

Workflow Software for Managing Large-Scale Job Campaigns at ALCF



CHRISTINE SIMPSON

Assistant Computational Scientist

Data Science Group

ALCF

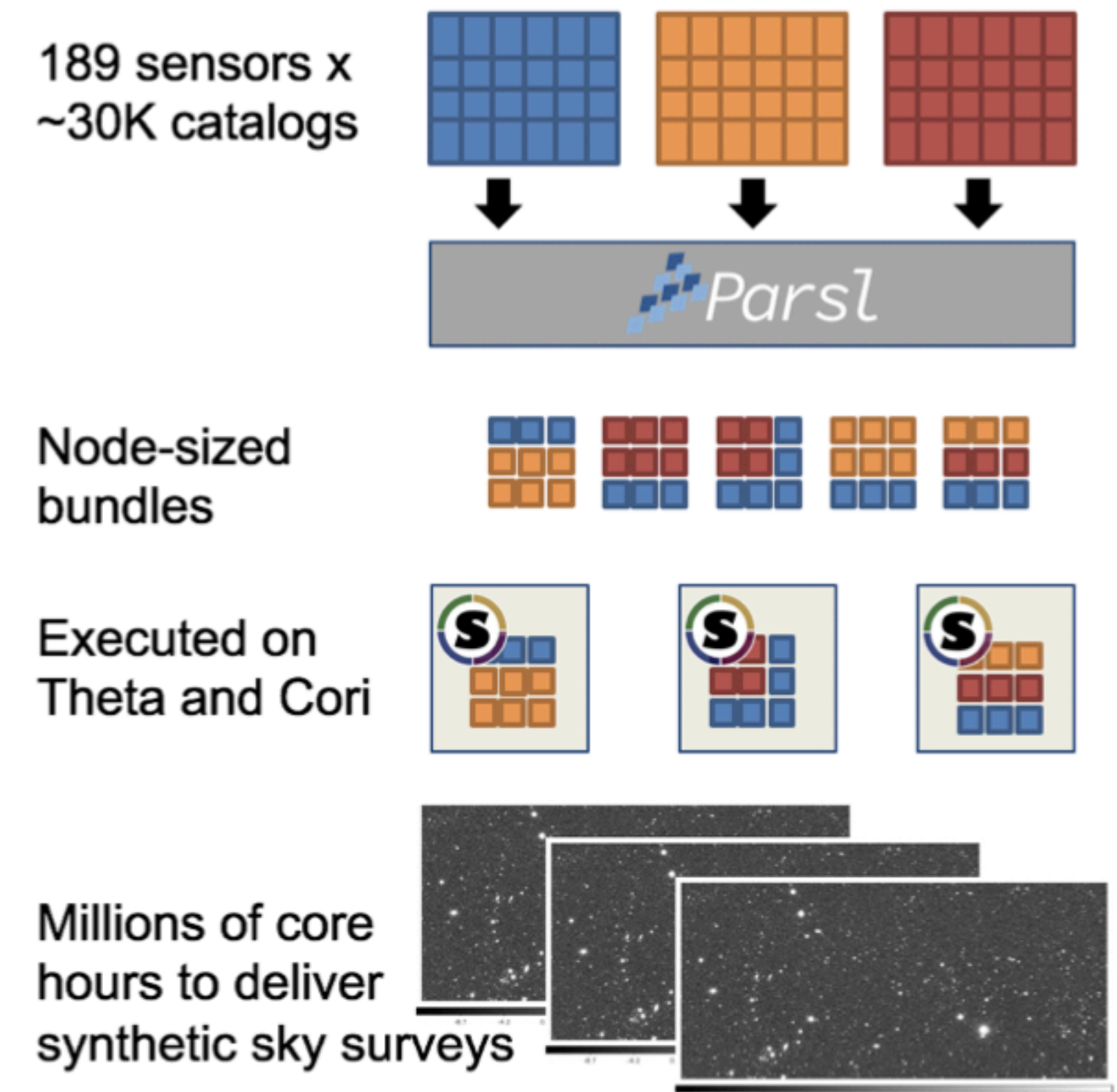
Outline

- In this talk we will address how to deploy large computational campaigns on ALCF machines using workflow tools
- The topics we will address:
 - What is a workflow?
 - How to approach workflows of varying sizes and what are workflow tools?
 - How to use the workflow tool Parsl on ALCF machines
 - How to use the workflow tool Balsam on ALCF machines
 - How to get more information and help

What does a large job campaign entail?

or “What is a workflow?”

- Large job campaigns, where a user runs a large number of jobs, are often discussed in the contexts of workflows
- A workflow is simply any collection of computational tasks run to achieve a result
- Workflows can vary in size, most users have some type of workflow whether or not they call it that
- We will discuss workflows in the context of HPC machines here at ALCF, but many of these considerations and tools apply to other HPC machines as well

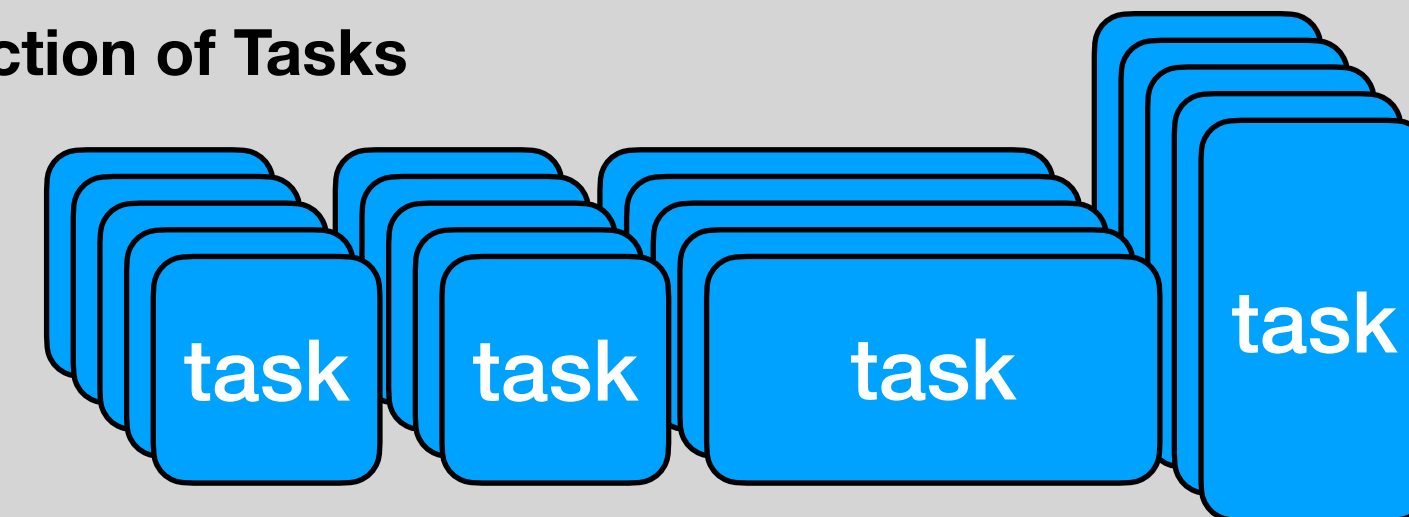


Villarreal et al. “Extreme Scale Survey Simulation with Python Workflows.”
Proceeding for eScience 2021

