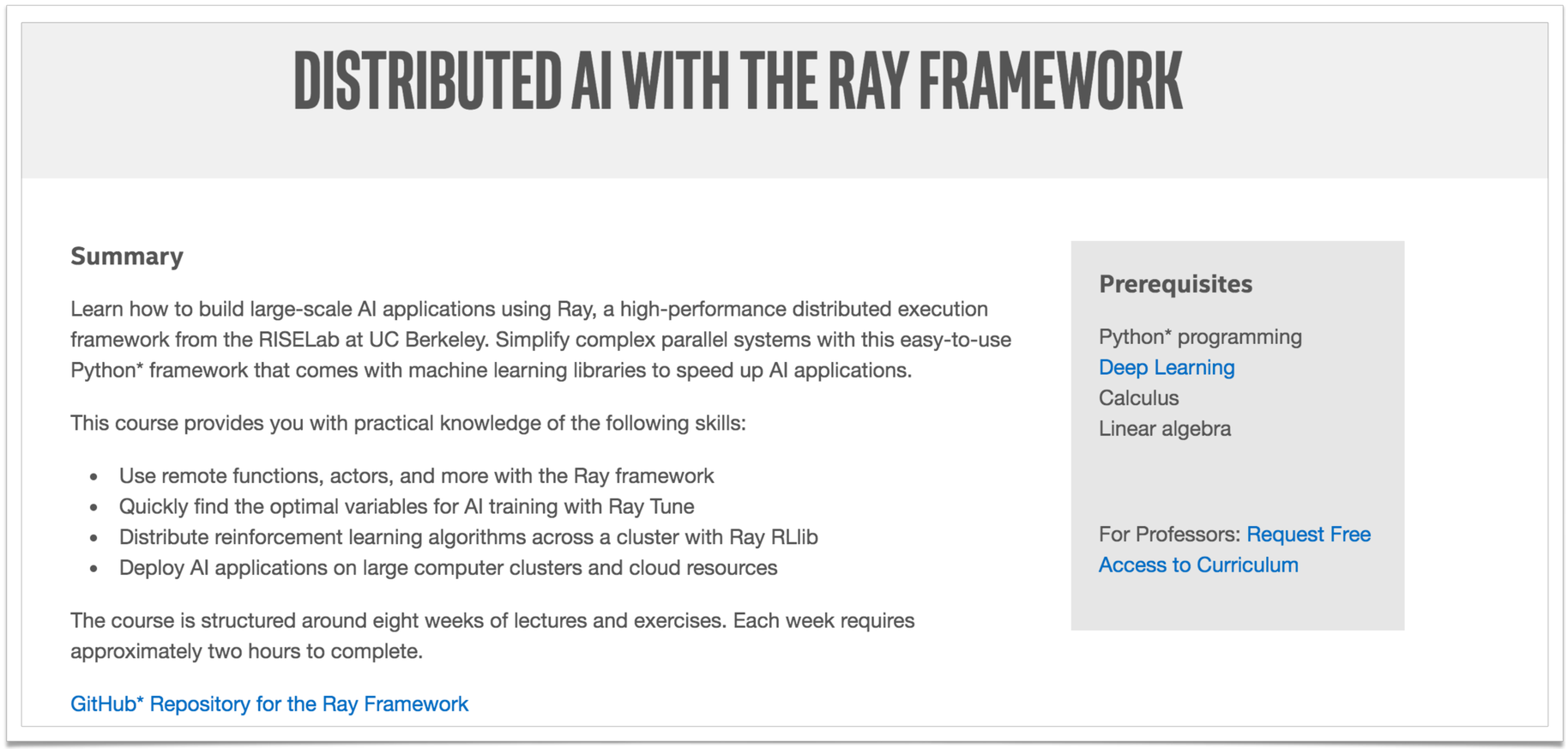
Summary

- We are exploring the applicability of a distributed execution framework (Ray) to ATLAS workflows,
- We have demonstrated a stand-alone prototype of a Ray-based ATLAS Event Service,
  - Shown to be scalable on Cori Haswell and KNL nodes,
  - Runs entirely from containers and is portable to other HPCs,
  - Plan is to use it as the default intermediate layer between Harvester and AthenaMP processes on compute nodes in Run 3 for large-scale production jobs.
- Longer-term-plan is to divide the ATLAS workflow into base components (Algorithms) and interface them directly to Ray.

BACKUP

Ray documentation and tutorials

- Ray has a very rich documentation hosted on readthedocs:
  - https://ray.readthedocs.io/en/latest/index.html,
- Hands-on tutorials with exercises available in form of jupyter notebooks,
- Since Feb 2019, Intel hosts an 8-week course about distributed AI computation with Ray: https://software.intel.com/en-us/ai/courses/distributed-AI-ray.



