UNIVERSITY OF OREGON

Scalable Observation System (SOS) for Scientific Workflows

Project Overview & Discussion

Chad D. Wood

Supervisors: Prof. Allen Malony and Kevin Huck

“So, where is this talk going?”

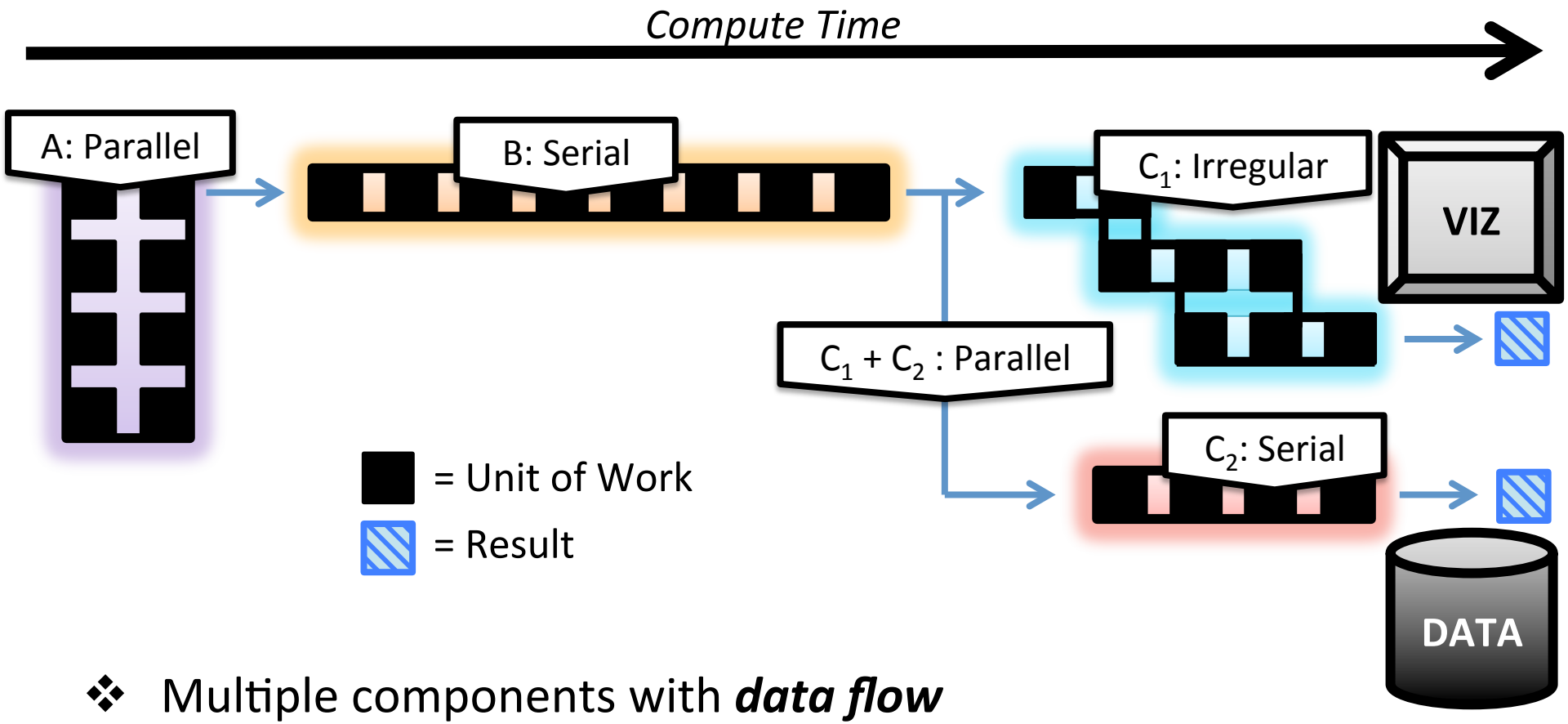
To advocate and demonstrate a ***run-time system*** designed to enable the ***characterization*** and ***analysis*** of complex scientific ***workflow performance*** at scale.

So, What is the Problem?

UNIVERSITY OF OREGON

- It is reasonable to want to see “information” during application execution
- Information could come from the application as well as from the environment in which the application is executing
- Application:
  Performance, problem-specific data and metadata, ...
- Environment:
  System state, resource usage, runtime properties, ...
- Multiple application components may be running together as a workflow, and higher-level workflow behavior might be interesting

Scientific Workflows



- Multiple components with ***data flow***
- Complex interactions with ***dynamic behavior***
- Components (or entire flows) may be *parallelized* differently
- ***Offline episodic performance analysis has limited benefits***

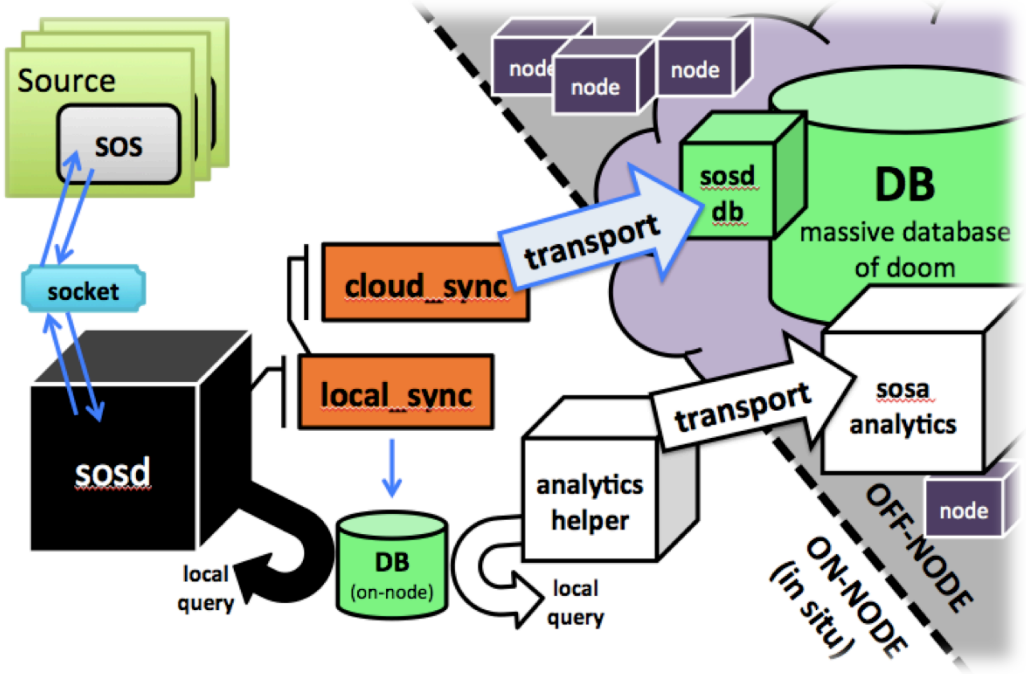
What are the Requirements?

UNIVERSITY OF OREGON

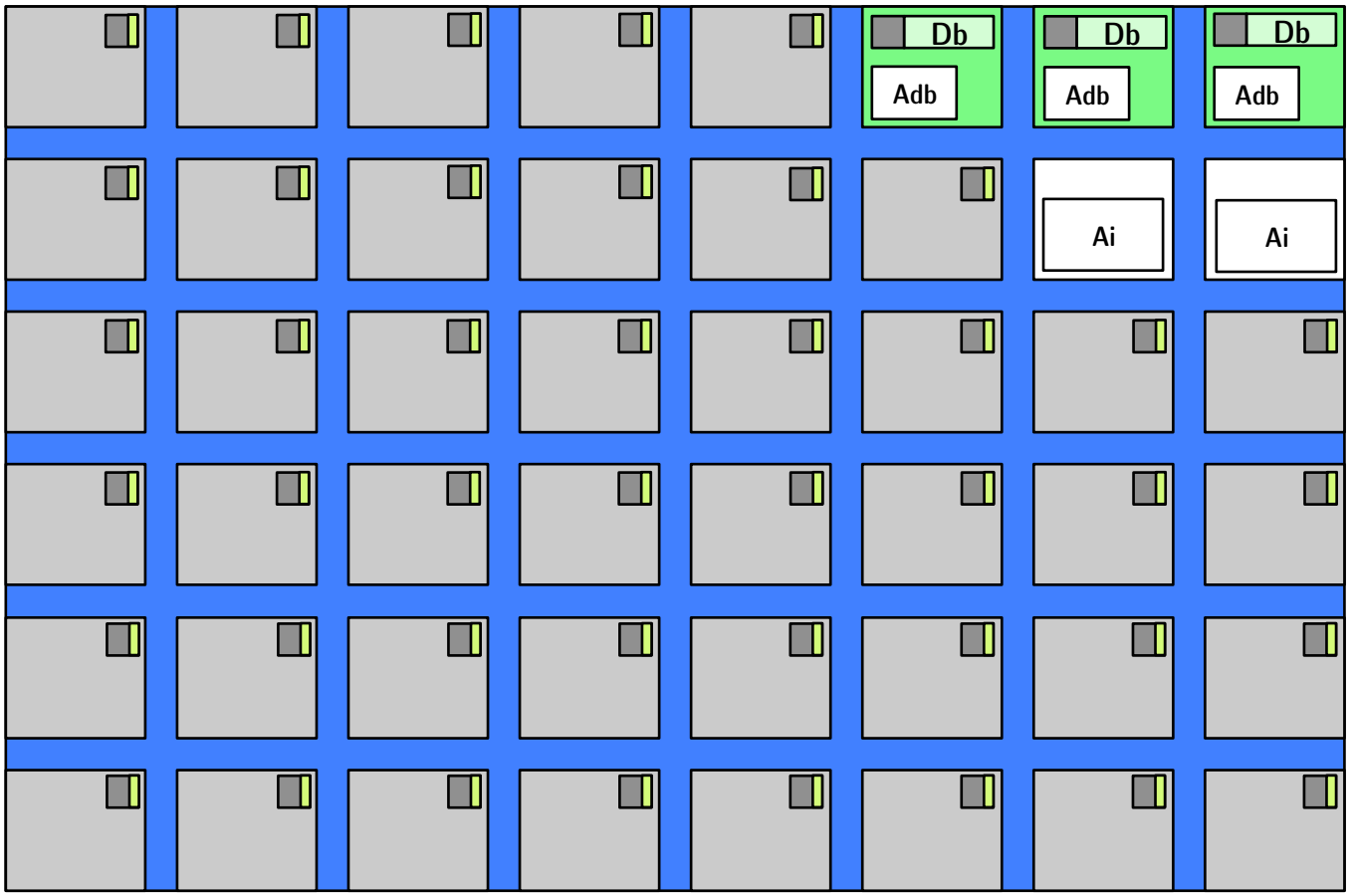
- Scalable
- Portable
- Easy to use
- Multi-purpose
- Multiple information sources
- Operates at the time of application (workflow) execution
- Supports in situ access
- Low overhead and low intrusion
- Ability to allocation additional resources to control overhead

Design Approach

- Base on a model of a “global” information space
- Utilize database technology
- Utilize MPI high-performance communication
- Build on launch support in scheduler
- Allow for additional (dedicated) resource allocation
- Flexible publishing interface
- SOS architecture

