PROCEEDINGS OF SCIENCE

GC3Pie: A Python framework for high-throughput computing

Sergio MAFFIOLETTI*
GC3: Grid Computing Competence Center
University of Zurich
E-mail: sergio.maffioletti@gc3.uzh.ch

Riccardo Murri
GC3: Grid Computing Competence Center
University of Zurich
E-mail: riccardo.murri@gmail.com

This paper present GC3Pie [7], a python library to ease the development of scalable and robust High Throughput data analysis tools. Most of the current distributed computing middlewares as well as most of the in-house grown scripts fall short in reaching the scaling and reliability factors required by the ever growing demand of large data analysis. GC3Pie provides mechanisms to automitise the execution and the monitoring of large collection of applications while, at the same time, provides simple data structures and interfaces to steer the behaviour of the underlying system in an application-centric perspective. The goal of GC3Pie is to embody the common execution and monitorig processing part of large data analysys while moving most decision making logic to the application level; like, for example, the reaction of certain types of failures, the validation of the application execution or the brokering of the computing resources driven by application fidelity metrics. This allows to write application specific tools that take full control of the underlying computing and data infrastructure, as opposite of current middleware stacks that are trying to embody the full control of the execution logic thus reducing the flexibility of the entire system as they prevent applications to define their own expected behaviour of the system.

EGI Community Forum 2012 / EMI Second Technical Conference
26-30 March, 2012
Munich, Germany

*Speaker.

1. Introduction

Modern science is progressively more involved in the use of scientific computing involving mathematical models with a hierarchy of complexity. Together with expanding capabilities in hardware, this results in exponential increase in computed data, which comes with a need for an array of quantitative analysis tools, all brought together in a seamless fashion to solve scientific problems. As a major consequence of this growth in digital computing the needs for support of data and compute infrastructure to carry out scientific investigations is growing.

GC3Pie addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details. The programming model specifies simple data structures and interfaces to steer the behavior of the underlying system in an application-centric perspective. GC3Pie maintains a separation of what applications are to be executed and how those executions are actually carried out on the underlying infrastructure. The first is under the control of the developer, while the second is exclusively the responsibility of the execution framework.

At the same time, GC3Pie remains agnostic on the execution control while moving relevant decision making logic to the application level; like, for example, the reaction of certain types of failures, the validation of the application execution or the brokering of the computing resources. This allows to write application specific tools that take full control of the underlying computing and data infrastructure, as opposite of current middleware stacks that are trying to embody the control of the execution logic thus reducing the flexibility at application level that are prevented to define their own expected behavior of the system.

2. What is GC3Pie

GC3Pie is a library of Python classes for running large job campaigns (high throughput) on diverse batch-oriented execution environments, including ARC-based computational grids [2]. It also provides facilities for implementing command-line driver scripts, in the form of Python object classes whose behavior can be customized by overriding specified object methods. GC3Pie comes with backends that allow to easily integrate SGE and LSF controlled batch clusters and with an ARC backends that makes fully use of the powerfull ARC client libraries to access ARC-based systems as well as gLite [6] CREAM-CE-based systems.

The GC3Pie is comprised of two main components:

GC3Libs

A python package for controlling the life-cycle of a Grid or batch computational job collections. GC3Libs provides services for submitting computational jobs to Grids and batch systems and controlling their execution, persisting job information, and retrieving the final output.

At the heart of the GC3Pie model is a generic *Application* object, that provides a high-level description of a computational job: list of input/output files, what command to run, resource requirements and limits, etc. GC3Pie translates this information into the job description format needed by the actual execution back-end selected, e.g., xRSL for ARC-based Grids, or a submis-

sion script for direct execution on a batch-queuing system. *Application* objects can be adapted to provide behavior customized to a specific use case.

GC3Apps

Driver scripts to run large job campaigns. There is a need in some scientific communities, to run large job campaigns to analyze a vast number of data files with the same application. The single-job level of control as provided by most of the current middleware's interfaces, in this case is not enough: you would have to implement "glue scripts" to control hundreds or thousand scripts at once. GC3Pie has provision for this, in the form of re-usable Python classes that implement a single point of control for job families.

The GC3Apps scripts are driver scripts that run job campaigns using the supported applications on a large set of input data. They can be used in production as-is, or adapted to suit your data processing needs.

2.1 High Throughput with GC3Pie

The programming model of GC3Pie relays on the following concepts:

- *gc3libs.Application*: it is the basic execution unit; this data structure contains the description as well as the application-specific decision logic; it has been conceived to encapsulate all the application-specific aspects of the execution. Generic `Application` class patterned after ARC's xRSL [1] model. At a minimum: provide application-specific command-line invocation. Through subclassing it is possible to customize pre- and post-processing, react on state transitions, set computational requirements based on input files, influence scheduling.

