gridengineapp

Release 19.3.0

Drew Dolgert

Aug 04, 2019

CONTENTS:

CHAPTER

ONE

ABOUT GRIDENGINEAPP

This library has two parts:

1. A set of Python functions to use qsub and qacct from Grid Engine.
2. An application framework for writing applications that run under Grid Engine and are testable on local machines.

1.1 Qsub and Qstat

These functions are responsible for formatting command arguments according to Grid Engine's rules for where dashes and commas go. They are also responsible for parsing XML output in order to produce dictionaries of the resulting data. They don't limit functionality, and they don't simplify it. You can do that in a layer on top of these tools, if you want.

1.2 Application Framework

An application within this framework presents its work as a graph of functions. The framework is responsible for executing subgraphs of functions using

1. Grid Engine,
2. Multiple subprocesses, respecting a memory limit on the machine,
3. A single process, or
4. Within unit tests.

The ability to run individual functions within unit tests enables a kind of testing called thread testing, which is a high-level test, from invocation to result.

With this framework, there is no need to make a separate `main()` for each UGE job. The level of abstraction is at a function level, where each function declares its memory, time, and CPU requirements. The framework will automatically create a `main()` with which to invoke individual jobs, if one isn't supplied.

1.3 Future Functionality

Because this framework is built around a graph of jobs, the graph structure is available for building tools to run an application within a larger framework and tools to check the status of jobs. While qstat tells you how each individual job is running, the graph of jobs tells you what that implies about whether other jobs have already run or are going to run.

CHAPTER

TWO

GRIDENGINEAPP TUTORIAL

This follows the `location_app` example in the examples directory.

2.1 The Application

We are going to build a graph of Jobs, where a Job is a class that holds code to run in a UGE job on the cluster. For instance, our code could use the locations hierarchy, in which case we would build the graph as follows:

```
import networkx as nx
import db_queries

def location_graph(gbd_round_id, location_set_version_id):
    location_df = db_queries.get_location_metadata(
        gbd_round_id=gbd_round_id, location_set_version_id=location_set_version_id)
    G = nx.DiGraph()
    G.add_nodes_from([
        (int(row.location_id), row._asdict())
        for row in location_df.itertuples()])
    # GBD encodes the global node as having itself as a parent.
    G.add_edges_from([
        (int(row.parent_id), int(row.location_id))
        for row in location_df[location_df.location_id != 1].itertuples()])
    return G
```

The NetworkX Library is a convenient way to build directed acyclic graphs. It has a good NetworkX Tutorial.

The main code required to use this framework is the Application class. It has the following parts:

```
class GridExample:
    """The class name will be used as the base name for cluster job names."""
    def __init__(self):
        """An init that takes no arguments, because it will be
        called for the children."""
        self.location_set_version_id = None
        self.gbd_round_id = None

    def add_arguments(parser):
        """The same argument parser is used for both the initial
        call to run all the jobs and each time a job is run.
        These arguments both decide the shape of the graph and,
        later, the exact job to run within that graph."""
        parser.add_argument("--location-set-version-id", type=int,
                            default=429)
```

(continues on next page)

(continued from previous page)

```
        parser.add_argument("--gbd-round-id", type=int, default=6)
        parser.add_argument("--job-idx", type=int, help="The job ID")

    def job_id_to_arguments(job_id):
        """Makes a list of arguments to add to a command line in
        order to run a specific job."""
        return ["--job-id", str(job_id)]

    def job_identifiers(self, args):
        """Given arguments, return the jobs specified.
        This could be used to subset the whole graph, for instance
        to run a slice through the locations from global to
        most-detailed locations."""
        if args.job_id:
            return [args.job_id]
        else:
            return self.job_graph().nodes

    def initialize(self, args):
        """Read the arguments in order to know what to do."""
        self.location_set_version_id = args.location_set_version_id
        self.gbd_round_id = args.gbd_round_id

    def job_graph(self):
        """Make the whole job graph and return it."""
        return location_graph(
            self.gbd_round_id, self.location_set_version_id)

    def job(self, location_id):
        """Make a job from its ID.
        We haven't said what this class is yet."""
        return LocationJob(location_id)
```

Most of that work is to define the job graph or parse arguments to specify parts of the job graph. The work we do is in a `Job` class.