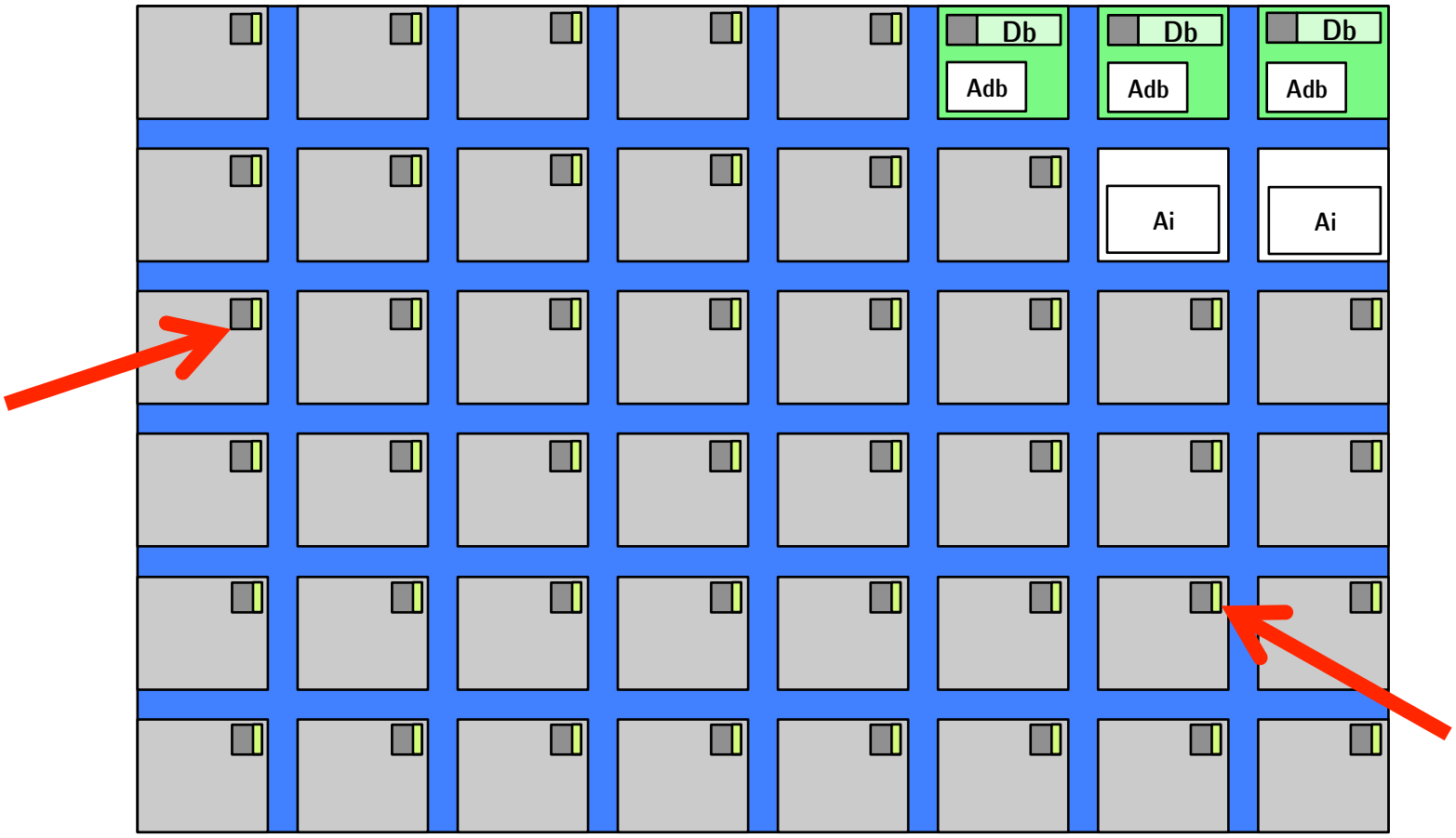


SOSflow: HPC Allocations



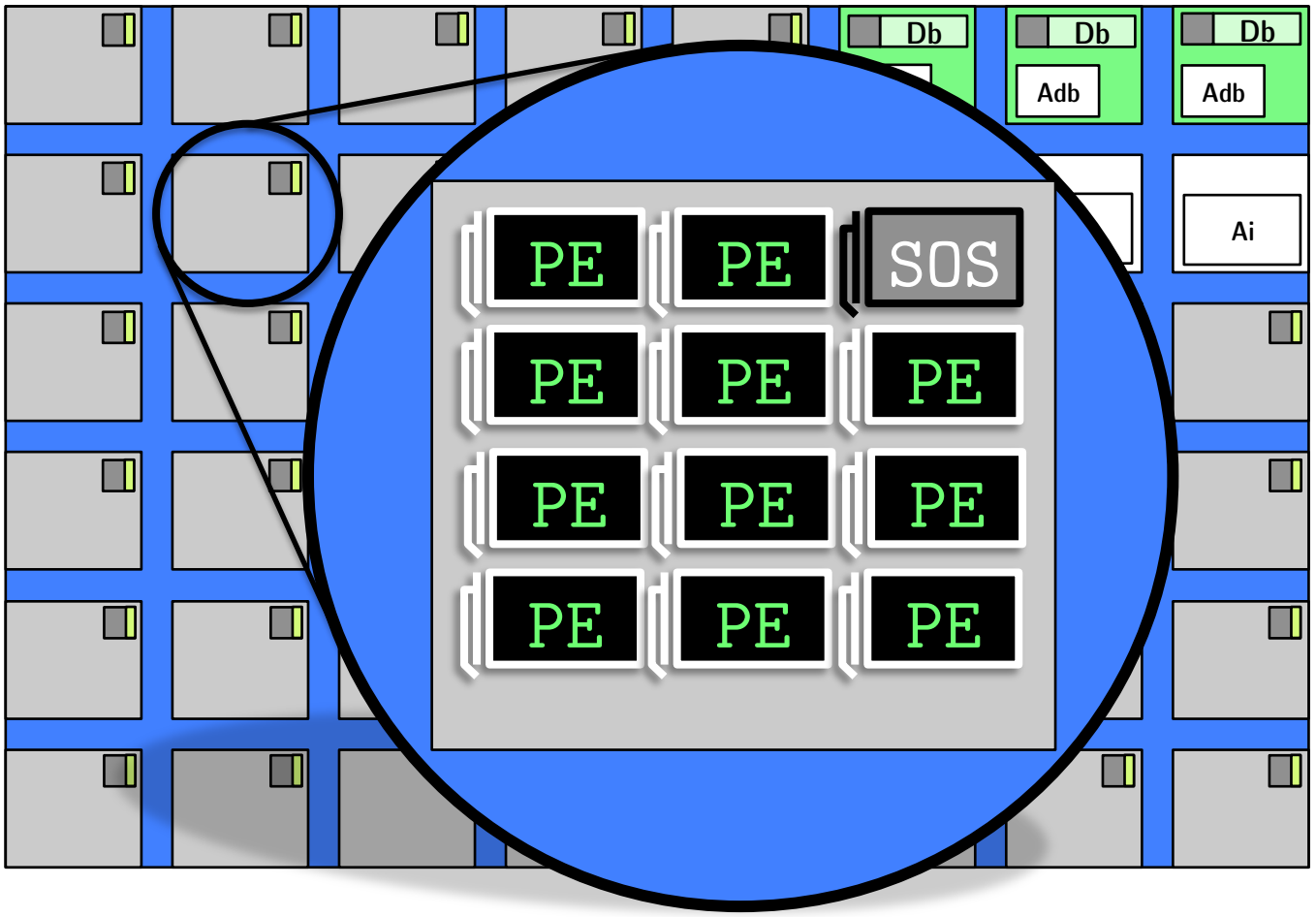
SOSflow forms a functional overlay.

SOSflow: HPC Allocations



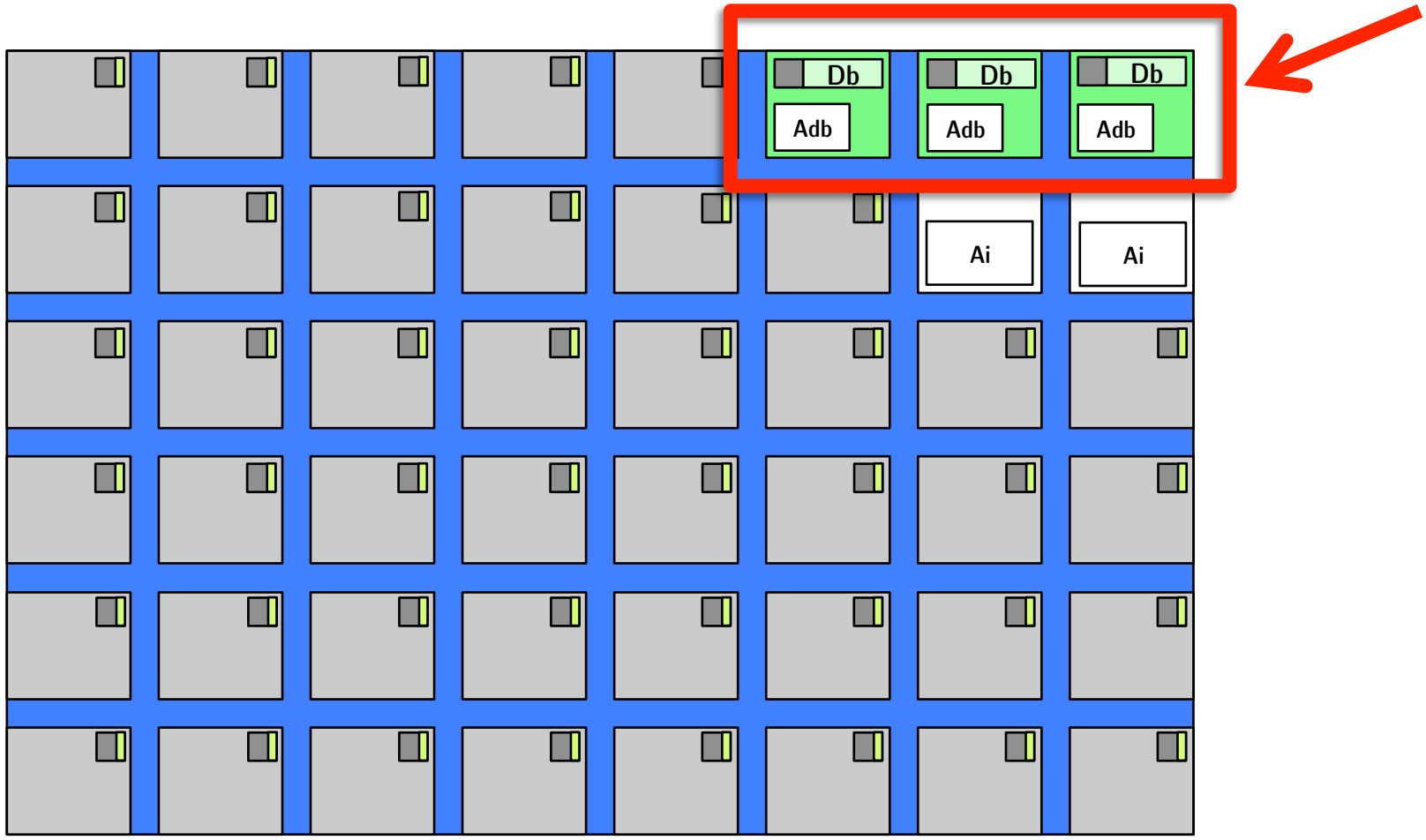
In situ **daemon** with its local **database**

