A HPC CO-SCHEDULER WITH REINFORCEMENT LEARNING

Abel Souza, * Kristiaan Pelckmans, Johan Tordsson

abel.souza@cs.umu.se

https://asouza.io

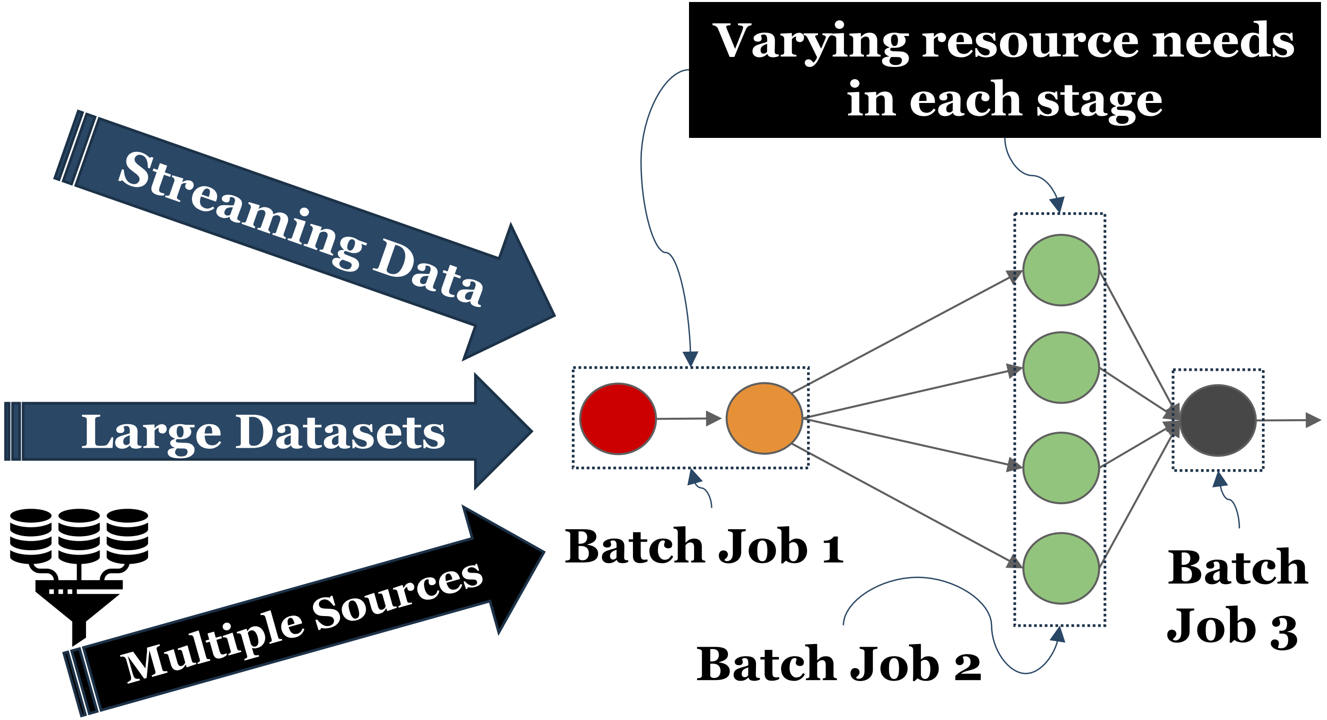
UMEÅ UNIVERSITY

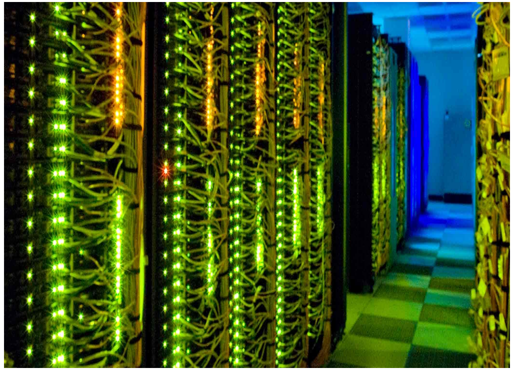
*

UPPSALA UNIVERSITET

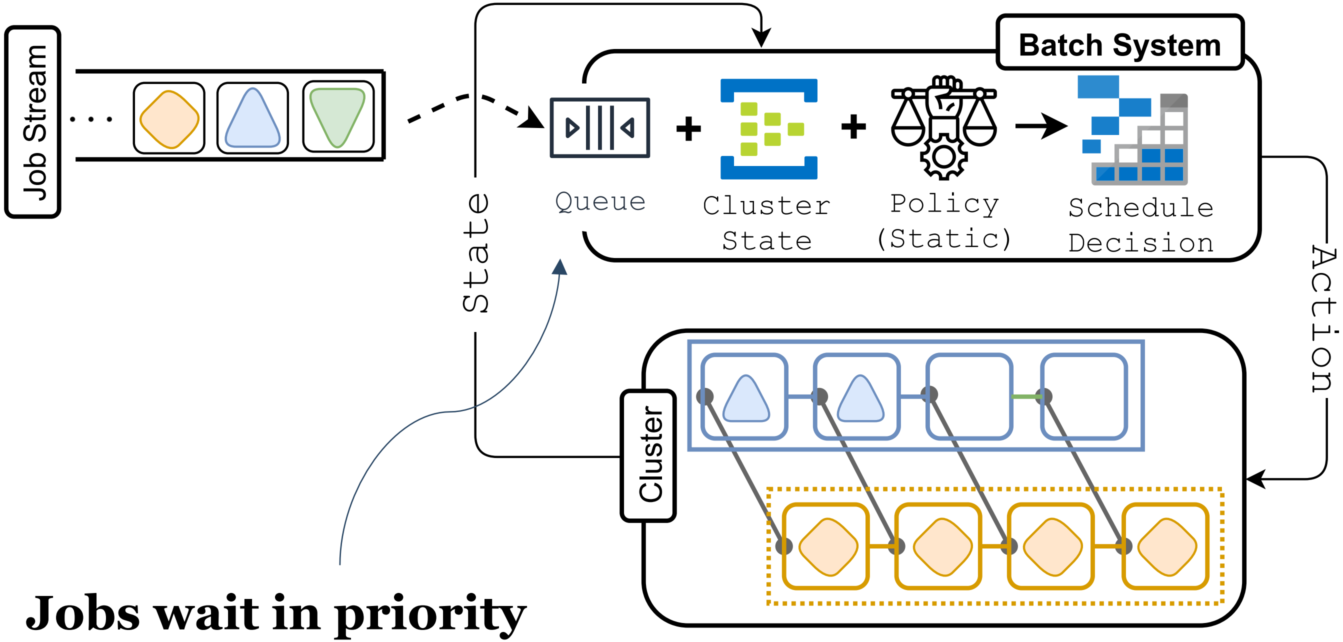
HPC JOBS

