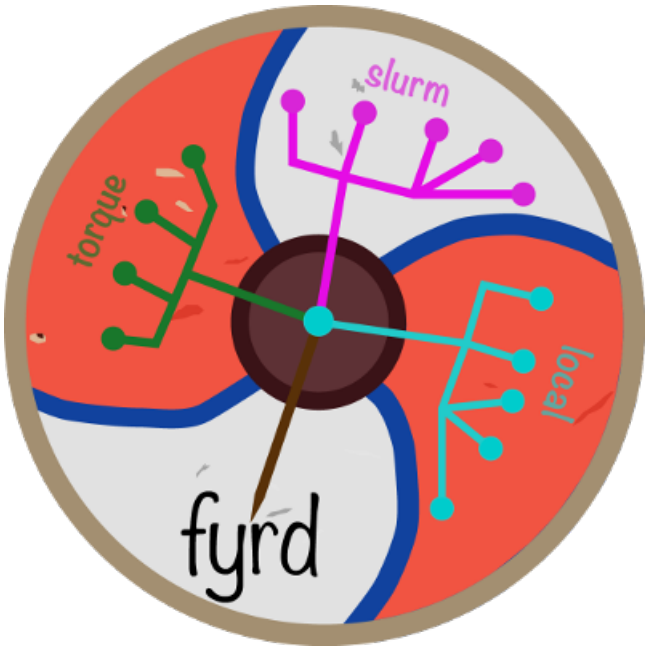


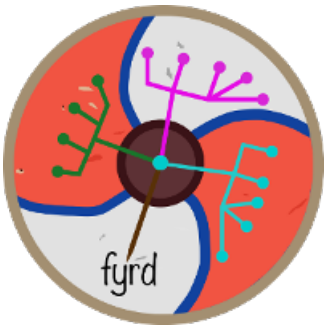
Fyrd Documentation

Release 0.6.1b9

Mike Dacre <mike.dacre@gmail.com>

Mar 07, 2018

Contents



Python job submission on torque and slurm clusters with dependency tracking. Pronounced 'feared' (sort of), Anglo-Saxon for an army, particularly an army of freemen (an army of nodes). The logo is based on a Saxon shield commonly used in these fyrds. This library used to be known as 'Python Cluster'.

Allows simple job submission with *dependency tracking and queue waiting* on either torque, slurm, or locally with the multiprocessing module. It uses simple techniques to avoid overwhelming the queue and to catch bugs on the fly.

It is routinely tested on Mac OS and Linux with slurm and torque clusters, or in the absence of a cluster, on Python versions 2.7.10, 2.7.11, 2.7.12, 3.3.0, 3.4.0, 3.5.2, 3.6.2, and 3.7-dev. The full test suite is available in the *tests* folder.

For complete documentation see the documentation site and the Fyrd.pdf document in this repository.

Contents:

CHAPTER 1

Getting Started

1.1 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 fyrd
job  = fyrd.submit(<command1>)
job2 = fyrd.submit(<command2>, depends=job1)
out  = job2.get()  # Will block until job completes
```

The *submit()* function is actually just a wrapper for the Job class. The same behavior as above can be obtained by initializing a *Job* object directly:

```
import fyrd
job  = fyrd.Job(<command1>)
job.submit()
job2 = fyrd.Job(<command2>, depends=job1).submit()
out  = job2.get()  # Will block until job completes
```

Note that as shown above, the submit method returns the *Job* object, so it can be called on job initialization. Also note that the object returned by calling the *submit()* function (as in the first example) is also a *Job* object, so these two examples can be used fully interchangeably.

1.2 Functions

The submit function works well with python functions as well as with shell scripts and shell commands, in fact, this is the most powerful feature of this package. For example:

```
import fyrd
def raise_me(something, power=2):
    return something**power
outs = []
if __name__ == '__main__':
    for i in range(80):
        outs.append(fyrd.submit(my_function, (i,), {'power': 2},
                                mem='10MB', time='00:00:30'))
    final_sum = 0
    for i in outs:
        final_sum += i.get()
    print(final_sum)
```

By default this will submit every instance as a job on the cluster, then get the results and clean up all intermediate files, and the code will work identically on a Mac with no cluster access, a slurm cluster, or a torque cluster, with no need to change syntax.

This is very powerful when combined with simple methods that split files or large python classes, to make this kind of work easier, a number of simple functions are provided in the helpers module, to learn more about that, review the Advanced Usage section of this documentation.

Function submission works equally well for submitting methods, however the original class object will not be updated, the method return value will be accurate, but any changes the method makes to *self* will not be returned from the cluster and will be lost.

1.2.1 Possible Infinate Recursion Error

Warning: in order for function submission to work, *fyrd* 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 statement.

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

This isn't true when submitting from an interactive session such as ipython or jupyter.

1.3 File Submission

If you want to just submit a job file that has already been created, either by this software or any other method, that can be done like this:

```
from fyrd 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 7 and 9. The dependencies can be omitted.

1.4 Keywords

The *Job* class, and therefore every submission script, accepts a large number of keyword arguments and synonyms to make job submission easy. Some good examples:

- cores

- mem (or memory)
- time (or walltime)
- partition (or queue)

The synonyms are provided to make submission easy for anyone familiar with the arguments used by either torque or slurm. For example:

```
job = Job('zcat huge_file | parse_file', cores=1, mem='30GB', time='24:00:00')
job = Job(my_parallel_function, cores=28, mem=12000, queue='high_mem')
for i in huge_list:
    out.append(submit(parser_function, i, cores=1, mem='1GB', partition='small'))
job = Job('ls /etc')
```

As you can see, optional keywords make submission very easy and flexible. The whole point of this software it to make working with a remote cluster in python as easy as possible.

For a full list of keyword arguments see the Keyword Arguments section of the documentation.

All options are defined in the fyrd.options module. If you want extra options, just submit an issue or add them yourself and send me a pull request.

1.5 Profiles

One of the issues with using keyword options is the nuisance of having to type them every time. More importantly, when writing code to work on any cluster one has to deal with heterogeneity between the clusters, such as the number of cores available on each node, or the name of the submission queue.

Because of this, *fyrd* makes use of profiles that bundle keyword arguments and give them a name, so that cluster submission can look like this:

```
job = Job('zcat huge_file | parse_file', profile='large')
job = Job(my_parallel_function, cores=28, profile='high_mem')
```

These profiles are defined in *~/.fyrd/profiles.txt* by default and have the following syntax:

```
[large]
partition = normal
cores = 16
nodes = 1
time = 24:00:00
mem = 32000
```

This means that you can now do this:

```
Job(my_function, profile='large')
```

You can create as many of these as you like.

While you can edit the profile file directly to add and edit profile, it is easier and more stable to use the console script:

..code:: shell

    fyrd profile list fyrd profile edit large time:02-00:00:00 mem=64GB fyrd profile edit DEFAULT partition:normal fyrd profile remove-option DEFAULT cores fyrd profile add silly cores:92 mem:1MB fyrd profile delete silly

The advantage of using the console script is that argument parsing is done on editing the profiles, so any errors are caught at that time. If you edit the file manually, then any mistakes will cause an Exception to be raised when you try to submit a job.

If no arguments are given the default profile (called 'DEFAULT' in the config file) is used.

Note: any arguments in the DEFAULT profile are available in all profiles if the are not manually overridden there. The DEFAULT profile cannot be deleted. It is a good place to put the name of the default queue.

CHAPTER 2

Configuration

Many program parameters can be set in the config file, found by default at *~/.fyrd/config.txt*.

This file has three sections with the following defaults:

[queue]:

```
max_jobs (int):     sets the maximum number of running jobs before
                    submission will pause and wait for the queue to empty
sleep_len (int):    sets the amount of time the program will wait between
                    submission attempts
queue_update (int): sets the amount of time between refreshes of the queue.
res_time (int):     Time in seconds to wait if a job is in an uncertain
                    state, usually preempted or suspended. These jobs often
                    resolve into running or completed again after some time
                    so it makes sense to wait a bit, but not forever. The
                    default is 45 minutes: 2700 seconds.
queue_type (str):   the type of queue to use, one of 'torque', 'slurm',
                    'local', 'auto'. Default is auto to auto-detect the
                    queue.
```