SOSflow: HPC Allocations



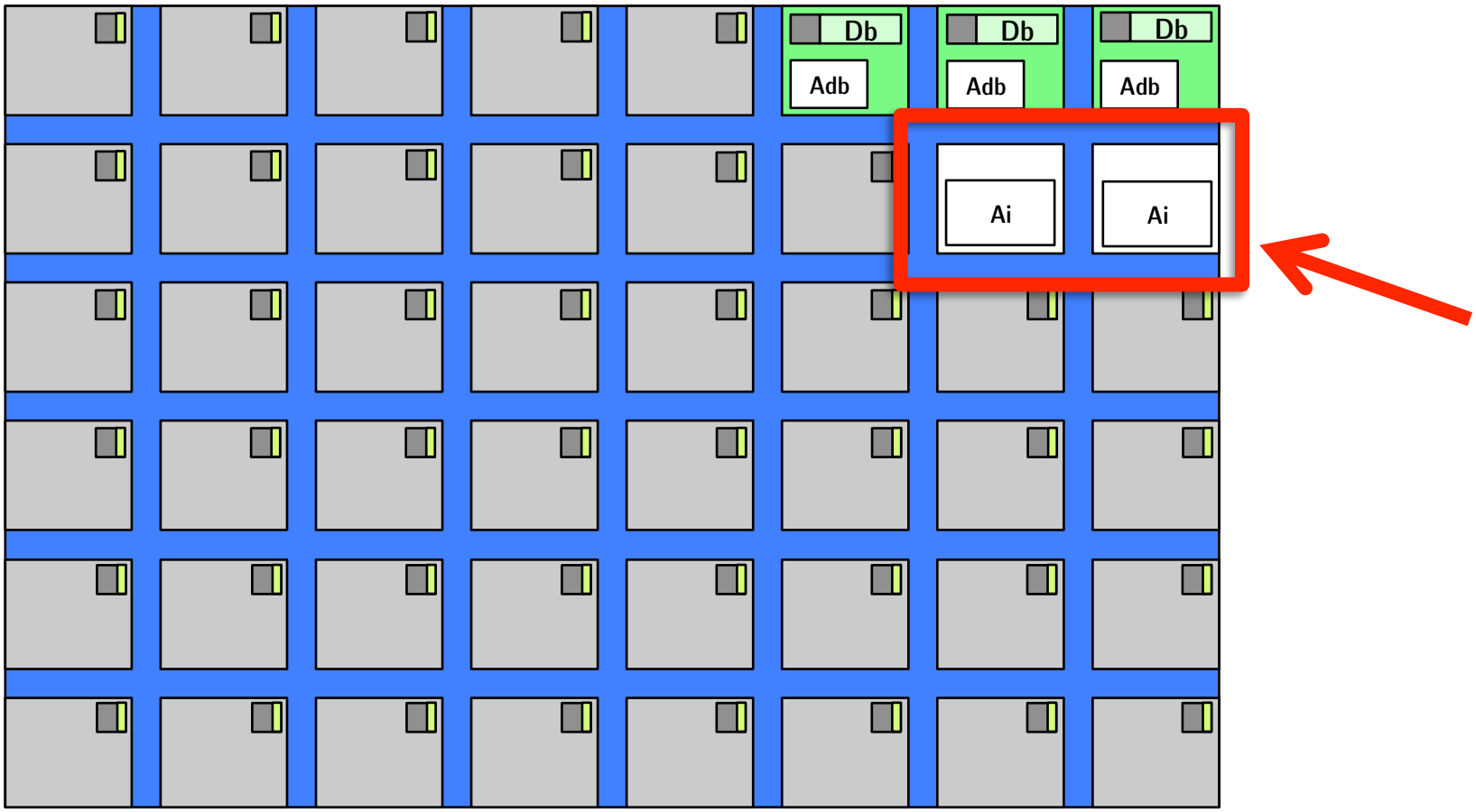
SOS lives side by side with your tasks

SOSflow: HPC Allocations



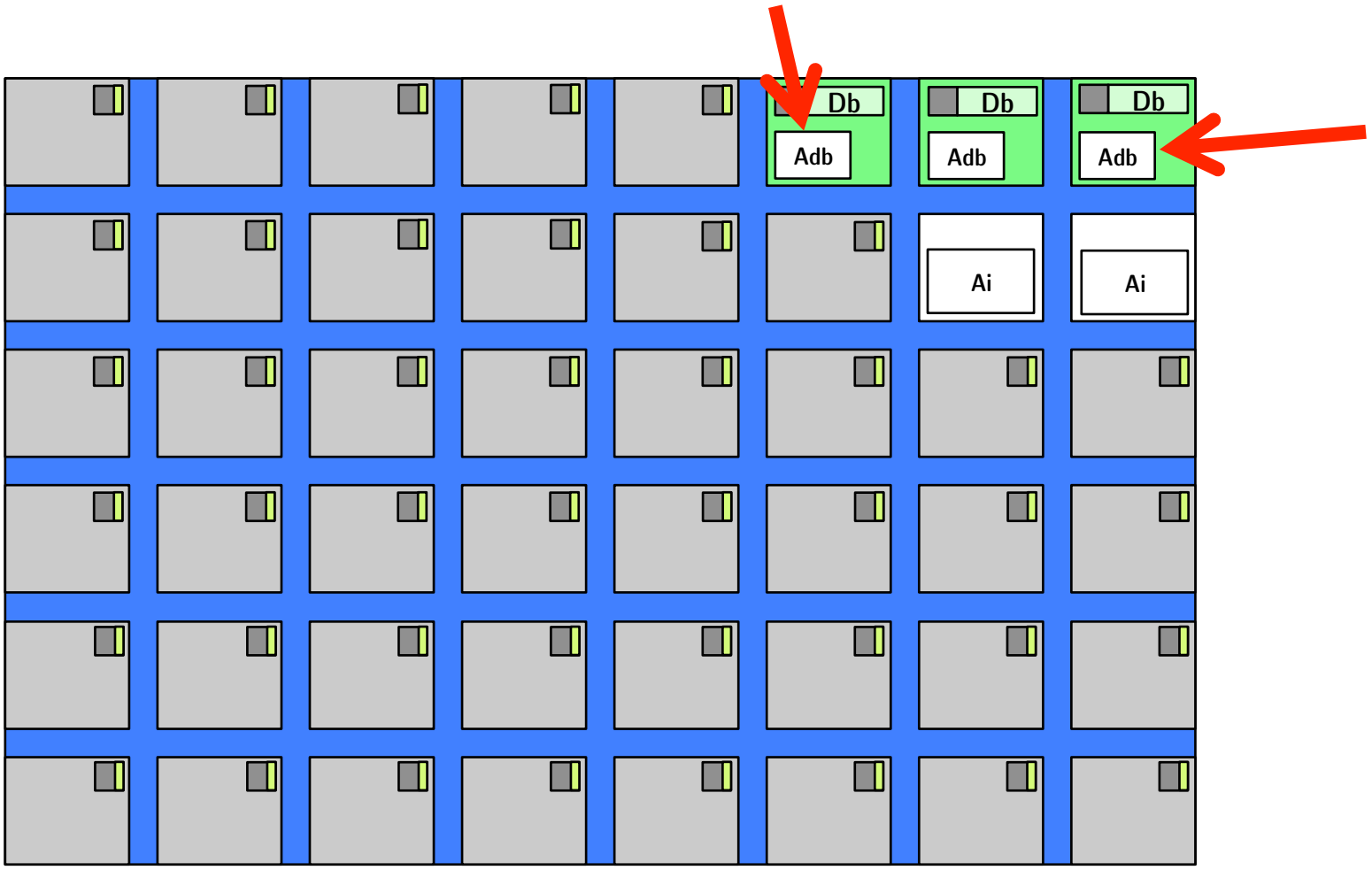
Dedicated nodes for **aggregate databases**

SOSflow: HPC Allocations



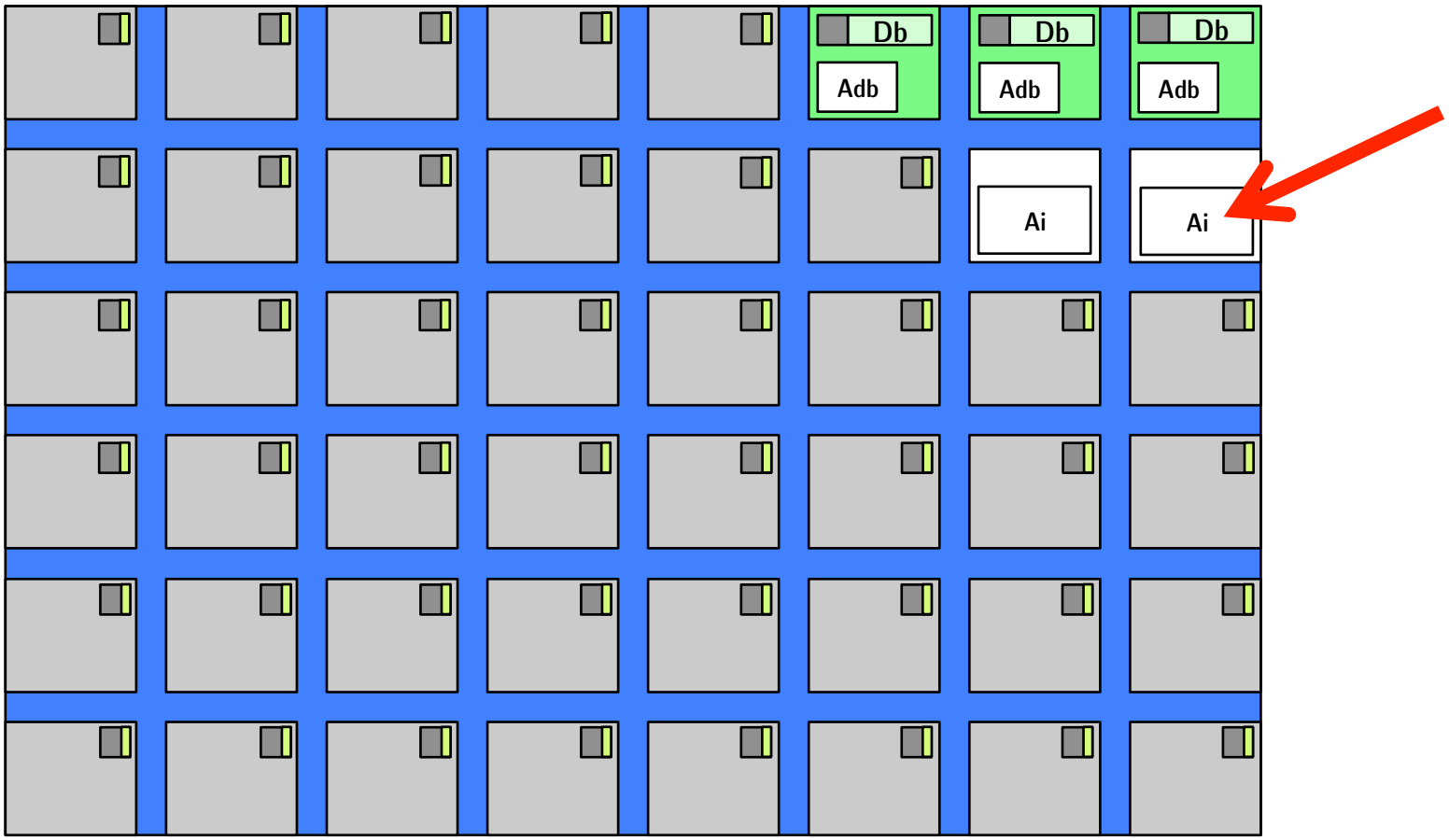
Dedicated nodes for **analytics processing**

