PyLauncher

Victor Eijkhout
Texas Advanced Computing Center

Motivation and basic usage

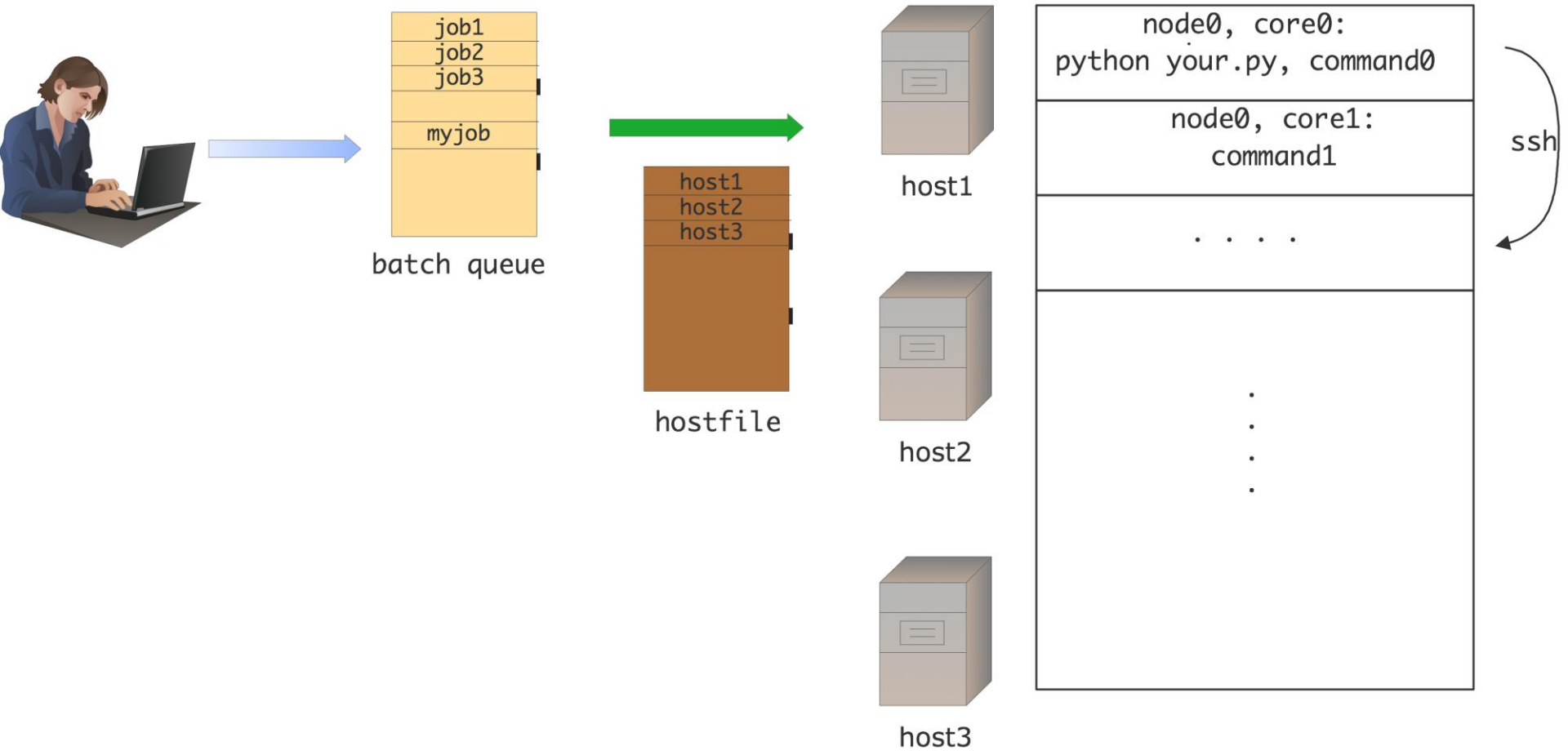
Why another launcher

- The shell-based launcher was static. This can lead to load unbalance
- Wish: restart
- Wish: dynamic task queue
- Wish: multicore jobs, MPI jobs
- Wish: customizability
- Meanwhile, the shell-based launcher has become quite a bit more sophisticated….

Availability

- Through the module system on TACC clusters
- Repository: https://bitbucket.org/VictorEijkhout/pylauncher

Structure


job1
job2
job3
myjob
batch queue
host1
host2
host3
hostfile
host1
host2
host3
node0, core0:
python your.py, command0
node0, core1:
command1
. . . .
ssh

Scenarios

- Single-threaded jobs, one per core
- Single-threaded jobs, need more memory
- Multi-threaded jobs
- MPI jobs
- Unifying theme: you have way more jobs than cores

Basic usage

- You need to write a little Python. Single core jobs:

```
#!/usr/bin/env python
import pylauncher
pylauncher.ClassicLauncher("commandlines")
```

- Multi-core and large memory jobs:

```
pylauncher.ClassicLauncher("commandlines",cores=4)
```

- Variable core:

```
pylauncher.ClassicLauncher("commandlines",cores=“file”)
```

- MPI

```
pylauncher.IbrunLauncher("commandlines",cores=“file”)
```

- MIC:

```
pylauncher.MICLauncher("commandlines",cores=“file”)
```

Input files

```
#
# example commandlines
#
echo "command 0"; sleep 21
echo "command 1"; sleep 14
# skip a few
echo "command 9"; sleep 29

echo "command 10"; sleep 16
```

```
# core count attached
5,echo "command 0"; sleep 21
5,echo "command 1"; sleep 14
3,echo "command 12"; sleep 13
3,echo "command 13"; sleep 24
3,echo "command 14"; sleep 28
```

