GC3Pie

Riccardo Murri, GC3, University of Zurich

2011-04-21

A typical high-throughput use case?

Run application *A* on a range of different inputs. Each input is a different file (or a set of files).

Then collect output files and post-process them, e.g., gather some statistics.

Typically implemented by a set of `sh` / `perl` scripts to drive execution on a local cluster.

Potential issues

1. **Portability:** Cannot run on a different cluster without rewriting all the scripts.
2. **Code reuse:** Scripts are often very tied to a certain purpose, so they are difficult to reuse.
3. **Heavy maintenance:** the more a script does its job well, the more you'll find yourself adding "generic" features and maintaining requests from other users.

What is GC3Libs?

GC3Libs is a Python library to drive application execution on Grids and SGE clusters.

GC3Libs provides ways to customize execution control based on application type, and compose applications to form complex execution patterns.

GC3Libs is part of a larger pack of tools called GC3Pie.

What is GC3Pie, then?

GC3Pie consists of:

- *GC3Libs:* Python library, aimed at programmers to drive application execution on Grids and clusters.
- *GC3Utils:* simple command-line interface to the core GC3Libs functionality: submit/monitor/kill a job, retrieve output, etc.
- *GC3Apps:* Driver scripts developed for specific groups, but that may be of independent general interest. (E.g., running the Rosetta Docking application on a large set of inputs.)

How is GC3Libs different? (I)

GC3Libs runs specific **applications**, not generic jobs.

That is, GC3Libs exposes `Application` classes whose programming interface is adapted to the specific task/computation a scientific application performs.

GC3Libs supports a few applications in the main library. (Our goal is to support more and more.)

You can add your own applications. You *have to* add you own applications.

How is GC3Libs different? (II)

GC3Libs can run applications in parallel, or sequentially, or any combination of the two, and do arbitrary processing of data in the middle.

Think of workflows, except you can write them in the Python programming language.

Which means, you can create them dynamically at runtime, adapting the schema to your problem.

How could this solve any issues?

Portability: GC3Libs aims at providing an abstraction over Grid and cluster resources: one single script is be able to run on different computational sites.

Code reuse: The application model, coupled with an object-oriented design, encourages writing more generic code that can be intergrated into the library. (We hope for community-contributed code in the event.)

Heavy maintenance: Generic features are part of the library. Focus on what makes your code special.

GC3Libs application model

An application is a subclass of the `gc3libs.Application` class.

Generic `Application` class patterned after ARC's xRSL model.

At a minimum: provide application-specific command-line invocation.

Advanced users can customize pre- and post-processing, react on state transitions, set computational requirements based on input files, influence scheduling. (This is standard OOP: subclass and override a method.)

An example from the library

```
app = RosettaDockingApplication(
   100, # number of decoys to compute
   '1brs.pdb', # input file
   flags_file='flags.txt', # optional
)
```

The `RosettaDockingApplication` class knows how to invoke Rosetta's `docking_protocol` program to compute N decoys of a given input file.

A basic example (I)

```
class SquareApplication(Application):
  """Compute the square of an integer, remotely."""
  def __init__(self, x):
    Application.__init__(
      self,
      executable = '/usr/bin/expr',
      arguments = [x, '*', x],
      inputs = [ ],
      outputs = [ ],
      stdout = "stdout.txt",
    )
```

This runs `expr x * x` and saves its output into `stdout.txt`

A basic example (II)

When the remote computation is done, the `postprocess` method is called with the path to the output.

```
def postprocess(self, output_dir):
  output_file = open(output_dir + self.stdout)
  output_value = output_file.read()
  self.result = int(output_value)
```

The above code sets `result` to the integer value computed by running `expr x * x`.

A simple high-throughput script structure. . .

1. Get access to the Grid (e.g., authentication step)
2. Prepare files for submission
3. Submit jobs
4. Monitor job status (loop)
5. Retrieve results
6. Postprocess and display

Core operations

Core operations: submit, update state, retrieve (a snapshot of) output, cancel job.

Core operations are **synchronous**.

Operations are always performed by a `Core` object. `Core` implements an overlay Grid on the resources specified in the configuration file.

Core operations: verb/object interface

Get an instance of `Core`:

```
g = Core(read_config_file(path))
```

Then you can operate on `Application` instances:

- submit: `g.submit(app)`
- monitor: `g.update_state(app)`
- fetch output: `g.fetch_output(app, dir)` (starts working as soon as application is RUNNING)
- cancel job: `g.kill(app)`
- free remote resources: `g.free(app)`

A simple high-throughput script, GC3Libs version

1. *Create a gc3libs.Core instance*
2. *Create instance(s) of the application class*
3. Submit applications
4. Monitor application status (loop)
5. Retrieve results
6. Postprocess and display

What if. . . ?

