PyLauncher Documentation

Release 2.0

Victor Eijkhout

Sep 13, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION AND GENERAL USAGE

This is the documentation of the pylauncher utility by Victor Eijkhout.

1.1 Motivation

There are various scenarios where you want to run a large number of serial or low-corecount parallel jobs. Many cluster scheduling systems do not allow you to submit a large number of small jobs (and it would probably lower your priority!) so it would be a good idea to package them into one large parallel job.

Let’s say that you have 4000 serial jobs, and your cluster allows you to allocate 400 cores, then packing up the serial jobs could be executed on those 400 cores, in approximately the time of 10 serial jobs.

The tool to do this packing is called a *parametric job launcher*. The ‘parametric’ part refers to the fact that most of the time your serial jobs will be the same program, just invoked with a different input parameter. One also talks of a ‘parameter sweep’ for the whole process.

A simple launcher scenario would take a file with command lines, and give them out cyclicly to the available cores. This mode is not optimal, since one core could wind up with a few processes that take much longer than the others. Therefore we want a dynamic launcher that keeps track of which cores are free, and schedules jobs there.

In a very ambitious scenario, you would not have a static list of commands to execute, but new commandlines would be generated depending on the ones that are finished. For instance, you could have a very large parameter space, and the results of finished jobs would tell you what part of space to explore next, and what part to ignore.

The pylauncher module supports such scenarios.

1.2 Here’s what I want to know: do I have to learn python?

Short answer: probably not. The pylauncher utility is written in python, and to use it you have to write a few lines of python. However, for most common scenarios there are example scripts that you can just copy.

Longer answer: only if you want to get ambitious. For common scenarios there are single function calls which you can copy from example scripts. However, the launcher is highly customizable, and to use that functionality you need to understand something about python’s classes and you may even have to code your own event loop. That’s the price you pay for a very powerful tool.

1.3 Realization

The pylauncher is a very customizable launcher utility. It provides base classes and routines that take care of most common tasks; by building on them you can tailor the launcher to your specific scenario.

Since this launcher was developed for use at the Texas Advanced Computing Center, certain routines are designed for the specific systems in use there. In particular, processor management is based on the SGE and SLURM job schedulers and the environment variables they define. By inspecting the source it should be clear how to customize the launcher for other schedulers and other environments.

If you write such customizations, please contact the author. Ideally, you would fork the repository https://bitbucket.org/VictorEijkhout/pylauncher and let the author re-absorb that.

CHAPTER

TWO

A QUICK TUTORIAL

2.1 Setup

You need to have the files `pylauncher.py` and `hostlist.py` in your `PYTHONPATH`. If you are at TACC, do `module load pylauncher` and all is good.

2.2 Batch operation

The most common usage scenario is to use the launcher to bundle many small jobs into a single batch submission on a cluster. In that case, just put

> module load python
>
> python your_launcher_file.py

in the jobscript.

If you are using TACC's stampede cluster, and you want to run the launcher script on the Intel Xeon PHI co-processor, do

> micrun /mic/python your_launcher_file.py

where '/mic/python' is the path to a python that is compiled for MIC. Currently no such python is officially available on Stampede.

2.3 Examples

There is an `examples` subdirectory with some simple scenarios of how to invoke the pylauncher.

2.3.1 Single-core jobs

In the simplest scenario, we have a file of commandlines, each to be executed on a single core.

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

import pylauncher

##
## Emulate the classic launcher, using a one liner
##
```

```
#pylauncher.ClassicLauncher("corecommandlines",debug="job+host+task")
pylauncher.ClassicLauncher("commandlines",debug="job")
```

where the commandlines file is:

```
####
#### This file was automatically generated by:
#### python make_commandlines.py 256 1 40
####
echo 0 >> /dev/null 2>&1  ; sleep 21
echo 1 >> /dev/null 2>&1  ; sleep 30
echo 2 >> /dev/null 2>&1  ; sleep 8
echo 3 >> /dev/null 2>&1  ; sleep 34
echo 4 >> /dev/null 2>&1  ; sleep 39
echo 5 >> /dev/null 2>&1  ; sleep 9
```

2.3.2 Constant count multi-core jobs

The next example uses again a file of commandlines, but now the launcher invocation specifies a core count that is to be used for each job.

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

import pylauncher

##
## Emulate the classic launcher, using a one liner
##

pylauncher.ClassicLauncher("commandlines",
                           debug="job+task+host+exec+command",
                           cores=2)
```

2.3.3 Variable count multi-core jobs

If we have multithreaded jobs, but each has its own core count, we add the core count to the file of commandlines, and we tell the launcher invocation that that is where the counts are found.

