User Guide for the Python Launcher at TACC

Victor Eijkhout

Version 1.5, October 2012

The python launcher is a utility[1] for executing many instances of a simple, possibly sequential, program in a single submitted batch job. For simple jobs, where a static list of commandlines is available, the user is only required to write a three line python script (examples below); for more complicated workflows, some python programming may be required.

1 General remarks on using the python launcher

The launcher module is used inside a batch script. If you do not know how to write a batch script, please consult the userguide for your cluster of choice. Inside your batch script, you execute a launcher script with python:

```
# .... the usual batch script stuff
module load pylauncher
python your_launcher_script.py
```

The python script needs to start with

```
import pylauncher
```

or

```
#!/usr/bin/env python
import pylauncher
```

if you want to invoke it as an executable shell script.

From now on we will take these lines for given and only discuss the launcher commands in the script.

Running the pylauncher leaves behind a temporary directory `pylauncher_tmpdir.123456`, with `123456` replaced by your job number. This directory can safely be deleted after the launcher job finishes.

1. We add the qualification 'python' since TACC also offers a more limited utility, simply called 'the launcher'.

2 Use cases

2.1 Classic launcher mode

In the 'classic launcher' setup, the user has made a file containing commandlines. These commandlines are executed by having this in the python script:

```
pylauncher.ClassicLauncher("commandlines")
```

where the `commandlines` file contains one commandline per line:

```
commandline1
echo "this is command 2" ; commandline2
if [ some_test ] ; then commandline3a ; else commandline3b ; fi
.... et cetera
```

Blank lines are allowed in the commandlines file, and comment lines starting with `#` are ignored.

The batch job that calls the script will execute these commandlines on all the nodes and cores that have been requested. Each line is executed in a separate shell, and this shell inherits the user environment and starts in the current working directory.

2.2 Launching multithreaded jobs

In the classic launcher setup each commandline is executed on a single core. If the commandline calls a multithreaded program, and therefore more cores are requested, the above call can be altered:

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

A variable number of cores can be accomodated too, but requires a more complicated file of commandlines.

```
pylauncher.CoreLauncher("corecommandlines")
```

where the input file `corecommandlines` is

```
ncores1,commandline1
ncores2,commandline2
... et cetera
```

2.3 Launching MPI jobs

What if you have a large number of MPI jobs that need a relatively small core count? The python launcher can accomodate this with

```
pylauncher.MPILauncher("mpicommandlines",cores=4)
```

for a constant core count, or

```
pylauncher.MPILauncher("mpicommandlines",cores="file")
```

for a variable core count, declared in the file:

```
ncores1,commandline1
ncores2,commandline2
... et cetera
```

Unfortunately in this mode the batch output file will contain a lot of noise, since each commandline is treated as a separate parallel job, submitted with `ibrun`.

2.4 A dynamic launcher

The following example code is meant to cover the use case where the job list is dynamically generated, possibly in response to jobs on the list finishing and being post-processed. The user now needs to program a python object that has two methods:

- `generate`. A function with no arguments. This method will be called when the launcher has cores available: by calling this method the launcher is requesting a new commandline from the user. The method should return two variables: the commandline, and the number of cores on which to execute it. In other words, your code for this method will typically look like

  ```
  def generate(self):
    job = "commandline"
    cores = 4
    return job,cores
  ```

  Two exceptions: if there are temporarily no commandlines available, the `generate` method should return `"stall",1`; if the job is finished the method should return `"stop",1`. (The number of cores is irrelevant in these cases.)
- `expire`. A function with one argument `id`. This function is called whenever a commandline finished, and the argument is then the id of that commandline. It is entirely up to the user to parse what this id means, and what action to perform accordingly. In the following example this function appends a new commandline to an internally maintained list, and the `generate` function then passes these commandlines to the launcher.

```
#
# a joblist object, which has methods for generating new commandlines
# and for postprocessing finished commands
#
class joblist():
    def __init__(self,list):
        self.list = list; self.njobs = len(self.list)
    def generate(self):
        if self.njobs<30:
            if len(self.list)>0:
                j = self.list.pop(); self.njobs += 1
                return j,1
            else:
                return "stall",1
```

```
        return "stop",1
    def expire(self,id):
        print "Processing expired task ",id
        self.list.append("sleep "+str(10+int(30*random.random())))

#
# we create a dynamic launcher job to that will create new commands
# while running; an initial job list is provided here.
#
job = pylauncher.DynamicLauncher(
    joblist( [ "command1", "command2", "command3" ] )
    )
```

3 Tracing

You can trace the workings of the launcher, to see when jobs start and finish, and where they are scheduled, by adding a `debug` value to the launcher call:

```
pylauncher.ClassicLauncher("commandlines",debug=1)
```

4 Customization

Many aspects of the launcher can be customized. Here are just a few examples.

- The name of the temporary directory can be set by a keyword argument:
  ```
  j = Job(launcherdir=".mylauncherdir",.....)
  j = LauncherJob(launcherdir=".mylauncherdir",.....)
  ```