ALCF INCITE GPU Hackathon May 20-22, 2025

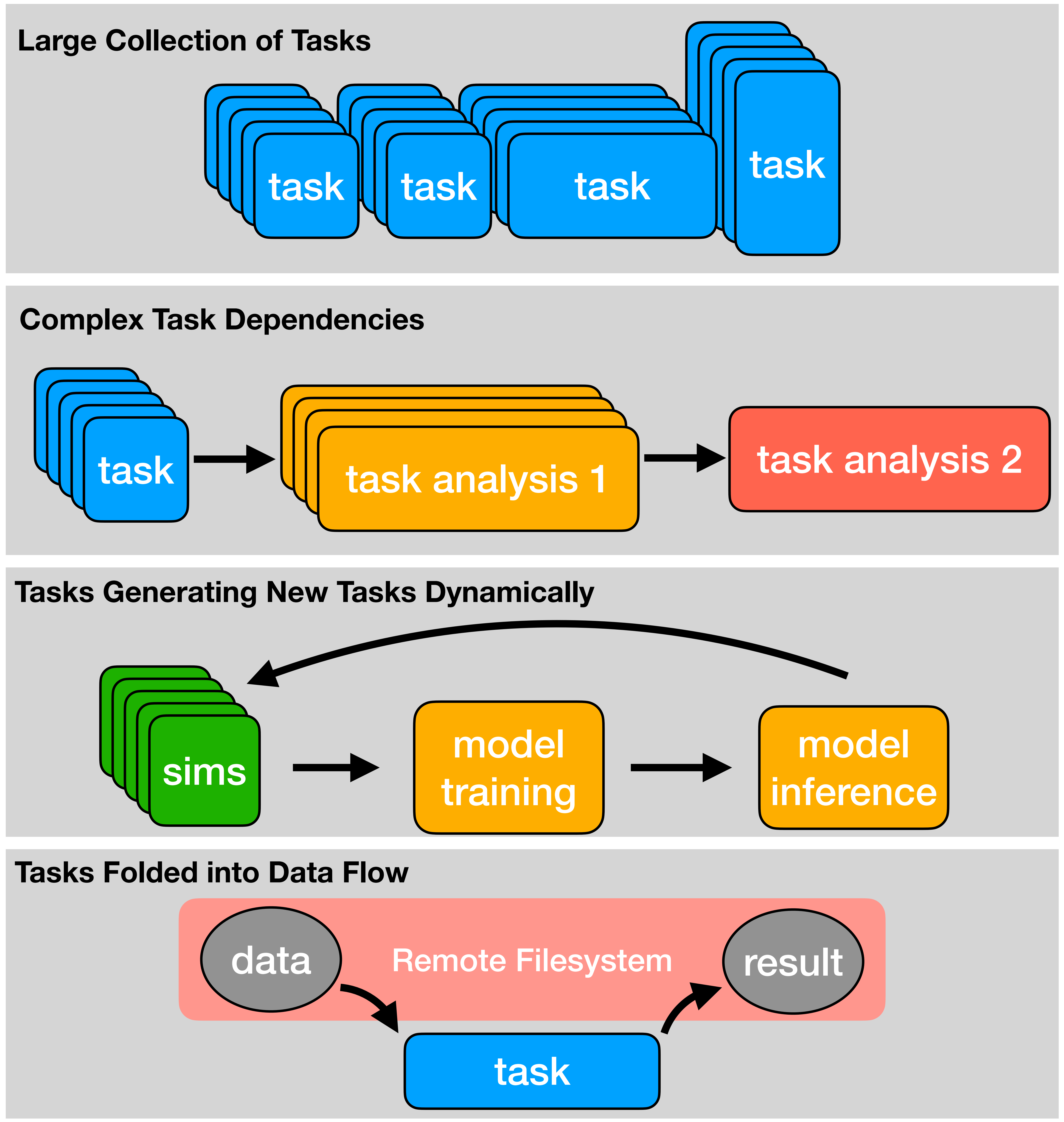
Ensembles and Workflows

Christine Simpson
Data Services & Workflows, ALCF

May 6, 2025

Ensembles & Workflows

- A workflow is simply any collection of computational tasks run to achieve a result
- Workflows can have complex dependencies, involve many different applications and/or data movement
- Workflows often include ensembles, which we will describe as a collection of tasks running the same application with similar inputs
- On ALCF machines, queueing policies usually require workflows and ensembles to be packed into large jobs


