Basic GC3Pie programming

GC3: Grid Computing Competence Center,
University of Zurich

Oct. 1, 2012

The basic purpose of GC3Pie

Run commands.

Just like the terminal, except GC3Pie can run them on *remote* computing resources.

(And it can run large collections of commands, but that's for later.)

Specifying commands to run, I

You need to “describe” an application to GC3Pie, in order for GC3Pie to use it.

This “description” is a blueprint from which many actual command instances can be created.

(A few such “descriptions” are already part of the core library.)

GC3Libs application model

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

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.)

A basic example, I

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

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

Always inherit from Application

Your application class must inherit from class `gc3libs.Application`

```
from gc3libs import Application
class SquareApplication@\HL{(Application)}@:
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

A basic example, III

Perform application-specific initialization first...

```
class SquareApplication(Application):
  def __init__(self, x):
    @\HL{\textbf{self}.to\_square = x}@
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

A basic example, IV

…but remember to call the `Application` constructor!

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    @\HL{Application.\_\_init\_\_(}@
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The arguments parameter, I

First, give the list of program arguments.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      @\HL{arguments=[}@
        @\HL{'/usr/bin/expr',}@
        @\HL{x, '*', x}@],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The `arguments` parameter, II

The first argument in the list is the name or path to the command to run.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        @\HL{'/usr/bin/expr',}@
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The `arguments` parameter, III

The rest of the list are arguments to the program, as you would type them at the shell prompt.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        @\HL{x, '*', x}@],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The inputs parameter, I

The `inputs` parameter holds a list of files that you want to *copy* to the location where the command is executed. (Remember: this might be a remote computer!)

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      @\HL{inputs=[ ],}@
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The `inputs` parameter, II

Input files retain their name during the copy.

For example:

```
inputs = [
  '/home/rmurri/values.dat',
  '/home/rmurri/stats.csv',
  ]
```

will make files *values.dat* and *stats.csv* available in the command execution directory.

The outputs parameter, I

The `outputs` argument list files that should be copied from the command execution directory back to your computer.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      @\HL{outputs=[ ],}@
      output_dir='squares.d',
      stdout="stdout.txt",
      join=True)
```

The `inputs` parameter, II

Output file names are *relative to the execution directory*. For example:

```
outputs = [ 'result.dat', 'program.log' ]
```

(Contrast with input files, which must be specified by *absolute path*, e.g., `/home/rmurri/values.dat`)

Any file with the given name that is found in the execution directory will be copied back. (*Where?* See next slides!)

If an output file is *not* found, this is *not* an error. In other words, output files are optional.

The output_dir parameter, I

The output_dir parameter specifies where output filess will be downloaded.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      @\HL{output\_dir='squares.d',}@
      stdout="stdout.txt",
      join=True)
```

The *output_dir* parameter, II

By default, GC3Pie does not overwrite an existing output directory: it will move the existing one to a backup name.

So, if `squares.d` already exists, GC3Pie will:

1. rename it to `squares.d.~1~`
2. create a new directory `squares.d`
3. download output files into the new directory

The `stdout` parameter

This specifies that the command's `standard output` should be saved into a file named `stdout.txt` and retrieved along with the other output files.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      @\HL{stdout="stdout.txt",}@
      join=True)
```

(The stderr parameter)

There's a corresponding stderr option for the command's *standard error* stream.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      arguments=[
        '/usr/bin/expr',
        x, '*', x],
      inputs=[ ],
      outputs=[ ],
      output_dir='squares.d',
      @\HL{stdout="stdout.txt",}@
      join=True)
```

The `join` parameter

This specifies that *stdout* and *stderr* should be merged into one single file (like happens on the terminal screen).

The default is *not* to join, i.e., keep *stdout* and *stderr* separate.

```
class SquareApplication(Application):
  def __init__(self, x):
    self.to_square = x
    Application.__init__(
      self,
      # ...
      outputs=[ ],
      output_dir='squares.d',
      stdout="stdout.txt",
      @\HL{join=True})
```

Now that we have an application,
what do we do with it?

A simple high-throughput script structure

This is the prototypical structure of a script for running jobs on remote computational resources.

1. Initialize computational resources
2. Prepare files for submission
3. Submit jobs
4. Monitor job status (loop)
5. Retrieve results
6. Postprocess and display

Core operations

GC3Pie provides a `Core` object to submit a job, update its state, retrieve (a snapshot of) the output, or cancel the job.

Core operations are **blocking**.

How to create a `Core` instance

An instance of core is created with the following two-liner, which reads the default configuration file and initializes the computational resources:

```
cfg = gc3libs.config.Configuration(
  *gc3libs.Default.CONFIG_FILE_LOCATIONS,
  auto_enable_auth=True)
core = gc3libs.core.Core(cfg)
```

Likely, you will only ever need no more than *one single* instance of `Core` in your scripts.

