Python Cluster Documentation

Release 0.6.1

Michael Dacre <mike.dacre@gmail.com>

Aug 24, 2016

Submit jobs to slurm or torque, or with multiprocessing.

Author	Michael D Dacre <mike.dacre@gmail.com>
License	MIT License, property of Stanford, use as you wish
Version	0.6.1b

Allows easy job submission with *dependency tracking and queue waiting* with either torque, slurm, or locally with the multiprocessing module. It uses simple techiques to avoid overwhelming the queue and to catch bugs (e.g. queue stalling) on the fly.

NOTE: While this software is extremely powerful for pure python-based cluster job submission, snakemake is possibly a better choice for very large workflows.

Contents:

CHAPTER

ONE

BASIC USAGE

1.1 Installation

This module will work with Python 2.7+ on Linux systems.

To install, use the standard python method:

```
git clone https://github.com/MikeDacre/python-cluster
cd python-cluster
python ./setup.py install --user
```

In general you want user level install even if you have sudo access, as most cluster environments share /home/<user> across the cluster, making this module available everywhere.

Note: While the name is *python-cluster* you will import it just as *cluster*:

```
import cluster
```

1.1.1 Prerequisites

The only external module that I use in this software is dill. It isn't 100% required but it makes function submission much more stable.

If you choose to use dill, it must be installed cluster wide.

1.1.2 Function Submission

In order to submit functions to the cluster, this module must import them on the compute node. This means that all of your python modules must be available on every compute node. To avoid pain and debugging, you can do this manually by running this on your loggin node:

```
freeze --local | grep -v '^\-e' | cut -d = -f 1 > module_list.txt
```

And then on the compute nodes:

```
cat module_list.txt | xargs pip install --user
```

This will ensure that all of your modules are installed globally.

In general it is a good idea to install modules as *–user* with pip to avoid this issue.

1.2 Simple Usage

1.2.1 Setting Environment

To set the environement, set queue.MODE to one of ['torque', 'slurm', 'local'], or run get_cluster_environment().

1.2.2 Simple Job Submission

At its simplest, this module can be used by just executing submit(<command>), where command is a function or system command/shell script. The module will autodetect the cluster, generate an intuitive name, run the job, and write all outputs to files in the current directory. These can be cleaned with clean_dir().

To run with dependency tracking, run:

```
import cluster
job  = cluster.submit(<command1>)
job2 = cluster.submit(<command2>, dependencies=job1)
exitcode, stdout, stderr = job2.get()  # Will block until job completes
```

1.2.3 Functions

The submit function works well with python functions as well as with shell scripts and shell commands.

However, in order for this to work, *cluster* ends up importing your original script file on the nodes. This means that all code in your file will be executed, so anything that isn't a function or class must be protected with an:

```
if __name__ == '__main__':
```

protecting statment.

If you do not do this you can end up with multi-submission and infinate recursion, which could mess up your jobs or just crash the job, but either way, it won't be good.

1.2.4 File Submission

If you want to just submit a file, that can be done like this:

```
from cluster import submit_file
submit_file('/path/to/script', dependencies=[7, 9])
```

This will return the job number and will enter the job into the queue as dependant on jobs 007 and 009. The dependencies can be omitted.

1.2.5 The Job Class

The core of this submission system is a *Job* class, this class allows easy job handling and debugging. All of the above commands work well with the Job class also, but more fine grained control is possible. For example:

```
my_job = """#!/bin/bash
parallel /usr/bin/parser {} ::: folder/*.txt
for i in folder/*.txt; do
    echo $i >> my_output.txt
```

```
    echo job_$i done!
fi"""
job = cluster.Job(my_job, cores=16)
job.submit()
job.wait()
print(job.stdout)
if job.exitcode != 0:
    print(job.stderr)
```

More is also possible, for a full description, see the API documentation here: Job Documentation

1.3 Queue Management

This module provides simple queue management functions

To generate a queue object, do the following:

```
import cluster
q = cluster.Queue(user='self')
```

This will give you a simple queue object containg a list of jobs that belong to you. If you do not provide user, all jobs are included for all users. You can provide *qtype* to explicitly force the queue object to contain jobs from one queing system (e.g. local or torque).