Large Collection of Tasks
task
task
task
task
Complex Task Dependencies
task
task analysis 1
task analysis 2
Tasks Generating New Tasks Dynamically
sims
model training
model inference
Tasks Folded into Data Flow
data
Remote Filesystem
result
task

Simple Case: Ensemble PBS job

Run many tasks in one job

- Useful for tasks with similar run times that have no subsequent dependencies
- May be sufficient for a small number of tasks with similar inputs
- Applications launched with `mpiexec` that assigns unique nodes to each task
- Ensemble jobs of multi-node tasks limited to 1000 tasks on Aurora
- No limit for tasks that use a single node (or are sub-node)

```bash
#!/bin/bash -l
#PBS -l select=8
#PBS -l walltime=0:30:00
#PBS -q debug-scaling
#PBS -A datascience
#PBS -l filesystems=home:flare

cd ${PBS_O_WORKDIR}

# MPI example w/ multiple runs per batch job
NNODES=`wc -l < $PBS_NODEFILE`

# Settings for each run: 2 nodes, 12 MPI ranks per node spread evenly
across cores, total of 4 tasks
NUM_NODES_PER_MPI=2
NRANKS_PER_NODE=12
NDEPTH=8
NTHREADS=1
NTOTRANKS=$(( NUM_NODES_PER_MPI * NRANKS_PER_NODE ))


# Increase value of suffix-length if more than 99 jobs
split --lines=${NUM_NODES_PER_MPI} --numeric-suffixes=1 \
      --suffix-length=2 $PBS_NODEFILE local_hostfile

for lh in local_hostfile*
do
  echo "Launching mpiexec w/ ${lh}"
  mpiexec -n ${NTOTRANKS} -ppn ${NRANKS_PER_NODE} \
          --hostfile ${lh} -depth=${NDEPTH} --cpu-bind depth \
          ./hello_affinity &
done
wait
```

Pinning applications to nodes with `mpiexec`

- `mpiexec` can be used to pin application to specific nodes with the `--hosts` or `--hostlist` options
- Node (host) names can be pulled from the `PBS_NODEFILE` during the job runtime

```
mpiexec -n 24 -ppn 12 --hostfile ./local_hostfile ./hello_affinity &
```

```
mpiexec -n 24 -ppn 12 --hosts $HOST_NAME1 $HOST_NAME2  ./hello_affinity &
```

- Use an `&` at the end of the line to launch the process in the background
- Always include `wait` at the end of the script to ensure it does not return before background processes complete