[jobs]:

```
clean_files (bool):    means that by default files will be deleted when job
                       completes
clean_outputs (bool):  is the same but for output files (they are saved
                       first)
file_block_time (int): Max amount of time to block after job completes in
                       the queue while waiting for output files to appear.
                       Some queues can take a long time to copy files under
                       load, so it is worth setting this high, it won't
                       block unless the files do not appear.
filepath (str):        Path to write all temp and output files by default,
                       must be globally cluster accessible. Note: this is
                       *not* the runtime path, just where files are written
                       to.
suffix (str):          The suffix to use when writing scripts and output
```

```
                        files
auto_submit (bool):     If wait() or get() are called prior to submission,
                        auto-submit the job. Otherwise throws an error and
                        returns None
generic_python (bool): Use /usr/bin/env python instead of the current
                        executable, not advised, but sometimes necessary.
profile_file (str):     the config file where profiles are defined.
```

[jobqueue]:

```
Sets options for the local queue system, will be removed in the future in
favor of database.

jobno (int):  The current job number for the local queue, auto-increments
              with every submission.
```

Example file:

```
[queue]
res_time = 2700
queue_type = auto
sleep_len = 1
queue_update = 2
max_jobs = 1000
bool = True

[jobs]
suffix = cluster
file_block_time = 12
filepath = None
clean_outputs = False
auto_submit = True
profile_file = /Users/dacre/.fyrd/profiles.txt
clean_files = True
generic_python = False

[jobqueue]
jobno = 9
```

The config is managed by fyrd/conf.py and enforces a minimum set of entries. If the config does not exist or any entries are missing, they will be created on the fly using the defaults defined in the defaults.

CHAPTER 3

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

> depends clean_files clean_outputs cores modules syspaths scriptpath outpath runpath suffix outfile errfile imports threads nodes features qos time mem partition account export begin

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).

3.1 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 `fyrd.options.options_to_string()`. For an excellent example of both approaches included in a single option, see the 'features' keyword above.

CHAPTER 4

Console Scripts

This software is primarily intended to be a library, however some management tasks are just easier from the console. For that reason, *fyrd* has a frontend console script that makes tasks such as managing the local config and profiles trivial, it also has modes to inspect the queue easily, and to wait for jobs from the console, as well as to clean the working directory.

4.1 fyrd

This software has uses a subcommand system to separate modes, and has six modes:

- *config* — show and edit the contents of the config file
- *profile* - inspect and manage cluster profiles
- *keywords* - print a list of current keyword arguments with descriptions for each
- *queue* - show running jobs, makes filtering jobs very easy
- *wait* - wait for a list of jobs
- *clean* - clean all script and output files in the given directory

Several of the commands have aliases (*conf* and *prof* being the two main ones)

4.1.1 Examples

```
fyrd prof list
fyrd prof add large cores:92 mem:200GB partition:high_mem time:00:06:00
```

```
fyrd queue  # Shows all of your current jobs
fyrd queue -a # Shows all users jobs
fyrd queue -p long -u bob dylan # Show all jobs owned by bob and dylan in the long␣
↪queue
```

```
fyrd wait 19872 19876
fyrd wait -u john

# Will block until all of bob's jobs in the long queue finish
fyrd queue -p long -u bob -l | xargs fyrd wait
```

```
fyrd clean
```

4.1.2 All Options

fyrd:

```
usage: fyrd [-h] [-v] {conf,prof,keywords,queue,wait,clean} ...

Manage fyrd config, profiles, and queue.

============   ======================================
Author         Michael D Dacre <mike.dacre@gmail.com>
Organization   Stanford University
License        MIT License, use as you wish
Version        0.6.2b9
============   ======================================

positional arguments:
  {conf,prof,keywords,queue,wait,clean}
    conf (config)       View and manage the config
    prof (profile)      Manage profiles
    keywords (keys, options)
                        Print available keyword arguments.
    queue (q)           Search the queue
    wait                Wait for jobs
    clean               Clean up a job directory

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Show debug outputs
```

fyrd conf:

```
usage: fyrd conf [-h] {show,list,help,update,alter,init} ...

This script allows display and management of the fyrd config file found
here: /home/dacre/.fyrd/config.txt.

positional arguments:
  {show,list,help,update,alter,init}
    show (list)         Show current config
    help                Show info on every config option
    update (alter)      Update the config
    init                Interactively initialize the config

optional arguments:
  -h, --help            show this help message and exit

Show usage::
    fyrd conf show [-s <section>]
```

```
Update usage::
    fyrd conf update <section> <option> <value>

*Values can only be altered one at a time*

To create a new config from scratch interactively::
    fyrd conf init [--defaults]
```

