GC3Pie

A Python framework for high-throughput computing

Riccardo Murri
<riccardo.murri@uzh.ch>

Grid Computing Competence Centre, University of Zurich
http://www.gc3.uzh.ch/

NorduGrid Conference, 2011-05-10

University of Zurich

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` or `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 of re-usable Python components to combine scientific applications into high-throughput workflows.

GC3Libs features:

- drive application execution on ARC and SGE clusters,
- customize execution control based on application type,
- 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:* re-usable Python components 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.)

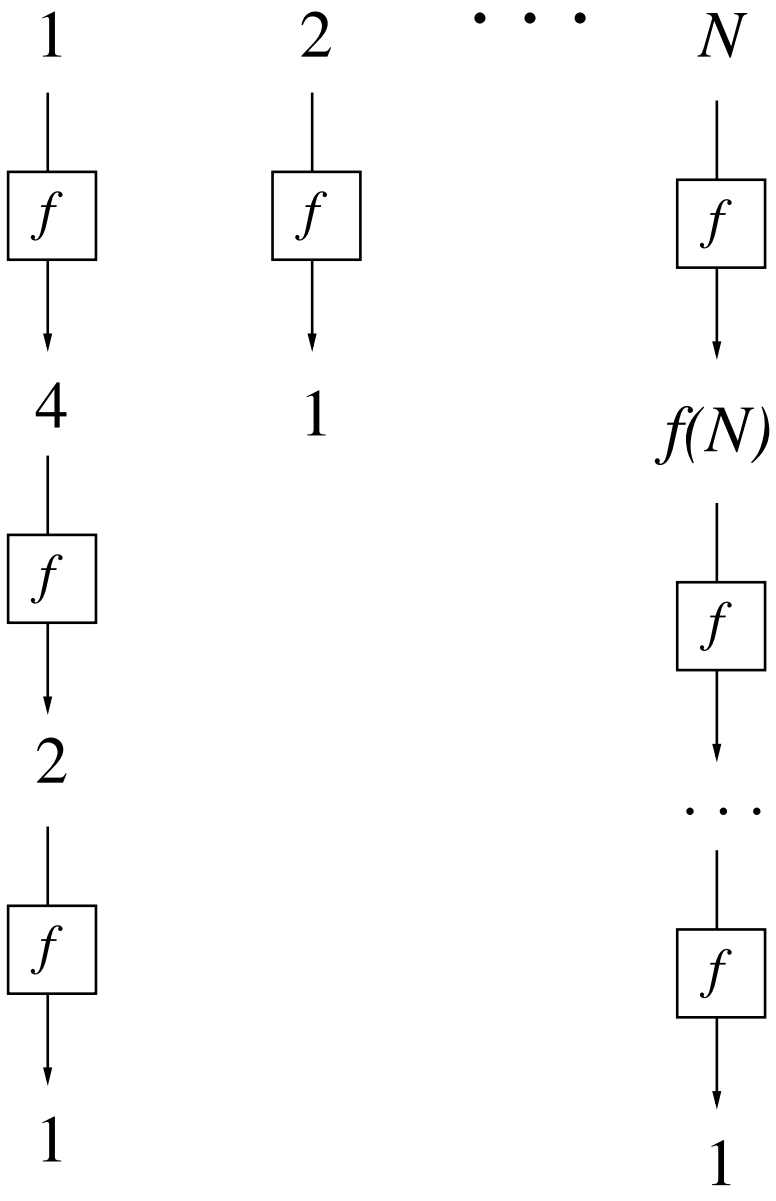
How could this solve any issues?

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.