DISTRIBUTED AI WITH THE RAY FRAMEWORK

Summary

Learn how to build large-scale AI applications using Ray, a high-performance distributed execution framework from the RISELab at UC Berkeley. Simplify complex parallel systems with this easy-to-use Python* framework that comes with machine learning libraries to speed up AI applications.

This course provides you with practical knowledge of the following skills:

- Use remote functions, actors, and more with the Ray framework
- Quickly find the optimal variables for AI training with Ray Tune
- Distribute reinforcement learning algorithms across a cluster with Ray RLlib
- Deploy AI applications on large computer clusters and cloud resources

The course is structured around eight weeks of lectures and exercises. Each week requires approximately two hours to complete.

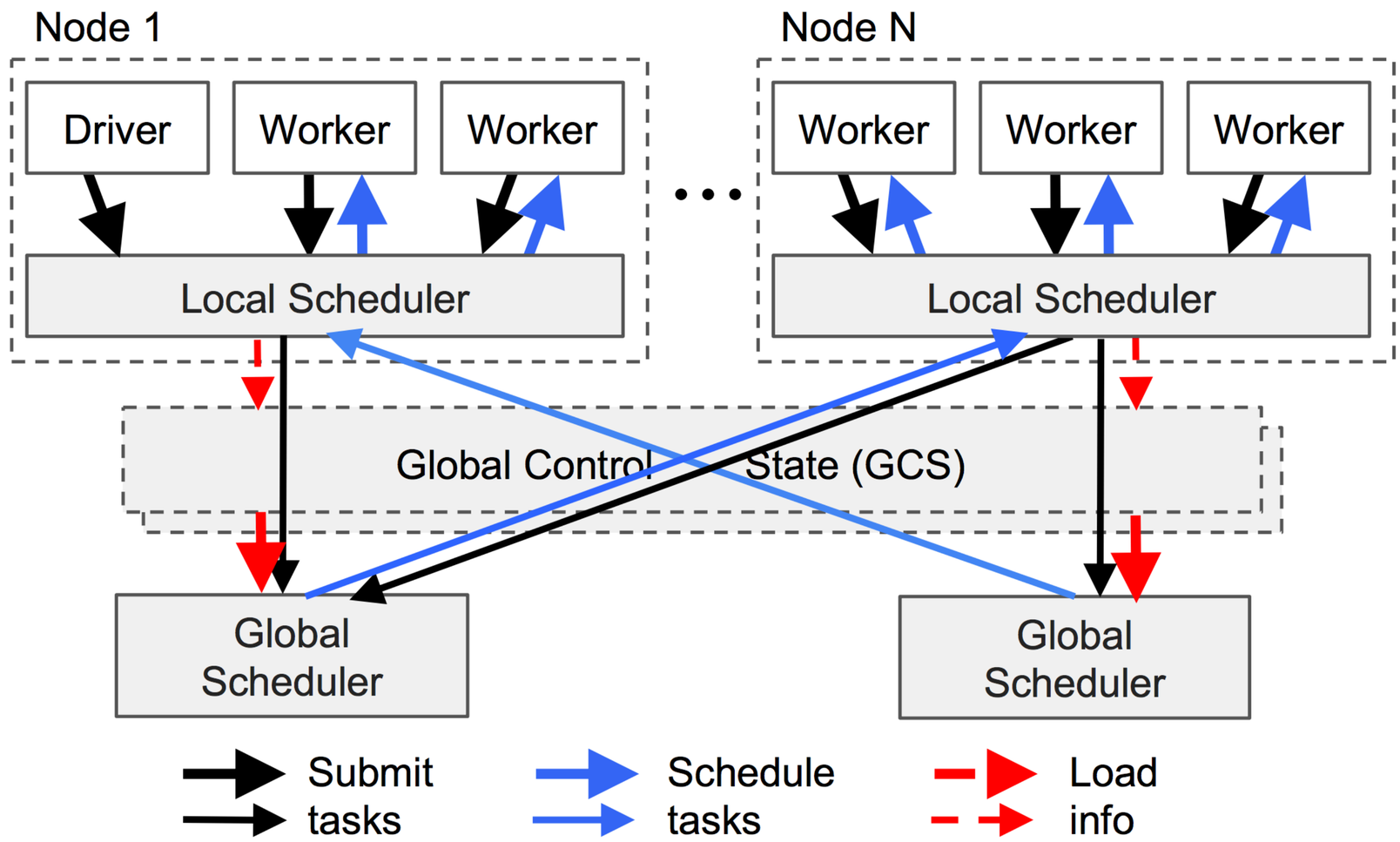
GitHub* Repository for the Ray Framework