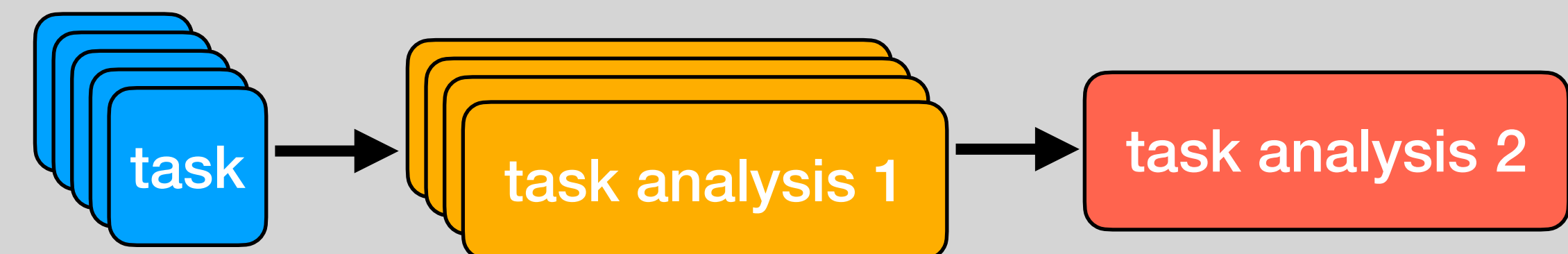
Common workflow patterns

- A large collection of tasks, perhaps with the same or different resource needs
- A collection of **dependent tasks** coupled to one another
- A collection of **tasks that generate new tasks** when they execute, perhaps from a ML or AI model
- **Tasks that need data** staged in from a remote location or staged out after the tasks complete

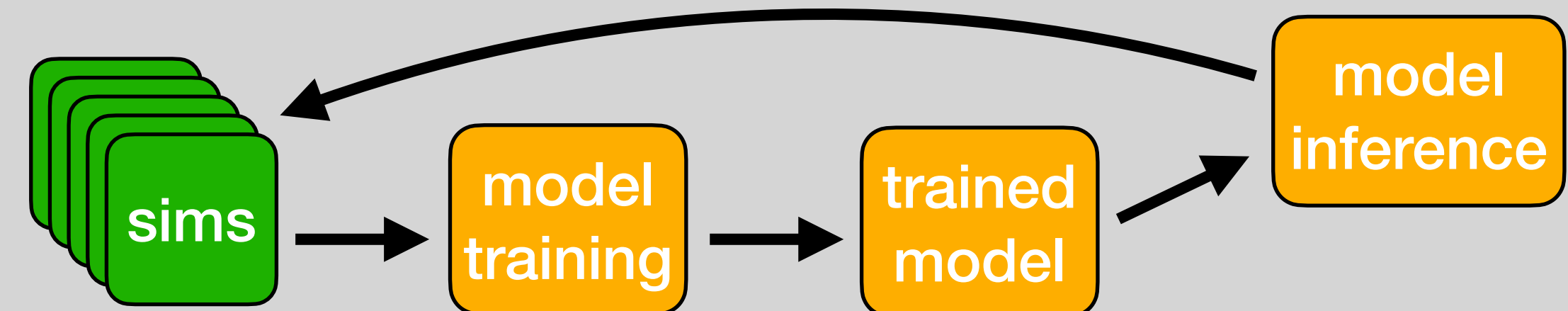
Large Collection of Tasks



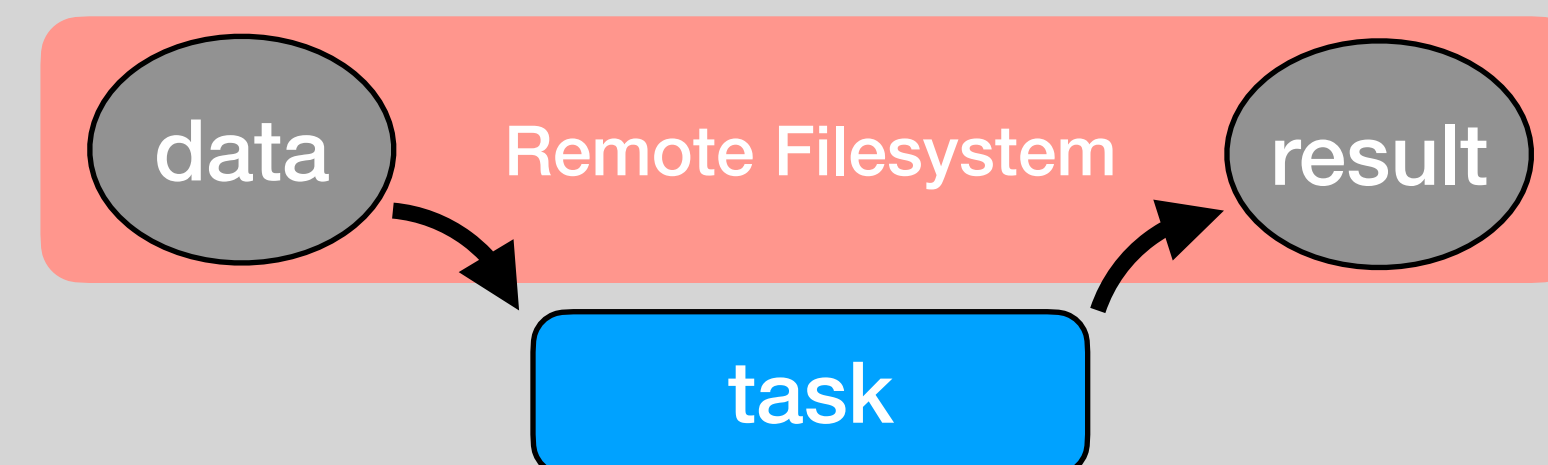
Complex Task Dependencies



Tasks Generating New Tasks Dynamically



Tasks Folded into Data Flow



A Large Collection of Tasks

Where many people start...