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



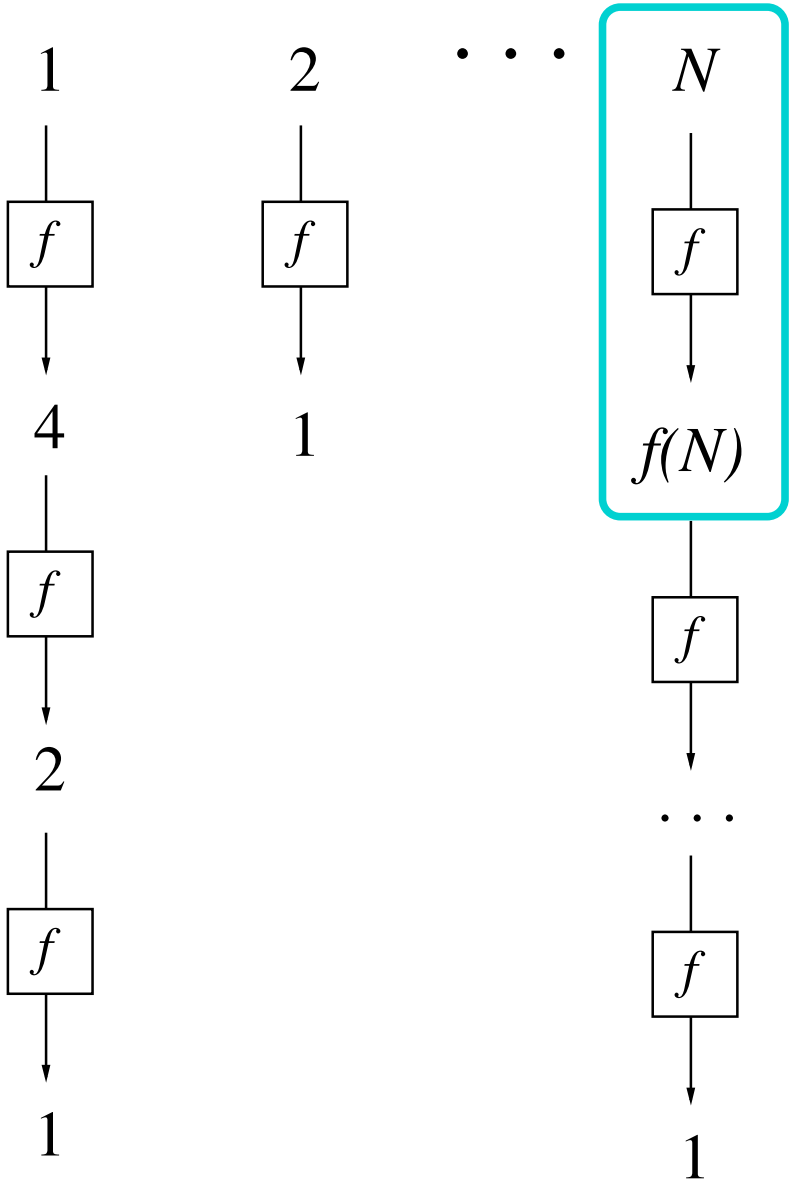
Define a function f, for n positive integer:

- 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 $S(n)$: $n \to f(n) \to f(f(n)) \to f(f(f(n))) \to \ldots$

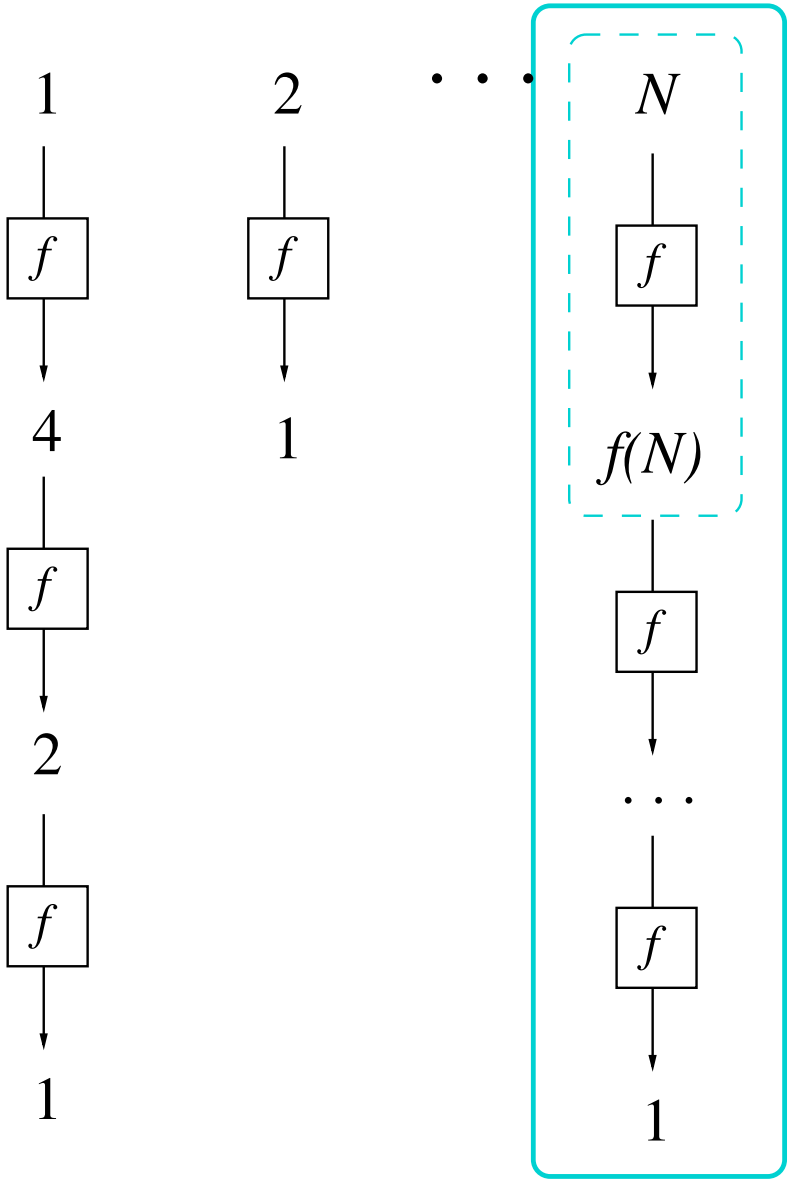
Conjecture: For every positive integer n, the sequence $S(n)$ eventually hits 1.