SOSflow: HPC Allocations



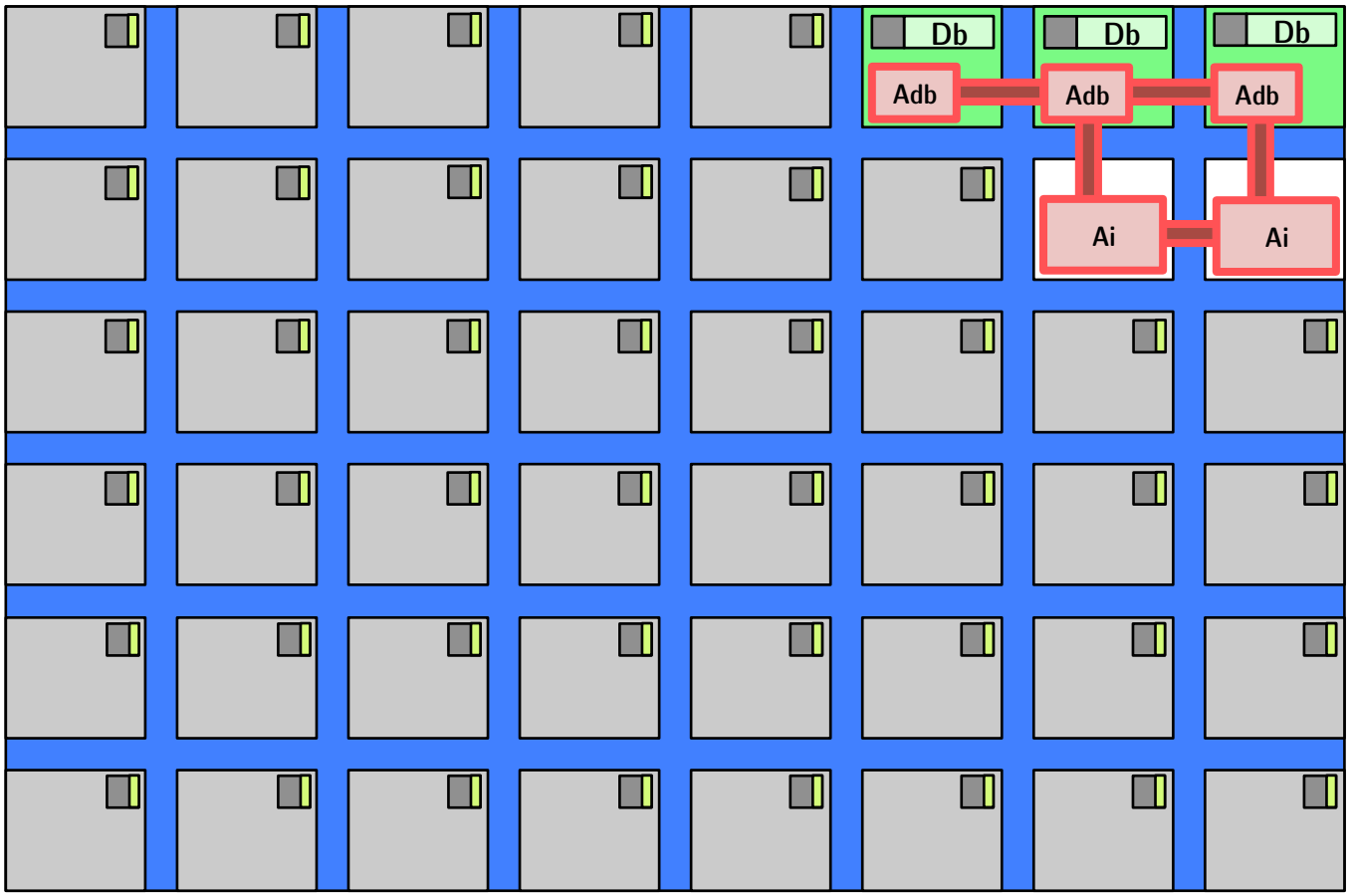
Co-located **analytics query modules**

SOSflow: HPC Allocations



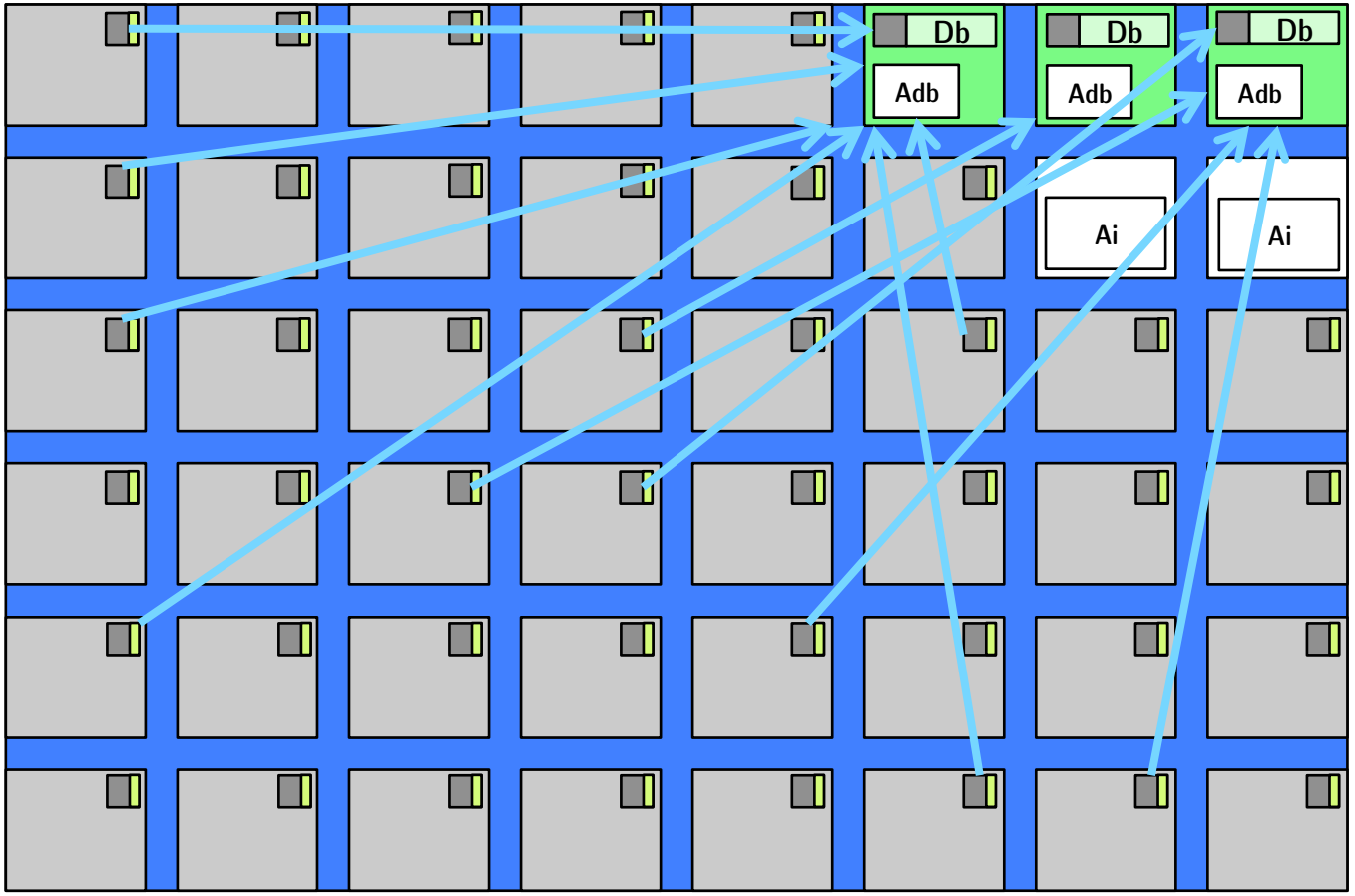
Independent ranks of **analytics engines**

SOSflow: HPC Allocations



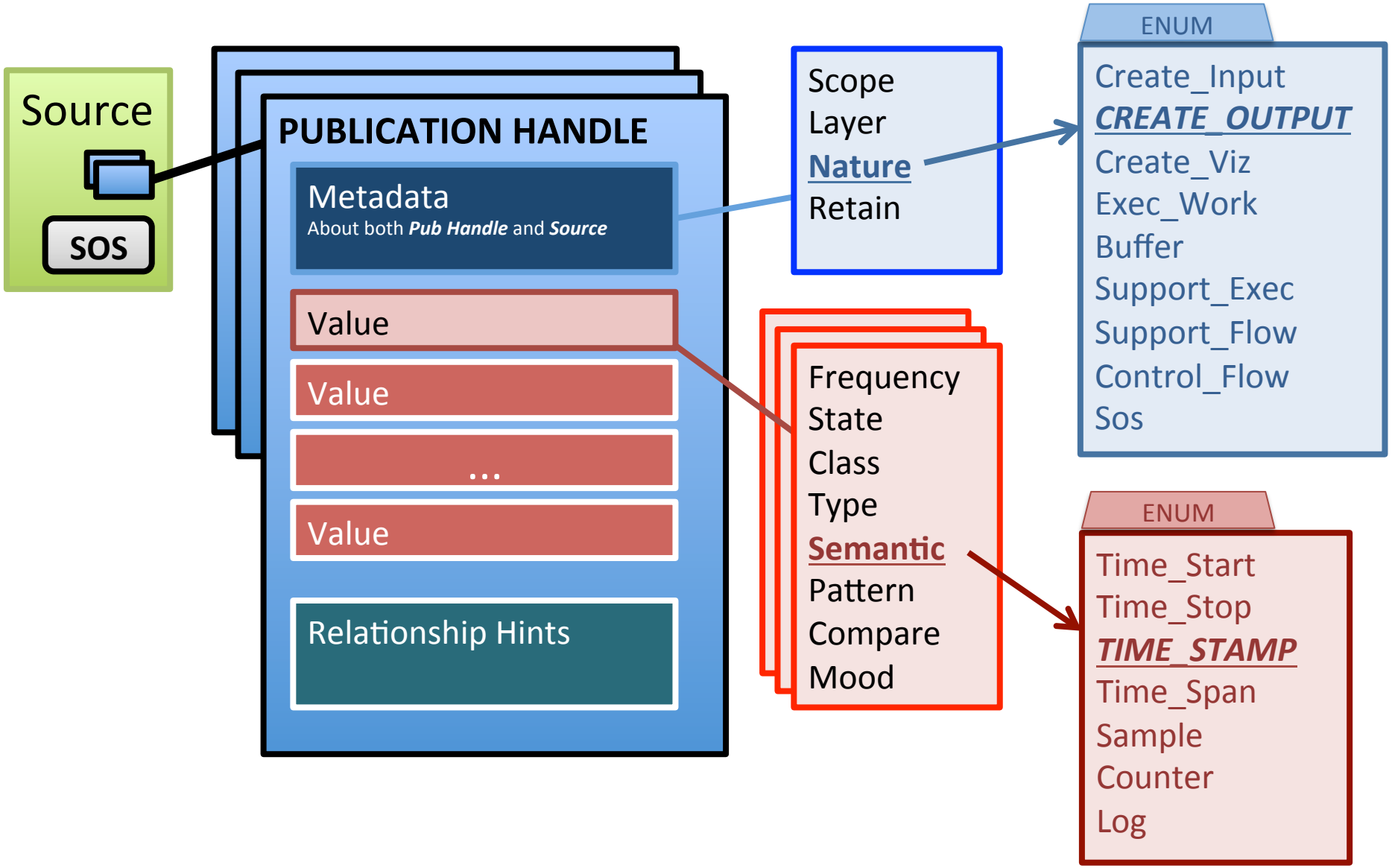
Analytics modules form independent communication channels

SOSflow: HPC Allocations



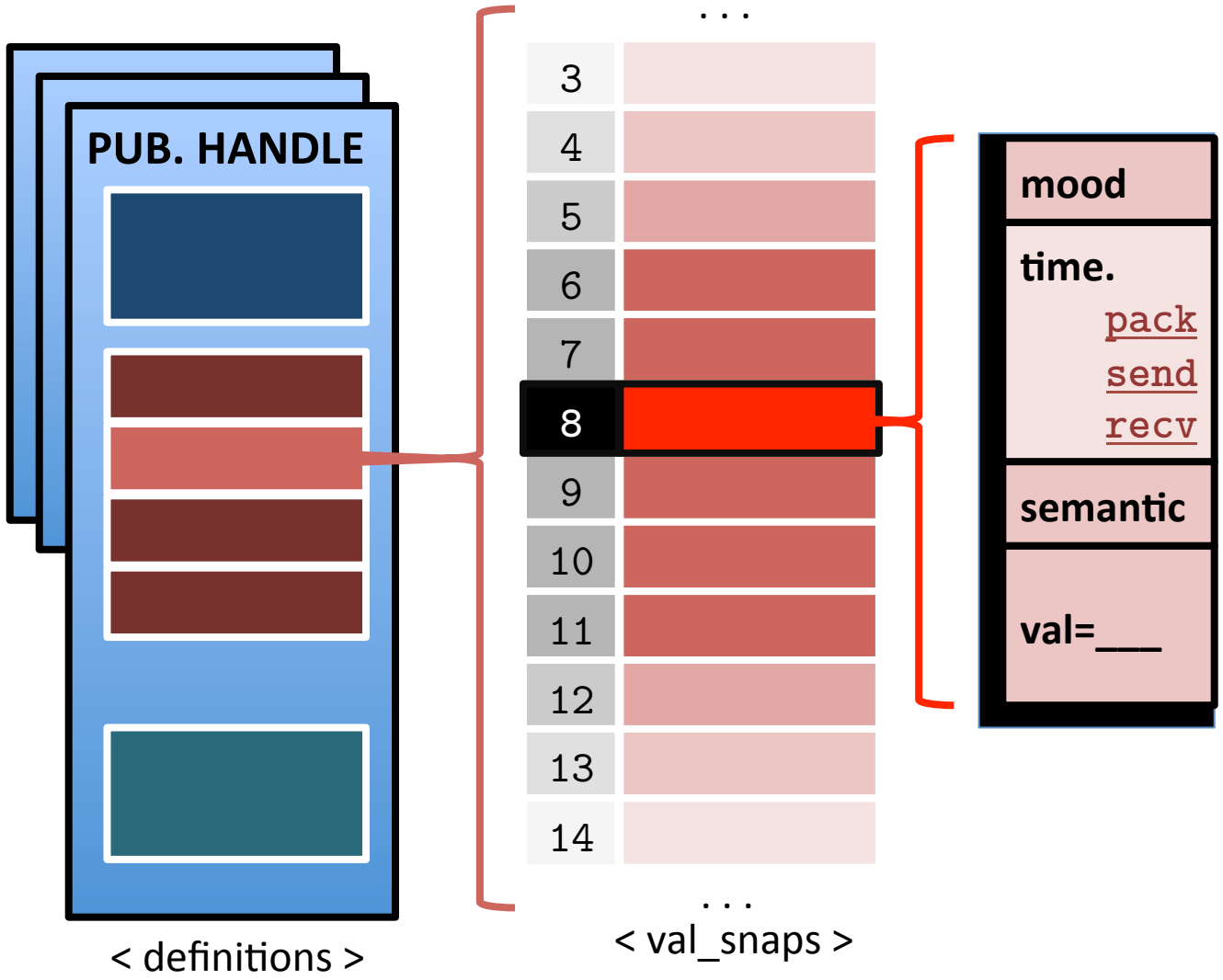
SOSflow data is **continuous** and **asynchronous**

SOSflow: Data Structure



SOSflow: Data Structure

Every value is conserved, with **its full history** and **evolving metadata**.