- Each line is one job, note blank lines and comments, core count for “file” option
- String PYL_ID gets replaced by command number

Some implementation details

- Each pylauncher run leaves behind a directory “pylauncher_tmpdir12345” where 12345 is jobid
- Each command line gets wrapped into a file “command1”, “command2”
- A successfully executed command leaves behind “expire1”, “expire2” &c

Checkpoint / restart

- There is also a file “queuestate”
- If your job runs out of time or crashes:

```
pylauncher.ClassicLauncher("commandlines",cores=4,
   resume=12345)
```

- This will re-issue all not-finished commands from the aborted job

Use at TACC

- “module load pylauncher/2.0”
- Define $TACC_PYLAUNCHER_DIR, sets $PYTHONPATH
- See “examples” and “documentation” subdirectories.

Unit tests

- The source is written for “nosetests”:
  all functions & classes that start with “test” are unit tests
- Currently 30 tests, some specific to TACC (meaning: SGE & SLURM)
- Takes about 1 minute to run, mostly because of “sleep” commands
- Some classes are only for testing: they generate dummy commands and command files

Customizing the pylauncher

Commandline classes

CommandlineGenerator

- “commandline” = < Command , Corecount >
- “command” can also be “stall” (dynamic case)
- Arguments: list=[c1,c2,…] nmax=0,>0,None
  - Nmax is None (default): iterate over the list
  - Nmax>0: stop after that many
  - nmax==0: wait for finish() call
- Method: finish() stop the generator, necessary for dynamic case

FileCommandlineGenerator

- Inherits from CommandlineGenerator
- Argument 1: filename
  - Each line is a Unix commandline
  - Line can optionally start with < [0-9]+ ”,” >
  - Blank lines and comments with “#” are skipped
- Argument 2 (optional): core count

DynamicCommandlineGenerator

- Inherits from CommandlineGenerator
- Method: append(command[,cores])

Host management classes

Node

- Description of a slot for a task
- Argument (opt): host=name, core=num, nodeid=num
- Methods: occupyWithTask(tid), release()
- Test: isfree()

HostList

- Contains a list of host,core pairs
- Derived class SGEHostList
- Derived class SLURMHostList
- *Derive your own class for LL, PBS, &c*
- Function: TACCHostlist gives SGE or SLURM hostlist depending on hostname

HostPool

- Maintains a pool of Node objects
- Arguments (opt): nhosts=num, hostlist=[h1,h2,…] or [h1;c1, h2;c2, …]
- Argument (opt): commandprefixer= (see next)
- Method: requestNodes(n) (returns HostLocator; see next)
- Method: occupyNodes(locator, taskid)
- Method: releaseNodesByTask(taskid)

prefixers

- Routine: takes a commandline and returns it with “ssh host” or so prefixed
- LocalPrefixer: identity
- SSHPrefixer(command,pool) returns:
  cd curdir ; env the_environment ; ssh pool command
- IbrunPrefixer(command,pool) returns:
  ibrun –o poolfirst –n poollength command

HostLocator

- Arguments: pool=…, offset=…, extent=…

TACCHostPool

- Inherits from HostPool
- By default uses SSHPrefixer

Task management classes

Completion

- Argument (optional): taskid
- Method: attach(command)
  returns command with completion action attached
- Method: test()

FileCompletion

- Inherits from Completion
- Completion based on creating “stamp” file
- Argument: taskid=nnn
- Argument (optional): stamproot=basename
- Argument (optional): stampdir=directory

Task

- Argument: completion=
- Argument (optional): size=nnn (default=1)
- Argument (optional): taskid=nnn
- Method: hasCompleted()
- Method: startonnodes(hostlocator,….)

TaskQueue

- Method: enqueue(task)
- Method: startQueued(hostpool)
- Test: isEmpty()

TaskGenerator

- Iteratable, yields a Task or “stall”
- Argument 1: commandlinegenerator
- Argument (opt): completion = function that gives Completion objects

Tieing it all together

LauncherJob

- Method: tick() submits queued jobs, returns:
  - “finished”: if generator stops, and all jobs finished
  - “continuing”: generator stopped, jobs still running
  - “stalling”: generator is stalled
  - “expired nnn”: reports on completed job
- Method: run() repeats tick until completed
- Argument (req’d): hostpool=
- Argument (req’d): taskgenerator=
- Argument (opt): prefixer, jobid, launcherdir

Writing simple launchers is easy!

```
def ClassicLauncher(commandfile,**kwargs):
    cores = kwargs.pop("cores",1)
    job = LauncherJob(
        hostpool=HostPool(hostlist=TACCHostList()),
        taskgenerator=TaskGenerator( FileCommandlineGenerator(commandfile,cores),
                                     completion=lambda x:FileCompletion( taskid=x,
                                                                         stamproot="expire",stampdir=workdir),
        **kwargs)
    job.run()
```

Customizing launchers is easy too!

```
# default run method
def run(self):
    """Invoke the launcher job, and call ``tick`` until all jobs are finished."""”
    self.starttime = time.time()
    while True:
        res = self.tick()
        if res=="finished":
            break
        if self.maxruntime>0 and time.time()-self.starttime>self.maxruntime:
            break
```

```
# post-processing run method
def run(self):
    …….
    if re.match(res,”expired”):
      # post-process
```

```
# dynamic launcher
def run(self):
    …….
    if re.match(res,”expired”):
      # post-process, and:
      self.taskgenerator.\
          commandlinegenerator.append\
              ( Commandline( newcommand) )
```