fyrd prof:

```
usage: fyrd prof [-h]
                 {show,list,add,new,update,alter,edit,remove-option,del-option,delete,
↪del}
                 ...

Fyrd jobs use keyword arguments to run (for a complete list run this script
with the keywords command). These keywords can be bundled into profiles, which
are kept in /home/dacre/.fyrd/profiles.txt. This file can be edited directly or␣
↪manipulated here.

positional arguments:
  {show,list,add,new,update,alter,edit,remove-option,del-option,delete,del}
    show (list)         Print current profiles
    add (new)           Add a new profile
    update (alter, edit)
                        Update an existing profile
    remove-option (del-option)
                        Remove a profile option
    delete (del)        Delete an existing profile

optional arguments:
  -h, --help            show this help message and exit

Show::
    fyrd prof show

Delete::
    fyrd prof delete <name>

Update::
    fyrd prof update <name> <options>

Add::
    fyrd prof add <name> <options>

<options>:
    The options arguments must be in the following format::
        opt:val opt2:val2 opt3:val3

Note: the DEFAULT profile is special and cannot be deleted, deleting it will
cause it to be instantly recreated with the default values. Values from this
profile will be available in EVERY other profile if they are not overriden
there. i.e. if DEFAULT contains `partition=normal`, if 'long' does not have
a 'partition' option, it will default to 'normal'.

To reset the profile to defaults, just delete the file and run this script
again.
```

fyrd keywords:

```
usage: fyrd keywords [-h] [-t | -s | -l]

optional arguments:
  -h, --help          show this help message and exit
  -t, --table         Print keywords as a table
  -s, --split-tables  Print keywords as multiple tables
  -l, --list          Print a list of keywords only
```

fyrd queue:

```
usage: fyrd queue [-h] [-u  [...] | -a] [-p  [...]] [-r | -q | -d | -b]
                  [-l | -c]

Check the local queue, similar to squeue or qstat but simpler, good for
quickly checking the queue.

By default it searches only your own jobs, pass '--all-users' or
'--users <user> [<user2>...]' to change that behavior.

To just list jobs with some basic info, run with no arguments.

optional arguments:
  -h, --help            show this help message and exit

queue filtering:
  -u  [ ...], --users  [ ...]
                        Limit to these users
  -a, --all-users       Display jobs for all users
  -p  [ ...], --partitions  [ ...]
                        Limit to these partitions (queues)

queue state filtering:
  -r, --running         Show only running jobs
  -q, --queued          Show only queued jobs
  -d, --done            Show only completed jobs
  -b, --bad             Show only completed jobs

display options:
  -l, --list            Print job numbers only, works well with xargs
  -c, --count           Print job count only
```

fyrd wait:

```
usage: fyrd wait [-h] [-u USERS] [jobs [jobs ...]]

Wait on a list of jobs, block until they complete.

positional arguments:
  jobs                  Job list to wait for

optional arguments:
  -h, --help            show this help message and exit
  -u USERS, --users USERS
                        A comma-separated list of users to wait for
```

fyrd clean:

```
usage: fyrd clean [-h] [-o] [-s SUFFIX] [-q {torque,slurm,local}] [-n] [dir]

Clean all intermediate files created by the cluster module.

If not directory is passed, the default if either scriptpath or outpath are
set in the config is to clean files in those locations is to clean those
directories. If they are not set, the default is the current directory.

By default, outputs are not cleaned, to clean them too, pass '-o'

Caution:
    The clean() function will delete **EVERY** file with
    extensions matching those these::

        .<suffix>.err
        .<suffix>.out
        .<suffix>.sbatch & .fyrd.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

positional arguments:
  dir                   Directory to clean (optional)

optional arguments:
  -h, --help            show this help message and exit
  -o, --outputs         Clean output files too
  -s SUFFIX, --suffix SUFFIX
                        Suffix to use for cleaning
  -q {torque,slurm,local}, --qtype {torque,slurm,local}
                        Limit deletions to this qtype
  -n, --no-confirm      Do not confirm before deleting (for scripts)
```

4.2 Aliases

Several shell scripts are provided in *bin/* to provide shortcuts to the *fyrd* subcommands:

- *my-queue* (or *myq*): *fyrd queue*
- *clean-job-files*: *fyrd clean*
- *monitor-jobs*: *fyrd wait*
- *cluster-keywords*: *fyrd keywords*

CHAPTER 5

Advanced Usage

Most of the important functionality is covered in the Getting Started section, and full details on the library are available in the API Reference section. This section just provides some extra information on Job and Queue management, and importantly introduces some of the higher-level options available through the helpers.