Looping is fine with a small number of jobs.

What if I want to run 10'000 jobs in a session? Do I have to loop/wait until all of them are finished?

What if my box crashes in the middle of the loop? Do I lose all running jobs?

What if the proxy expires just in the middle of the loop?

How do I manage authentication with GC3Libs?

You don't.

GC3Libs will check that there is always a valid proxy and certificate when attempting Grid operations, and if necessary, renew it.

GC3Libs provide a specific authentication module, that abstracts on the various authentication models. It can be used to ease/automate authentication steps when accessing the Grid.

Persisting jobs

GC3Libs provides a simple persistence framework:

- save a live `Application` to disk, return “persistent ID”
- load a saved application given its “persistent ID”
- delete a saved application
- list IDs of saved applications (very simplistic! **your input needed:** what kind of query/select operations should we support?)

Filesystem-based storage (1 job, 1 file). But interface is generic, could use SQL, MongoDB, etc.

Implemented on top of Python’s `pickle` module: it can persist any kind of object, not just jobs.

Asynchronous operations

The `Engine` class provides all core operations, with a non-blocking interface.

Calling core methods on an `Engine` instance returns immediately to the caller; operations are actually executed when you call the `Engine.progress()` method.

Which you can do in a separate thread, thus achieving asynchronous operation.

The `Engine` class

Same programmatic interface as the `Core` class: you can use an `Engine` instance every time a `Core` is needed.

The `progress()` method will advance jobs through their lifecycle; use state-transition methods to take application-specific actions. (E.g., post-process output data.)

An engine can automatically persist the jobs, if you so wish. (Just pass it a `Store` instance at construction time.)

A high-throughput script with GC3Libs, revisited

1. *Create a gc3libs.core.Core instance*
2. *Create a gc3libs.persistence.FilesystemStore instance*
3. *Create a gc3libs.core.Engine instance*
4. *Load saved jobs into it*
5. Create *new* instance(s) of the application class
6. *Let engine manage jobs until all are done*
7. ~~Retrieve results~~ (the `Engine` does it)
8. Postprocess and display

Tasks 1.-4., and 6.-7. are automatically done by the `SessionBasedScript` class.

A high-throughput script with GC3Libs, *the code*

```
core = Core(read_config_file)
store = FilesystemStore(directory)
apps = [ store.load(jobid)
         for jobid in file(session, 'r') ]
for arg in new_args:
  apps += Application(arg, ...)
engine = Engine(core, apps, store)
engine.wait() # call progress() until done
total_global_postprocess()
```

This is the **actual** structure of the GRosetta/GGamess/GRunDB scripts! (But you would just subclass `SessionBasedScript`.)

Job dependency management

GC3Libs provides Directed Acyclic Graph support.

DAGs are created programmatically from Python code.

Which means, no graphical editor. But also means you can create workflows on-the-fly as your computation proceeds.

The *3n+1* conjecture, a fictitious use case

Define a function $f(n)$, for n positive integer:

- if n is even, then $f(n)=n / 2$,
- if n is odd, then $f(n)=3n+1$,

For every positive integer n, form the sequence $S(n)$: $n \to f(n) \to f(f(n)) \to f(f(f(n))) \to \ldots$

For example:

- $S(1) = 1 \to 4 \to 2 \to 1$,
- $S(2) = 2 \to 1$,
- $S(3) = 3 \to 10 \to 5 \to 16 \to 4 \to 2 \to 1$,

The *3n+1* conjecture (II)

Conjecture: For every positive integer n, the sequence $S(n)$ eventually hits 1.

Imagine you want to check the *3n+1* conjecture up to some very large number N.

- For each $n = 1, \dots, N$, run a task $H(n)$ that computes $S(n)$ until it hits 1.
- Each task $H(n)$ is a sequence of computational jobs $J(n,k)$, where $J(n,k+1)$ applies function f to the result of $J(n,k)$.

The $J(n,k)$ must be run **sequentially** (over k).

The $H(n)$ can be run in **parallel**.

The *3n+1* conjecture (III)

Let's define the simple application *J(n,k)* that computes *f*:

```
class HotpoApplication(Application):
  def __init__(self, n):
    Application.__init__(
      self,
      executable = '/usr/bin/expr',
      arguments = (
          # run `expr n / 2` if n is even
          [n, '/', n] if n % 2 == 0
          # run `expr 1 + 3 * n` if n is odd
          else [1, '+', 3, '*', n]),
      stdout = "stdout.txt",
    )
```

Composition of tasks (I)

The unit of job composition is called a `Task` in GC3Libs.