- 1 - 20 tasks
 - Each task can be run as its own job that a user can setup by hand, i.e. a user creates a directory, writes a batch script, and submits a job to the scheduler for each task
- 20 - 100 tasks
 - Human scaling starts to become uncomfortable
 - A few bash scripts that automate run setup and submission may be a good solution.
- > 100 tasks
 - Queuing policies start to become an issue, and running each task as its own job becomes difficult. For example, Polaris only allows for 100 queued jobs at one time.
 - Running ensemble batch jobs that launch multiple mpiexec/aprun calls in one script could be a solution, but it's not efficient if there's a large variance in run times
<https://docs.alcf.anl.gov/theta/queueing-and-running-jobs/example-job-scripts/#running-many-jobs-at-the-same-time>

A Large Collection of Tasks

Where many people start...

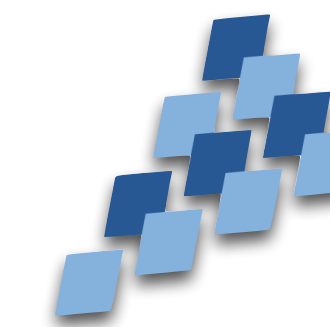
- > 100 tasks
 - Queuing policies start to become an issue, and running each task as its own job becomes difficult. For example, Polaris only allows for 100 queued jobs at one time.
 - Running ensemble batch jobs that launch multiple mpiexec/aprun calls in one script could be a solution, but it's not efficient if there's a large variance in run times.
- > 1000 tasks, at this scale several considerations make a workflow tool a must
 - Packing tasks efficiently into a batch job to minimize idle node time is important
 - How can a user judge the outcome of runs? Looking at >1000 files is difficult
 - What happens if some number of your tasks fail? How do you restart them?
- For context, on ALCF machines:
 - Theta: 280K concurrent processes possible, 1 per CPU.
 - Polaris: 1984 concurrent processes, 1 per GPU
 - Aurora: 60K concurrent processes, 1 per GPU

A workflow tool is the solution

What is a workflow tool?

- A workflow tool is a piece of software that orchestrates the execution of large numbers of tasks on compute resources, handling dependencies, data flows, and errors/timeouts
- What a workflow tool can provide for:
 - Running many tasks concurrently and one after another in one batch job
 - Task dependencies
 - Automated error handling and restarts
 - Data movement into/out of the file system
- There are many tools! A few that are used at ALCF are Balsam, Parsl and Fireworks
- There are also hyperparameter search and ML tools that couple with workflow tools such as DeepHyper, libEnsemble & Colmena, which can be useful for some workflows, but we won't cover those tools today

Parsl



A parallel programming library for Python

- Simple installation with pip
- Apps define how to run tasks
 - Python apps call Python functions
 - Bash apps call external applications
- Apps return futures: a proxy for a result that might not yet be available
- Apps run concurrently respecting dependencies
- Community of 70+ developers, including many at ALCF, UIUC & UChicago including Kyle Chard and Yadu Babuji

```
To install:  
  
$ module load conda  
$ conda create -n parsl  
$ conda activate parsl  
$ pip install parsl
```

```
@python_app  
def hello ():  
    return 'Hello World!'  
  
print(hello().result())
```



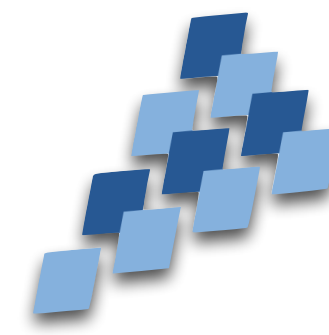
Hello World!

```
@bash_app  
def echo_hello(stdout='echo-hello.stdout'):  
    return 'echo "Hello World!"'  
  
echo_hello().result()  
  
with open('echo-hello.stdout', 'r') as f:  
    print(f.read())
```



Hello World!