time.pack
stored by client

time.send
pushed to daemon

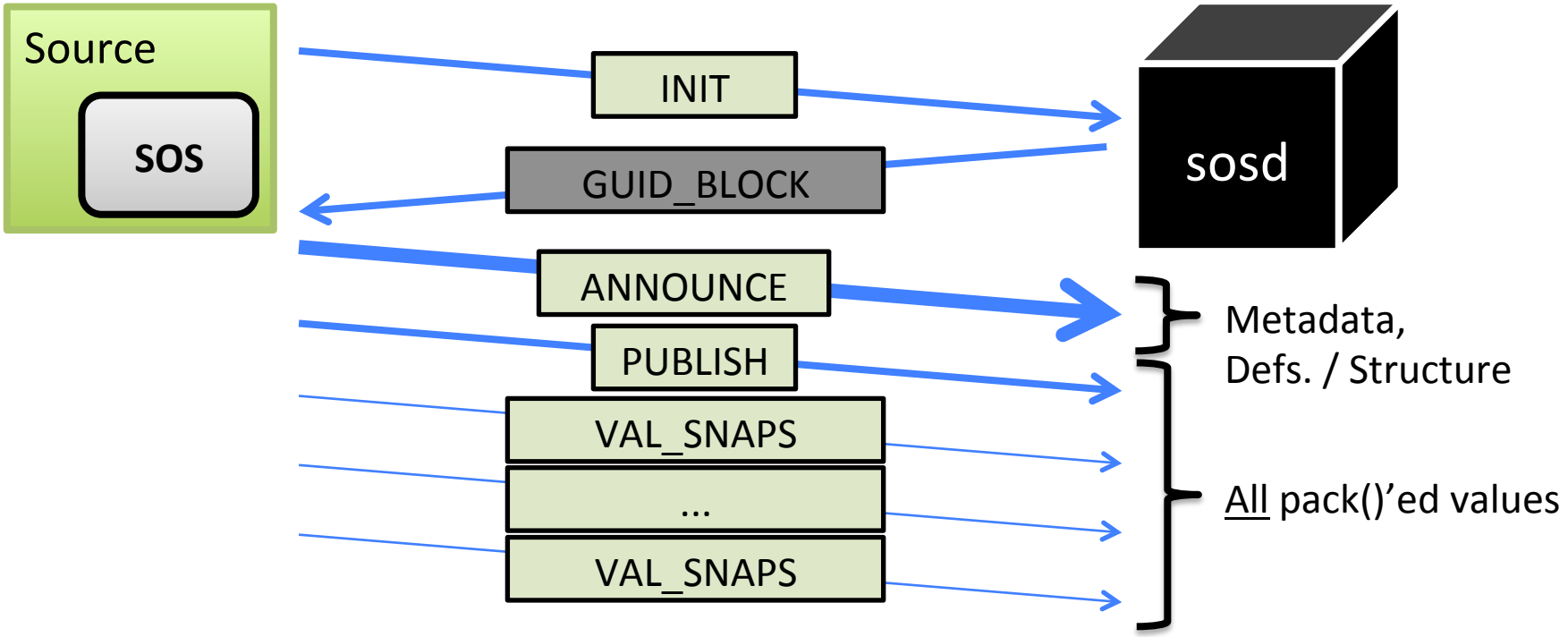
time.recv
injected into db

SOSflow: Easy to use API

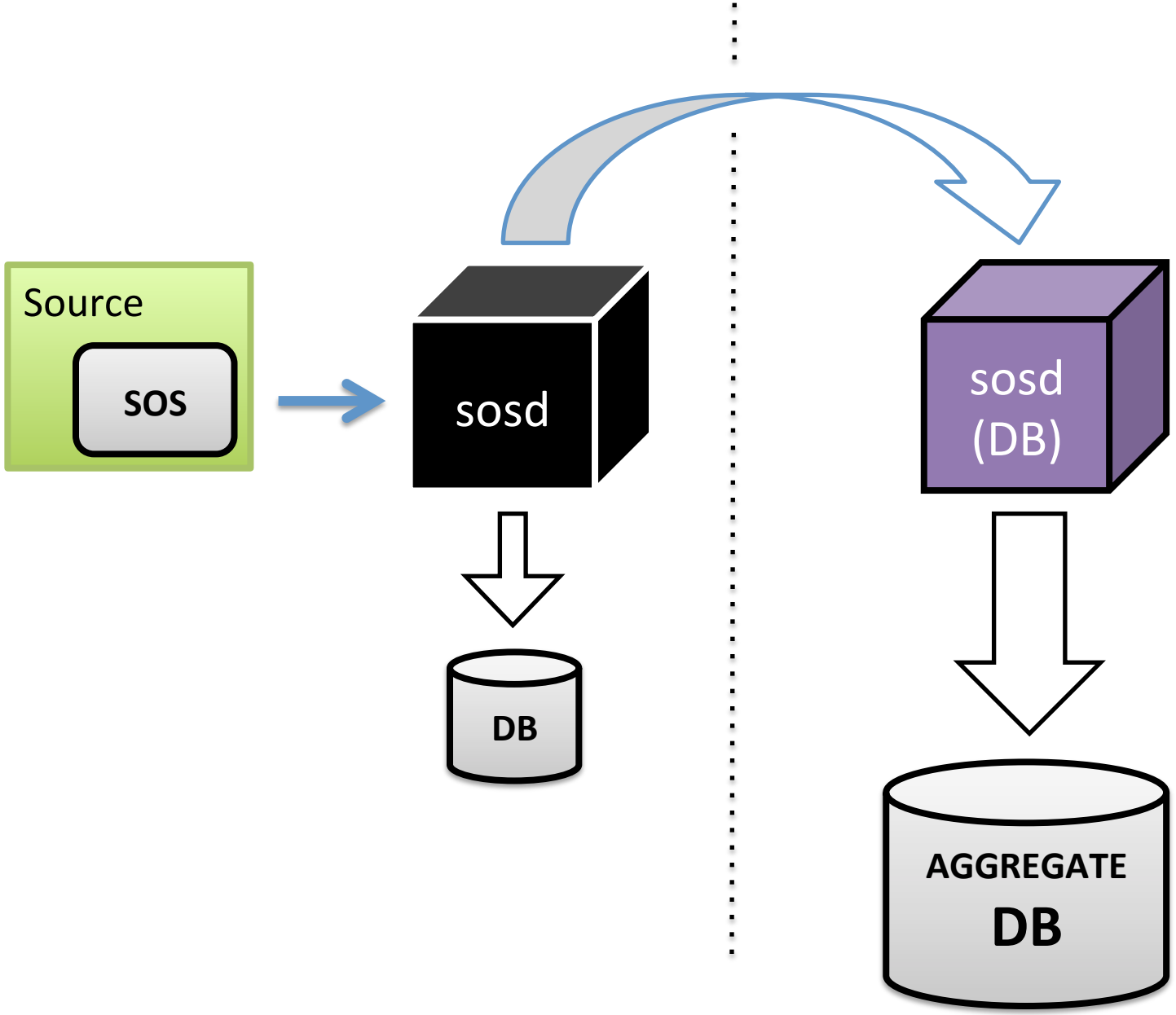
UNIVERSITY OF OREGON

```
int main(int argc, char *argv[]) {
    int      var_int    = 10;
    double   var_double = 88.8;
    char     var_string = "Hello, SOS.";
    SOS_pub *pub;

    SOS_init( &argc, &argv, SOS_ROLE_CLIENT );


    pub = SOS_new_pub("demo");

    SOS_pack(pub, "example_int", SOS_VAL_TYPE_INT,    (SOS_val) var_int    );
    SOS_pack(pub, "example_dbl", SOS_VAL_TYPE_DOUBLE, (SOS_val) var_double );
    SOS_pack(pub, "example_str", SOS_VAL_TYPE_STRING, (SOS_val) var_string );

    SOS_announce(pub);
    SOS_publish(pub);


                                . . .

    SOS_finalize();

    return (EXIT_SUCCESS);
}
```

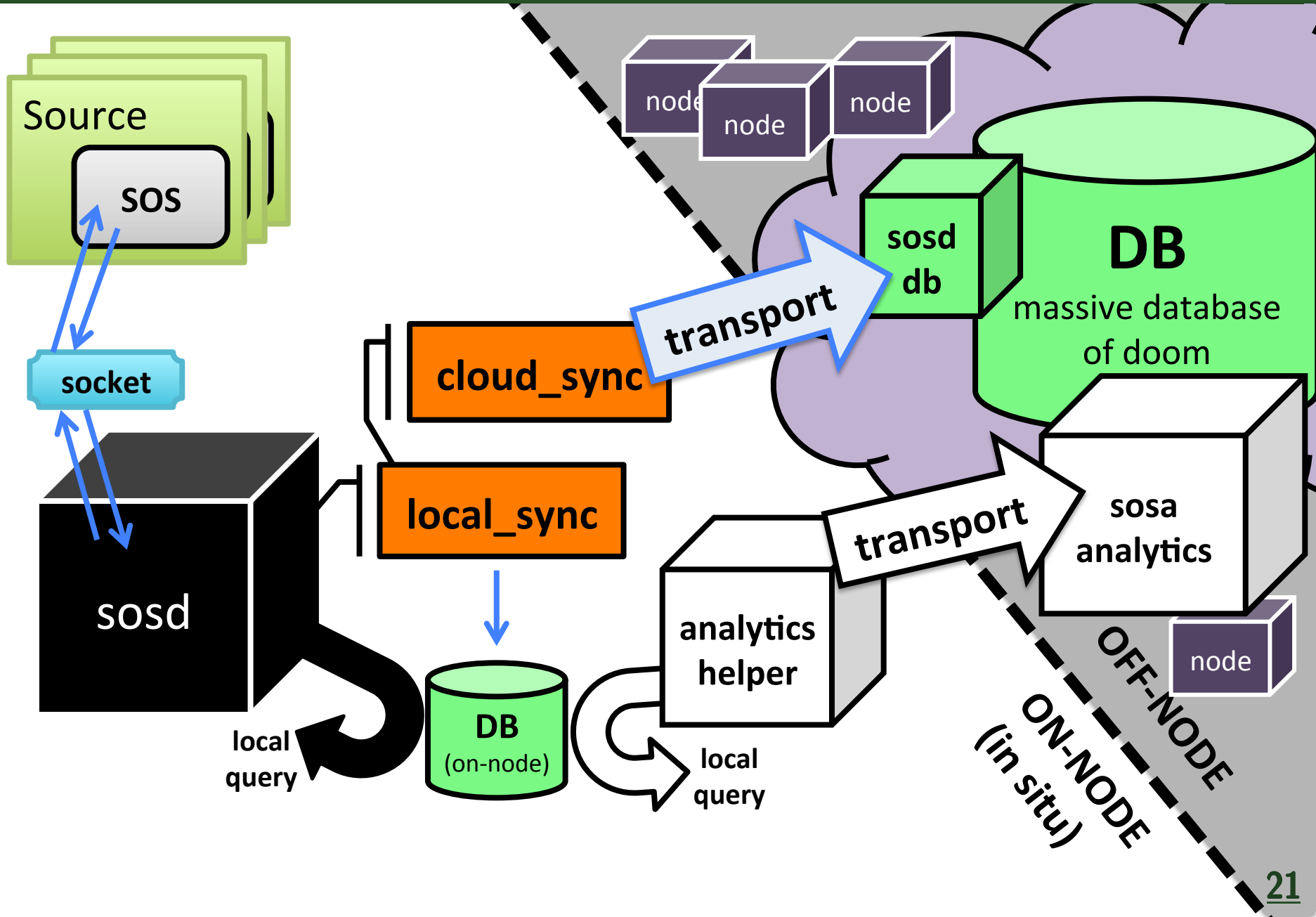
SOSflow: In Situ Socket Communication Protocol



SOSflow: Distributed Asynchronous Runtime (Simple)



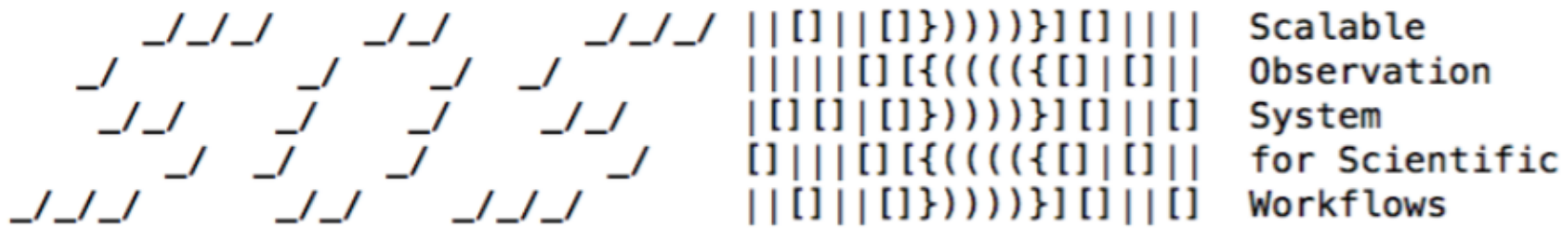
SOSflow: Distributed Asynchronous Runtime



SOSflow: Where it Runs

- NERSC
  - Cori
  - Edison
- LLNL
  - CAB
  - Catalyst
- University of Oregon
  - ACISS