2.2 The Job Class

A Job itself inherits from a base class, `Job`. The most important parts of the Job are its run method and outputs. The run method does the work, and the framework uses the list of outputs to check whether the job completed. The class's initialization is done by the Application class, so we can pass in whatever helps initialize the Job:

```
class LocationJob(Job):
    def __init__(self, location_id, gbd_round_id):
        super().__init__()
        out_file = Path("/data/home") / f"{location_id}.hdf"
        self.outputs["paf"] = FileEntity(out_file)

    @property
    def resources(self):
        """These can be computed from arguments to init."""
        return dict(
            memory_gigabytes=1,
            threads=1,
```

(continues on next page)

(continued from previous page)

```
            run_time_minutes=1,
        )

    def run(self):
        pass  # Make that output file.
```

The outputs are a dictionary of objects that check whether a file is in a state where we consider this job to have done its work. The `FileEntity` checks that the file exists. The `PandasEntity` can check that particular data sets exist in the file.

The list of outputs enables the framework to know which jobs have definitely completed. We can also define `self.inputs`, which enable the framework to set up mock inputs, so that we can test individual jobs in a larger graph, without first running the whole graph.

2.3 The Child Job Main

Finally, at the bottom of the file, under the Application, we put a snippet that is the `main()` for the jobs:

```
if __name__ == "__main__":
    app = GridExample()
    exit(entry(app))
```

This framework looks for this specifically in the same file as the application class. If it doesn't find one, it will attempt to make its own version of a `main()`.

2.4 Running

2.4.1 Debug One Job Locally

In order to start one job locally, you can run it with, in this case:

```
$ python location_app.py --job-idx 1 --pdb
```

The `--pdb` will make the job drop into an interactive debugger when it encounters an exception.

2.4.2 Check Outputs Match Inputs

One way to see that the graph is well-formed is to supply both an input list and an output list to each job and run the whole of it using an automatic mocking:

```
$ python location_app.py --mock
```

Because there is no `--job-idx` argument, it will try to run the whole graph. Because there is no `--grid-engine` argument, it will run it as functions within this process, and the `--mock` argument tells it to skip the real `run()` method and, instead, use the `self.outputs` to generate fake files. The `self.inputs` check that the correct fake files exist when a Job first starts.

2.4.3 Run on the Cluster

On the cluster, start the whole thing with the command:

```
$ python location_app.py --grid-engine --project proj_forecasting
```

It will launch jobs and return immediately. Those jobs will all have the same name, something like `location_app23f824_37`, where the first part is the application name, and then there are six hexadecimal characters that are (probably) unique for this job, and then an identifier for the particular location running.

The framework looks at each Job's run times in order to determine which queue to use.

2.4.4 Smaller Run on One Node

If there is less work to do, it may be easier to run this application interactively, using all the cores of a node. In that case, login to a node, allocating, maybe 16 GB of memory. Then run:

```
$ python location_app.py --memory-limit 16
```

Then it will run all jobs as subprocesses, ensuring it doesn't exceed that memory limit in GB.

CHAPTER

THREE

QSUB TUTORIAL

Grid Engine is an abstraction of the Unix process. It makes a Unix process on a remote machine look like a Unix process on a local machine by giving the user access to its standard in and standard out streams, its Job ID (equivalent to process ID), and allowing the user to start, pause, and kill these jobs.

This library gives a program a tested interface to using Grid Engine commands. It reduces the probability of mistyping while making every last feature available.

3.1 Main Success Sequence

Let's look at a typical sequence of events. Suppose we want to submit 300 jobs, for every cause of death. Start by making a template that describes all of the jobs:

```
from gridengineapp import QsubTemplate
template = QsubTemplate()
template.P = "proj_forecasting"
template.q = "all.q"
template.l["fthread"] = "30"
template.l["m_mem_free"] = "5G"
template.l["h_rt"] = "00:05:00"
```

Now we want to submit a bunch of jobs:

```
from gridengineapp import qsub
job_id = list()
for cause in range(300):
    job_id.append(qsub(template, ["/ihme/code/borlaug/run.sh", cause]))
```

That gives you a bunch of job IDs. If the list of job IDs is small, you might find their status by passing them into `qstat`:

```
from gridengineapp import qstat
jobs = qstat(job_list=job_id)
for job in jobs:
    if "error" in job.status:
        print(f"Job {job.name} in error")
```

The job status is a set of strings, where the strings can be "idle", "held", "migrating", "queued", "running", "suspended", "transfering", "deleted", "waiting", "exiting", "written", "error", or "waiting4osjid".