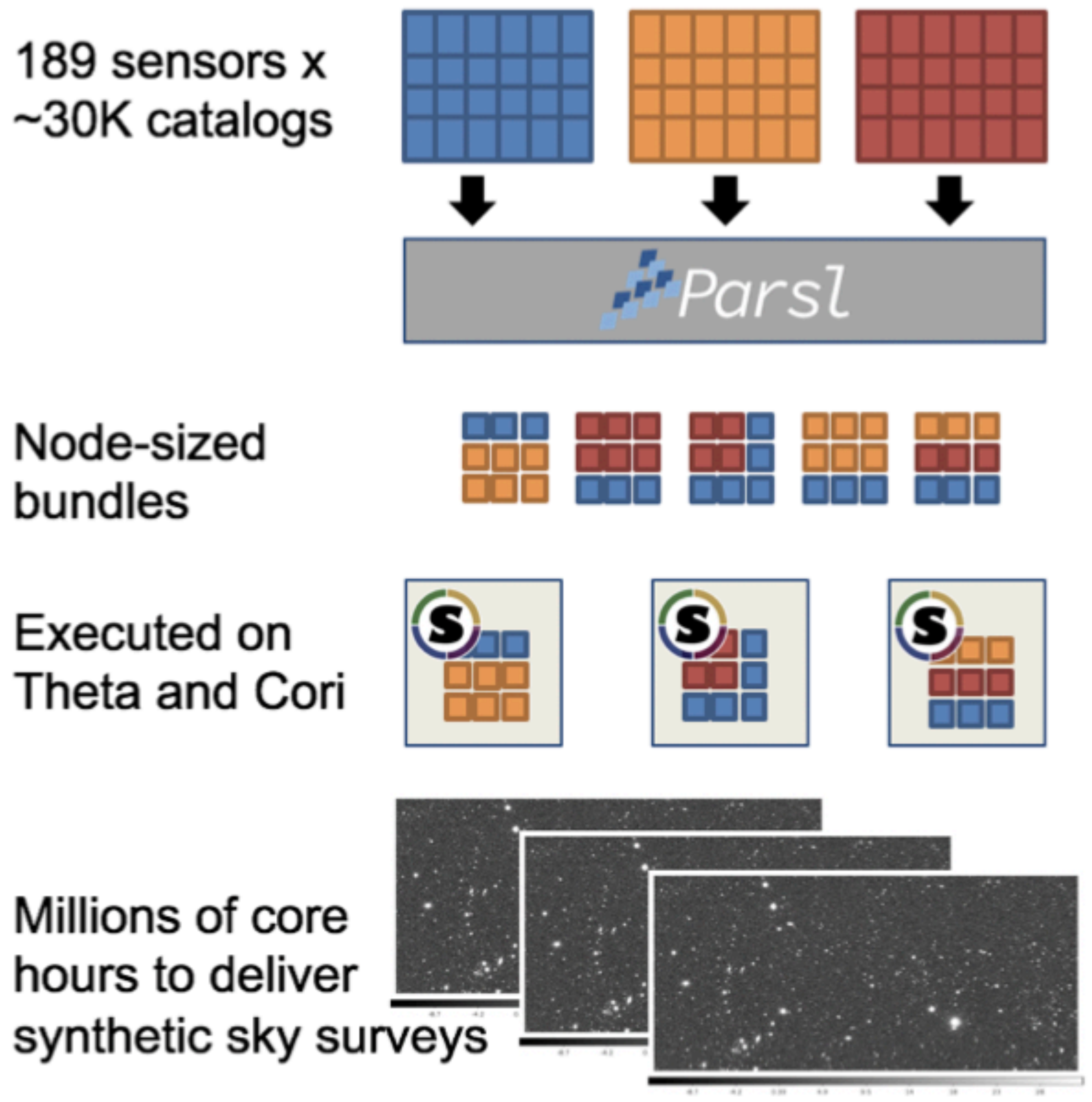
Limitations of Ensemble PBS jobs

- If task run times are varied, PBS ensemble jobs give poor node utilization, e.g. nodes are not refilled when tasks complete
- Large task ensembles may encounter bottlenecks on the launching node (e.g. run out of file descriptors) or encounter slow launching times
- Complex dependencies can be hard to manage in a shell script, no checkpointing functionality, automatic task logs, retries, etc.
- When these issues arise, using a workflow tool is recommended

What is a workflow tool?

Why should you consider using one?

- 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 do for your workload:
  - Run many tasks concurrently and/or one after another asynchronously across one or many batch jobs
  - Manage task dependencies
  - Automate error handling and restarts of tasks
  - Manage data movement into/out of the file system needed for tasks
- Many tools can be run on our systems, we will show some details of how to run Parsl, developed at UChicago and Globus Labs


189 sensors x ~30K catalogs
Parsl
Node-sized bundles
Executed on Theta and Cori
Millions of core hours to deliver synthetic sky surveys

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

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
- Workflow contained within memory
- Configuration (assignment of tasks to hardware) set by user, separate from workflow logic and application definitions
- Apps return futures: a proxy for a result that might not yet be available
- Apps run concurrently, respecting dependencies
- Community of 70+ developers, several at UChicago & ANL, part of Globus Labs

```python
@python_app
def hello ():
    return 'Hello World!'

print(hello().result())
```

```
Hello World!
```

```python
@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!
```