- Software:
  - OpenMPI
  - MPICH
  - Slurm
  - PBS

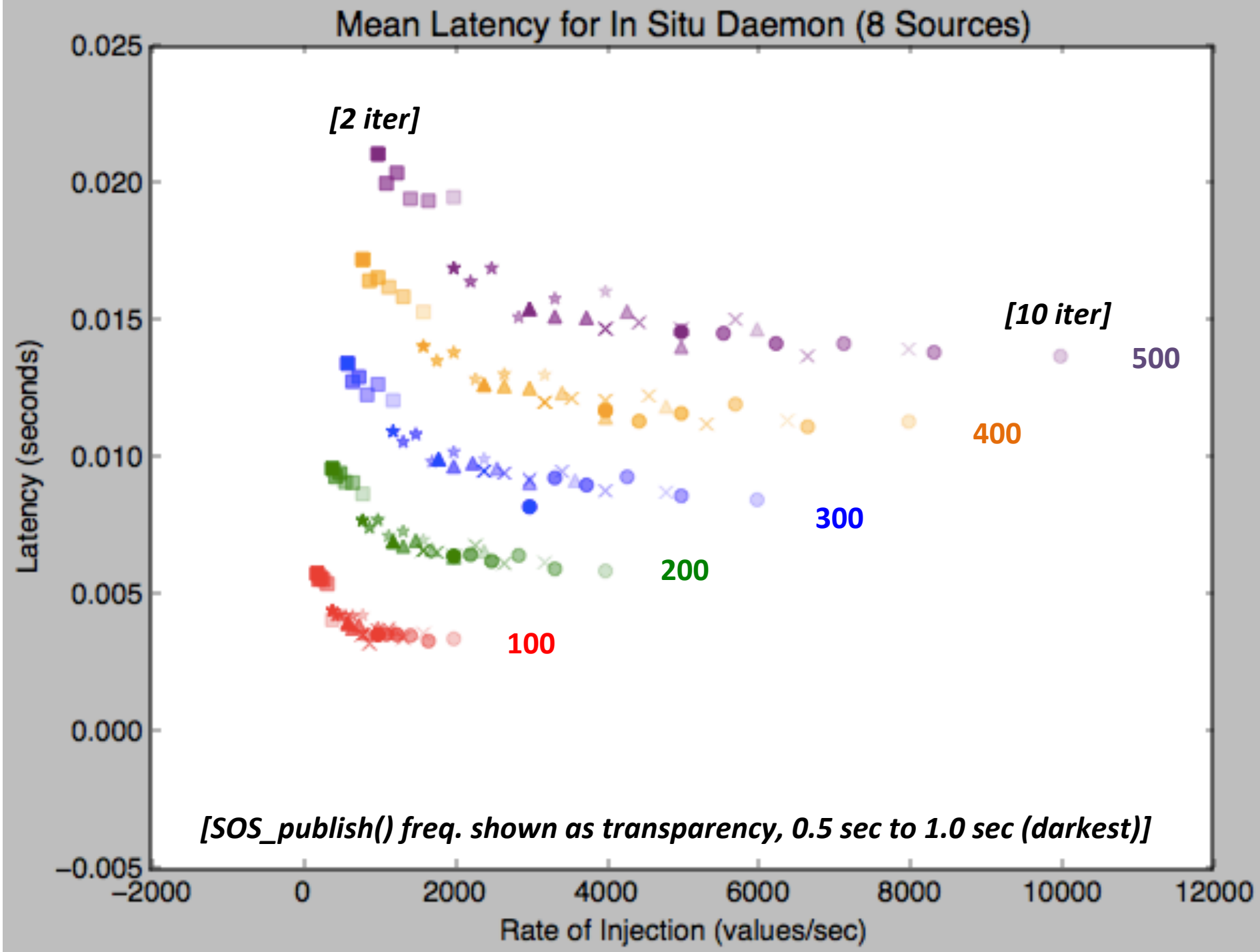


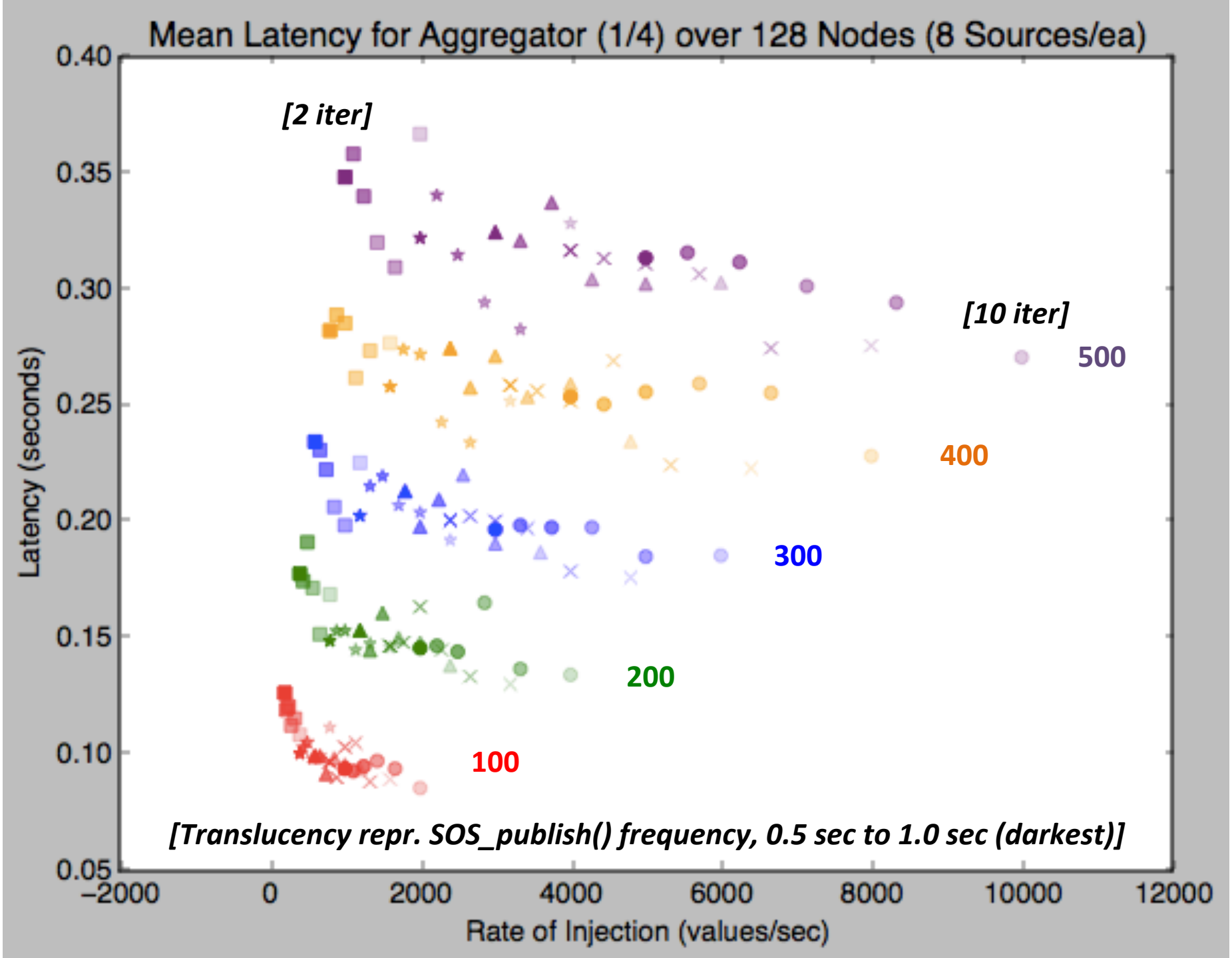
SOSflow: Evaluation

UNIVERSITY OF OREGON

- Experimental Setup
  - Explore performance of work-in-progress implementation
  - Synthetic and real-world cases
  - What is the latency cost of being async?

- Synthetic Sweep of Parameters
  - Iterations: 2 to 10, steps of 2
  - Size: 100 to 500 unique values per pub, steps of 100
  - Delay: 0.5 to 1.0 second, each 0.1 second

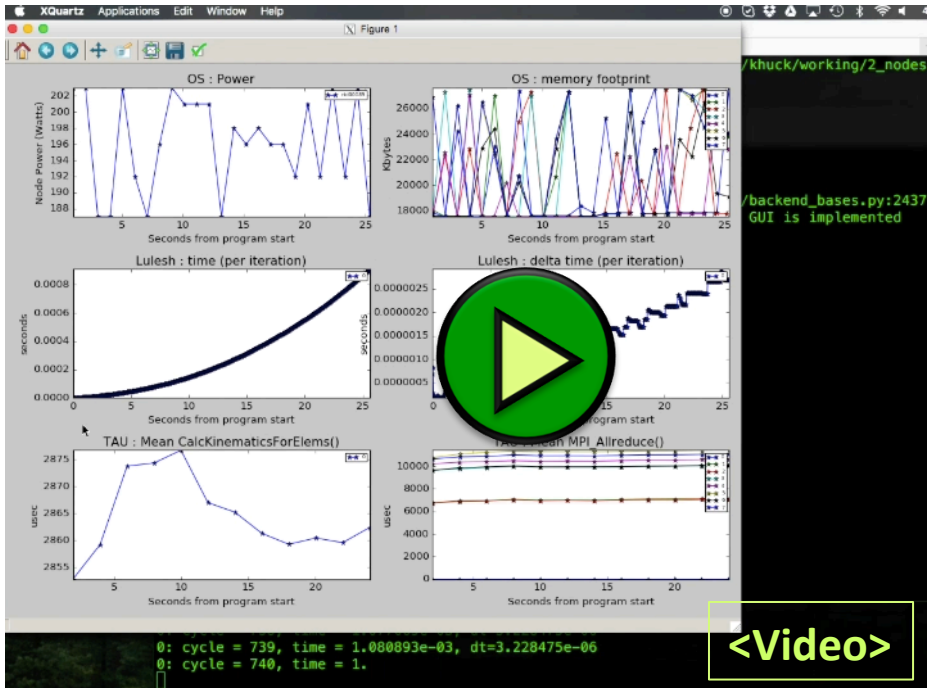

Mean Latency for In Situ Daemon (8 Sources)
Latency (seconds)
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
−2000
0
2000
4000
6000
8000
10000
12000
Rate of Injection (values/sec)
[2 iter]
[10 iter]
500
400
300
200
100
[SOS_publish() freq. shown as transparency, 0.5 sec to 1.0 sec (darkest)]

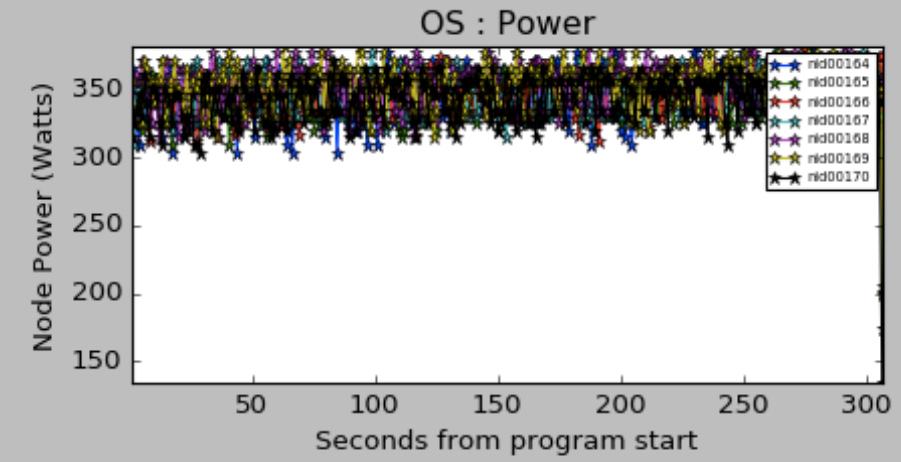

Mean Latency for Aggregator (1/4) over 128 Nodes (8 Sources/ea)
[2 iter]
[10 iter]
500
400
300
200
100
[Translucency repr. SOS_publish() frequency, 0.5 sec to 1.0 sec (darkest)]
Latency (seconds)
Rate of Injection (values/sec)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
−2000
0
2000
4000
6000
8000
10000
12000