The *3n+1* conjecture on a Grid? *(I)*



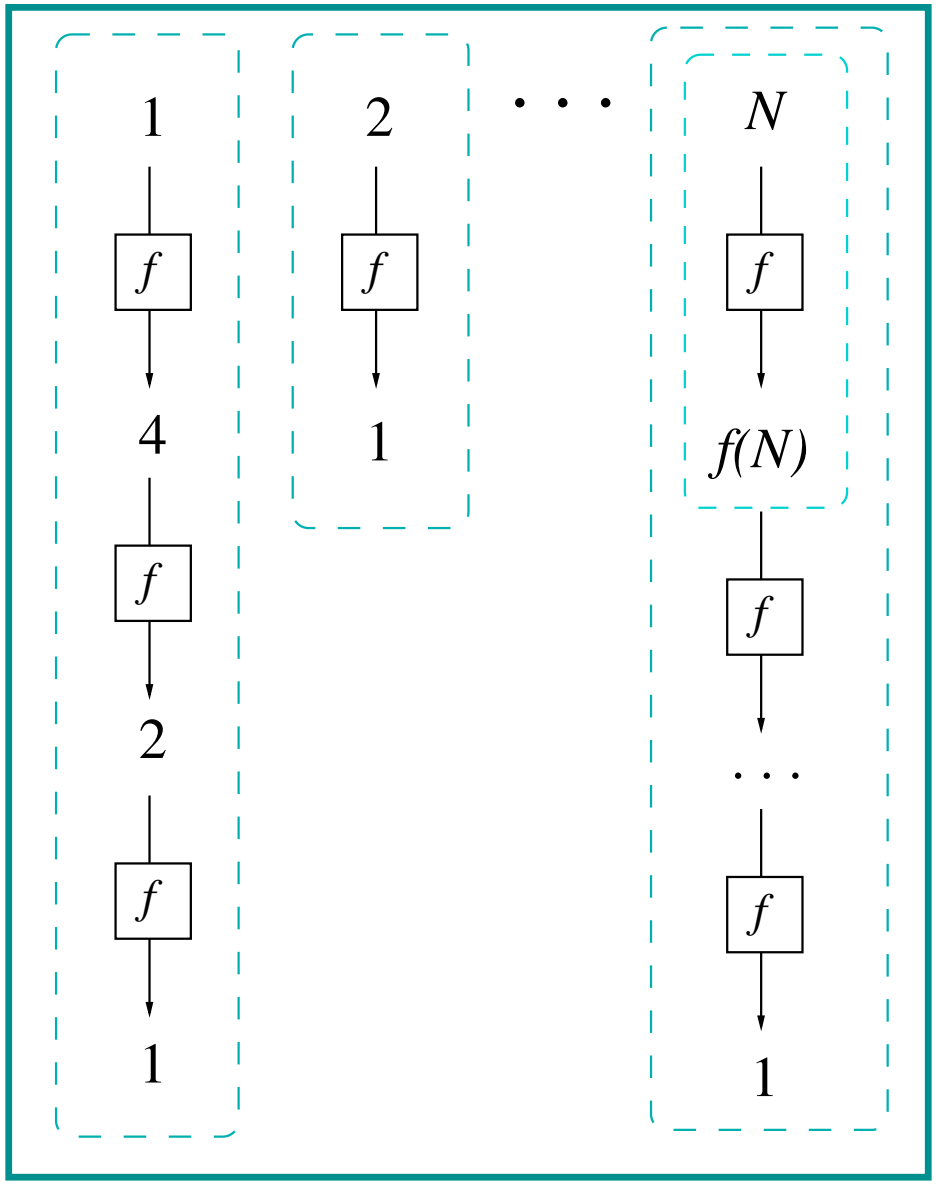
A computational job $J(n,k)$, applies function f to the result of $J(n,k)$.