Prerequisites

Python* programming
Deep Learning
Calculus
Linear algebra

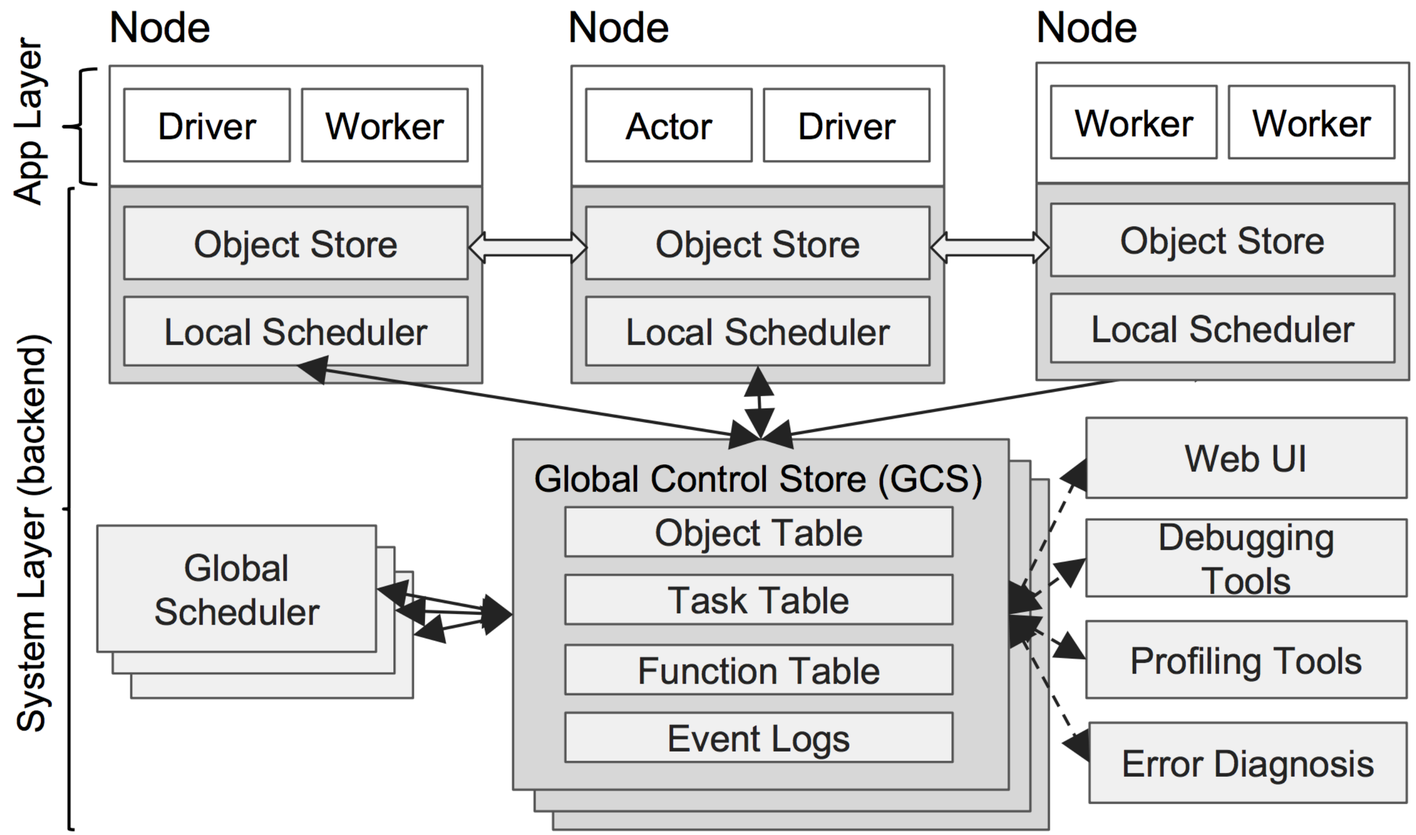
For Professors: Request Free Access to Curriculum

- One driver application (running on any compute node) controls all nodes in a cluster (HPC) that are connected via TCP to a redis server,
- Tasks are first scheduled locally (Local Scheduler) if resources are available, otherwise they are scheduled globally via the Global Scheduler.



Ray Application Layer

- Ray maintains three types of processes:
  - *Driver*: a process executing the user program,
  - *Worker*: a stateless process that executes tasks invoked by the driver or another worker. Workers are started automatically and execute tasks serially without maintaining a local state,
  - *Actor*: a stateful process that executes only the method it exposes. They execute methods serially and each method depends on the state resulting from the previous execution.



Ray Functions and Actors

- A Ray parallel application is constructed with python decorations:

Task executed at a worker

```
@ray.remote
def simpleFunction(a, b):
    # wait for 5 seconds
    time.sleep(5)
    # return sum
    return a + b
```

Actor process

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value
```

```
# this returns immediately
r = simpleFunction.remote(2, 4)

# this will be executed
# after 5 seconds
print( ray.get(r) )
```

Driver application