5.1 The Job Class

The core of this submission system is the *Job* class, this class builds a job using keyword arguments and profile parsing. The bulk of this is done at class initialization and is covered in the getting started section of this documentation and on job submission with the *submit()* method. There are several other features of this class to be aware of though.

5.1.1 Script File Handling

Torque and slurm both require submission scripts to work. In the future these will be stored by fyrd in a database and submitted from memory, but for now they are written to disk.

The creation and writing of these scripts is handled by the Script and Function classes in the fyrd.submission_scripts module. These classes pass keywords to the options_to_string() function of the options method, which converts them into a submission string compatible with the active cluster. These are then written to a script for submission to the cluster.

The *Function* class has some additional functionality to allow easy submission of functions to the cluster. It tries to build a list of all possible modules that the function could need and adds import statements to all of them to the function submission script. It then pickles the submitted function and arguments to a pickle file on the disk, and writes a python script to the same directory.

This python script unpickles the function and arguments and runs them, pickling either the result or and exception, if one is raised, to the disc on completion. The submission script calls this python script on the cluster nodes.

The script and output files are written to the path defined by the *.filepath* attribute of the *Job* class, which is set using the 'filepath' keyword argument. If not set, this directory defaults to the directory set in the filepath section of the config file or the current working directory. Note that this path is independent of the *.runpath* attibute, which is where the code will actually run, and also defaults to the current working directory.

5.1.2 Job Output Handling and Retrieval

The correct way to get outputs from within a python session is to call the *.get()* method of the *Job* class. This first calls the *.wait()* method, which blocks until job completion, and then the *.fetch_outputs()* method which *calls get_output, get_stdout, and get_stderr, which save all function outputs, STDOUT, and STDERR to the class*. This means that outputs can be accessed using the following *Job* class attributes:

- *.output* — the function output for functions or STDOUT for scripts
- *.stdout* — the STDOUT for the script submission (always present)
- *.stderr* — the STDERR for the script submission (always present)

This makes job output retrieval very easy, but it is sometimes not what you want, particularly if outputs are very large (they get loaded into memory).

The *wait()* method will not save any outputs. In addition *get()* can be with the *save=False* argument, which means it will fetch the output (or STDOUT) only, but will not write them to the class itself.

Note: By default, *get()* also deletes all script and output files. This is generally a good thing as it keeps the working directory clean, but it isn't always what you want. To prevent outputs from being deleted, pass *delete_outfiles=False* to *get()*, or alternatively set the *.clean_outputs* attribute to *False* prior to running *get()*. To prevent the cleaning of any files, including the script files, pass *cleanup=False* or set *.clean_files* to *False*.

clean_files and *clean_outputs* can also be set globally in the config file.

5.2 Job Files

All jobs write out a job file before submission, even though this is not necessary (or useful) with multiprocessing. This will change in a future version.

To ensure files are obviously produced by this package and that files are unique the file format is name.number.random_string.suffix.extension. These are:

name: Defined by the *name=* argument or guessed from the function/script number: A number count of the total jobs with the same name already queued random_string: An 8-character random string suffix: A string defined in the config file, default 'cluster' extension: An obvious extension such as '.sbatch' or '.qsub'

To change the directory these files are written to, set the filedir item in the config file or use the 'filedir' keyword argument to Job or submit.

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

It is sometimes useful to set the filedir setting in the config to a single directory accessible cluster-wide. This avoids cluttering the current directory, particularly as outputs can be retrieved so easily from within python. If you are going to do this set the 'clean_files' and 'clean_outfiles' arguments in the config file to avoid cluttering the directory.

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.

5.3 Helpers

The fyrd.helpers module defines several simple functions that allow more complex job handling.

The helpers are all high level functions that are not required for the library but make difficult jobs easy to assist in the goal of trivially easy cluster submission.