To get a dictionary of all jobs, running jobs, queued jobs, and complete jobs, use:

```
q.jobs
q.running
q.complete
q.queued
```

Every job has a number of attributes, including owner, nodes, cores, memory.

1.4 Advanced Usage

1.4.1 Keyword Arguments

To make submission easier, this module defines a number of keyword arguments in the options.py file that can be used for all submission and Job() functions. These include things like 'cores' and 'nodes' and 'mem'.

The following is a complete list of arguments that can be used in this version:

```
Used in every mode::
cores:      Number of cores to use for the job
            Type: int; Default: 1
modules:    Modules to load with the `module load` command
            Type: list; Default: None
filedir:    Folder to write cluster files to, must be accessible to the compute
            nodes.
            Type: str; Default: .
dir:        The working directory for the job
            Type: str; Default: path argument
suffix:     A suffix to append to job files (e.g. job.suffix.qsub)
            Type: str; Default: cluster
```

```
outfile:    File to write STDOUT to
            Type: str; Default: None
errfile:    File to write STDERR to
            Type: str; Default: None

Used for function calls::
imports:    Imports to be used in function calls (e.g. sys, os) if not provided,
            defaults to all current imports, which may not work if you use complex
            imports. The list can include the import call, or just be a name, e.g.
            ['from os import path', 'sys']
            Type: list; Default: None

Used only in local mode::
threads:    Number of threads to use on the local machine
            Type: int; Default: 8

Options that work in both slurm and torque::
nodes:      Number of nodes to request
            Type: int; Default: 1
features:   A comma-separated list of node features to require
            Type: list; Default: None
time:       Walltime in HH:MM:SS
            Type: str; Default: 12:00:00
mem:        Memory to use in MB (e.g. 4000)
            Type: ['int', 'str']; Default: 4000
partition:  The partition/queue to run in (e.g. local/batch)
            Type: str; Default: None
account:    Account to be charged
            Type: str; Default: None
export:     Comma separated list of environmental variables to export
            Type: str; Default: None

Used for slurm only::
begin:      Start after this much time
            Type: str; Default: None
```

In addition some synonyms are allowed:

```
cpus:                           cores
memory:                         mem
queue:                          partition
depend, dependencies, dependency: depends
```

Note: Type is enforced, any provided argument must match that python type (automatic conversion is attempted), the default is just a recommendation and is not currently used. These arguments are passed like regular arguments to the submission and Job() functions, eg:

```
Job(nodes=1, cores=4, mem='20MB')
```

This will be interpretted correctly on any system. If torque or slurm are not available, any cluster arguments will be ignored. The module will attempt to honor the cores request, but if it exceeds the maximum number of cores on the local machine, then the request will be trimmed accordingly (i.e. a 50 core request will become 8 cores on an 8 core machine).

Adding your own keywords

There are many more options available for torque and slurm, to add your own, edit the options.py file, and look for CLUSTER_OPTS (or TORQUE/SLURM if your keyword option is only availble on one system). Add your option

using the same format as is present in that file. The format is:

```
('name', {'slurm': '--option-str={}', 'torque': '--torque-option={}',
          'help': 'This is an option!', 'type': str, 'default': None})
```

You can also add list options, but they must include 'sjoin' and 'tjoin' keys to define how to merge the list for slurm and torque, or you must write custom option handling code in `cluster.options.options_to_string()`. For an excellent example of both approaches included in a single option, see the 'features' keyword above.

I happily accept pull requests for new option additions (any any other improvements for that matter).

1.4.2 Profiles and the Config File

To avoid having to enter all keyword arguments every time, profiles can be used. These profiles can store any of the above keywords and drastically simplify submission. For example:

```
job = submit(my_function, profile='large')
```

Instead of:

```
job = submit(my_funtion, nodes=2, cores=16, mem='64GB', partition='bigjobs',
             features=['highmem'], export='PYTHONPATH')
```

These profiles are saved in a config file at ~/.python-cluster and can be editted in that file directly, or using the below functions. To edit them in the file directly, you must make sure that the section is labelled 'prof_<name>' where <name> is whatever you want it to be called. e.g.:

```
[prof_default]
nodes = 1
cores = 16
time = 24:00:00
mem = 32000
```

Note: a default profile must always exist, it will be added back if it does not exist.