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

import pylauncher

##
## Emulate the classic launcher, using a one liner
##

pylauncher.ClassicLauncher("corecommandlines",
                           debug="job+task+host+exec+command",
                           cores="file",
                           )
```

```
#
# Automatically generated commandlines
#
5,echo "command 0"; sleep 21
5,echo "command 1"; sleep 14
5,echo "command 2"; sleep 23
5,echo "command 3"; sleep 13
5,echo "command 4"; sleep 29
5,echo "command 5"; sleep 12
5,echo "command 6"; sleep 23
```

2.3.4 MPI parallel jobs

If your program uses the MPI library and you want to run multiple instances simultaneously, use the `IbrunLauncher`.

```
4,./parallel 0 10
4,./parallel 1 10
4,./parallel 2 10
4,./parallel 3 10
4,./parallel 4 10
4,./parallel 5 10
4,./parallel 6 10
4,./parallel 7 10
4,./parallel 8 10
4,./parallel 9 10
```

This example uses a provided program, `parallel.c` of two parameters:

- the job number
- the number of seconds running time

The program will report the size of its communicator, that is, on how many cores it is running.

2.3.5 Job restarting

If your job runs out of time, it will leave a file `queuestate` that describes which tasks were completed, which ones were running, and which ones were still scheduled to fun. You can submit a job using the `ResumeClassicLauncher`:

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

import pylauncher

##
## This resumes a classic launcher from a queuestate file
##

pylauncher.ResumeClassicLauncher("queuestate",debug="job")
```

CHAPTER

THREE

IMPLEMENTATION

3.1 Commandline generation

The term ‘commandline’ has a technical meaning: a commandline is a two-element list or a tuple where the first member is the Unix command and the second is a core count. These commandline tuples are generated by a couple of types of generators.

The `CommandlineGenerator` base class handles the basics of generating commandlines. Most of the time you will use the derived class `FileCommandlineGenerator` which turns a file of Unix commands into commandlines.

Most of the time a commandline generator will run until some supply of commands run out. However, the `DynamicCommandlineGenerator` class runs forever, or at least until you tell it to stop, so it is good for lists that are dynamically replenished.

3.2 Host management

We have an abstract concept of a node, which is a slot for a job. Host pools are the management structure for these nodes: you can query a host pool for sufficient nodes to run a multiprocess job.

A host pool has associated with it an executor object, which represents the way tasks (see below) are started on nodes in that pool. Executors are also discussed below.

3.3 Task management

Tasks are generated internally from a `TaskGenerator` object that the user can specify. The `TaskQueue` object is created internally in a `LauncherJob`. For the `completion` argument of the `TaskGenerator`, see below.

3.3.1 Executors

At some point a task needs to be executed. It does that by applying the `execute` method of the `Executor` object of the `HostPool`. (The thinking behind attaching the execution to a host pool is that different hostpools have different execution mechanisms.) Executing a task takes a commandline and a host locator on which to execute it; different classes derived from `Executor` correspond to different spawning mechanisms.

3.3.2 Task Completion

Task management is largely done internally. The one aspect that a user could customize is that of the completion mechanism: by default each commandline that gets executed leaves a zero size file behind that is branded with the task number. The TaskQueue object uses that to detect that a task is finished, and therefore that its Node objects can be released.

Task generators need completions dynamically generated since they need to receive a job id. You could for instance specify code such as the following; see the example launchers.

```
completion=lambda x:FileCompletion( taskid=x,
             stamproot="expire",stampdir="workdir")
```

3.4 Jobs

All of the above components are pulled together in the LauncherJob class. Writing your own launcher this way is fairly easy; see the TACC section for some examples of launchers.

CHAPTER

FOUR

TACC SPECIFICS AND EXTENDABILITY TO OTHER INSTALLATIONS

The pylauncher source has a number of classes and routines that are tailored to the use at the Texas Advanced Computing Center. For starters, there are two classes derived from `HostList`, that parse the hostlists for the SGE and SLURM scheduler. If you use Load Leveler or PBS, you can write your own using these as an example.

Two utility functions may help you in writing customizations.

CHAPTER

FIVE

TACC LAUNCHERS

CHAPTER

SIX

TRACING AND PROFILING

It is possible to generate trace output during a run and profiling (or summary) information at the end.

6.1 Trace output

You can get various kinds of trace output on your job. This is done by specifying a `debug=....` parameter to the creation of the various classes. For the easy case, pass `debug="job+host+task"` to a launcher object.

Here is a list of the keywords and what they report on:

- host: for `HostPool` objects.
- command: for `CommandlineGenerator` objects.
- task: for `Task` and `TaskGenerator` objects.
- exec: for `Executor` objects. For the SSHExecutor this prints out the contents of the temporary file containing the whole environment definition.
- ssh: for `SSHExecutor` objects.
- job: for `LauncherJob` objects.

6.2 Final reporting

Various classes can produce a report. This is intended to be used at the end of a job, but you can do it really at any time. The predefined launchers such as `ClassicLauncher` print out this stuff by default.

CHAPTER SEVEN

TESTING

The pylauncher.py source file has a large number of unittests that are designed for the nosetests framework: all routines and classes starting with `test` are only for testing purposes.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search