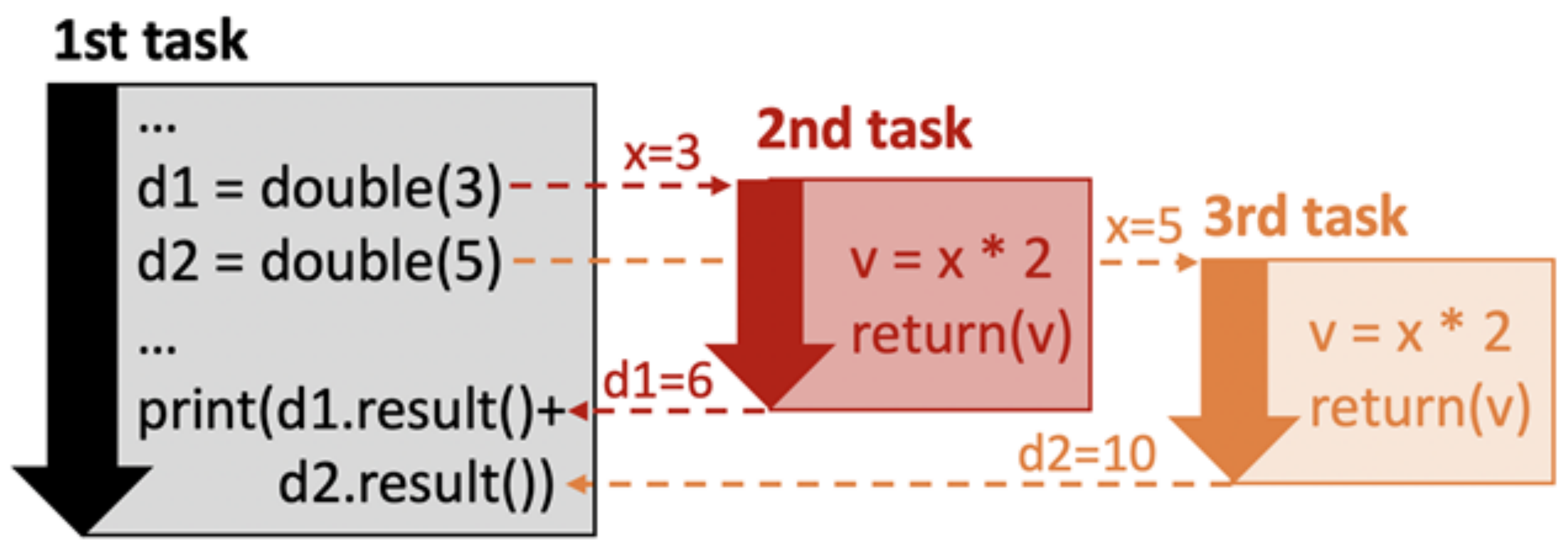
`Parsl` Apps and Futures

How tasks are made and linked

- Parsl extends the Python `concurrent.futures` module
- Tasks are created by invoking apps that return an AppFuture
- A “future” is a proxy for a result that may or may not yet exist
- Task dependencies can be built by passing the AppFuture from one task to another