3.2 Submit a Restartable Job

There are a few reasons a job might want to ask the scheduler to rerun it on a different node. For instance, the node where it starts could be missing mount points so that files aren't found. The node could have an out-of-date version of an important piece of software. In these cases, you can start your code with a check of the node and, if it looks bad, restart the job:

```
from gridengineapp import restart_count

# This command increments the restart count, stored in file
# in the logging directory.
restarted = restart_count()
if not Path("/ihme/forecasting").exists():
    if restarted < 3:
        LOGGER.error("Node missing data mount point. Restarting")
        exit(99)
    else:
        LOGGER.error("Node missing data mount point. Restart limit reached.")
        exit(1)
```

CHAPTER

FOUR

TASK ARRAYS

The basic unit of work is a Job. Task arrays are copies of a job. In Grid Engine, a task array is specified using:

```
qsub -P project -t 1-10 script.sh
```

Here, there are 10 tasks with task ids from 1 to 10. This could also be from 1 to 1 and would still be a task array. That's even a nice way to test a task array.

In a Grid Engine job, task IDs are specified by an environment variable, `SGE_TASK_ID`, which will be the integer number, or, if this isn't a task array, the variable will be undefined or the string value "undefined."

How do we handle this for our Grid Engine Application? Make a Job that clones to become a task array:

```
class MyJob:
    def __init__(self, task_id=None):
        super().__init__()
        self.task_id = task_id
        if self.task_id:
            self.outputs[f"out{task_id}"] = FileEntity("out.hdf")

    def clone_task(self, task_id):
        return MyJob(task_id)

    ... and the usual methods follow ...
```

When the task runs, it will be cloned with the task id. This way, we can instantiate multiple tasks at the same time. This task id is guaranteed to be greater than zero.

CHAPTER

FIVE

APPLICATION REQUIREMENTS

5.1 Problem to Solve

Forecasting builds a lot of applications that run on the cluster. The main challenge for our cluster is that cluster nodes often fail to run a job that would otherwise run. When we write code, we get distracted by this circumstance. This package focuses on making code testable first, while also dealing with node misconfiguration as a secondary problem. Our code has more bugs than the nodes do, so making work testable is the primary problem to solve.

These applications have to be

1. modifiable
2. reliable
3. usable

in that order. They don't need a lot of crazy capability. They don't need much security.

There are a ton of frameworks to run applications. We've had trouble finding ones that let us make code that runs under Grid Engine and is can still be run in a test harness.

5.2 Stakeholders

- Stein Emil Vollset - P.I., who wants things timely.
- Amanda Smith - Project officer, who wants things usable.
- Serkan - who wants us not to abuse the cluster.

5.3 Core Capabilities

1. Run Python code with memory, CPU, and time requirements on the Fair cluster.
2. Run the same Python code under pytest both on the cluster and on desktops.

5.4 Scenario Summaries

- S1. Define a Python function for every country and run it for all countries on the cluster.
- S2. Modeler runs a single country under a debugger.

- S3. Modeler changes the code, deletes a subset of the files, and tells the program to redo all steps that depend on the deleted files.
- S4. Define a hierarchical set of jobs, with an aggregation step at the end. Run a mock version of this, in a single Unix process, in order to verify that each step creates files needed by the next step, so they all connect correctly.
- S5. Rerun one job in the middle of a graph of jobs in order to run it under a debugger.
- S6. If a job is slow, and the modeler cannot ssh into the node, then the modeler asks Grid Engine to delete that job. The modeler then asks the application to resubmit that job and all jobs that depend on it.
- S7. If a job runs into an error that comes from node misconfiguration, such as inability to reach a file on a shared filesystem, then it can raise an exception that results in rerunning the job.

5.5 Feature List

- F1. Run an application under Grid Engine.
- F2. Run an application as a single local process.
- F3. Run an application as multiple Unix processes on a local machine.
- F4. Continue a application, which means looking at which outputs it hasn't made and starting the jobs that make those outputs.
- F5. Rerun a single job within an application, if the application detects a problem with its local node, only for Grid Engine.
- F6. Wait synchronously for all jobs within an application to complete.
- F7. Configure logging to go to the known logging directory.
- F8. Run a single job within an application as a local process under a debugger.
- F9. Run all jobs in an application that depend on a particular job.
- F10. Test that an application is well-formed before trying to run it.
- F11. During asynchronous execution, tell the client the name of the job and the job id of the last job to execute.
- F12. Collect metrics on a job as a function of the parameters.
- F13. Check common node misconfiguration problems, such as missing filesystem mounts.
- F14. Launch jobs without requiring the client to write a Python main function. Write that file for them, so that they can call a job using the class of the application.
- F15. Use task arrays to run jobs that have multiplicity.
- F16. Allow the developer to configure Grid Engine job templates and logging.