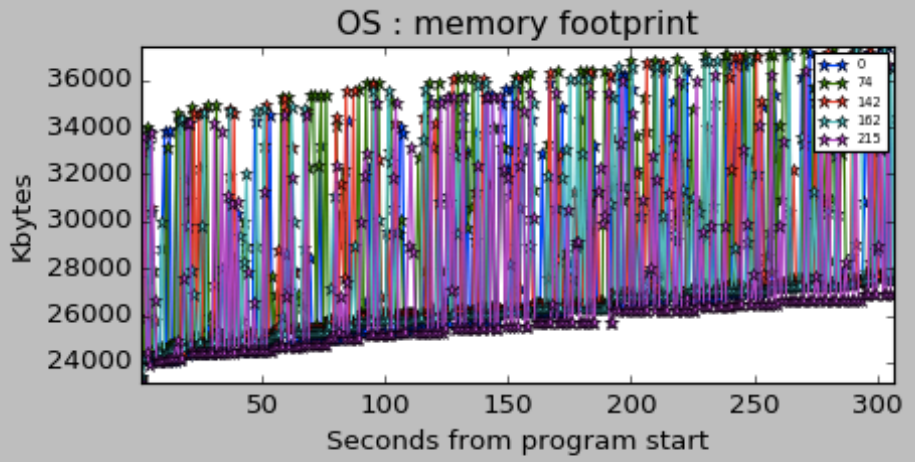
SOSflow: Evaluation

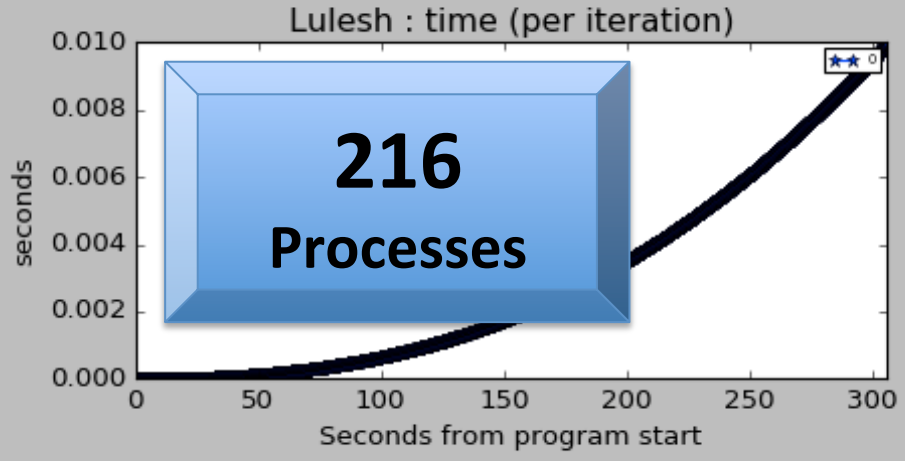
UNIVERSITY OF OREGON

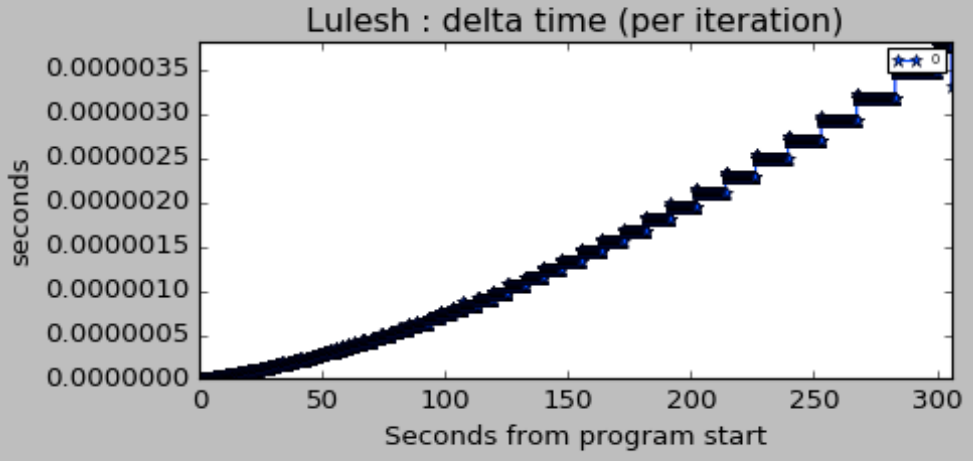
- Real-World Scenario
  - TAU Instrumented LULESH on Cori
  - TAU reports results to SOSflow on a timer
  - LULESH calls SOSflow API directly at iteration
  - SOSflow gathers metrics from the OS

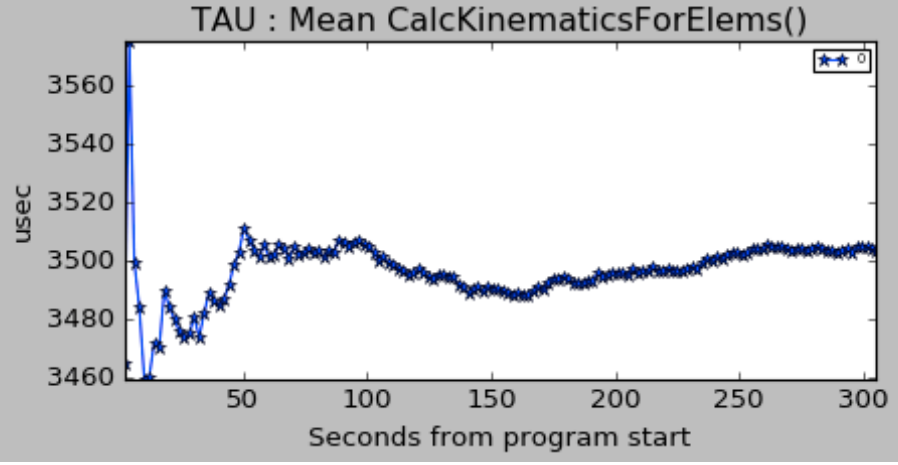


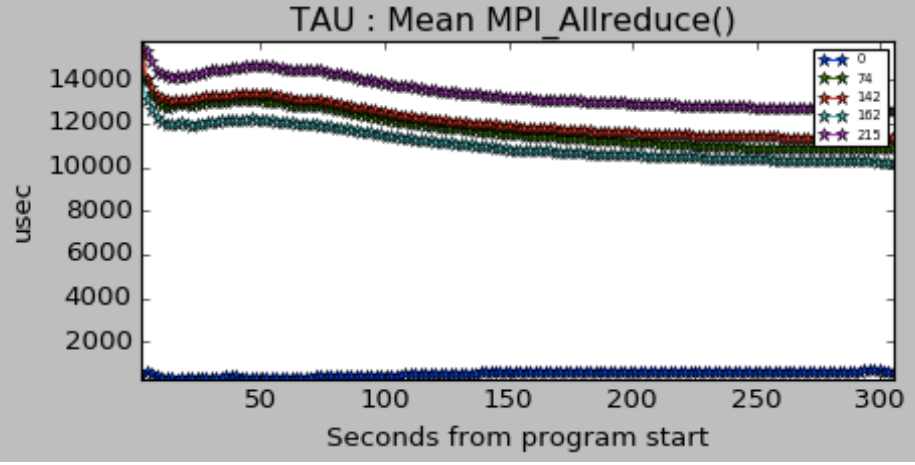

OS : Power
Node Power (Watts)
Seconds from program start
nid00164
nid00165
nid00166
nid00167
nid00168
nid00169
nid00170