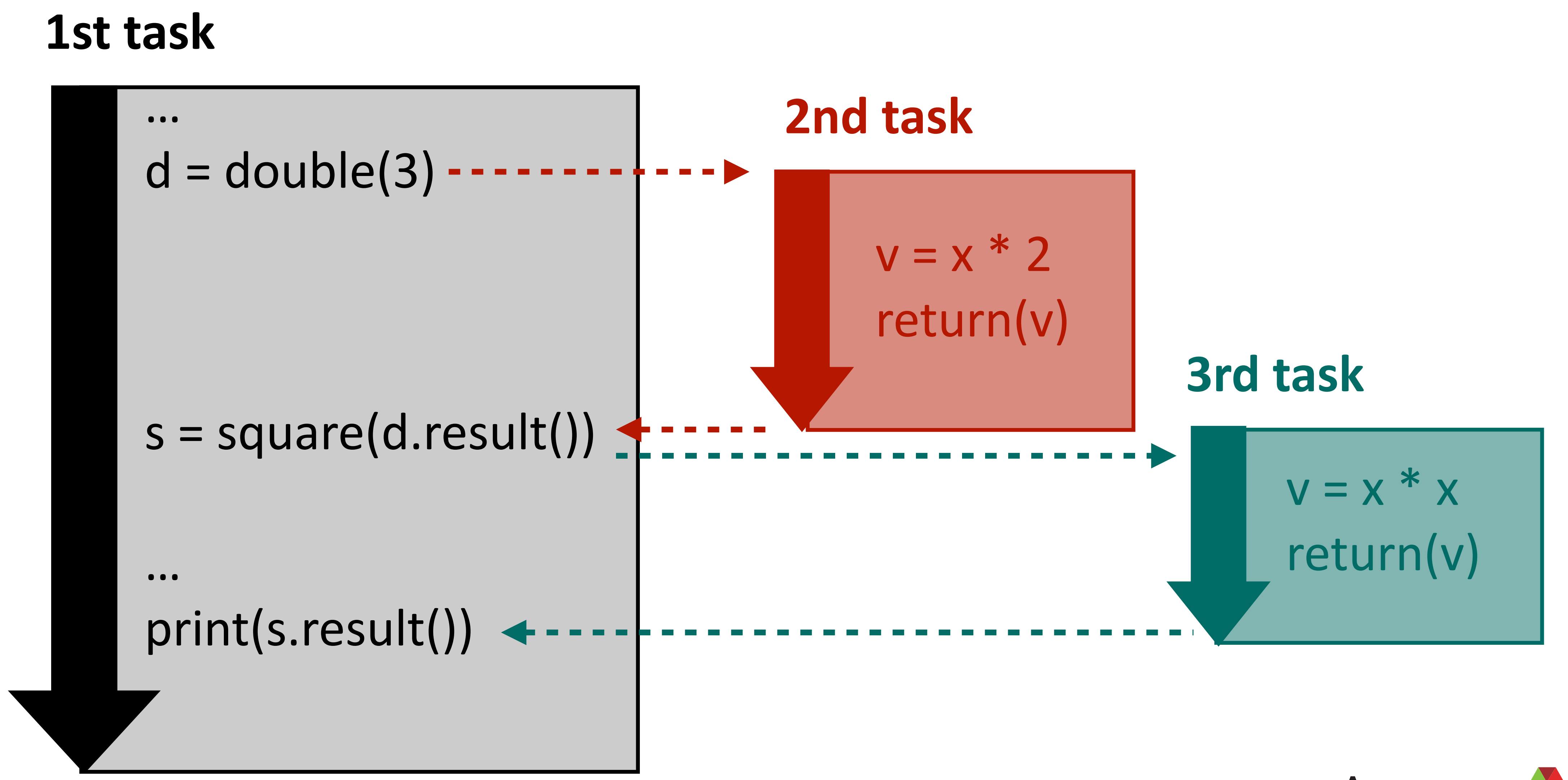
Concurrent Tasks

```
@python_app
def double(x):
  v = x * 2
  return v
```



Dependent Tasks

```
@python_app
def double(x):
  v = x * 2
  return v

@python_app
def square(x):
  v = x * x
  return v
```



Parsl Config

- Parsl assigns tasks to workers for execution.
- Resources (nodes, GPUS, CPUS) assigned to workers are configured by the user.
- Two possible Config types (see Aurora docs for more):
  - "Head-less" Config that launches workflow from within PBS job (shown here on the right)
  - Externally launched workflow that sends jobs to PBS from the login node
- Many other Config options, see Parsl docs for details

Example Config: 1 GPU tile per worker, launch within PBS job

```
tile_names = [f'{gid}.{tid}' for gid in range(6) for tid in range(2)]
start_threads = [1,9,17,25,33,41,53,61,69,77,85,93]
threads_by_tile = [f"{st}-{st+7},{st+104}-{st+111}" for st in start_threads]
cpu_affinity = "list:"+":".join(threads_by_tile)

# Get the number of nodes:
node_file = os.getenv("PBS_NODEFILE")
with open(node_file,"r") as f:
    node_list = f.readlines()
    num_nodes = len(node_list)

aurora_single_tile_config = Config(
    executors=[
        HighThroughputExecutor(
            # Ensures one worker per GPU tile on each node
            available_accelerators=tile_names,
            max_workers_per_node=12,
            # Distributes threads to workers in a way optimized for Aurora
            cpu_affinity=cpu_affinity,
            # Increase if you have many more tasks than workers
            prefetch_capacity=0,
            # Options that specify properties of PBS Jobs
            provider=LocalProvider(
                # Number of nodes job
                nodes_per_block=num_nodes,
                launcher=MpiExecLauncher(bind_cmd="--cpu-bind",
                                         overrides="--ppn 1"),
                init_blocks=1,
                max_blocks=1,
            ),),],)
```