CHAPTER

SIX

TESTING PLAN

6.1 Running Tests

Install the `cascade_config` repository from the local versioning system before running tests, so that it knows local directories:

```
cd tests
pytest --fair
```

This will run tests using up to 4GB of memory and including job submission to the cluster queues. When testing on a laptop or on Github, exclude the `--fair` flag.

6.2 Systems Under Test

This software is a single Python package. It gets installed into the same Python virtual environment, or Conda environment, that the client application uses. That's not complicated, but there can be many concurrent Unix processes that run within this same environment, on the same or different machines.

- For a Grid Engine job, there is a parent invocation of the `gridengineapp.entry()` method, done on a Grid Engine submission host. This asks Grid Engine to launch multiple jobs, each of which is a Python main that calls `gridengineapp.entry()` on remote hosts.
- For a multiprocess job, there is a parent invocation of `gridengineapp.entry()` on a local host. This then uses subprocesses to run multiple Unix processes, each of which calls `gridengineapp.entry()` to run a single Job.
- For a whithin-process job, the parent invocation then runs each job as a function, in order, on the local host. The arguments to this function need to be similar to what a Grid Engine run would see.

We can think of this as six kinds of systems under test, for the parent invocation and the child job runs of Grid Engine, multiprocess, and within-process.

6.3 Ways to Partition Testing

By command-line argument We can think of combinations of command-line arguments as a way to partition testing at a high level. These arguments are defined in the `entry()` function. There are sets of arguments:

- Grid engine arguments
- Multiprocessing arguments
- Graph sub-selection arguments

- Arguments about how to run individual jobs (mock, pdb, logging).

By Scenario We could make several applications and run them through the steps described in the requirements scenarios. So make a new application and run it through different things a user would do.

CHAPTER

SEVEN

LIBRARY REFERENCE

gridengineapp.**qstat**(*effective_user=None*, *job_list=None*)

Get status of all jobs in the job_list belonging to the given user:

```
import getpass
user = getpass.getuser()
job_info = qstat(user, "dm_38044_*")
```

Parameters

- **effective_user** (*str*) – The user ID.
- **job_list** (*str*) – Can be model version IDs, or a job name, or a job name with a wildcard. See `man sge_types`.

Returns Information about the jobs.

Return type List[FlyWeightJob]

gridengineapp.**qstat_short**(*effective_user=None*)

Calling qstat without -j gets a much smaller result that just has job information.

Parameters **effective_user** (*str*) – Request qstat for this user's jobs. Default is to use the current user.

Returns A list of jobs with less information than what `qstat -j` shows.

Return type List[MiteWeightJob]

gridengineapp.**qsub**(*template*, *command*)

Runs a qsub command with a template. By using the template, as described below, this function makes it easier to create a default set of qsub settings and overwrite them, job by job, without doing string manipulation.

We can either try to put a super-thoughtful interface on qsub, or we let the user manage its arguments. This focuses on making it a little easier to manage arguments with the template.

Parameters

- **template** – Suitable for *template_to_args*.
- **command** (*List[str]*) – A list of strings to pass to qsub.

Returns The model version ID. It's a str because it isn't an int. Can you add 37 to it? No. Is it ordered? That's not guaranteed. Does it sometimes have a ".1" at the end? Yes. That makes it a string.

Return type str

The template argument is a dictionary where each entry corresponds to an argument to `qsub`. Here are the rules:

- If the argument is a flag with no argument, set `template[flag] = None`.
- If the argument is a flag with a true or false, set `template[flag] = True`, or `False`.
- If the argument is a comma-separated list, set the value to a list, `template["dc"] = ["LD_LIBRARY_PATH", "CC"]`.
- If the argument is a set of key-value pairs, set the value to a dictionary, `template["l"] = dict(m_mem_free="16G", fthreads=16)`.

gridengineapp.**qsub_template**()

Basic template for qsub. This means that any flags that can have multiple copies are already included in the data structure. So you can do `template["l"]["intel"]` without having to check that "l" exists.

```
template = qsub_template()
template["q"] = "all.q"
template["P"] = "proj_forecasting"
template["l"]["h_rt"] = "12:00:00"
args = template_to_args()
assert "-q all.q" in " ".join(args)
```

class gridengineapp.**FlyWeightJob**(*job_jsonlike*)

Sits on top of the parsed XML to answer job questions. The *for_each_member* creates a reasonable Pythonic data structure. What's missing at that point is knowing what tag correspondes to what human information. We layer that here and will add what we need when we need it.