The *3n+1* conjecture on a Grid? *(II)*



A sequence $H(n)$ of jobs computes the chain $n \to f(n) \to \ldots \to 1$.

The *3n+1* conjecture on a Grid? *(III)*



Run one sequence $H(n)$ per each $n = 1, \ldots, N$.

The can all run in **parallel**.

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.)

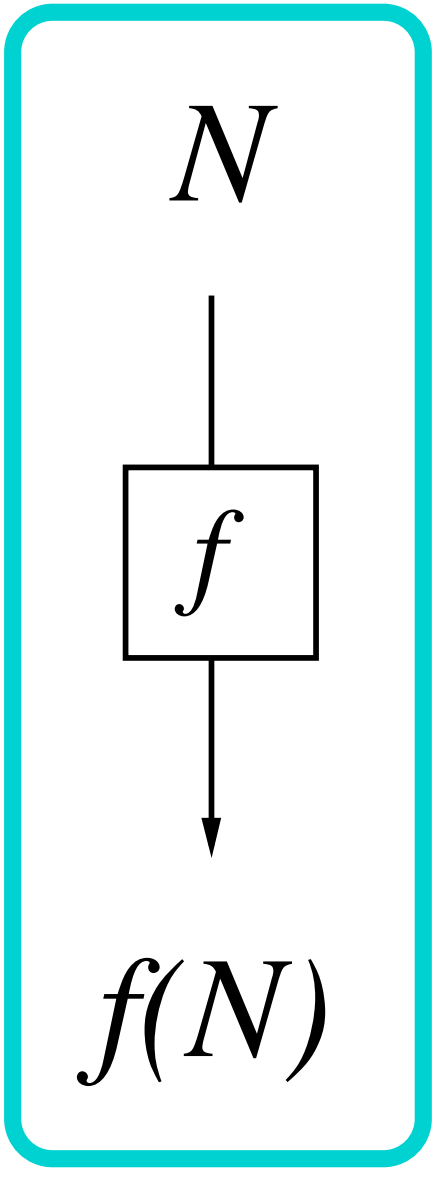
Supported applications (as of May 2011)

As of May 2011, GC3Libs has support for the following applications in the library core:

- GAMESS(US); 60K jobs in 2011.
- Rosetta (`minirosetta` and `docking_protocol`); 110K jobs run in 2011 alone!
- TURBOMOLE

. . . plus a few user-developed codes.

The *3n+1* conjecture (IV)



Let's define the simple application 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.

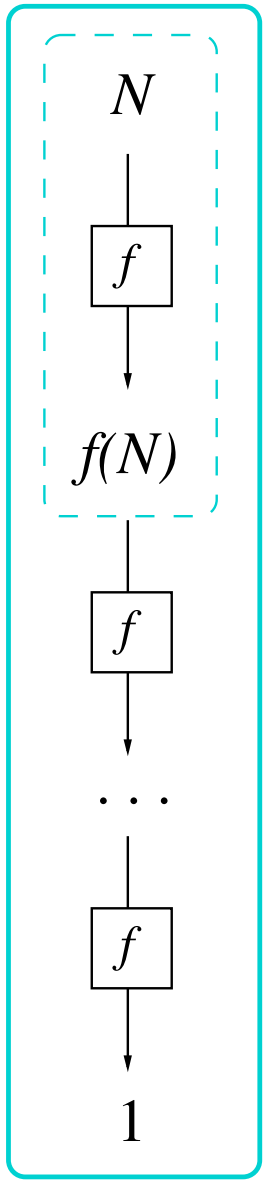
Workflows are built by composing tasks in different ways. A “workflow” is a task, too.

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.