5.3.1 Pandas

The most important function in *fyrd.helpers* is *parapply()*, which allows the user to submit a *pandas.DataFrame.apply* method to the cluster in parallel by splitting the DataFrame, submitting jobs, and then recombining the DataFrame at the end, all without leaving any temp files behind. e.g.:

```
df = pandas.read_csv('my_huge_file.txt')
df = fyrd.helpers.parapply(100, df, long_running_function, profile='fast')
```

That command will split the dataframe into 100 pieces, submit each to the cluster as a different job with the profile 'fast', and then recombine them into a single DataFrame again at the end.

parapply_summary behaves similarly but assumes that the function summarizes the data rather than returning a DataFrame of the same size. It thus runs the function on the resulting DataFrame also, allowing all dfs to be merged. e.g.:

```
df = fyrd.helpers.parapply_summary(df, numpy.mean)
```

This will return just the mean of all the numeric columns, *parapply* would return a DataFrame with duplicates for every submitted job.

5.3.2 Running on a split file

The splitrun function behaves similarly to the *parapply()* function, with the exception that it works on a filesystem file instead, which it splits into pieces. It then runs your job on all of the pieces and attempts to recombine them, deleting the intermediate files as it goes.

If you specify an output file, the outputs are merged and places into that file, otherwise, if the outputs are a string (always true for scripts), the function returns a merged string. If the outputs are not strings, then the function just returns a list out outputs that you will have to combine yourself.

The key to this function is that if the job is a script, it must at a minimum contain '{file}' where the file argument goes, and if the job is a function it must contain and argument or keyword argument that matches '<file>'.

If you expect the job to have and output, you must provide the *outfile=* argument too, and be sure that '{outfile}' is present in the script, if a script, or '<outfile>' is in either args or kwargs if a function.

In addition, you should pass *inheader=True* if the input file has a header line, and *outheader=True* if the same is true for the outfile. It is very important to pass these arguments, because they both will strip the top line from a file if True. Importantly, if *inheader* is *True* on a file without a header, the top line will appear at the top of every broken up file.

Examples:

```
script = """my_long_script --in {file} --out {outfile}"""
outfile = fyrd.helpers.splitrun(
    100, 'huge_file.txt', script, name='my_job', profile='long',
    outfile='output.txt', inheader=True, outheader=True
)
```

```
output = fyrd.helpers.splitrun(
    100, 'huge_file.txt', function, args=('<file>',), name='my_job',
    profile='long', outfile='output.txt', inheader=True, outheader=True
)
```

5.4 Queue Management

Queue handling is done by the Queue class in the fyrd.queue module. This class calls the fyrd.queue.queue_parser iterator which in turn calls either fyrd.queue.torque_queue_parser or fyrd.queue.slurm_queue_parser depending on the detected cluster environment (set by *fyrd.queue.QUEUE_MODE* and overridden by the 'queue_type' config option if desired (not necessary, queue type is auto-detected)).

These iterators return the following information from the queue:

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

These pieces of information are used to create QueueJob objects for every job, which are stored in the *Queue.jobs* attribute (a dictionary). The *Queue* class provides several properties, attributes, and methods to allow easy filtering of these jobs.

Most important is the *QueueJob.state* attribute, which holds information on the current state of that job. To get a list of all states in the queue, call the *Queue.job_states* property, which will return a list of states in the queue. All of these states are also attributes of the *Queue* class, for example:

```
fyrd.Queue.completed
```

returns all completed jobs in the queue as a dictionary (a filtered copy of the *.jobs* attribute).

Note: torque states are auto-converted to slurm states, as slurm states are easier to read. e.g. 'C' becomes 'completed'.

The most useful method of *Queue* is *wait()*, it will take a list of job numbers or *Job* objects and wait until all of them are complete. This method is called by the *Job.wait()* method, and can be called directly to wait for an arbitrary number of jobs.