Experimental data are increasingly processed on HPC systems as scientific workflows





HPC CONCEPTUAL ARCHITECTURE: SPACE SHARING



Jobs wait in priority queues until free resources are available

- Common HPC scheduling policies are static and do not allow collocation

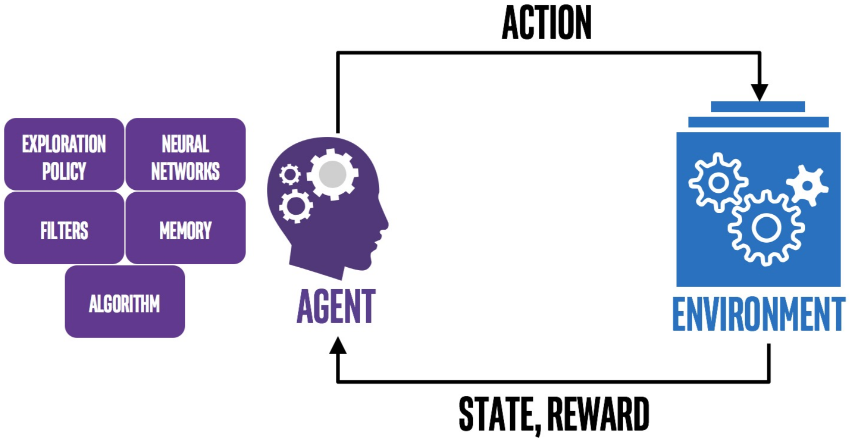
- These policies do not take into consideration cluster state history

REINFORCEMENT LEARNING IN SCHEDULING

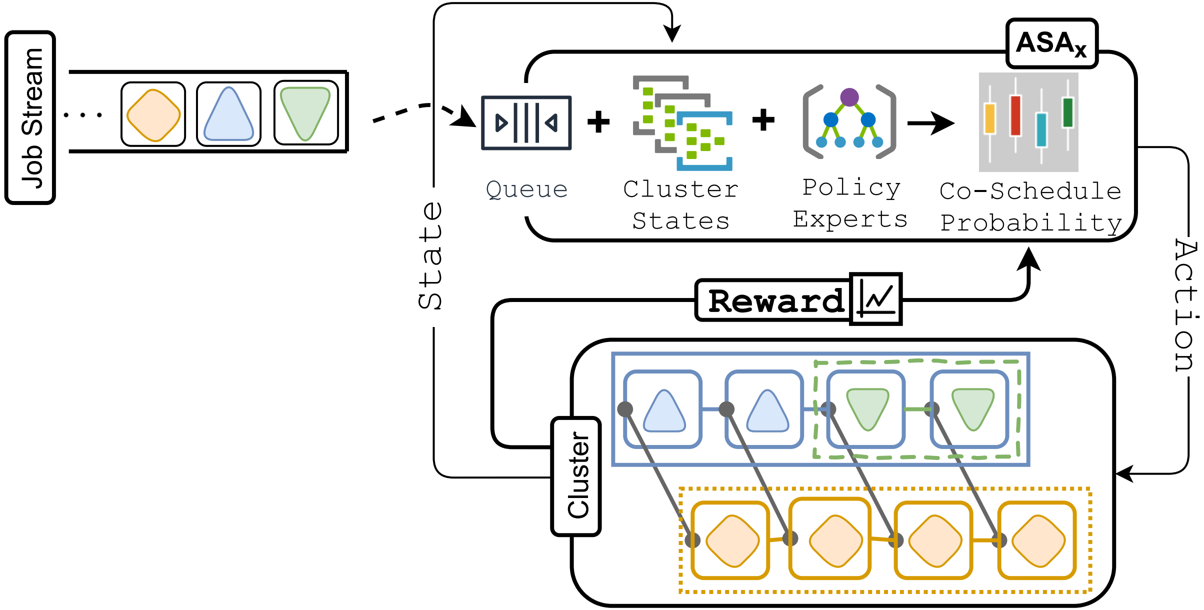
- Reinforcement Learning is a mathematical framework that optimizes rewards by acting on the environment

- Integrating it into HPC, we would have:

    - **Agent**: Scheduler
    - **Environment**: The cluster
    - **Action**: Resource allocation
    - **Reward**: How faster/slower jobs run
    - **State**: Key performance Indicators



ASAX – THE ADAPTIVE SCHEDULING AND EXTENDED ARCHITECTURE



- Each scheduling decision is evaluated prior and post (Reward) collocation
- Policy experts evaluate analytically the schedule (Action) probability of success for a given cluster state
- RL algorithm bounds error to avoid slowing applications down

ASAX – A HPC CO-SCHEDULER WITH REINFORCEMENT LEARNING

- ASAx transparently abstracts the resource management from applications
  - Fault-tolerance
  - Resource isolation/control
  - Elasticity
  - Runs on top of Apache Mesos

- The architecture allows new scheduling capabilities to be assessed:
  - We present a Reinforcement-Learning scheduling algorithm to estimate how overcommitting resources affect jobs runtimes
  - The algorithm adapts to mistakes and new situations

ASAX – THE CO-SCHEDULING ALGORITHM

Algorithm 1 ASA_X

Require:

Queued Job_b

Job_a allocation satisfying Job_b `#In terms of resource`

m co-allocation actions a, e.g. $m = 10$ and $a = \{a_0 = 0\%, \ldots, a_{m-1} = 90\%\}$

Initialise $\alpha_{0i} = \frac{1}{n}$ for $i = 1, \ldots, n$ `#metrics in state` $\mathbf{x}$`, e.g. CPU, Mem.`

Handled by Mesos

Linux CFS bandwidth cgroups control

```
for t = 1, 2, ... do
    Initialise co-allocation risk r_ti = 0 for each i-th metric
    #Co-allocation assessment:
    while max_i r_ti ≤ 1 do
        Evaluate Job_a's state x:
            Compute each ith-DT metric expert p_i(x) ∈ R^m
            Aggregate Job_a's co-schedule probability as p = Σ_{i=1}^n α_{t-1,i} p_i(x) ∈ R^m
        j = sample one action from a according to p
        Allocate a_j from Job_a's to co-schedule Job_b
        Compute Job_a's performance loss |ℓ(a_j)| ≤ 1 due to co-allocation
        For all i, update Job_a's risk r_ti = r_ti + p_i(a_j) ℓ(a_j)
    end while
    For Job_a and for all i, update α_{t,i} as
end for=0
```

for $t = 1, 2, \ldots$ **do**

Initialise co-allocation risk $\mathbf{r}_{ti} = 0$ for each i-th metric

`#Co-allocation assessment:`

while $\max_i \mathbf{r}_{ti} \leq 1$ **do**

Evaluate Job_a's state $\mathbf{x}$:

Compute each ith-DT metric expert $\mathbf{p}_i(\mathbf{x}) \in \mathbb{R}^m$

Aggregate Job_a's co-schedule probability as $\mathbf{p} = \sum_{i=1}^{n} \alpha_{t-1,i} \mathbf{p}_i(\mathbf{x}) \in \mathbb{R}^m$

j = sample one action from a according to $\mathbf{p}$

Allocate a_j from Job_a's to co-schedule Job_b

Compute Job_a's performance loss $|\ell(a_j)| \leq 1$ due to co-allocation

For all i, update Job_a's risk $\mathbf{r}_{ti} = \mathbf{r}_{ti} + \mathbf{p}_i(a_j)\ell(a_j)$

end while

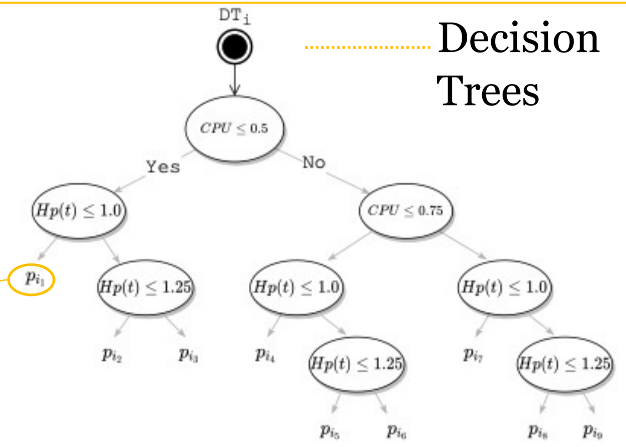
For Job_a and for all i, update $\alpha_{t,i}$ as

$$\alpha_{t,i} = \frac{\alpha_{t-1,i}}{N_t} \times e^{-\gamma_t \mathbf{r}_{ti}}$$

where N_t is a normalising factor such that $\sum_{i=1}^{n} \alpha_{t,i} = 1$.

end for=0

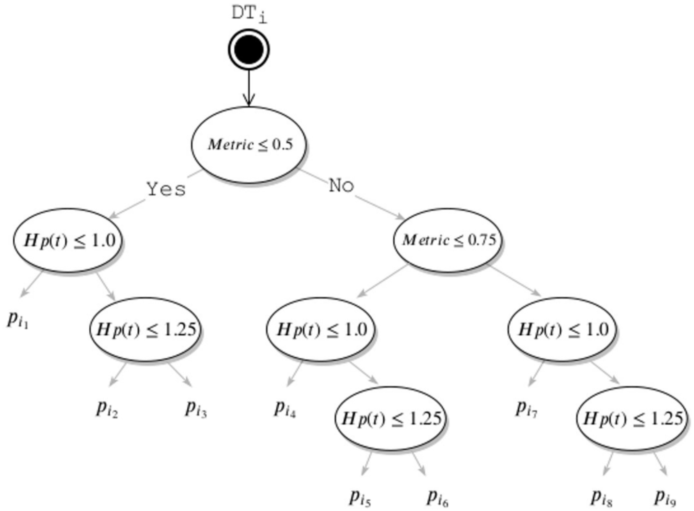
Adapts learning when “too much losses”