Alternatively, the functions `cluster.config_file.set_profile()` and `cluster.config_file.get_profile()` can be used:

```
cluster.config_file.set_profile('small', {'nodes': 1, 'cores': 1,
                                          'mem': '2GB'})
cluster.config_file.get_profile('small')
```

To see all profiles run:

```
config_file.get_profile()
```

Other options are defined in the config file, including the maximum number of jobs in the queue, the time to sleep between submissions, and other options. To see these run:

```
cluster.config_file.get_option()
```

You can set options with:

```
cluster.config_file.set_option()
```

The defaults can be directly edited in `config_file.py`, they are clearly documented.

1.4.3 Job Files

All jobs write out a job file before submission, even though this is not necessary (or useful) with multiprocessing. In local mode, this is a *.cluster* file, in slurm is is a *.cluster.sbatch* and a *.cluster.script* file, in torque it is a *.cluster.qsub* file. 'cluster' is set by the suffix keyword, and can be overridden.

To change the directory these files are written to, use the 'filedir' keyword argument to Job or submit.

NOTE: This directory *must* be accessible to the compute nodes!!!

All jobs are assigned a name that is used to generate the output files, including STDOUT and STDERR files. The default name for the out files is STDOUT: name.cluster.out and STDERR: name.cluster.err. These can be overwridden with keyword arguments.

All Job objects have a `clean()` method that will delete any left over files. In addition there is a clean_job_files script that will delete all files made by this package in any given directory. Be very careful with the script though, it can clobber a lot of work all at once if it is used wrong.

1.4.4 Dependecy Tracking

Dependency tracking is supported in all modes. Local mode uses a unique queueing system that works similarly to torque and slurm and which is defined in jobqueue.py.

To use dependency tracking in any mode pass a list of job ids to submit or submit_file with the *dependencies* keyword argument.

1.4.5 Logging

I use a custion logging script called logme to log errors. To get verbose output, set logme.MIN_LEVEL to 'debug'. To reduce output, set logme.MIN_LEVEL to 'warn'.

1.5 Code Overview

There are two important classes for interaction with the batch system: Job and Queue. The essential flow of a job submission is:

```
job = Job(command/function, arguments, name)
job.write()  # Writes the job submission files
job.submit() # Submits the job
job.wait()   # Waits for the job to complete
job.stdout   # Prints the output from the job
job.clean()  # Delete all of the files written
```

You can also wait for many jobs with the Queue class:

```
q = Queue(user='self')
q.wait([job1, job2])
```

The jobs in this case can be either a Job class or a job number.

CHAPTER

TWO

SCRIPTS

This package contains a few little helper scripts to make your life easier. These are not required in order to use the cluster library.

2.1 clean_job_files

```
usage: clean_job_files [-h] [-d DIR] [-s SUFFIX] [-q {torque,slurm,local}]
                       [-n] [-v VERBOSE]

Clean all intermediate files created by the cluster module from this dir.

================================================================================

        AUTHOR: Michael D Dacre, mike.dacre@gmail.com
  ORGANIZATION: Stanford University
       LICENSE: MIT License, property of Stanford, use as you wish
       CREATED: 2016-34-15 15:06
 Last modified: 2016-06-16 10:42

   DESCRIPTION: Uses the cluster.job.clean_dir() function

       CAUTION: The clean() function will delete **EVERY** file with
                extensions matching those these::
                    .<suffix>.err
                    .<suffix>.out
                    .<suffix>.sbatch & .cluster.script for slurm mode
                    .<suffix>.qsub for torque mode
                    .<suffix> for local mode
                    _func.<suffix>.py
                    _func.<suffix>.py.pickle.in
                    _func.<suffix>.py.pickle.out

================================================================================

optional arguments:
  -h, --help            show this help message and exit
  -d DIR, --dir DIR     Directory to clean
  -s SUFFIX, --suffix SUFFIX
                        Directory to clean
  -q {torque,slurm,local}, --qtype {torque,slurm,local}
                        Limit deletions to this qtype
  -n, --no-confirm      Do not confirm before deleting (for scripts)
```

```
  -v VERBOSE, --verbose VERBOSE
                        Show debug information
```

- search

CHAPTER

THREE

API DOCUMENTATION

The following documentation is primarily built from the docstrings of the actual source code and can be considered an API reference.

3.1 Queueing

The most import thing is the *Queue()* class which does most of the queue mangement. In addition, *get_cluster_environment()* attempts to autodetect the cluster type (torque, slurm, normal) and sets the global cluster type for the whole file. Finally, the *wait()* function accepts a list of jobs and will block until those jobs are complete.

The Queue class is actually a wrapper for a few simple queue parsers, these call *qstat -x* or *squeue* and *sacct* to get job information, and return a simple tuple of that data with the following members:

```
job_id, name, userid, partition, state, node-list, node-count, cpu-per-node, exit-code
```

The Queue class then converts this information into a Queue.QueueJob object and adds it to the internal *jobs* dictionary within the Queue class. This list is now the basis for all of the other functionality encoded by the Queue class. It can be accessed directly, or sliced by accessing the *completed*, *queued*, and *running* attributes of the Queue class, these are used to simply divided up the jobs dictionary to make finding information easy.

class cluster.**Queue**(*user=None*, *qtype=None*)
Handle torque, slurm, or multiprocessing objects.

All methods are transparent and work the same regardless of queue type.

Queue.queue is a list of jobs in the queue. For torque and slurm, this is all jobs in the queue for the specified user. In local mode, it is all jobs added to the pool, Queue must be notified of these by adding the job object to the queue directly with add().

exception **QueueError**
Simple Exception wrapper.

class Queue.**QueueJob**
Only used for torque/slurm jobs in the queue.

Queue.**can_submit**(*max_queue_len=None*)
Return True if R/Q jobs are less than max_queue_len.

If max_queue_len is None, default from config is used.

Queue.**update**()
Refresh the list of jobs from the server, limit queries.

Queue.**wait**(*jobs*)
Block until all jobs in jobs are complete.

Note: update time is dependant upon the queue_update parameter in your ~/.cluster file.

In addition, wait() will not return until between 1 and 3 seconds after a job has completed, irrespective of queue_update time. This allows time for any copy operations to complete after the job exits.

Jobs A job or list of jobs to check. Can be one of: Job or multiprocessing.pool.ApplyResult objects, job ID (int/str), or a object or a list/tuple of multiple Jobs or job IDs.

Returns True on success False or nothing on failure.

Queue.**wait_to_submit**(*max_queue_len=None*)
Wait until R/Q jobs are less than max_queue_len.

If max_queue_len is None, default from config is used.

cluster.queue.**get_cluster_environment**()
Detect the local cluster environment and set MODE globally.

Uses which to search for sbatch first, then qsub. If neither is found, MODE is set to local.

Returns MODE variable ('torque', 'slurm', or 'local')

cluster.queue.**wait**(*jobs*)
Wait for jobs to finish.

Jobs A single job or list of jobs to wait for. With torque or slurm, these should be job IDs, with local mode, these are multiprocessing job objects (returned by submit())

cluster.queue.**check_queue**(*qtype=None*)
Raise exception if MODE is incorrect.

cluster.queue.**queue_parser**(*qtype=None*, *user=None*)
Call either torque or slurm qtype parsers depending on qtype.

Qtype Either 'torque' or 'slurm', defaults to current MODE

User optional user name to pass to queue to filter queue with

Yields job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

cluster.queue.**torque_queue_parser**(*user=None*)
Iterator for torque queues.

Use the *qstat -x* command to get an XML queue for compatibility.

User optional user name to pass to qstat to filter queue with

Yields job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

numcpus is currently always 1 as most torque queues treat every core as a node.

cluster.queue.**slurm_queue_parser**(*user=None*)
Iterator for slurm queues.

Use the *squeue -O* command to get standard data across implementation, supplement this data with the results of *sacct*. sacct returns data only for the current user but retains a much longer job history. Only jobs not returned by squeue are added with sacct, and they are added to *the end* of the returned queue, i.e. *out of order with respect to the actual queue*.

User optional user name to pass to squeue to filter queue with

Yields job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

3.2 Job Management

Job management is handeled by the *Job()* class, full instructions on using this class are above, in particular review the 'Keyword Arguments' section above.

The methods of this class are exposed by a few functions that aim to make job submission easier. The foremost of these is *submit()* which can take as little as a single command and execute it. *make_job()* and *make_job_file()* work similarly but just return a Job object, or write the file and then return the Job object respectively. *clean()* takes a list of Job objects and runs their internal *clean()* methods, deleting all written files.

There are two additional functions that are completely independent of the Job object: *submit_file()* and *clean_dir()*. *submit_file()* uses similar methods to the Job class to submit a job to the cluster, but it does not involve the job class at all, instead just submitting an already created job file. It can do dependency tracking in the same way as a job file, but that is all. *clean_dir()* uses the file naming convention established in the Job class (and defined separately here) to delete all files in a directory that look like they could be made by this module. It has an autoconfirm feature that can be activated to avoid accidental clobbering.

class `cluster.`**Job**(*command*, *args=None*, *name=None*, *path=None*, *qtype=None*, *profile=None*, ***kwds*)

Information about a single job on the cluster.

Holds information about submit time, number of cores, the job script, and more.

submit() will submit the job if it is ready wait() will block until the job is done get() will block until the job is done and then unpickle a stored

output (if defined) and return the contents

clean() will delete any files created by this object

Printing the class will display detailed job information.

Both wait() and get() will update the queue every two seconds and add queue information to the job as they go.

If the job disappears from the queue with no information, it will be listed as 'completed'.

All jobs have a .submission attribute, which is a Script object containing the submission script for the job and the file name, plus a 'written' bool that checks if the file exists.

In addition, SLURM jobs have a .exec_script attribute, which is a Script object containing the shell command to run. This difference is due to the fact that some SLURM systems execute multiple lines of the submission file at the same time.

Finally, if the job command is a function, this object will also contain a .function attribute, which contains the script to run the function.

clean(*delete_outputs=False*)

Delete all scripts created by this module, if they were written.

If delete_outputs is True, also delete the stdout and stderr files, but get their contents first.

get()

Block until job completed and return exit_code, stdout, stderr.

get_exitcode(*update=True*)

Try to get the exitcode.

get_stderr(*update=True*)

Read stdout file if exists and set self.stdout, return it.

get_stdout(*update=True*)

Read stdout file if exists and set self.stdout, return it.

submit(*max_queue_len=None*)
Submit this job.

Max_queue_len if specified (or in defaults), then this method will block until the queue is open enough to allow submission.

To disable max_queue_len, set it to 0. None will allow override by the default settings in the config file, and any positive integer will be interpretted to be the maximum queue length.

Returns self

update()
Update status from the queue.

update_queue_info()
Set queue_info from the queue even if done.

wait()
Block until job completes.

write(*overwrite=True*)
Write all scripts.

cluster.**submit**(*command*, *args=None*, *name=None*, *path=None*, *qtype=None*, *profile=None*, ***kwargs*)
Submit a script to the cluster.

Command The command or function to execute.

Args Optional arguments to add to command, particularly useful for functions.

Name The name of the job.

Path Where to create the script, if None, current dir used.

Qtype 'torque', 'slurm', or 'normal'

Profile The name of a profile saved in the config_file

Kwargs Keyword arguments to control job options

There are many keyword arguments available for cluster job submission. These vary somewhat by queue type. For info run:

cluster.options.option_help()

Returns Job object

cluster.job.**submit_file**(*script_file*, *dependencies=None*, *threads=None*, *qtype=None*)
Submit a job file to the cluster.

If qtype or queue.MODE is torque, qsub is used; if it is slurm, sbatch is used; if it is local, the file is executed with subprocess.

This function is independent of the Job object and just submits a file.

Dependencies A job number or list of job numbers. In slurm: *–dependency=afterok:* is used For torque: *-W depend=afterok:* is used

Threads Total number of threads to use at a time, defaults to all. ONLY USED IN LOCAL MODE

Returns job number for torque or slurm multiprocessing job object for local mode

cluster.job.**make_job**(*command*, *args=None*, *name=None*, *path=None*, *qtype=None*, *profile=None*, ***kwargs*)
Make a job file compatible with the chosen cluster.

If mode is local, this is just a simple shell script.

Command The command or function to execute.

Args Optional arguments to add to command, particularly useful for functions.

Name The name of the job.

Path Where to create the script, if None, current dir used.

Qtype 'torque', 'slurm', or 'normal'

Profile The name of a profile saved in the config_file

There are many keyword arguments available for cluster job submission. These vary somewhat by queue type. For info run:

cluster.options.option_help()

Returns A Job object

cluster.job.**make_job_file**(*command*, *args=None*, *name=None*, *path=None*, *qtype=None*, *profile=None*, ***kwargs*)

Make a job file compatible with the chosen cluster.

If mode is local, this is just a simple shell script.

Command The command or function to execute.

Args Optional arguments to add to command, particularly useful for functions.

Name The name of the job.

Path Where to create the script, if None, current dir used.

Qtype 'torque', 'slurm', or 'normal'

Profile The name of a profile saved in the config_file

Kwargs Keyword arguments to control job options

There are many keyword arguments available for cluster job submission. These vary somewhat by queue type. For info run:

cluster.options.option_help()

Returns Path to job script

cluster.job.**clean**(*jobs*)

Delete all files in jobs list or single Job object.

cluster.job.**clean_dir**(*directory='.'*, *suffix='cluster'*, *qtype=None*, *confirm=False*)

Delete all files made by this module in directory.

CAUTION: The clean() function will delete EVERY file with

extensions matching those these:: .<suffix>.err .<suffix>.out .<suffix>.sbatch & .<suffix>.script for slurm mode .<suffix>.qsub for torque mode .<suffix> for local mode _func.<suffix>.py _func.<suffix>.py.pickle.in _func.<suffix>.py.pickle.out

Directory The directory to run in, defaults to the current directory.

Qtype Only run on files of this qtype

Confirm Ask the user before deleting the files

Returns A set of deleted files

3.3 Options

All keyword arguments are defined in dictionaries in the *options.py* file, alongside function to manage those dictionaries. Of particular importance is *option_help()*, which can display all of the keyword arguments as a string or a table. *check_arguments()* checks a dictionary to make sure that the arguments are allowed (i.e. definied), it is called on all keyword arguments in the package.

The way that option handling works in general, is that all hardcoded keyword arguments must contain a dictionary entry for 'torque' and 'slurm', as well as a type declaration. If the type is NoneType, then the option is assumed to be a boolean option. If it has a type though, *check_argument()* attmepts to cast the type and specific idiosyncracies are handled in this step, e.g. memory is converted into an integer of MB. Once the arguments are sanitized *format()* is called on the string held in either the 'torque' or the 'slurm' values, and the formatted string is then used as an option. If the type is a list/tuple, the 'sjoin' and 'tjoin' dictionary keys must exist, and are used to handle joining.

The following two functions are used to manage this formatting step.

option_to_string() will take an option/value pair and return an appropriate string that can be used in the current queue mode. If the option is not implemented in the current mode, a debug message is printed to the console and an empty string is returned.

options_to_string() is a wrapper around *option_to_string()* and can handle a whole dictionary of arguments, it explicitly handle arguments that cannot be managed using a simple string format.

`cluster.options.`**`option_help`**(*qtype=None*, *mode='string'*)
Print a sting to stdout displaying information on all options.

Qtype If provided only return info on that queue type.

Mode string: Return a formatted string print: Print the string to stdout table: Return a table of lists

`cluster.options.`**`check_arguments`**(*kwargs*)
Make sure all keywords are allowed.

Raises OptionsError on error, returns sanitized dictionary on success.

Note: Checks in SYNONYMS if argument is not recognized, raises OptionsError if it is not found there either.

`cluster.options.`**`options_to_string`**(*option_dict*, *qtype=None*)
Return a multi-line string for slurm or torque job submission.

Option_dict Dict in format {option: value} where value can be None. If value is None, default used.

Qtype 'torque', 'slurm', or 'local': override queue.MODE

`cluster.options.`**`option_to_string`**(*option*, *value=None*, *qtype=None*)
Return a string with an appropriate flag for slurm or torque.

Option An allowed option definied in options.all_options

Value A value for that option if required (if None, default used)

Qtype 'torque', 'slurm', or 'local': override queue.MODE

3.4 Config File

Profiles are combinations of keyword arguments that can be called in any of the submission functions. They are handled in the *config_file.py* file which just adds an abstraction layer on top of the builtin python ConfigParser script.

The config file also contains other options that can be managed with the *get_option()* and *set_option()* functions. Profiles are wrapped in a *Profile()* class to make attribute access easy, but they are fundamentally just dictionaries of keyword arguments. They can be created with *cluster.config_file.Profile({kewywds})* and then written to a file with that class' *write()* method. The easiest way to interact with profiles is with the *get_profile()* and *set_profile()* functions. These make it very easy to go from a dictionary of keywords to a profile.

Profiles can then be called with the *profile=* keyword in any submission function or Job class.

class `cluster.config_file.`**`Profile`**(*name*, *kwds*)
A job submission profile. Just a thin wrapper around a dict.

`write`()
Write self to config file.

`cluster.config_file.`**`get_option`**(*section=None*, *key=None*, *default=None*)
Get a single key or section.

Section The config section to use (e.g. queue, prof)

Key The config key to get (e.g. 'max_jobs')

Default If the key does not exist, create it with this default value.

Returns None if key does not exist.

`cluster.config_file.`**`set_option`**(*section*, *key*, *value*)
Write a config key to the config file.

`cluster.config_file.`**`set_profile`**(*name*, *args*)
Write profile to config file.

`cluster.config_file.`**`get_profile`**(*profile=None*)
Return a profile if it exists, if None, return all profiles.

`cluster.config_file.`**`delete`**(*section*, *key=None*)
Delete a config item.

If key is not provided deletes whole section.

`cluster.config_file.`**`get_config`**()
Load defaults from file.

3.5 Local Queue Implementation

The local queue implementation is based on the multiprocessing library and is not intended to be used directly, it should always be used via the Job class because it is somewhat tempramental. The essential idea behind it is that we can have one JobQueue class that is bound to the parent process, it exclusively manages a single child thread that runs the *job_runner()* function. The two process communicate using a *multiprocessing.Queue* object, and pass *cluster.jobqueue.Job* objects back and forth between them.

The Job objects (different from the Job objects in *job.py*) contain information about the task to run, including the number of cores required. The job runner manages a pool of *multiprocessing.Pool* tasks directly, and keeps the total running cores below the total allowed (default is the system max, can be set with the threads keyword). It backfills smaller jobs and holds on to larger jobs until there is enough space free.

This is close to what torque and slurm do, but vastly more crude. It serves as a stopgap to allow parallel software written for compute clusters to run on a single machine in a similar fashion, without the need for a pipeline alteration. The reason I have reimplemented a process pool is that I need dependency tracking and I need to allow some processes to run on multiple cores (e.g. 6 of the available 24 on the machine).

The *job_runner()* and *Job* objects should never be accessed except by the JobQueue. Only one JobQueue should run at a time (not enforced), and by default it is bound to *cluster.jobqueue.JQUEUE*. That is the interface used by all other parts of this package.

class `cluster.jobqueue.`**JobQueue**(*cores=None*)
Monitor and submit multiprocessing.Pool jobs with dependencies.

add(*function*, *args=None*, *kwargs=None*, *dependencies=None*, *cores=1*)
Add function to local job queue.

Function A function object. To run a command, use the run.cmd function here.

Args A tuple of args to submit to the function.

Kwargs A dict of keyword arguments to submit to the function.

Dependencies A list of job IDs that this job will depend on.

Cores The number of threads required by this job.

Returns A job ID

get(*job*)
Return the output of a single job

restart(*force=False*)
Kill the job queue and restart it.

update()
Get fresh job info from the runner.

wait(*jobs=None*)
Wait for a list of jobs, all jobs are the default.

class `cluster.jobqueue.`**Job**(*function*, *args=None*, *kwargs=None*, *depends=None*, *cores=1*)
An object to pass arguments to the runner.

`cluster.jobqueue.`**job_runner**(*jobqueue*, *outputs*, *cores=None*, *jobno=None*)
Run jobs with dependency tracking.

Must be run as a separate multiprocessing.Process to function correctly.

Jobqueue A multiprocessing.Queue object into which Job objects must be added. The function continually searches this Queue for new jobs. Note, function must be a function call, it cannot be anything else. function is the only required argument, the rest are optional. tuples are required.

Outputs A multiprocessing.Queue object that will take outputs. A dictionary of job objects will be output here with the format:: {job_no => Job} **NOTE**: function return must be picklable otherwise this will raise an exception when it is put into the Queue object.

Cores Number of cores to use in the multiprocessing pool. Defaults to all.

Jobno What number to start counting jobs from, default 1.

3.6 Logme

This is a package I wrote myself and keep using because I like it. It provides syslog style leveled logging (e.g. 'debug'->'info'->'warn'->'error'->'critical') and it implements colors and timestamped messages.

The minimum print level can be set module wide at runtime by changing *cluster.logme.MIN_LEVEL*.

`cluster.logme.`**`log`**(*message*, *level='info'*, *logfile=None*, *also_write=None*, *min_level=None*, *kind=None*)

Print a string to logfile.

Message The message to print.

Logfile Optional file to log to, defaults to STDERR. Can provide a logging object

Level 'debug'|'info'|'warn'|'error'|'normal' Will only print if level > MIN_LEVEL

'debug':	'<timestamp> DEBUG –> '
'info':	'<timestamp> INFO –> '
'warn':	'<timestamp> WARNING –> '
'error':	'<timestamp> ERROR –> '
'critical':	'<timestamp> CRITICAL –> '

Also_write 'stdout': print to STDOUT also. 'stderr': print to STDERR also. These only have an effect if the output is not already set to the same device.

Min_level Retained for backwards compatibility, min_level should be set using the logme.MIN_LEVEL constant.

Kind synonym for level, kept to retain backwards compatibility

3.7 Other Functions

Some other wrapper functions are defined in *run.py*, these are just little useful knick-knacks that make function submission and queue management possible.

`cluster.run.`**`cmd`**(*command*, *args=None*, *stdout=None*, *stderr=None*, *tries=1*)

Run command and return status, output, stderr.

Command Path to executable.

Args Tuple of arguments.

Stdout File or open file like object to write STDOUT to.

Stderr File or open file like object to write STDERR to.

Tries Int: Number of times to try to execute 1+

Returns exit_code, STDOUT, STDERR

`cluster.run.`**`which`**(*program*)

Replicate the UNIX which command.

Taken verbatim from: stackoverflow.com/questions/377017/test-if-executable-exists-in-python

Program Name of executable to test.

Returns Path to the program or None on failure.

cluster.run.**open_zipped**(*infile*, *mode='r'*)
Open a regular, gzipped, or bz2 file.

Returns text mode file handle.

If infile is a file handle or text device, it is returned without changes.

cluster.run.**split_file**(*infile*, *parts*, *outpath=''*, *keep_header=True*)
Split a file in parts parts and return a list of paths.

NOTE: Linux specific (uses wc).

Outpath The directory to save the split files.

Keep_header Add the header line to the top of every file.

3.8 Indices and tables

- genindex
- search
- modindex
- search