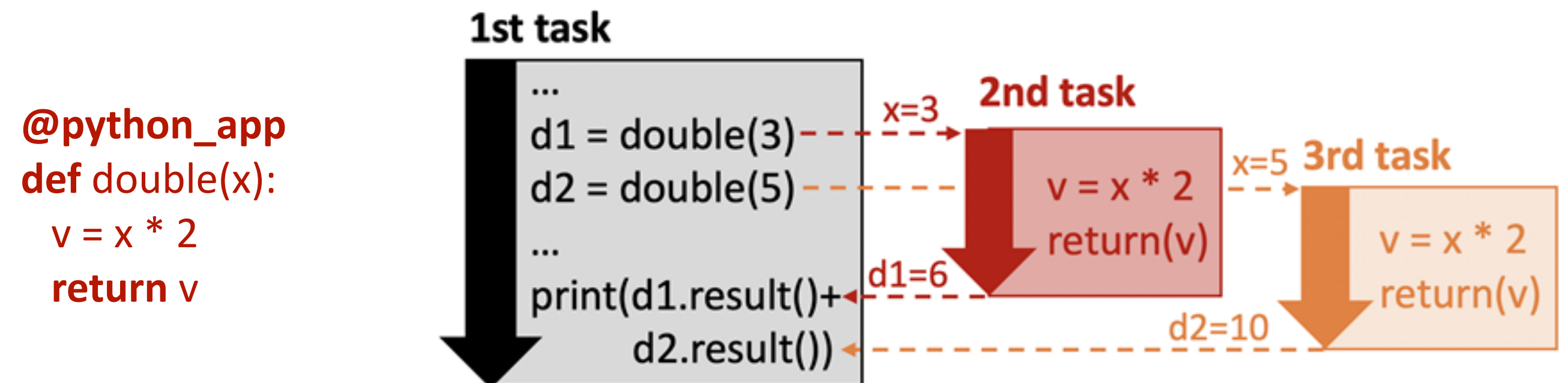
Parsl Apps and Futures



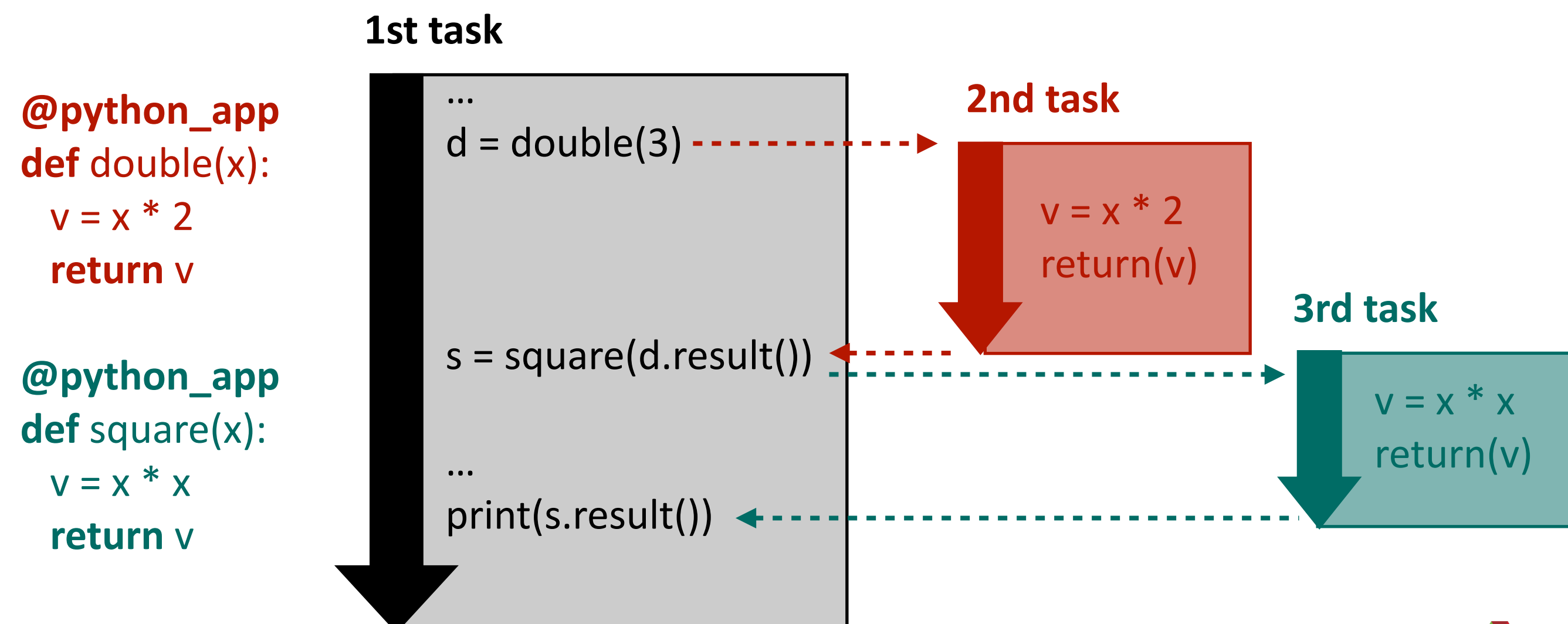
How tasks are made and linked

Concurrent Tasks

- Parsl extends the Python `concurrent.futures` module
- Tasks are created by invoking apps that return an AppFuture
- Task dependencies can be created by passing the AppFuture from one task to another



Dependent Tasks



A complete Parsl example - hello_cuda.py

```
# hello_cuda.py
import parsl
from parsl import bash_app
from config import polaris_config

# Load config for polaris
parsl.load(polaris_config)

# Application that says hello to each GPU
@bash_app
def hello_device(stdout='hello.stdout', stderr='hello.stderr'):
    return 'echo "Hello Polaris CUDA device "$CUDA_VISIBLE_DEVICES'

# Create futures calling 'hello_device', store them in list 'tasks'
tasks = []
for i in range(4):
    tasks.append(hello_device(stdout=f"hello_{i}.stdout"))

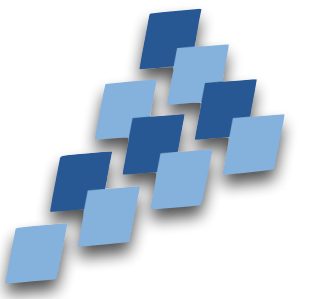
# Wait on futures to return, when they do, print results
for i,t in enumerate(tasks):
    if t.result() == 0:
        with open(f"hello_{i}.stdout", "r") as f:
            print(f.read())

# Workflow complete!
print("Hello tasks completed")
```

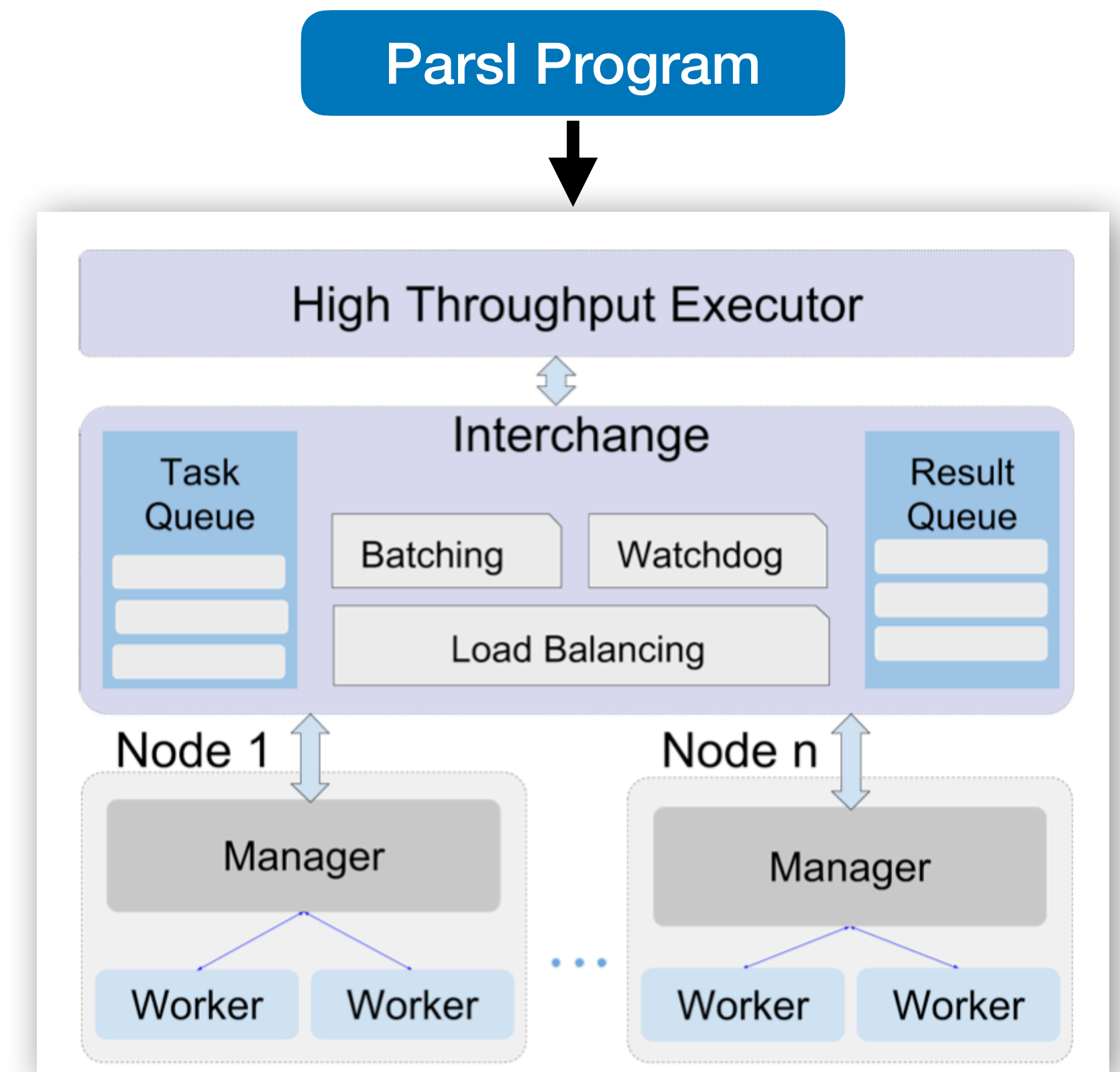
- First load a **Config** object from a file `config.py`. The `Config` tells Parsl how to schedule jobs and distribute tasks over compute nodes
- **The app** echos the value of `CUDA_VISIBLE_DEVICES`. Each task should report a different value if pinned to different GPUs
- **Create 4 tasks** by calling the app 4 times, creating 4 futures saved in the list `tasks`.
- Finally, this script will wait for each task future by calling `t.result()`. Bash apps return 0 for success.
- The wait time will depend on the queuing time and execution time of the tasks. You may wish to put the script in the background or use `screen` if you are running a longer example.
- Executing this script from a Polaris login node starts a process that creates task futures, writes a Polaris batch script, submits it, & waits for results. If futures are outstanding when the job completes, Parsl will create more jobs.

```
> python hello_cuda.py
Hello Polaris CUDA device 0
Hello Polaris CUDA device 1
Hello Polaris CUDA device 2
Hello Polaris CUDA device 3
Hello tasks completed
```