Decision Trees

Recommendation of what action to take (CPU %)

HAPPINESS



The *Hp(t)* (happiness) metric => if an application - in relation to a *target* - is doing "fine" at time *t*, then Hp(*t*) is equal or larger than **1.0**, and lower than 1.0 otherwise.

$$Hp(t) = \frac{|t_{Deadline} - t| * \#RemainingTasks}{\#Tasks/s}$$

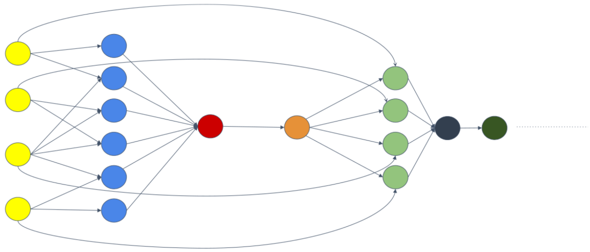
- Decision trees influences how **p** is built:
  - *Metric* can be any system's measurable
  - It can also be any **user-defined targets**, like deadline, latency, #Tasks/s, incoming data, or a combination of these, like Hp(t)
  - Each leaf has different weights with associated (**p**$_{ij}$) vectors impacting **p**
    - For instance, if a task has a high CPU %, then the scheduler should have a small probability to allocate high CPU quotas to extra tasks
    - This means the action to allocate "small quotas" is more likely to minimize the loss, e.g. pi7 means DTi evaluated CPU % > 0.75, then task collocation should be *less* recommended

EVALUATION (1/2)

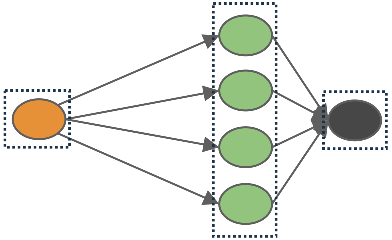
- System:
  - **NUMA Scale***: 6 nodes, 24-cores AMD Opteron, 185 GB
  - Slurm (18.08), default backfilling (BF)
- Four workflows:
  - Montage: I/O Intensive
  - BLAST: CPU Intensive
  - Statistics: Network Intensive
  - Synthetic: CPU + Memory Intensive
  - **Strategies: Slurm (BF), 50/50 CPU (Static), and ASAx**
- Submitted all at once, different job sizes configurations
  - 45 jobs in total
- **Three cluster sizes**
  - 64, 128, and 256 cores
  - 8x, 4x, and 2x occupation