An `Application` is the primary instance of a `Task`.

However, a single task can be composed of many applications. A task is a composite object: tasks can be composed of other tasks.

Workflows are built by composing tasks in different ways. A “workflow” is a task, too.

Composition of tasks (II)

The `SequentialTask` class takes a list of jobs and executes them one after the other. Subclass and override the `next()` method to determine early exit conditions, or to modify the list dynamically.

The `ParallelTask` class takes a list of jobs and executes all of them in parallel. It's done when all jobs are done: there's an implicit synchronization barrier at the end.

Composition of tasks (III)

`Application`, `SequentialTask` and `ParallelTask` are all subclasses of the same `Task` interface.

So, you can create sequential collections of parallel jobs, parallel collections of sequential collections, etc.

Plus, collections can be mutated at run-time.

An `Engine` really manages a list of tasks, so we are really scripting workflows here.

The *3n+1* conjecture (IV)

Now string together the *J(n,k)* to compute a single sequence *S(n)*:

```
class HotpoSequence(SequentialTask):
  def __init__(self, n):
    # compute first iteration of /f/
    self.tasks = [ MyApplication(n) ]
    SequentialTask.__init__(self, self.tasks)
  def next(self, k):
    last_computed_value = self.tasks[k].result
    if last_computed_value == 1:
      return TERMINATED
    else:
      self.tasks.append(MyApplication(last_computed_value)
       return RUNNING
```

The *3n+1* conjecture (V)

Parallel tasks are independent by definition, so it's even easier to create a collection:

```
tasks = ParallelTaskCollection([
          HotpoSequence(n) for n in range(1, N) ])
```

We can run such a collection like any other `Application`.

Have fun proving the *3n+1* conjecture! ;-)

A real use case: dynamic programming / value function iteration (I)

Let a function *U(t, a, b, c)* be given. Assume that computing *U* is computationally intensive.

We want to compute a function *f(x,t)* such that:

- *f(x,0)* has a given value (boundary condition)
- $f(x,t+1) = U(t, f(x-1,t), f(x,t), f(x+1,t))$

Can this be computed with a GC3Pie task collection?

A real use case: dynamic programming / value function iteration (II)

Computing *f(x,t+1)* depends on the computed values of *f(x±1,t)*.

But computing *f(x,t)* is independent of *f(x',t)*.

So we have a **sequence** of **parallel** jobs:

- a parallel collection *P(t)* computes *f(x,t)* for every *x*
- a sequential collection *S* computes *P(1)*, then *P(2)*, etc. . .

Again on issues (I)

Portability: different computational resources have different conceptual/programming models. Taking the minimal common set of features is often not enough.

Solving this might need some radical programming change. Until that is sorted out, we have a constantly changing API, and even that might not support the features you need.

Again on issues (II)

Code reuse: At the end of the day, this very much depends on the user: so far users have been building sh/perl scripts to glue together Python scripts provided by GC3Libs.

Is this "just" a language/habit issue?

Again on issues (III)

Heavy maintenance: GC3Libs provide classes for frequently-used patterns in high-throughput application scripting and programming.

So, you get standard/generic features for free, and can concentrate on implementing the part that is specific to *your* application.

The usual disclaimer of warranty

GC3Libs in active development. Long list of features requests, planned improvements, and bug reports.

Small team: work is being prioritized according to requests from users, but sometimes it takes quite some time from request to implementation.

Any questions?

GC3Pie home page: http://gc3pie.googlecode.com

Source code: `svn co http://gc3pie.googlecode.com/svn`

Mailing list: gc3pie@googlegroups.com

Thank you!

Technical details

The following slides discuss a few technical details that complement the introduction.

(But they are actually only relevant if you are trying to do some GC3Libs programming.)

Application lifecycle

GC3Libs `Application` objects mimic POSIX processes life-cycle. There's a *single TERMINATED state*, whatever the job outcome.

As with POSIX processes, you have to inspect the exit code and signals to determine the cause of "job death".

- If `os.WIFSIGNALED(app) = False` then job run to completion: check exit code!
- If `os.WIFSIGNALED(app) = True` then some error occurred before end of application code.

Grid- and batch-system errors are encoded as "pseudo-signals". E.g., if `os.WTERMSIG(app) = 124` then job was killed by remote batch system.

Core operations: self-action interface

Get an instance of core, then "attach" an application to it:

```
g = Core(read_config_file())
app.attach(g)
```

The application can now operate on itself:

- submit: `app.submit()`
- monitor: `app.update_state()`
- etc.

Combined with state-transition methods, this gives a way to embed job control logic in the `Application` object.

Think of automatic resubmission if certain conditions are met.