Parsl Config



- How does Parsl know how to execute its tasks on Polaris compute resources?
- The Config object specifies details of the provider, executors, connection channel, allocation size, queues, durations, and data management options.
- Different machines need different Configs. The Parsl documentation has example Config objects for all ALCF machines: <https://parsl.readthedocs.io/en/stable/userguide/configuring.html>



Babuji et.al. "Parsl: Pervasive Parallel Programming in Python." ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). 2019.

Parsl Config for Polaris

- With this Config, Parsl will submit 1 node batch jobs that run for 10 minutes each to the debug queue
- It will only submit one batch job at a time, but increase max_blocks to submit more jobs at once
- The HighThroughputExecutor with the MPIExecLauncher are common Config settings for Polaris. Always use the PBSProProvider for Polaris.
- The MPIExecLauncher uses mpiexec to place one manager on each MPI Rank and one MPI Rank per node. It does not refer to how the Apps are executed.

```
# config.py
from parsl.config import Config
from parsl.providers import PBSProProvider
from parsl.executors import HighThroughputExecutor
from parsl.launchers import MpiExecLauncher
from parsl.addresses import address_by_hostname
```

```
polaris_config = Config(
    executors=[
        HighThroughputExecutor(
            available_accelerators=4,
            address=address_by_hostname(),
            cpu_affinity="alternating",
            prefetch_capacity=0,
            start_method="spawn",
            provider=PBSProProvider(
                account="datascience",
                queue="debug",
                worker_init="source /path/to/env; cd /path/to/rundir",
                walltime="0:10:00",
                scheduler_options="#PBS -l filesystems=home:eagle",
                launcher=MpiExecLauncher(
                    bind_cmd="--cpu-bind",
                    overrides="--depth=64 --ppn 1"
                ),
                select_options="ngpus=4",
                nodes_per_block=1,
                min_blocks=0,
                max_blocks=1,
                cpus_per_node=64,
            ),
        ],
    )
```