- **Metrics**:
  - Total Runtime
  - Response Time
  - Waiting Time

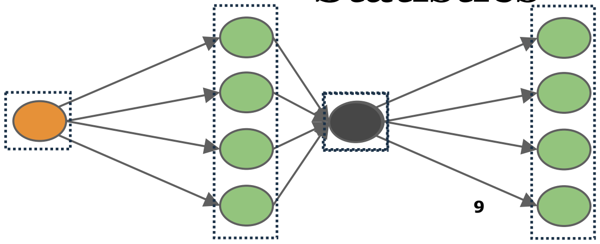
Montage



BLAST



Statistics



* https://www.numascale.com/

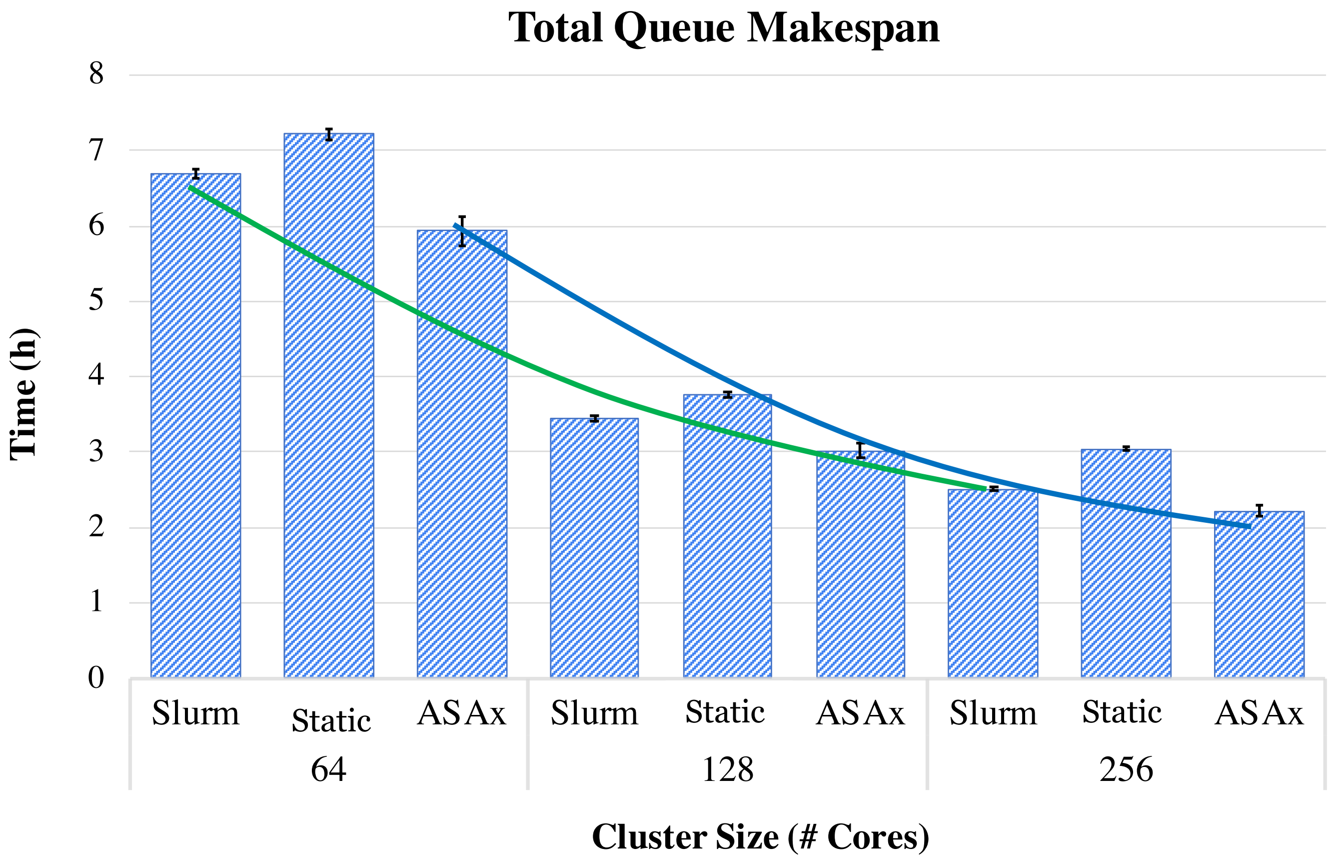
EVALUATION (2/2)

- **System sensors**:
  - CPU %
  - Memory %
  - Job Type
    - Parallel or
    - Sequential
  - Time interval
  - Happiness Metric (Hp(t))

