HPC WORKFLOW MANAGEMENT WITH CHAPEL

Ben Albrecht, HPE
June 4, 2021
Coordinating many monolithic applications is a challenge for the HPC user community:

- This impacts scientists across many domains such as astronomy, computational chemistry, and bioinformatics.

These workflows often begin as a simple shell script or collection of scripts:
- Depending on the complexity of the workflow, this may become unwieldy.
- Users can adopt a workflow framework or code their workflow in a more advanced programming language.

This talk will explore Chapel as a language to write HPC workflows.
The patterns explored in this talk are motivated by those found in Cray HPO

- They are not a comprehensive list of workflow patterns

Cray HPO is a hyperparameter optimization framework developed at HPE

Background on hyperparameter optimization (HPO):

- In data science, a model is trained on a dataset
- Parameters are internal values in the model that are used to make predictions, e.g. slope and offset in $y=mx+b$
- Hyperparameters are external values that impact how the model is trained, e.g. learning rate in SGD
- Hyperparameter optimization is a process of tuning hyperparameters to minimize a metric, e.g. 1-accuracy
- There is a wide breadth of HPO strategies: grid, random, genetic, bayesian, and more advanced variations
Cray HPO essentially acts as a distributed workflow manager for tuning the hyperparameters

```python
from crayai import hpo  # crayai.hpo implementation is written in Chapel

# Specify training kernel and the resources on which to execute HPO training
evaluator = hpo.Evaluator('python3 source/train.py', workload_manager='slurm', nodes=16)

# Specify hyperparameter search space
params = hpo.Params([['--lr', 0.001, (1e-5, 0.1)],
                      ['--optimizer', 'Adam', ['Adam', 'Adadelta', 'Nadam']]])

# Specify optimizer and its metaparameters
optimizer = hpo.GeneticOptimizer(evaluator,
                                  generations=10,
                                  pop_size=16,
                                  mutation_rate=0.15)

# Optimize hyperparameters, results are stored in a csv file
optimizer.optimize(params)
```
WORKFLOW BUILDING BLOCKS
• Running other programs and interacting with their outputs is the basic building block of workflows
• In Chapel, running other programs can be done with the Spawn module:

```chapel
use Spawn;

var process = spawn(['./run-simulation']);
```
LAUNCHING A SUBPROCESS

- Spawning a subprocess is a non-blocking operation.
  - Blocking on the completion of the program can be done with a ‘wait’ or ‘communicate’ call:

```javascript
use Spawn;

var process = spawn(['./run-simulation'],
                     stdout=PIPE);

process.wait();
for line in process.stdout.readlines() {
    writeln(line);
}
```
LAUNCHING A DISTRIBUTED SUBPROCESS

For interacting with HPC workload managers and launchers, one can capture interface into a function:

```javascript
use Slurm;

const jobid = salloc(numNodes=32, timeout=120);
var process = srun('./run-simulation', numNodes=32, id=jobid);
process.wait();
```

Abstracting away workload manager specifics enables portability across HPC systems:

```javascript
use Jobs;

var job = new Job(workloadManager=Launchers.pbs, launcher=Launchers.aprun);
job.alloc(numNodes=32, timeout=120);
var process = job.run('./run-simulation', numNodes=32);
process.wait();
```
LAUNCHING SUBPROCESSES IN PARALLEL

- Launching subprocesses in parallel is necessary for many workloads
- The `forall` loop is the ideal construct for this functionality:
  - Creates a concurrent task mapped to available resources, such as cores, gpus, nodes, etc.

```typescript
var simulationCommands: [1..n] string = getCommands(n);

forall simulationCommand in simulationCommands {
    var process = spawn(simulationCommand, stdout=PIPE, stderr=PIPE);
    process.wait();
}
```

- Each task will block on the `process.wait()` call, allowing the subprocess to complete before starting a new one
- This `forall` maps to the local available resources, i.e. number of cores on the machine
To map the number of concurrent tasks to the available distributed resources, use the iterator explicitly:

```plaintext
use Jobs;

config const nodesPerSim = 8;
config const numNodes = 128;

var job = new Job(workloadManager=Launchers.slurm);
var inputs: [0..<n] string = readInputs('inputs.txt');
const numParallelTasks = (numNodes / nodesPerSim):int;

job.alloc(numNodes, timeout=240);
forall i in simulationCommands._value.these(tasksPerLocale=numParallelTasks) {
    var process = job.run(simulationCommand, numNodes=nodesPerSim);
    process.wait();
}
```

This will allocate 128 nodes and run 16 concurrent tasks that each launch a process with 8 nodes
VARIABLE NUMBER OF NODES PER DISTRIBUTED APPLICATION

Varying the number of nodes in the distributed application requires tracking the available node pool

```
use ChapelLocks;
var nodesAvailable = numNodes;
var lock: chpl_LocalSpinlock;
...
forall i in simulationCommands._value.these(tasksPerLocale=numParallelTasks) {
    localNumNodes = randStream.getNext(1,8);
    waitForNodes(localNumNodes);
    var process = job.run(simulationCommand, numNodes=localNumNodes);
    process.wait();
}
```
VARIABLE NUMBER OF NODES PER DISTRIBUTED APPLICATION

```java
proc waitForNodes(localNumNodes) {
    // Wait until nodes are available
    while true {
        lock.lock();
        if nodesAvailable >= localNumNodes {
            // Remove nodes from node pool
            nodesAvailable -= localNumNodes;
            lock.unlock();
            break;
        } else {
            lock.unlock();
            Time.sleep(1);
        }
    }
    // Add nodes back to node pool when done
    lock.lock();
    nodesAvailable += localNumNodes;
    lock.unlock();
}
```
I/O is an important component of any workflow application

The output generated by the subprocess can be accessed through various mechanisms:

- Parsing subprocess stdout
- Parsing files generated by subprocess
- Connecting with the subprocess and communicating in memory, e.g. over ZMQ
- Writing results to a database from the subprocess
ADVANTAGES AND DISADVANTAGES OF CHAPEL

Advantages
- Modern programming language with modern features and productivity
  - Generics, type inference, native C/python interop, etc.
- Parallel constructs built into the language
  - Parallel loops, atomics, tasks, etc.
- Performance
  - Not always important in workflow space, but can be important for sufficiently large workflow programs
  - Some workflow tools use complex algorithms in their feedback loops, e.g. Bayesian optimization in HPO
- Compiling and statically linking produces a dependency-free binary

Disadvantages
- Compilation is time consuming
- Standard library and module ecosystem not as mature as other languages such as Python
ALTERNATIVE OPTIONS FOR BUILDING WORKFLOWS

- Shell scripting
- Other modern programming languages: Python, Julia, Go, Rust, etc.
- Workflow frameworks, such as Apache Airflow
- Domain-specific workflow frameworks, such as Bioclipse
- Domain-specific languages for workflow automation, such as swift scripting language
SUMMARY

- Developing HPC workflows is a challenge faced by many scientists and engineers.
- Chapel has many appealing features making it a competitive choice for developing workflows:
  - Modern PL features
  - Parallel constructs
  - Performance
  - Portability
THANK YOU