`Parsl` Workflow Example

- Load `Config` object in workflow
- Add app decorators to parsl apps
- Workflow script must wait on results of all tasks created to ensure they complete before the script returns
- This example shows `@python_app` but compiled executables can be executed with `@bash_app`

```python
import parsl
from parsl import python_app
from config import aurora_single_tile_config


@python_app
def double(x):
    return 2*x
@python_app
def square(x):
    return x*x

with parsl.load(aurora_single_tile_config):

    # Ensemble of double tasks
    d = []
    for i in range(10):
        d.append(double(i))

    # Dependent ensemble of square tasks
    s = []
    for i in range(10):
        s.append(square(d[i]))

    # Wait on and print results
    for i in range(10):
        print(s[i].result())
```

Other tools

- There are many other tools that can be used on our systems, please come talk to us if you need help running your preferred tool
- A common tool at the facility is Balsam which was developed at ALCF
  - Balsam uses a **database model**, applications and tasks are stored in a centralized database that tracks the progress of tasks
  - Balsam has a **Python API** and **command line interface** and is optimized for running **MPI** applications
  - Access to the Balsam database requires permission, email support@alcf.anl.gov for access
- Workflows tools that have a data management layer can be useful in some cases: SmartSim, Dragon, and ADIOS2 are all tools with this capability that run on Aurora.
- Remote workflows that involve remote data movement and triggering can be enabled by Globus Compute and Globus Flows

Conclusions

- Small ensembles may be run with PBS ensemble jobs
- Large ensembles and complex workflows should use workflow tools
- Parsl is one such tool that can execute compiled applications or native python code, handle complex dependencies, refill nodes, and manage work over multiple PBS jobs.
- Other tools are available, for example database tools like Balsam or tools that include a data management layer like Dragon or Adios2.
- There are many workflow tools out there and if you need help running your preferred tool, please let us know

Extra Slides

Use `mpiexec` to launch ensemble of single-tile tasks

- Use `gpu_tile_compact.sh` to assign each application run to a unique tile

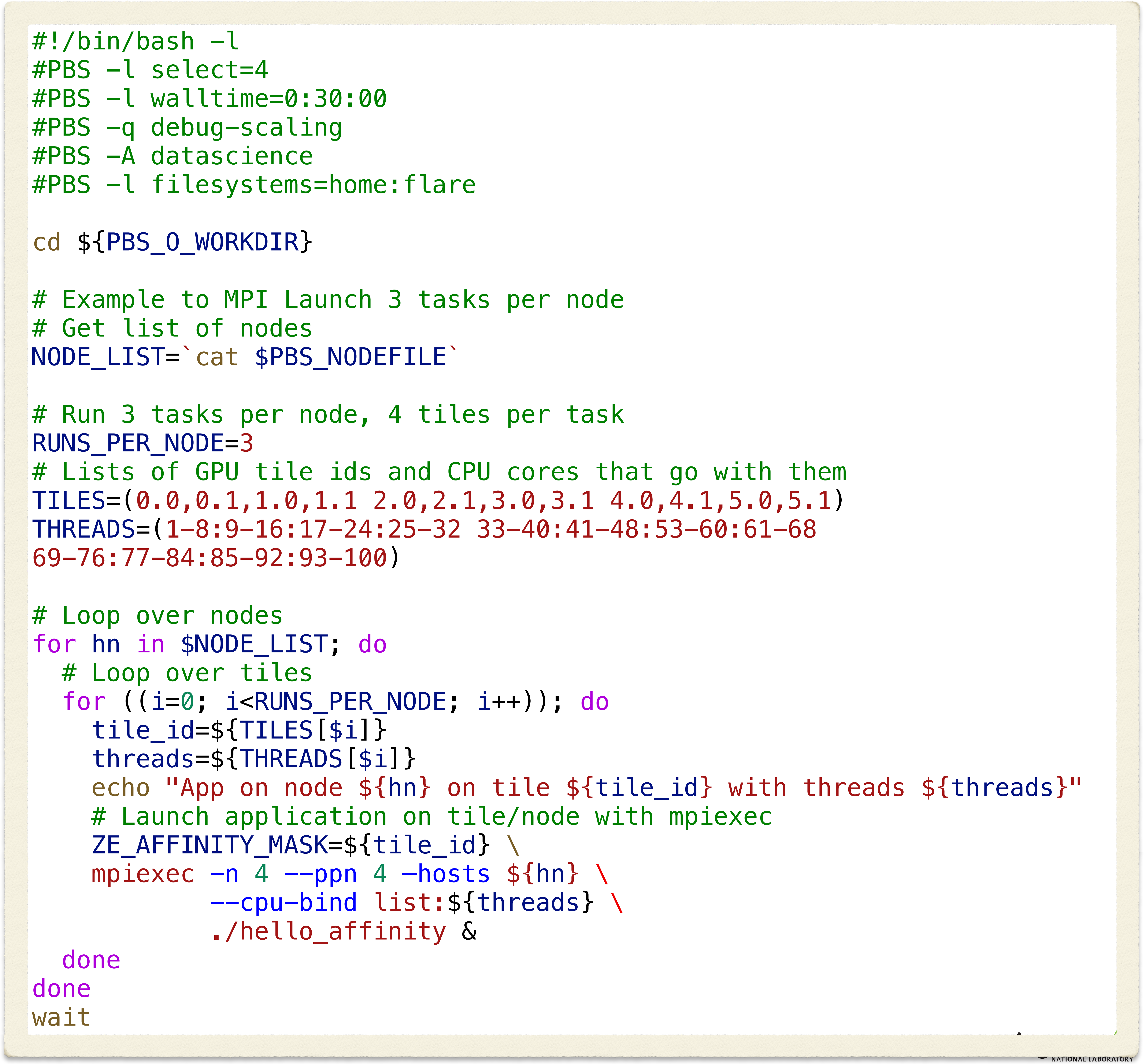
- If your application takes inputs that need to vary, a wrapper bash script similar to `gpu_tile_compact.sh` can be used to introduce logic to assign parameters