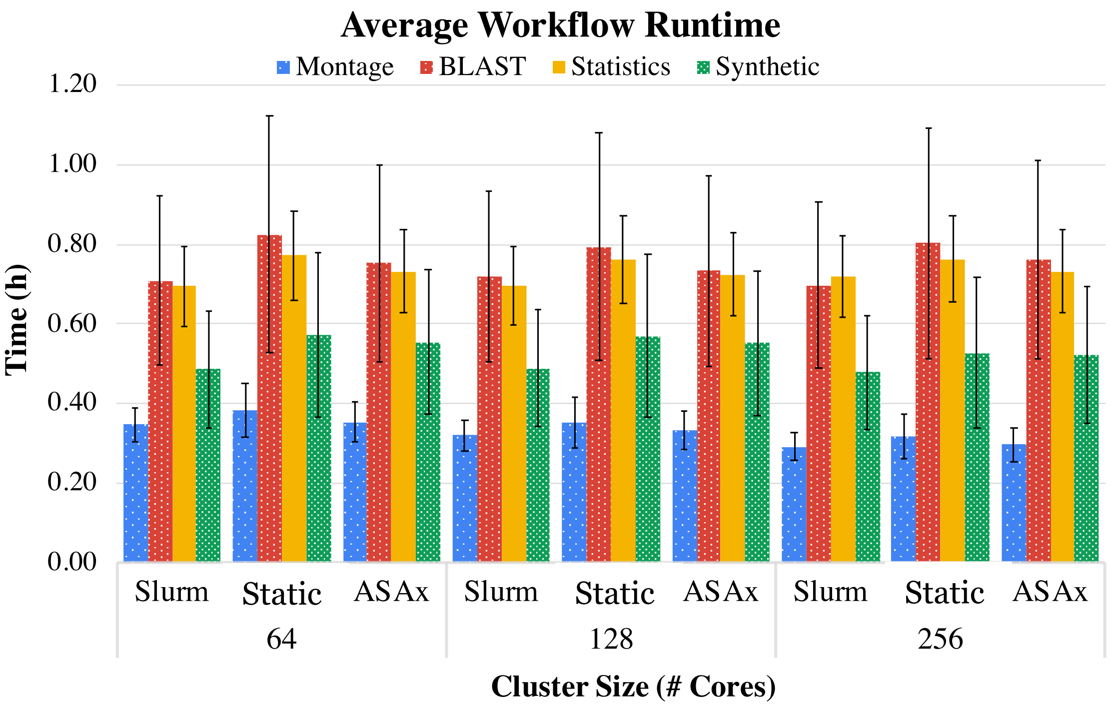
$$Hp(t) = \frac{|t_{Deadline} - t| * \#RemainingTasks}{\#Tasks/s}$$

The *Hp(t)* (happiness) metric => if an application - in relation to a *target* - is doing “fine” at time *t*, then Hp(*t*) is equal or larger than **1.0**. It is lower than 1.0 otherwise.

RESULTS (1/3) – TOTAL QUEUE MAKESPAN



RESULTS (2/3) – AVG WORKFLOW RUNTIME



- The fig. basically shows different runs for the same application with different number of cores in each run.
- If an application is scalable, the more cores, the faster the application runs, the larger the stdev.
- When the application is not scalable, more idle resources, thus the stdev will be small between the runs.

RESULTS (3/3) – RESPONSE TIME

	Cluster Size	Cluster Load	Waiting Time (h)	CPU Util. (%)	Response Time (h)
Slurm	64	8x	3.5±1%	53±5	4.4±1%
	128	4x	1.5±1%	45±5	2.4±1%
	256	2x	0.5±1%	32±6	1.4±1%
Static	64	8x	1.7±1%	90±5	4.8±1%
	128	4x	0.8±1%	92±3	2.8±1%
	256	2x	0.3±1%	89±5	2.0±1%
ASA_X	64	8x	1.8±5%	82±7	3.5±3%
	128	4x	1.0±8%	84±5	1.2±2%
	256	2x	0.3±7%	83±4	0.4±2%

CONCLUSIONS: SMART CO-SCHEDULING

- Low slow downs for majority of workloads
- ASAx has lower response time than Slurm and ASA
  - 20% improvements in response time over Slurm
  - Even with high cluster loads (8x)
- Measurements where new jobs come to the stream needs to be assessed
  - How would ASAx adapt to this?
  - Happiness metric seems to capture such behavior well
- How can we dynamically add/remove new decision trees to adapt scheduling at runtime?

A HPC CO-SCHEDULER WITH REINFORCEMENT LEARNING

Abel Souza, * Kristiaan Pelckmans, Johan Tordsson

abel.souza@cs.umu.se

https://asouza.io

UMEÅ UNIVERSITY

*

UPPSALA UNIVERSITET