The *3n+1* conjecture (V)



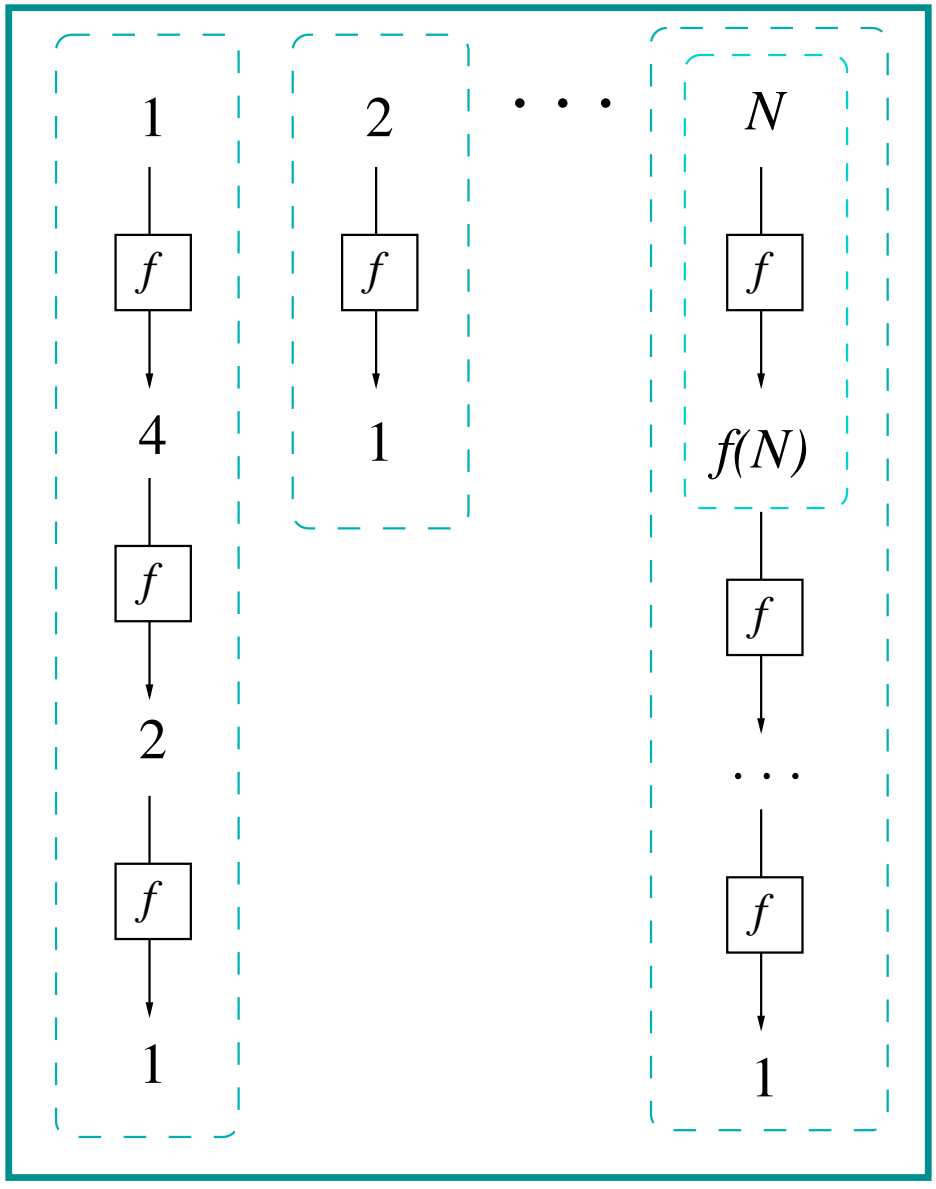
Now string together applications to compute a single sequence:

```
class HotpoSequence(SequentialTask):

  def __init__(self, n):
    # compute first iteration of f
    self.tasks = [ HotpoApplication(n) ]
    SequentialTask.__init__(self, self.tasks)

  def next(self, k):
    last = self.tasks[k].result
    if last == 1:
      return TERMINATED
    else:
      self.tasks.append(MyApplication(last)
       return RUNNING
```

The *3n+1* conjecture (VI)



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 `Task`.

A simple high-throughput script structure

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

The `Engine` class

Implements core operations on applications, with *non-blocking* semantics.

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.)

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.

A high-throughput script, revisited with GC3Libs

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

You only need to implement 2. and 5.; the rest is done by the `SessionBasedScript` class.

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.

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!