job_dict = None

Dictionary containing all information from qstat.

property status

Set of strings like: idle, running, as a set. This can be in more than one state at a time, such as `{"queued", "waiting"}`, which we know as qw.

property tasks

FlyWeightTasks for tasks in the job. Can be none.

property name

As given by the `-N` qsub option.

property job_id

The job ID, as in 2349272.

property task_cnt

How many tasks are associated with this job. Jobs contain tasks, and it's the tasks that run, have statuses, and have CPU times.

class gridengineapp.**FlyWeightTask**(*task_jsonlike*)

Responsible for presenting task-specific information from qstat. Every job contains at least one task. Task arrays have one or more tasks.

task_dict = None

Dictionary containing all information from qstat.

property number

Tasks within a job are numbered from 1.

property status

Status is a set of strings.

property restarted
Bool: Whether this task did restart.

property hostname
Hostname where this task will run, is running, or has run.

class gridengineapp.**MiteWeightJob**(*job_jsonlike*)
Like the FlyWeightJob, this represents a Job. This one includes everything in the simplified version of qstat.

job_dict = None
Dictionary with all information from qstat.

property status
Set of strings like: idle, running, as a set. This can be in more than one state at a time, such as `{"queued", "waiting"}`, which we know as qw.

property tasks
FlyWeightTasks for tasks in the job. Can be none.

property name
As given by the `-N` qsub option.

property job_id
The job ID, as in 2349272.

property task_cnt
How many tasks are associated with this job. Jobs contain tasks, and it's the tasks that run, have statuses, and have CPU times.

class gridengineapp.**GridParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

error(*message*)
Override the base class because it calls sys.exit. This library uses the parser as an internal tool, not just for user interaction. For instance, it's used in testing, where an exception is more appropriate.

If the status is 0, that means nothing is wrong, and the user requested `--help`, so, yes, exit.

Otherwise, print a message to standard error and raise an exception.

exception gridengineapp.**ArgumentError**
An error for command-line arguments.

class gridengineapp.**FileEntity**(*file_path*)
Responsible for making a path that is writable for a file.

Parameters **relative_path** (*Path|str*) – Path to the file, relative to a root.

property path
Return a full file path to the file, given the current context.

validate()
Validate by checking file exists.

Returns None, on success, or a string on error.

mock()
Touch the file into existence.

remove()
Delete, unlink, remove the file. No error if it doesn't exist.

class gridengineapp.**PandasFile**(*file_path*, *required_frames=None*)
Responsible for validating a Pandas file.

Parameters

- **file_path** (*Path|str*) – Path to the file.
- **required_frames** (*Dict[str,set]*) – Map from the name of the dataset, as specified by the Pandas `key` argument, to a list of columns that should be in that dataset.

validate()

Returns None, on success, or a string on error.

mock()
Touch the file into existence.

class gridengineapp.**ShelfFile**(*file_path*, *required_keys=None*)
Responsible for validating a Python shelf file.

Parameters

- **file_path** (*Path|str*) – Path to the file.
- **required_keys** (*Set[str]*) – String names of variables to find in the file.

validate()
Validates that there are variables named after the required keys. :returns: None, on success, or a string on error.

mock()
Touch the file into existence.

remove()
Delete, unlink, remove the file. No error if it doesn't exist.

gridengineapp.**check_complete**(*identify_job*, *check_done*, *timeout=3600*)
Submit a job and check that it ran. If the job never shows up in the queue, and it didn't run, that's a failure. If it shows up in the queue and goes over the timeout, we abandon it, because these are tests.

Parameters

- **identify_job** (*function*) – True if it's this job.
- **check_done** (*function*) – True if job is done.
- **timeout** (*float*) – How many seconds to wait until calling the job lost.

Returns None

gridengineapp.**entry**(*app*, *arg_list=None*)
This starts the application. Use it with:

```
if __name__ == "__main__":
    application = MyApplication()
    entry(application)
```

Parameters

- **app** (*application.Application*) – The main application to run.
- **arg_list** (*Namespace|SimpleNamespace*) – Arguments to the command line. This is usually None and is used for testing. Pass this around instead of using sys.argv because pytest makes it hard to set sys.argv.

gridengineapp.**execution_ordered**(*graph*)

This iterator orders the nodes such that they go depth-first. This is chosen so that the data has the most locality during computation. It's not strictly depth-first, but depth-first, given that all predecessors must be complete before a node executes.

exception gridengineapp.**NodeMisconfigurationError**

Failures whose faults may be due to node misconfiguration.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX