PyLauncher Documentation

Release 2.0

Victor Eijkhout

September 26, 2013

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:

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

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`.

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

import pylauncher

pylauncher.IbrunLauncher("parallellines",cores="file",
                         debug="job+host+task+exec")
```

```
4,./parallel 0 30
4,./parallel 1 30
4,./parallel 2 30
4,./parallel 3 30
4,./parallel 4 30
4,./parallel 5 30
4,./parallel 6 30
4,./parallel 7 30
4,./parallel 8 30
4,./parallel 9 30
```

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.

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.

class `pylauncher.`**`CommandlineGenerator`**(***kwargs*)

An iteratable class that generates a stream of `Commandline` objects.

The behaviour of the generator depends on the `nmax` parameter:

- nmax is None: exhaust the original list
- nmax > 0: keep popping until the count is reached; if the initial list is shorter, someone will have to fill it, which this class is not capable of
- nmax == 0 : iterate indefinitely, wait for someone to call the `finish` method

In the second and third scenario it can be the case that the list is empty. In that case, the generator will yield a COMMAND that is `stall`.

Parameters

- **list** – (keyword, default []) initial list of Commandline objects
- **nax** – (keyword, default None) see above for explanation

`finish`()

Tell the generator to stop after the commands list is depleted

`next`()

Produce the next Commandline object, or return an object telling that the generator is stalling or has stopped

class `pylauncher.`**`CommandlineGenerator`**(***kwargs*)

An iteratable class that generates a stream of `Commandline` objects.

The behaviour of the generator depends on the `nmax` parameter:

- nmax is None: exhaust the original list
- nmax > 0: keep popping until the count is reached; if the initial list is shorter, someone will have to fill it, which this class is not capable of
- nmax == 0 : iterate indefinitely, wait for someone to call the `finish` method

In the second and third scenario it can be the case that the list is empty. In that case, the generator will yield a COMMAND that is `stall`.

Parameters

- **list** – (keyword, default []) initial list of Commandline objects
- **nax** – (keyword, default None) see above for explanation

abort()
Stop the generator, even if there are still elements in the commands list

finish()
Tell the generator to stop after the commands list is depleted

next()
Produce the next Commandline object, or return an object telling that the generator is stalling or has stopped

class pylauncher.**FileCommandlineGenerator**(*filename*, ***kwargs*)
Bases: pylauncher.CommandlineGenerator

A generator for commandline files: blank lines and lines starting with the comment character '#' are ignored

- cores is 1 by default, other constants allowed.
- cores=='file' means the file has << count,command >> lines
- if the file has core counts, but you don't specify the 'file' value, they are ignored.

Parameters

- **filename** – (required) name of the file with commandlines
- **cores** – (keyword, default 1) core count to be used for all commands
- **dependencies** – (keyword, default False) are there task dependencies?

class pylauncher.**DynamicCommandlineGenerator**(***kwargs*)
Bases: pylauncher.CommandlineGenerator

A CommandlineGenerator with an extra method:

`append`: add a Commandline object to the list

The 'nmax=0' parameter value makes the generator keep expecting new stuff.

append(*command*)
Append a unix command to the internal structure of the generator

class pylauncher.**DirectoryCommandlineGenerator**(*command_directory*, *commandfile_root*, ***kwargs*)
Bases: pylauncher.DynamicCommandlineGenerator

A CommandlineGenerator object based on finding files in a directory.

Parameters

- **command_directory** – (directory name, required) directory where commandlines are found; unlike launcher job work directories, this can be reused.

- **commandfile_root** – (string, required) only files that start with this, followed by a dash, are inspected for commands. A file can contain more than one command.
- **cores** – (keyword, optional, default 1) core count for the commandlines.

next()
List the directory and iterate over the commandfiles:

•ignore any open files, which are presumably still being written

•if they are marked as scheduled, ignore

•if there is a file `finish-nnn`, mark job nnn as finished

•if they are not yet scheduled, call `append` with a `Commandline` object

If the finish name is present, and all scheduled jobs are finished, finish the generator.

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.

class `pylauncher.`**Node**(*host=None*, *core=None*, *nodeid=-1*)
A abstract object for a slot to execute a job. Most of the time this will correspond to a core.

A node can have a task associated with it or be free.

isfree()
Test whether a node is occupied

occupyWithTask(*taskid*)
Occupy a node with a taskid

release()
Make a node unoccupied

class `pylauncher.`**HostList**(*list=*[ ], *tag=''*)
Object describing a list of hosts. Each host is a dictionary with a `host` and `core` field.

Arguments:

•list : list of hostname strings

•tag : something like `.tacc.utexas.edu` may be necessary to ssh to hosts in the list

This is an iteratable object; it yields the host/core dictionary objects.

append(*h*, *c=0*)
Arguments:

•h : hostname

•c (optional, default zero) : core number

class `pylauncher.`**HostPoolBase**(***kwargs*)
A base class that defines some methods and sets up the basic data structures.

Parameters

- **commandexecutor** – (keyword, optional, default=‘‘LocalExecutor‘‘) the `Executor` object for this host pool
- **workdir** – (keyword, optional) the workdir for the command executor
- **debug** – (keyword, optional) a string of debug types; if this contains ‘host’, anything derived from `HostPoolBase` will do a debug trace

append_node(*host='localhost'*, *core=0*)
Create a new item in this pool by specifying either a Node object or a hostname plus core number. This function is called in a loop when a `HostPool` is created from a `HostList` object.

final_report()
Return a string that reports how many tasks were run on each node.

occupyNodes(*locator*, *taskid*)
Occupy nodes with a taskid

Argument: * locator : HostLocator object * taskid : like the man says

release()
If the executor opens ssh connections, we want to close them cleanly.

releaseNodesByTask(*taskid*)
Given a task id, release the nodes that are associated with it

request_nodes(*request*)
Request a number of nodes; this returns a HostLocator object

unique_hostnames(*pool=None*)
Return a list of unique hostnames. In general each hostname appears 16 times or so in a HostPool since each core is listed.

class pylauncher.**HostPool**(***kwargs*)
Bases: pylauncher.HostPoolBase

A structure to manage a bunch of Node objects. The main internal object is the `nodes` member, which is a list of Node objects.

Parameters

- **nhosts** – the number of slots in the pool; this will use the localhost
- **hostlist** – HostList object; this takes preference over the previous option
- **commandexecutor** – (optional) a prefixer routine, by default LocalExecutor

class pylauncher.**HostLocator**(*pool=None*, *extent=None*, *offset=None*)
A description of a subset from a HostPool. A locator object is typically created when a task asks for a set of nodes from a HostPool. Thus, a locator inherits the executor from the host pool from which it is taken.

The only locator objects allowed at the moment are consecutive subsets.

Parameters

- **pool** – HostPool (optional)
- **extent** – number of nodes requested
- **offset** – location of the first node in the pool

class pylauncher.**DefaultHostPool**(***kwargs*)
Bases: pylauncher.HostPool

A HostPool object based on the hosts obtained from the `HostListByName` function, and using the `SSHExecutor` function.

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.

class `pylauncher.`**`Task`**(*command*, ***kwargs*)
A Task is an abstract object associated with a commandline

Parameters

- **command** – (required) Commandline object; note that this contains the core count
- **completion** – (keyword, optional) Completion object; if unspecified the trivial completion is used.
- **taskid** – (keyword) identifying number of this task; has to be unique in a job, also has to be equal to the taskid of the completion
- **debug** – (keyword, optional) string of debug keywords

`hasCompleted`()
Execute the completion test of this Task

`line_with_completion`()
Return the task's commandline with completion attached

`start_on_nodes`(***kwargs*)
Start the task.

Parameters

- **pool** – HostLocator object (keyword, required) : this describes the nodes on which to start the task
- **commandexecutor** – (keyword, optional) prefixer routine, by default the commandexecutor of the pool is used

This sets `self.startime` to right before the execution begins. We do not keep track of the endtime, but instead set `self.runningtime` in the `hasCompleted` routine.

class `pylauncher.`**`TaskQueue`**(***kwargs*)
Object that does the maintains a list of Task objects. This is internally created inside a `LauncherJob` object.

`enqueue`(*task*)
Add a task to the queue

`final_report`()
Return a string describing the max and average runtime for each task.

`find_recently_completed`()
Find the first recently completed task. Note the return, not yield.

`isEmpty`()
Test whether the queue is empty and no tasks running

`startQueued`(*hostpool*)
for all queued, try to find nodes to run it on; the hostpool argument is a HostPool object

class `pylauncher.`**`TaskGenerator`**(*commandlines*, ***kwargs*)
iterator class that can yield the following:

•a Task instance, or

- the keyword `stall`; this indicates that the commandline generator is stalling and this will be resolved when the outer environment does an `append` on the commandline generator.
- the `pylauncherBarrierString`; in this case the outer environment should not call the generator until all currently running tasks have concluded.
- the keyword `stop`; this means that the commandline generator is exhausted. The `next` function can be called repeatedly on a stopped generator.

You can iterate over an instance, or call the `next` method. The `next` method can accept an imposed taskcount number.

Parameters

- **commandlinegenerator** – either a list of unix commands, or a CommandlineGenerator object
- **completion** – (optional) a function of one variable (the task id) that returns Completion objects
- **debug** – (optional) string of requested debug modes

next(*imposedcount=None*)
Deliver a Task object, or a special string:

- "stall" : the commandline generator will give more, all in good time
- "stop" : we are totally done

pylauncher.**TaskGeneratorIterate**(*gen*)
In case you want to iterate over a TaskGenerator, use this generator routine

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.

class pylauncher.**Executor**(***kwargs*)
Class for starting a commandline on some actual computing device.

All derived classes need to define a `execute` method.

Parameters

- **catch_output** – (keyword, optional, default=True) state whether command output gets caught, or just goes to stdout
- **workdir** – (optional, default="pylauncher_tmpdir_exec") directory for exec and out files
- **debug** – (optional) string of debug modes; include "exec" to trace this class

Important note: the `workdir` should not already exist. You have to remove it yourself.

workdir_is_safe()
Test that the working directory is (in) a subdirectory of the cwd

wrap(*command*)
Take a commandline, write it to a small file, and return the commandline that sources that file

class pylauncher.**LocalExecutor**(***kwargs*)
Bases: pylauncher.Executor

Execute a commandline locally, in the background.

Parameters **prefix** – (keyword, optional, default null string) for recalcitrant shells, the possibility to specify '/bin/sh' or so

class pylauncher.**SSHExecutor**(***kwargs*)
Bases: pylauncher.Executor

Prepend a command with an ssh to the pool; this also does a `cd` to the current directory, and sets up the current environment.

Note: environment variables with a space, semicolon, or parentheses are not transferred.

Parameters

- **command** – a unix command, including semicolons and whatnot
- **pool** – a HostLocator object
- **workdir** – if this is None, the ssh connection will cd to the current directory, otherwise it will go to this workdir. If this is a relative path, it is taken relative to the current directory.

execute(*usercommand*, ***kwargs*)
Execute a commandline by

•making an ssh connection to the host locator; this uses `paramiko`;

•cd to the current directory;

•setting up the environment (we filter out variables with semicolon, space, or parentheses in the name or value); and

•executing the command in the background

Parameters **pool** – (required) either a Node or HostLocator

class pylauncher.**IbrunExecutor**(***kwargs*)
Bases: pylauncher.Executor

An Executor derived class for the shift/offset version of ibrun that is in use at TACC

Parameters

- **pool** – (required) `HostLocator` object
- **stdout** – (optional) a file that is open for writing; by default `subprocess.PIPE` is used

execute(*command*, ***kwargs*)
Much like `SSHExecutor.execute()`, except that it prefixes with `ibrun -n -o`

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.

class pylauncher.**Completion**(*taskid=0*)
Define a completion object for a task.

The base class doesn't do a lot: it immediately returns true on the completion test.

attach(*txt*)
Attach a completion to a command, giving a new command

test()
Test whether the task has completed

class pylauncher.**FileCompletion**(**kwargs)
Bases: pylauncher.Completion

FileCompletion is the most common type of completion. It appends to a command the creation of a zero size file with a unique name. The completion test then tests for the existence of that file.

Parameters

- **taskid** – (keyword, required) this has to be unique. Unfortunately we can not test for that.
- **stampdir** – (keyword, optional, default is self.stampdir, which is ".") directory where the stampfile is left
- **stamproot** – (keyword, optional, default is "expire") root of the stampfile name

attach(*txt*)
Append a 'touch' command to the txt argument

stampname()
Internal function that gives the name of the stamp file, including directory path

test()
Test for the existence of the stamp file

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.

class pylauncher.**LauncherJob**(**kwargs)
LauncherJob class. Keyword arguments:

Parameters

- **hostpool** – a HostPool instance (required)
- **taskgenerator** – a TaskGenerator instance (required)
- **delay** – between task checks (optional)
- **debug** – list of keywords (optional)
- **gather_output** – (keyword, optional, default None) filename to gather all command output

tick()
This routine does a single time step in a launcher's life, and reports back to the user. Specifically:

•It tries to start any currently queued jobs. Also:

•If any jobs are finished, it detects exactly one, and reports its ID to the user in a message `expired 123`

•If there are no finished jobs, it invokes the task generator; this can result in a new task and the return message is `continuing`

- if the generator stalls, that is, more tasks will come in the future but none are available now, the message is `stalling`
- if the generator is finished and all jobs have finished, the message is `finished`

After invoking the task generator, a short sleep is inserted (see the `delay` parameter)

run()
Invoke the launcher job, and call `tick` until all jobs are finished.

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.

class `pylauncher.`**`SGEHostList`**`(`*`**kwargs`*`)`
Bases: `pylauncher.HostList`

class `pylauncher.`**`SLURMHostList`**`(`*`**kwargs`*`)`
Bases: `pylauncher.HostList`

`pylauncher.`**`HostListByName`**`(`*`**kwargs`*`)`
Give a proper hostlist. Currently this work for the following TACC hosts:

- `ls4`: Lonestar, using SGE
- `stampede`: Stampede, using SLURM
- `mic`: Intel Xeon PHI co-processor attached to a compute node

We return a trivial hostlist otherwise.

class `pylauncher.`**`DefaultHostPool`**`(`*`**kwargs`*`)`
Bases: `pylauncher.HostPool`

A HostPool object based on the hosts obtained from the `HostListByName` function, and using the `SSHExecutor` function.

Two utility functions may help you in writing customizations.

`pylauncher.`**`HostName`**`()`
This just returns the hostname. See also `ClusterName`.

`pylauncher.`**`ClusterName`**`()`
Assuming that a node name is along the lines of `c123-456.cluster.tacc.utexas.edu` this returns the second member. Otherwise it returns None.

`pylauncher.`**`JobId`**`()`
This function is installation dependent: it inspects the environment variable that holds the job ID, based on the actual name of the host (see `HostName`): this should only return a number if we are actually in a job.

CHAPTER

FIVE

TACC LAUNCHERS

pylauncher.**ClassicLauncher**(*commandfile*, ***kwargs*)

A LauncherJob for a file of single or multi-threaded commands.

The following values are specified for your convenience:

•hostpool : based on HostListByName

•commandexecutor : SSHExecutor

•taskgenerator : based on the `commandfile` argument

•completion : based on a directory `pylauncher_tmp` with jobid environment variables attached

Parameters

- **commandfile** – name of file with commandlines (required)
- **cores** – number of cores (keyword, optional, default=1)
- **workdir** – directory for output and temporary files (optional, keyword, default uses the job number); the launcher refuses to reuse an already existing directory
- **debug** – debug types string (optional, keyword)

pylauncher.**IbrunLauncher**(*commandfile*, ***kwargs*)

A LauncherJob for a file of small MPI jobs.

The following values are specified for your convenience:

•hostpool : based on HostListByName

•commandexecutor : IbrunExecutor

•taskgenerator : based on the `commandfile` argument

•completion : based on a directory `pylauncher_tmp` with jobid environment variables attached

Parameters

- **commandfile** – name of file with commandlines (required)
- **cores** – number of cores (keyword, optional, default=4, see `FileCommandlineGenerator` for more explanation)
- **workdir** – directory for output and temporary files (optional, keyword, default uses the job number); the launcher refuses to reuse an already existing directory
- **debug** – debug types string (optional, keyword)

pylauncher.**MICLauncher**(*commandfile*, ***kwargs*)

A LauncherJob for execution entirely on an Intel Xeon Phi.

See `ClassicLauncher` for an explanation of the parameters. The only difference is in the use of a LocalExecutor. Treatment of the MIC cores is handled in the `HostListByName`.

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.

class `pylauncher.`**`HostPoolBase`**(***kwargs*)

A base class that defines some methods and sets up the basic data structures.

Parameters

- **commandexecutor** – (keyword, optional, default=‘‘LocalExecutor‘‘) the `Executor` object for this host pool
- **workdir** – (keyword, optional) the workdir for the command executor
- **debug** – (keyword, optional) a string of debug types; if this contains ‘host’, anything derived from `HostPoolBase` will do a debug trace

`final_report`()

Return a string that reports how many tasks were run on each node.

class pylauncher.**TaskQueue**(***kwargs*)
Object that does the maintains a list of Task objects. This is internally created inside a LauncherJob object.

final_report()
Return a string describing the max and average runtime for each task.

class pylauncher.**LauncherJob**(***kwargs*)
LauncherJob class. Keyword arguments:

Parameters

- **hostpool** – a HostPool instance (required)
- **taskgenerator** – a TaskGenerator instance (required)
- **delay** – between task checks (optional)
- **debug** – list of keywords (optional)
- **gather_output** – (keyword, optional, default None) filename to gather all command output

final_report()
Return a string describing the total running time, as well as including the final report from the embedded HostPool and TaskQueue objects.

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.

class `pylauncher.`**`ListCommandlineGenerator`**`(`***kwargs*`)`
A generator from an explicit list of commandlines.

•cores is 1 by default, other constants allowed.

class `pylauncher.`**`CountedCommandGenerator`**`(`***kwargs*`)`
This class is only for the unit tests, it produces a string of ‘echo 0’, ‘echo 1’ et cetera commands.

Parameters

- **nmax** – (keyword, default=-1) maximum number of commands to generate, negative for no maximum
- **command** – (keyword, default==‘‘echo‘‘) the command that will do the counting; sometimes it’s a good idea to replace this with `/bin/true`
- **catch** – (keyword, default None) file where to catch output

class `pylauncher.`**`SleepCommandGenerator`**`(`***kwargs*`)`
Generator of commandlines ‘echo 0 ; sleep trand’, ‘echo 1 ; sleep trand’ where the sleep is a random amount.

Parameters

- **tmax** – (keyword, default 5) maximum sleep time
- **tmin** – (keyword, default 1) minimum sleep time
- **barrier** – (keyword, default 0) if >0, insert a barrier statement every that many lines

class `pylauncher.`**`RandomSleepTask`**`(`***kwargs*`)`
Make a task that sleeps for a random amount of time. This is for use in many many unit tests.

Parameters

- **taskid** – unique identifier (keyword, required)
- **t** – maximum running time (keyword, optional; default=10)
- **tmin** – minimum running time (keyword, optional; default=1)
- **completion** – Completion object (keyword, optional; if you leave this unspecified, the next two parameters become relevant
- **stampdir** – name of the directory where to leave the stamp file (optional, default=current dir)
- **stamproot** – filename stemp for the stamp file (optional, default=”sleepexpire”)

class pylauncher.**OneNodePool**(*node*, ***kwargs*)
This class is mostly for testing: it allows for a node to function as a host pool so that one can start a task on it.

pylauncher.**MakeRandomCommandFile**(*fn*, *ncommand*, ***kwargs*)
Make file with commandlines and occasional comments and blanks.

Parameters **cores** – (keyword, default=1) corecount, if this is 1 we put nothing in the file, larger values and "file" (for random) go into the file

pylauncher.**MakeRandomSleepFile**(*fn*, *ncommand*, ***kwargs*)
make file with sleep commandlines and occasional comments and blanks

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX

p

INDEX

R

S

T

U

W