To wait for all jobs from a given user, you can do this:

```
q = fyrd.Queue()
q.wait(q.get_user_jobs(['bob', 'fred']))
```

This task can also be accomplished with the console application:

```
fyrd wait <job> [<job>...]
fyrd wait -u bob fred
```

The method can actually be simply accessed as a function instead of needing the *Queue* class:

```
fyrd.wait([1,2,3])
```

To generate a *Queue* object, do the following:

```
import fyrd
q = fyrd.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 is a *QueueJob* class and has a number of attributes, including owner, nodes, cores, memory.

5.5 Config

Many of the important options used by this software are set in a config file and can be managed on the console with *fyrd conf* . . . .

For full information see the Configuration section of this documentation.

5.6 Logging

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

CHAPTER 6

API Reference

6.1 fyrd.queue

The core class in this file is the *Queue()* class which does most of the queue management. 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 relies on a few simple queue parsers defined by the *torque_queue_parser* and *slurm_queue_parser* functions. These call *qstat -x* or *squeue* and *sacct* to get job information, and yield 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 divide up the jobs dictionary to make finding information easy.

6.1.1 fyrd.queue.Queue

Methods

6.1.2 fyrd.queue functions

parsers

utilities

6.2 fyrd.job

Job management is handled by the *Job()* class. This is a very large class that defines all the methods required to build and submit a job to the cluster.

It accepts keyword arguments defined in *fyrd.options* on initialization, which are then fleshed out using profile information from the config files defined by *fyrd.conf*.

The primary argument on initialization is the function or script to submit.

Examples:

```
Job('ls -lah | grep myfile')
Job(print, ('hi',))
Job('echo hostname', profile='tiny')
Job(huge_function, args=(1,2) kwargs={'hi': 'there'},
    profile='long', cores=28, mem='200GB')
```

6.2.1 fyrd.job.Job

Methods

6.3 fyrd.submission_scripts

This module defines to classes that are used to build the actual jobs for submission, including writing the files. *Function* is actually a child class of *Script*.

6.4 fyrd.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. defined), it is called on all keyword arguments in the package.

To see keywords, run *fyrd keywords* from the console or *fyrd.option_help()* from a python session.

The way that option handling works in general, is that all hard-coded 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()* attempts to cast the type and specific idiosyncrasies 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.

6.5 fyrd.conf

fyrd.conf handles the config (*~/.fyrd/config.txt*) file and the profiles (*~/.fyrd/profiles.txt*) file.

Profiles are combinations of keyword arguments that can be called in any of the submission functions. Both the config and profiles are just ConfigParser objects, *conf.py* merely adds an abstraction layer on top of this to maintain the integrity of the files.

6.5.1 config

The config has three sections (and no defaults):

- queue — sets options for handling the queue
- jobs — sets options for submitting jobs
- jobqueue — local option handling, will be removed in the future

For a complete reference, see the config documentation : Configuration

Options can be managed with the *get_option()* and *set_option()* functions, but it is actually easier to use the console script:

```
fyrd conf list
fyrd conf edit max_jobs 3000
```

6.5.2 profiles

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.conf.Profile(name, {keywds})* and then written to a file with the *write()* method.

The easiest way to interact with profiles is not with class but with the *get_profile()*, *set_profile()*, and *del_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.

As with the config, profile management is the easiest and most stable when using the console script:

```
fyrd profile list
fyrd profile add very_long walltime:120:00:00
fyrd profile edit default partition:normal cores:4 mem:10GB
fyrd profile delete small
```

fyrd.conf.Profile

6.6 fyrd.helpers

The helpers are all high level functions that are not required for the library but make difficult jobs easy to assist in the goal of trivially easy cluster submission.

The functions in *fyrd.basic* below are different in that they provide simple job submission and management, while the functions in *fyrd.helpers* allow the submission of many jobs.

6.7 fyrd.basic

This module holds high level functions to make job submission easy, allowing the user to skip multiple steps and to avoid using the *Job* class directly.

submit(), *make_job()*, and *make_job_file()* all create *Job* objects in the background and allow users to submit jobs. All of these functions accept the exact same arguments as the *Job* class does, and all of them return a *Job* object.

submit_file() is different, it simply submits a pre-formed job file, either one that has been written by this software or by any other method. The function makes no attempt to fix arguments to allow submission on multiple clusters, it just submits the file.

clean() takes a list of job objects and runs the *clean()* method on all of them, *clean_dir()* uses known directory and suffix information to clean out all job files from any directory.

6.8 fyrd.local

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 temperamental. 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 *fyrd.local.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 *fyrd.local.JQUEUE*. That is the interface used by all other parts of this package.

6.8.1 fyrd.local.JobQueue

6.8.2 fyrd.local.job_runner

6.9 fyrd.run

6.10 fyrd.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*.