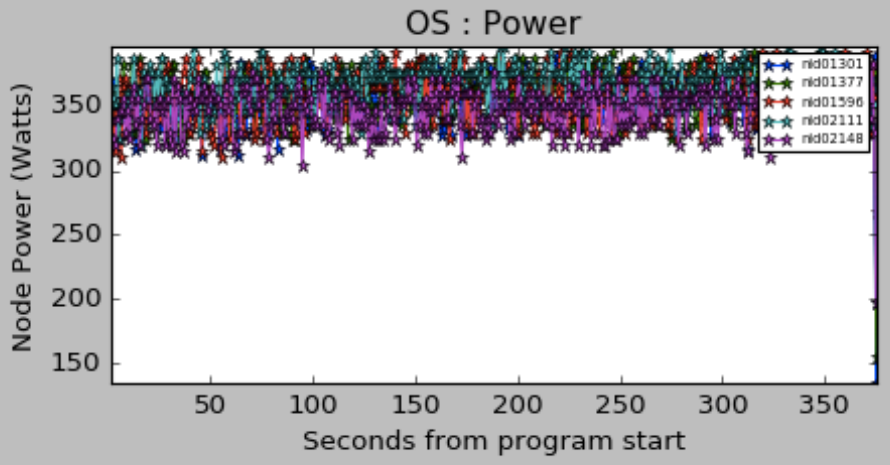

OS : memory footprint
Kbytes
Seconds from program start
0
74
142
162
215

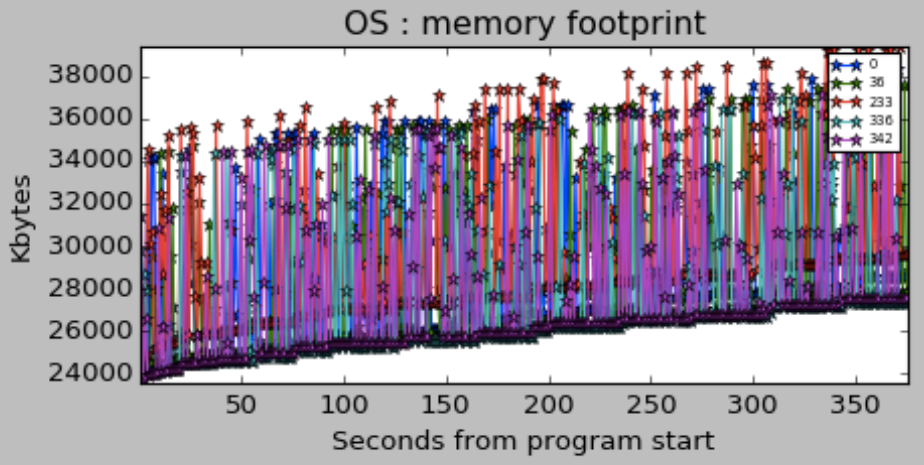

Lulesh : time (per iteration)
seconds
Seconds from program start
0
216
Processes

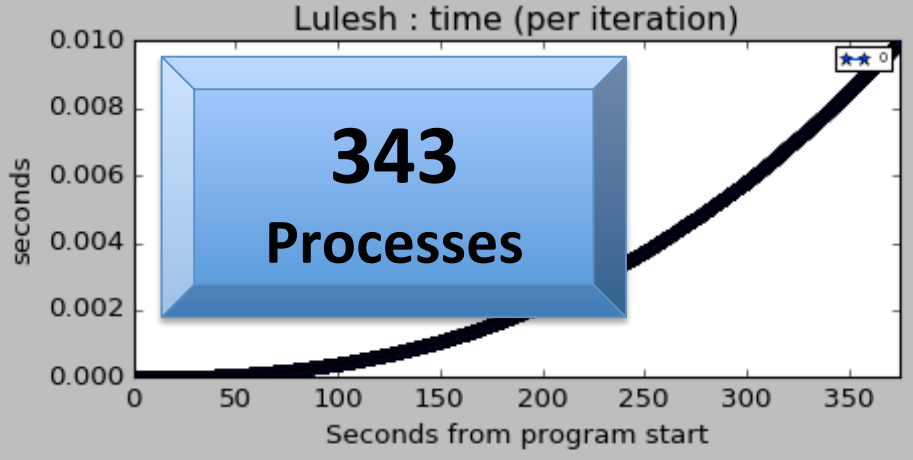

Lulesh : delta time (per iteration)
seconds
Seconds from program start
0

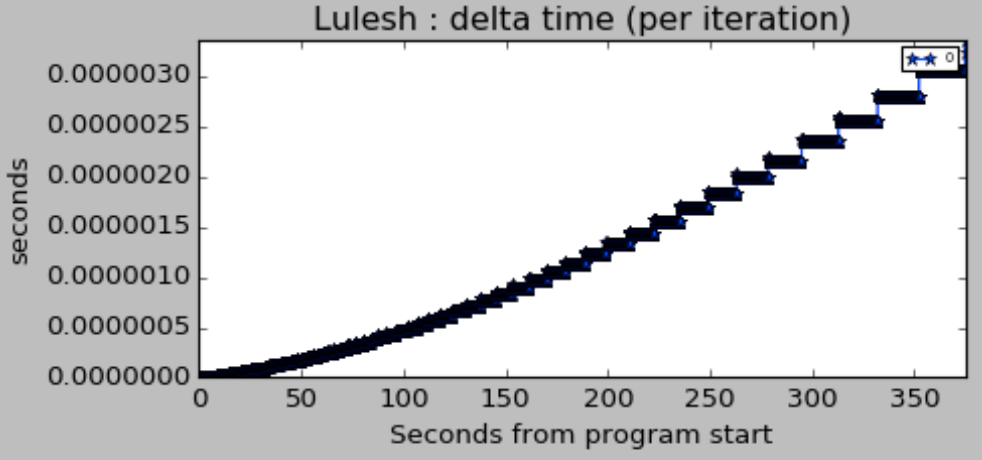

TAU : Mean CalcKinematicsForElems()
usec
Seconds from program start
0

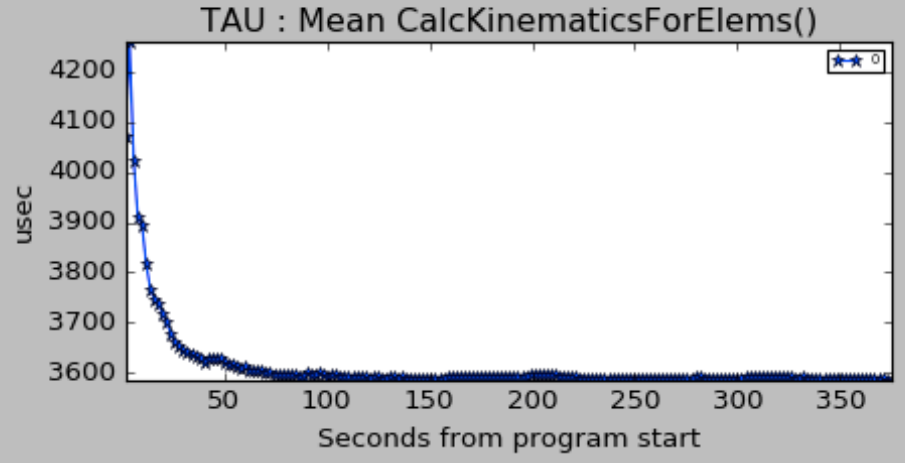

TAU : Mean MPI_Allreduce()
usec
Seconds from program start
0
74
142
162
215

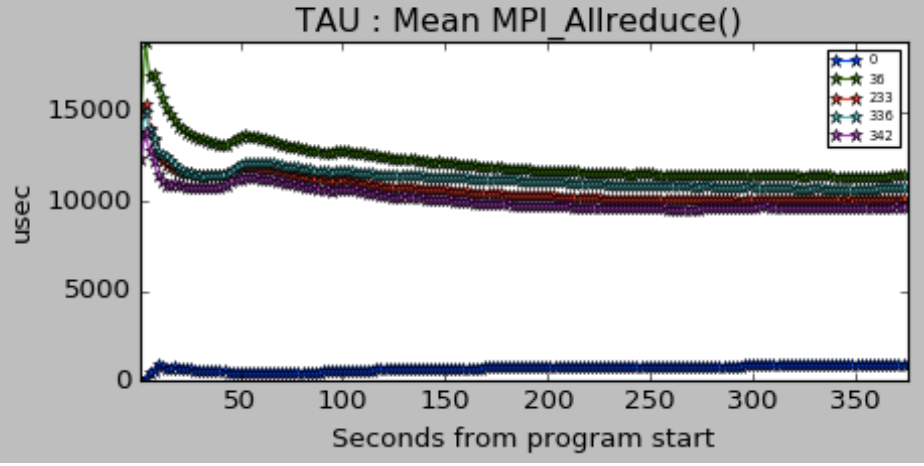

OS : Power
Node Power (Watts)
Seconds from program start
nid01301
nid01377
nid01596
nid02111
nid02148

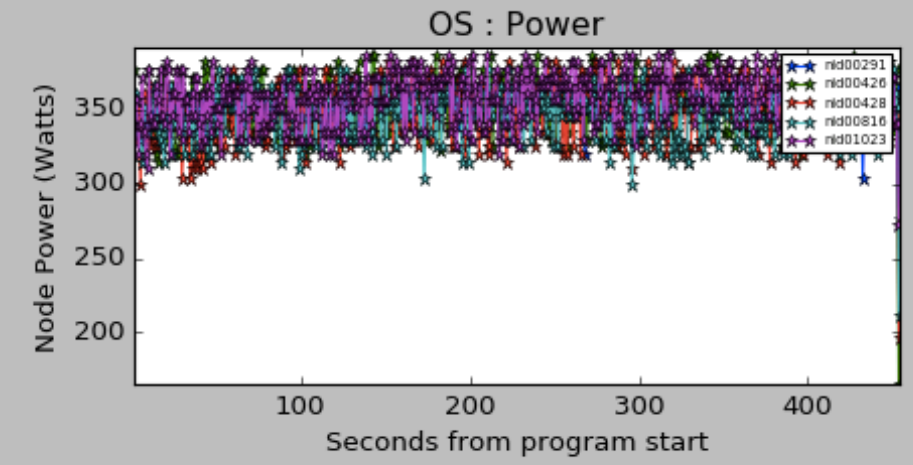

OS : memory footprint
Kbytes
Seconds from program start
0
36
233
336
342

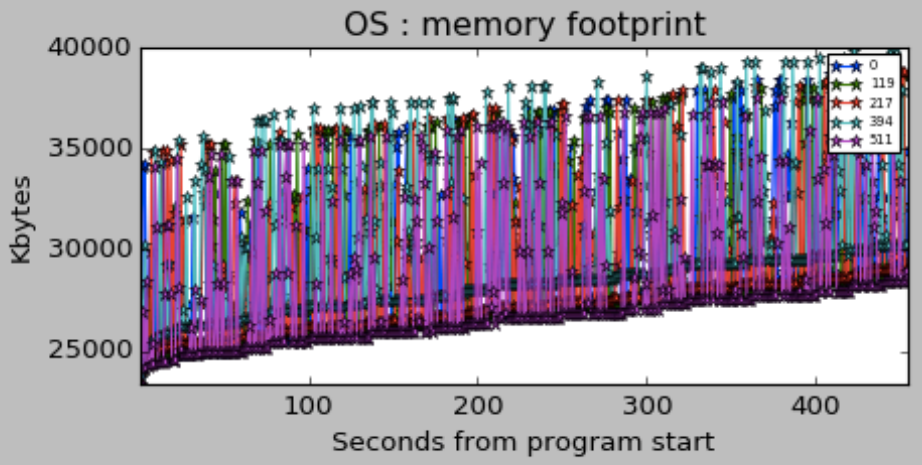

Lulesh : time (per iteration)
seconds
Seconds from program start
343
Processes

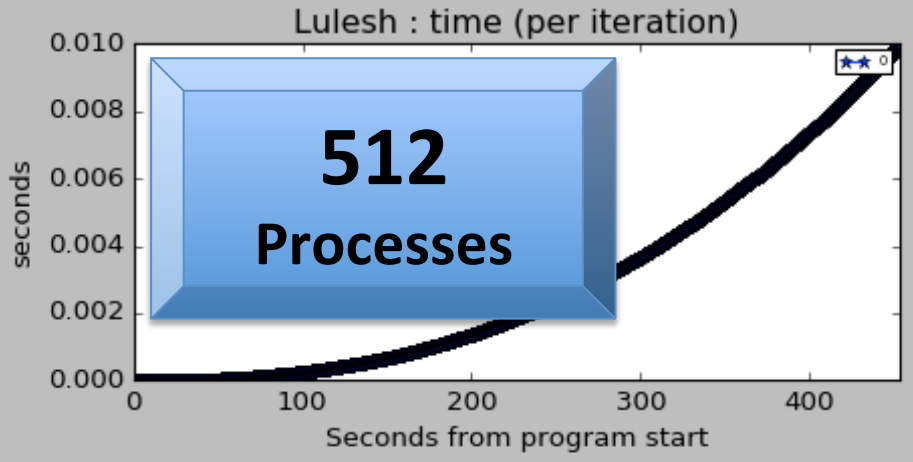

Lulesh : delta time (per iteration)
seconds
Seconds from program start

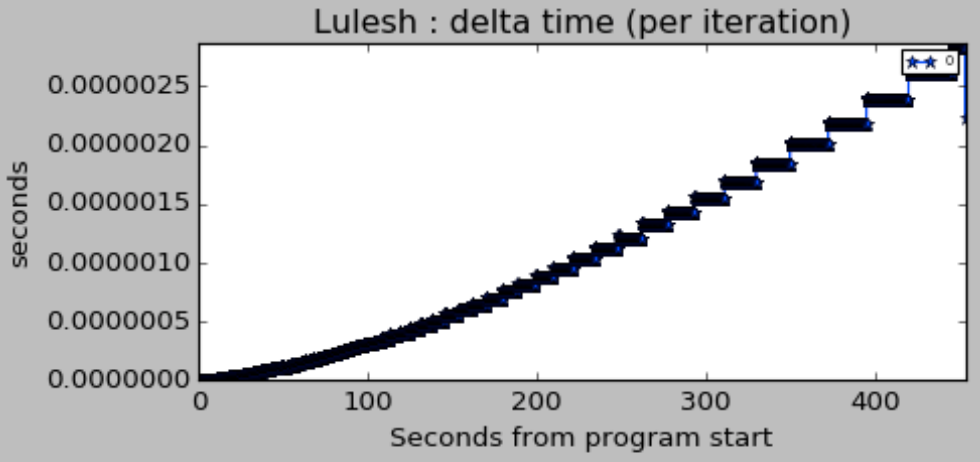

TAU : Mean CalcKinematicsForElems()
usec
Seconds from program start

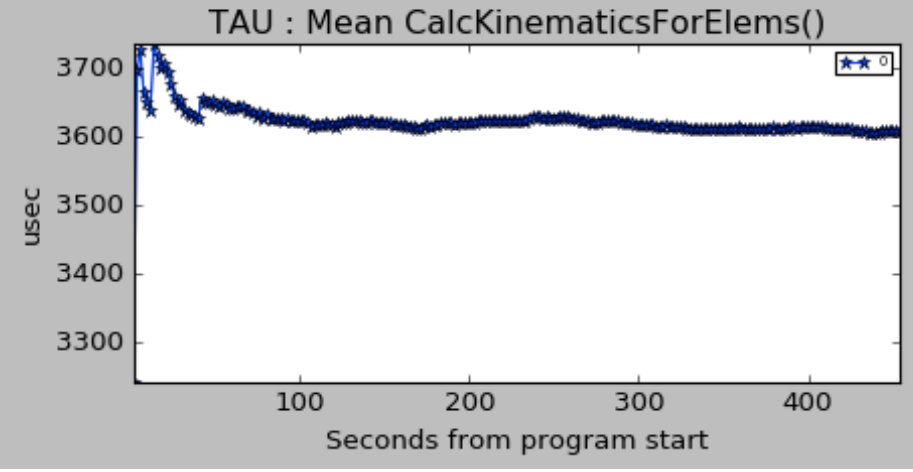

TAU : Mean MPI_Allreduce()
usec
Seconds from program start
0
36
233
336
342

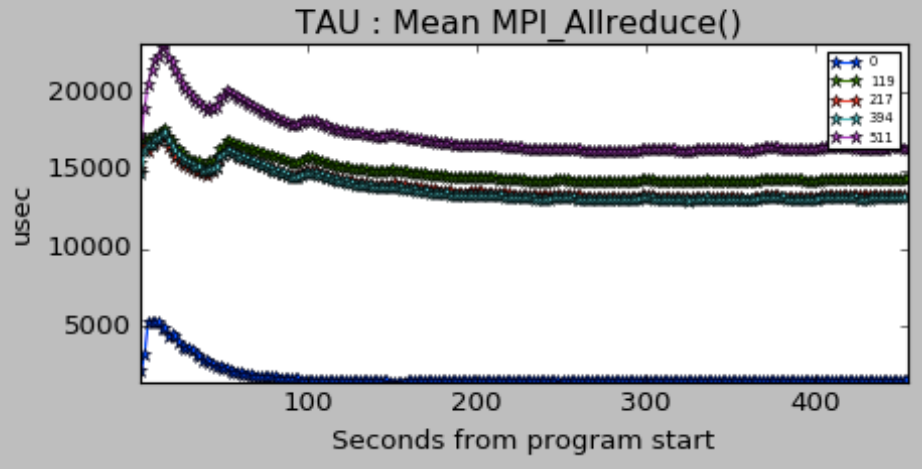

OS : Power
Node Power (Watts)
350
300
250
200
100
200
300
400
Seconds from program start
nid00291
nid00426
nid00428
nid00816
nid01023


OS : memory footprint
Kbytes
40000
35000
30000
25000
100
200
300
400
Seconds from program start
0
119
217
394
511


Lulesh : time (per iteration)
seconds
0.010
0.008
0.006
0.004
0.002
0.000
0
100
200
300
400
Seconds from program start
0
512
Processes


Lulesh : delta time (per iteration)
seconds
0.0000025
0.0000020
0.0000015
0.0000010
0.0000005
0.0000000
0
100
200
300
400
Seconds from program start
0


TAU : Mean CalcKinematicsForElems()
usec
3700
3600
3500
3400
3300
100
200
300
400
Seconds from program start
0


TAU : Mean MPI_Allreduce()
usec
20000
15000
10000
5000
100
200
300
400
Seconds from program start
0
119
217
394
511

Future Work

UNIVERSITY OF OREGON

- Performance improvements
- Integrate more automatic data gathering for node-level metrics
- Support for deep analytics
- Testing with additional real-world workflows and operating environments