- Again, this is not an optimal approach for a large number of tasks and/or tasks with varying run times

```
#!/bin/bash -l
#PBS -l select=8
#PBS -l walltime=0:30:00
#PBS -q debug-scaling
#PBS -A datascience
#PBS -l filesystems=home:flare

cd ${PBS_O_WORKDIR}

# Example to MPI Launch 1 task per tile
# Get number of nodes
NNODES=`wc -l < $PBS_NODEFILE`

TASKS_PER_NODE=12
NTASKS=$(( NNODES * TASKS_PER_NODE ))

NDEPTH=8
NTHREADS=1

# This mpiexec call will launch NTASKS copies of the application
# Wrapping the application with gpu_tile_compact.sh will ensure
# each hello_affinity task will be pinned to one unique GPU
mpiexec -n ${NTASKS} \
        --ppn ${TASKS_PER_NODE} \
        --depth=${NDEPTH} \
        --cpu-bind depth \
        gpu_tile_compact.sh \
        ./hello_affinity
```

Run multiple MPI application tasks per node

- Similar to multi-node task ensemble, but nodes will be shared by 3 concurrent tasks
- Use ZE_AFFINITY_MASK to pin task to GPUS
- Again, this is not an optimal approach for a large number of tasks with varying run times



```bash
#!/bin/bash -l
#PBS -l select=4
#PBS -l walltime=0:30:00
#PBS -q debug-scaling
#PBS -A datascience
#PBS -l filesystems=home:flare

cd ${PBS_O_WORKDIR}

# Example to MPI Launch 3 tasks per node
# Get list of nodes
NODE_LIST=`cat $PBS_NODEFILE`

# Run 3 tasks per node, 4 tiles per task
RUNS_PER_NODE=3
# Lists of GPU tile ids and CPU cores that go with them
TILES=(0.0,0.1,1.0,1.1 2.0,2.1,3.0,3.1 4.0,4.1,5.0,5.1)
THREADS=(1-8:9-16:17-24:25-32 33-40:41-48:53-60:61-68
69-76:77-84:85-92:93-100)

# Loop over nodes
for hn in $NODE_LIST; do
  # Loop over tiles
  for ((i=0; i<RUNS_PER_NODE; i++)); do
    tile_id=${TILES[$i]}
    threads=${THREADS[$i]}
    echo "App on node ${hn} on tile ${tile_id} with threads ${threads}"
    # Launch application on tile/node with mpiexec
    ZE_AFFINITY_MASK=${tile_id} \
    mpiexec -n 4 --ppn 4 -hosts ${hn} \
            --cpu-bind list:${threads} \
            ./hello_affinity &
  done
done
wait
```