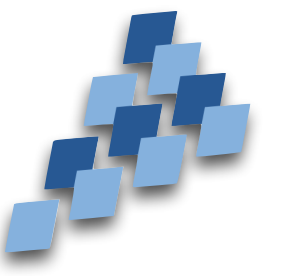
sets 4 workers per node, one per GPU

Activate environment, load modules, set env variables for app here

sets 1 manager per node

sets how many nodes per block aka per job

sets how many blocks/jobs will run concurrently

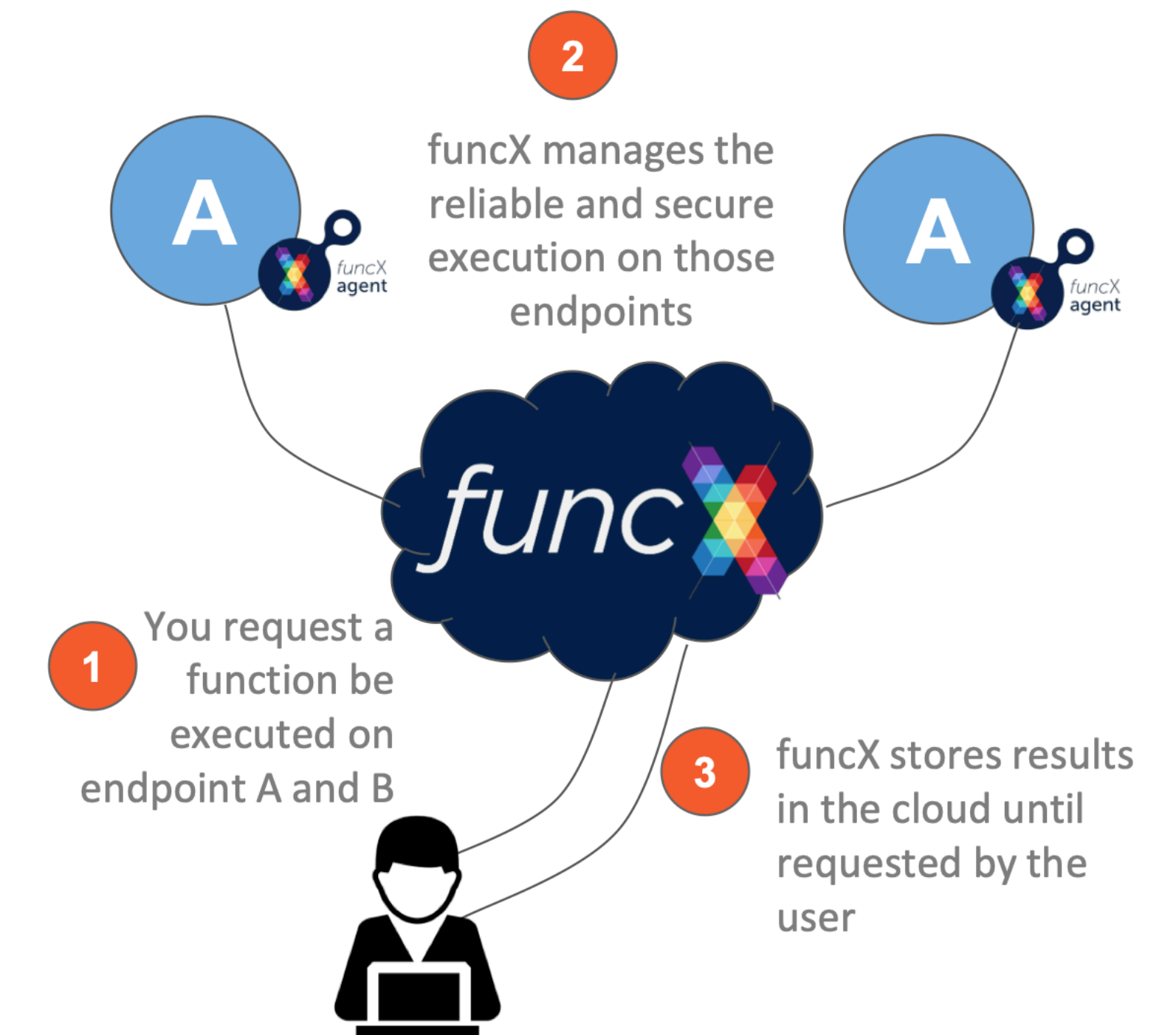


Parsl Integration with other Tools

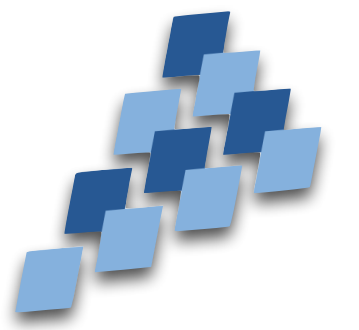


funcX/Globus Compute

- Parsl is part of a wider ecosystem of tools through Globus Labs, including funcX, now called Globus Compute
 - Globus Compute allows for the remote execution of tasks, on a fire-and-forget model.
 - The user sets up an endpoint on an ALCF machine that connects to the cloud. The user can send tasks to the endpoint from anywhere, which are then run on the machine.
 - Globus Compute uses Parsl to execute tasks on the target machine, and therefore your Parsl config can be easily adapted to be used by Globus Compute



Why Choose Parsl?



- Parsl is an excellent choice for High Throughput workflows, e.g. workflows running many tasks per second on single nodes or single CPUs/GPUs
- Parsl can manage complex and dynamic task dependencies
- Parsl is lightweight, it does not depend on a database, for example
- Parsl is portable and platform agnostic (the Config is the only thing that needs to be changed)
- Parsl is native to Python, a great choice for Python applications
- Parsl is part of the Globus community and has integration with many of their other tools
- Parsl has extensive documentation and an active Slack community supporting users

<https://parsl.readthedocs.io/en/stable/>

Balsam Workflow Management Tool



A unified platform to manage high-throughput workflows across the HPC landscape

- Balsam was developed at ALCF and is used for deploying workflows on DOE HPC machines
- Balsam uses a database model, applications and tasks are stored in a centralized database on a server that tracks the progress of tasks, called jobs
- Can execute external apps and native Python apps
- Has a Python API and command line interface
- Centralized server allows for inter-machine workflows
- Requires login to the Balsam server at ALCF where the database is hosted, and therefore an ALCF account
- Supported configurations for all ALCF machines, and machines at NERSC & OLCF

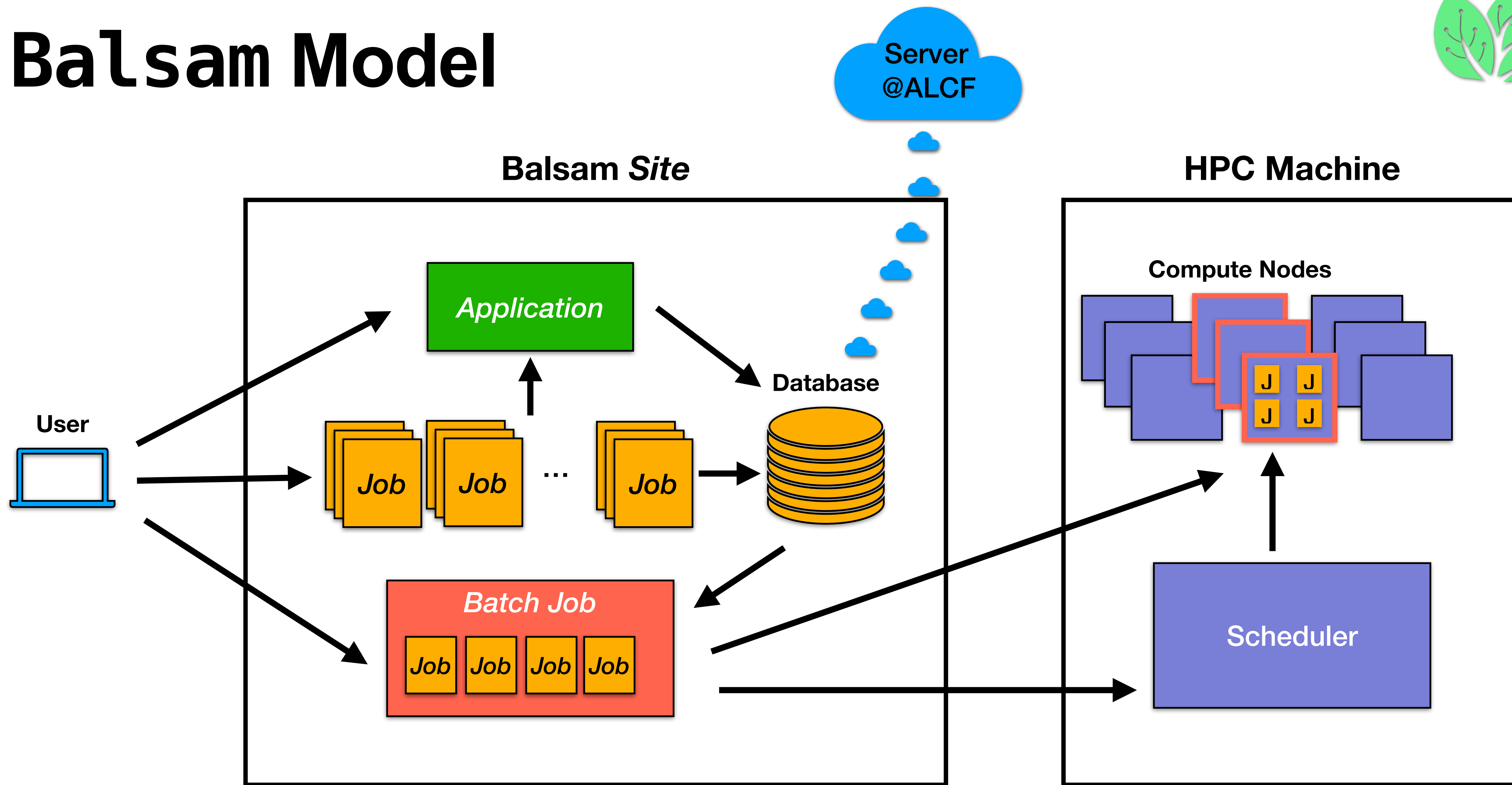
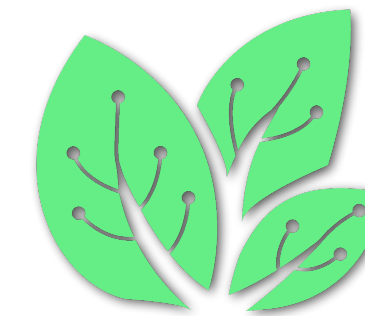
To install:

```
$ module load conda
$ conda create -n balsam
$ conda activate balsam
$ pip install --pre balsam
```

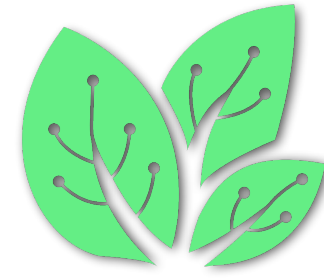
To login to the server:

```
$ balsam login
```

Balsam Model



The Balsam Site



- The first step to creating a workflow in Balsam is to make a Balsam Site
- A Site is composed of two elements
 - A workspace directory on the machine file system containing data, logs & settings
 - A background process running on the machine that communicates with the machine scheduler and the database on the server
- Sites are 'active' when the background process is running and communicating with the server. The Site needs to be active for the workflow to work
- Sites are configured for the machine where they are located and setup automatically for the user

From a Polaris login node, first login:

```
$ balsam login
```

Next, navigate to where you want your Site on the file system and create the site:

```
$ balsam site init -n polaris_tutorial polaris_tutorial
```

```
[?] Select system: Polaris (ALCF)
```

```
Perlmutter-GPU (NERSC)
```

```
Cori-KNL (NERSC)
```

```
> Polaris (ALCF)
```

```
MacOS/Linux (Local)
```

```
Perlmutter-CPU (NERSC)
```

```
Cori-Haswell (NERSC)
```

```
Sunspot (ALCF)
```

```
Summit (OLCF)
```

```
Cooley (ALCF)
```

```
Theta-KNL (ALCF)
```

```
Theta-GPU (ALCF)
```

```
$ cd polaris_tutorial
```

```
$ balsam site start
```

```
Started Balsam site daemon [pid 12383] on polaris-login-02
```

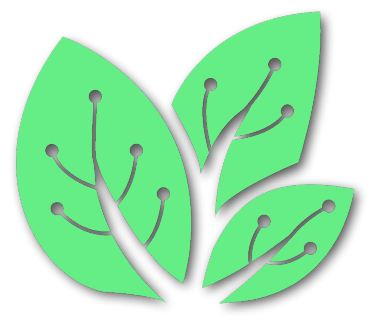
Query the status of your Site:

```
$ balsam site ls
```

ID	Name	Path	Active
514	polaris_tutorial	.../polaris_tutorial	Yes

A Simple Balsam Example

Running a compiled executable, e.g. LAMMPS (lmp)



```
from balsam.api import ApplicationDefinition, Job, BatchJob

class Lammps(ApplicationDefinition):

    site = "polaris_tutorial"

    def shell_preamble(self):
        return f'source /path/to/envs.sh'

    command_template = 'lmp -in /path/to/input.in -var tinit {{tinit}}'

Lammps.sync()

initial_temps = [0.7, 1.0, 1.5]
jobs = [Lammps.submit(workdir=f"LJ/{n}",
                    parameters={"tinit": tinit})
        for n, tinit in enumerate(initial_temps)]

jobs = Job.objects.bulk_create(jobs)

site = Site.objects.get("polaris_tutorial")
BatchJob.objects.create(
    site_id=site.id,
    num_nodes=2,
    wall_time_min=10,
    job_mode="mpi",
    project="datascience",
    queue="debug",)
```

Application

- Includes executable in `command_template`
- Can pass input parameters with `{{ }}`
- Can set environment variables, load modules in `shell_preamble`



Jobs

- Each job runs an application
- Required to set a `workdir` for each job
- Can vary inputs, resources for each job, set tags

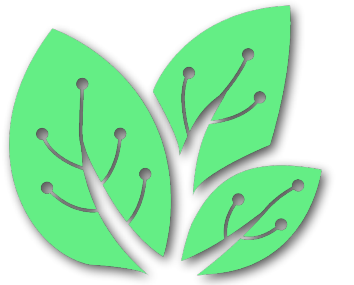


Batch Job

- Creates Job on the scheduler that runs the Balsam 'jobs' saved in the database
- 'mpi' mode means the executable in the application will be passed to `mpiexec`