Reference: http://gc3pie.readthedocs.org/en/latest/gc3libs/api.html#module-gc3libs.config

Core operations: verb/object interface

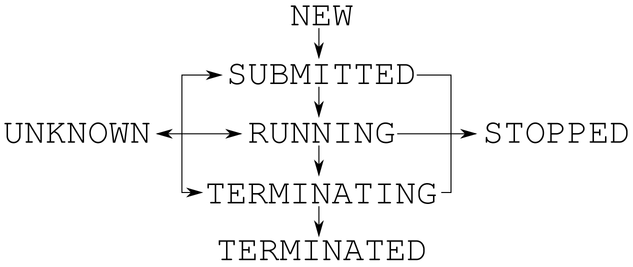
`Core` instances operate on `Application` instances:

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

Reference: http://gc3pie.readthedocs.org/en/latest/gc3libs/api.html#gc3libs.core.Core

Application lifecycle

`Application` objects can be in one of several states.



The current state is stored in the `.execution.state` instance attribute.

Reference:

http://gc3pie.readthedocs.org/en/latest/gc3libs/api.html#gc3libs.Run.state

Application lifecycle: state NEW

$$\begin{tikzcd}
\textbf{NEW} \arrow[d] \\
\texttt{SUBMITTED} \arrow[d] \\
\texttt{RUNNING} \arrow[d] \\
\texttt{TERMINATING} \arrow[d] \\
\texttt{TERMINATED}
\end{tikzcd}$$

NEW is the state of "just created" Application objects.

The Application has not yet been sent off to a compute resource: it only exists locally.

Application lifecycle: state SUBMITTED

$$\begin{tikzcd}
\texttt{NEW} \arrow[d] \\
\texttt{\textbf{SUBMITTED}} \arrow[d] \\
\texttt{RUNNING} \arrow[d] \\
\texttt{TERMINATING} \arrow[d] \\
\texttt{TERMINATED}
\end{tikzcd}$$

SUBMITTED applications have been successfully sent to a computational resource.

(The transition to *RUNNING* happens automatically, as we do not control the remote execution.)

Application lifecycle: state RUNNING

$$\begin{tikzcd}
\texttt{NEW} \arrow[d] \\
\texttt{SUBMITTED} \arrow[d] \\
\textbf{RUNNING} \arrow[d] \\
\texttt{TERMINATING} \arrow[d] \\
\texttt{TERMINATED}
\end{tikzcd}$$

RUNNING state happens when the computational job associated to an application starts executing on the computational resource.

Application lifecycle: state TERMINATING

$$\begin{tikzcd}
\texttt{NEW} \arrow[d] \\
\texttt{SUBMITTED} \arrow[d] \\
\texttt{RUNNING} \arrow[d] \\
\boldsymbol{\texttt{TERMINATING}} \arrow[d] \\
\texttt{TERMINATED}
\end{tikzcd}$$

TERMINATING state when a computational job has finished running, for whatever reason.

(Transition to *TERMINATED* only happens when `fetch_output` is called.)

Application lifecycle: state TERMINATED

$$\begin{tikzcd}
\texttt{NEW} \arrow[d] \\
\texttt{SUBMITTED} \arrow[d] \\
\texttt{RUNNING} \arrow[d] \\
\texttt{TERMINATING} \arrow[d] \\
\textbf{\texttt{TERMINATED}}
\end{tikzcd}$$

A job is *TERMINATED* when its final output has been retrieved and is available locally.

The exit code of *TERMINATED* jobs can be inspected to find out whether the termination was successful or unsuccessful, or if the program was forcibly ended.

A successful run or not?

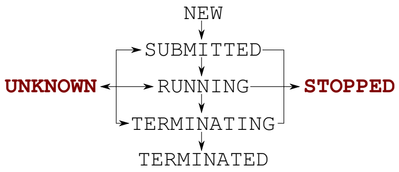
There's a *single TERMINATED state*, whatever the job outcome.

You have to inspect the exit code and signals to determine the cause of "job death".

The `.execution.exitcode` instance attribute holds the numeric exitcode of the executed command, or `None` if the command has not finished running yet.

The `.execution.signal` instance attribute is non-zero if the program was killed by a signal (e.g., memory error / segmentation fault).

Application lifecycle: error states



Two jobs indicate an error state, i.e., impossibility to carry a job to termination.

A job is in *STOPPED* state when its execution has been blocked at the remote site and GC3Pie cannot recover automatically. User or sysadmin intervention is required.

A job is in *UNKNOWN* state when GC3Pie can no longer monitor it at the remote site. (As this might be due to network failures, jobs *can* get out of *UNKNOWN* automatically.)

A simple high-throughput script, GC3Libs version

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

What about post-processing?

When the remote computation is done, the `terminated` method of the application instance is called.

The path to the output directory is available as `self.output_dir`.

```
def terminated(self):
  output_file = open(self.output_dir+'/'+self.stdout)
  output_value = output_file.read()
  self.result = int(output_value)
```

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

Putting it all together

The script square.py provides an example of how all these work together.

Exercise A: Copy the square.py script to a file `cpuinfo.py` and modify it to run the command `cat /proc/cpuinfo` instead of squaring a number. Run it and verify that it gave correct information about your computer's CPU.

Exercise B: Modify the `cpuinfo.py` script from the previous exercise, and add a post-processing method to extract the CPU model information and print it. (The CPU model information is in the line that starts with "model name".)