Asynchronous Node Packing in Balsam

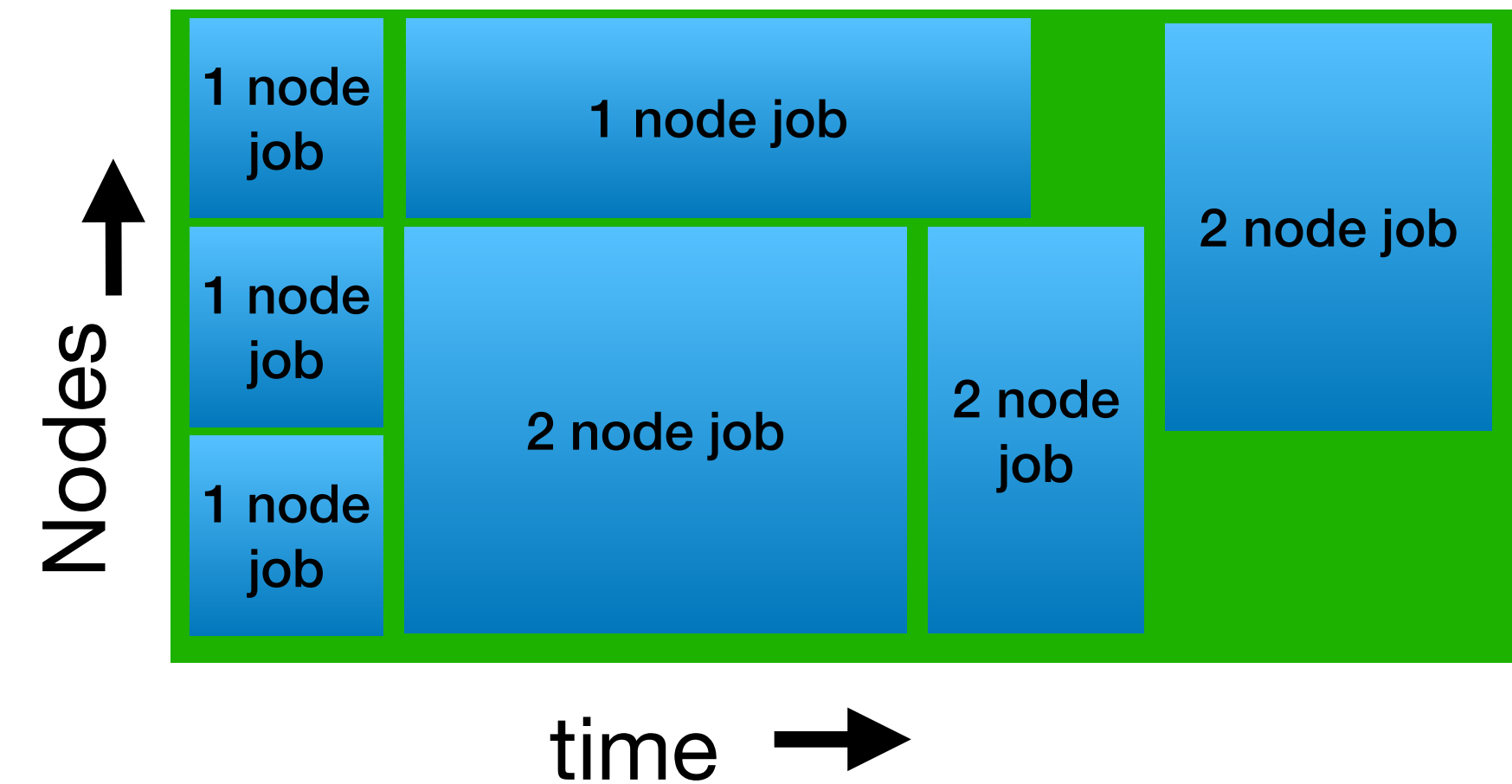


Many tasks, many different resource needs

- The user can define different resource needs for each job, even jobs calling the same Application
- The Balsam model of job packing will dynamically fill available nodes within a batch job to achieve maximal utilization of nodes
- An example of how to specify resources for a Balsam job:

```
Job(app_id="Lammps",
    site_name="polaris_tutorial",
    workdir="demo",
    parameters={"tinit":1.5},
    wall_time_min=5, # an estimate for the run time
    num_nodes=2,
    ranks_per_node=4,
    gpus_per_rank=1,
    threads_per_rank=8, #This sets OMP_NUM_THREADS
    launch_params={"cpu_bind":"depth"},
    tags={"parameter_test":"temp"},
)
```


3 Node Batch Job running 7 Balsam jobs requiring different run times and node numbers



<https://argonne-lcf.github.io/balsam/user-guide/jobs/#defining-compute-resources>



Why Choose Balsam?

- Balsam is an excellent choice to run large collections of multi-node MPI jobs
- Balsam can manage complex job dependencies between jobs on the same or *different* machines
- Balsam uses a database which makes tracking job provenance and performance straightforward
- Balsam's database is hosted in the cloud which allows for workflows that span machines
- Balsam has support for data transfers via Globus Online 
- Balsam has both static and elastic queuing modes that allows for different levels of control in submitting Batch Jobs to the scheduler
- Balsam is preconfigured for all ALCF machines, little machine specific setup is required
- Balsam has documentation and an active Slack where users can get help

<https://argonne-lcf.github.io/balsam/>

Where to get more information

- **Parsl**

- docs: <https://parsl.readthedocs.io/en/stable/>
- github: <https://github.com/Parsl/parsl>
- slack: <https://parsl-project.org/support.html>
- Globus Compute (formally funcX): <https://funcx.org/>

- **Balsam**

- docs: <https://argonne-lcf.github.io/balsam/>
- github: <https://github.com/argonne-lcf/balsam>
- slack: https://join.slack.com/t/balsam-workflows/shared_invite/zt-1t0736hsz-6hxsmC~0MBFpuP~WvouwWQ
- Recent workflows workshop materials (includes materials on how to run GNU Parallel, Parsl, Balsam & Fireworks on Polaris): <https://github.com/CrossFacilityWorkflows/DOE-HPC-workflow-training>
- Workflows community (group where you can discover new workflow tools & connect with workflows community) : <https://workflows.community/>

Summary

How to deploy large job campaigns

- Projects often need to deploy large numbers of jobs with complex dependencies on ALCF machines
- ALCF supports multiple tools to facilitate this work, we discussed two, Parsl and Balsam, and how to use them on Polaris
- These tools are well suited to handle high throughput workflows that run many tasks per second and workflows that run large numbers of multi-node MPI jobs
- Other tools can be used at the facility such as Fireworks and GNU Parallel
- Many of these tools can be coupled to hyperparameter space search tools like DeepHyper, Colmena or libEnsemble
- Currently, Theta and Polaris can support large scale workflows running thousands to 10s of thousands of concurrent processes. Aurora will enable even larger workflows, supporting over 60K concurrent processes on GPUs. Workflow tools such as Parsl and Balsam will allow users to make use of these capabilities.