- *gc3libs.core.Engine*: it implements core operations on applications, with *non-blocking*semantics; it is a logical container of Applications to be executed. Engine behaves around a `progress()` method that advance jobs through their lifecycle; use state-transition methods to take application-specific actions (e.g. pre- post-processing, application-level failure detection and reaction, resubmission)

- *gc3libs.Persistency*: any robust and fault-tolerant high throughput tool need to provide a mechanism to persist statefull information during the execution life-cycle. This allows to recover the execution from any sort of interruption. GC3Pie provides a hierarchy of persistency objects that can be used to store application execution information (e.g. FilesystemStore, DBStore)

- *gc3libs.Task* Composition: a Task is a container for an Application object. Tasks could be group together forming a Collection of Tasks. The execution logic for a Collection of task is used to define workflows.

A prototypical driver script for analyzing large data-sets should

1. identify the input files (this is normally left outside the control of GC3Pie as it could be specific of each usecase)

2. Create a *gc3libs.core.Engine* instance and load saved jobs into it
3. Create new instance(s) of the *gc3libs.Application* class for each valid input sequence
4. attach the *application* collection to the engine.
5. Let *engine* manage and control the executions until all are done
6. Verify the results as part of the *application* post-processing step

2.2 Tasks composition: Workflows

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

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.

GC3Pie provides composition operators, that allow treating a collection of tasks as a single whole [3]: compositions operators can be arbitrarily nested, allowing the creation of complex workflows, including workflows with cycles in them, and dynamic workflows whose structure is created at runtime by the script itself. Two basic composition operators are provided, upon which others can be created (by subclassing).

The *ParallelTaskCollection* operator takes a list of tasks and creates a task that executes them all independently and, compatibly with the allowed degree of job parallelism, concurrently.

The *SequentialTaskCollection* operator takes a list of tasks and creates a task that executes them in the sequence; a decision method is invoked in between each step, which can terminate execution early (e.g., in case of errors), but also alter the list of planned tasks.

3. GC3Pie in action: `gcrypto`

`gcrypto` is a utility for supporting the large scale execution of cryptographic applications like the factorization of the 768-bit number RSA-768 by the number field sieve factoring method [4]. Like a for-loop, the `gcrypto` driver script takes as input three mandatory arguments:

- *RANGE_START*: initial value of the range (e.g., 800000000)
- *RANGE_END* final value of the range (e.g., 1200000000)
- *SLICE* extent of the range that will be examined by a single job (e.g., 1000)

For example::

```
$ gcrypto 800000000 1200000000 1000
```

will produce 400000 jobs; the first job will perform calculations on the range 800000000 to 800000000+1000, the 2nd one will do the range 800001000 to 800002000, and so on.

Jobs progress is monitored and, when a job is done, output is retrieved back.

The `gcrypto` command keeps a record of jobs (submitted, executed and pending) in a session file (set name with the '-s' option); at each invocation of the command, the status of all recorded jobs is updated, output from finished jobs is collected, and a summary table of all known jobs is printed.

Due to the very large number of embarrassingly parallel jobs that could be created with this usecase, `gcrypto` uses a *ChunkedParameterSweep* collection to group all executions. This particular data structure is like *ParallelTaskCollection* as it allows to define a parallel execution of the defined tasks, but generate a sequence of jobs with a parameter varying from 'min_value' to 'max_value' in steps of 'step'; the constructor for this class reads:

```
def __init__(self, jobname, min_value, max_value, step, chunk_size,
grid=None)
```

Only *chunk_size* jobs are generated at a time, to distribute the burden of job creation along the whole run. `Gcrypto` has been used in production on the NGI-CH national infrastructure SMSCG [5] with an initial range 800M - 1200M with step 1000; that produced 1.6M cpu_hours. SMSCG is mostly based on the ARC middleware.

4. Related Works

Ganga [8] is an easy-to-use frontend for job definition and management, implemented in Python. It has been mainly developed to meet the needs of the ATLAS and LHCb for a Grid user interface. Ganga supports large scale job handling through various plugins deriving from template ones like the "splitting" and the "merging". This approach has been demonstrated to be effective for parameter-sweep like type of computations. While Ganga provides data structure to define large collection of Job management similar to those defined by GC3Pie, it does not provide mechanisms to let applications to influence the job management lifecycle (e.g. take application-specific actions upon job state transition). While GC3Pie comes with fully fledged support for Task and Collection compositions, Ganga only provides composition primitives in the form of splitting and merging.

5. Conclusions

GC3Pie is used to implement end-to-end solutions for a large variety of scientific usecases. GC3Pie is centered around the controlled execution of a collection of applications. It automatically handles the execution process by optimizing the submission to the underlying computing infrastructure, by monitoring the progress of the entire collection, by handling individual failures through resubmission or termination, and by retrieving and organizing the results.

GC3Pie is a tool to write complex application clients by moving the control and decision logic from the middleware stack to the application level. it will grow in providing at application level as much information and related directives as possible to represent the status of the underlying system. In such a way, the application client developers will have a more structured framework to write complex execution logic (e.g. application-driven resource brokering, reliability metric, etc.)

References

[1] http://www.nordugrid.org/documents/xrsl.pdf

[2] The next-generation ARC middleware. O. Appleton, D. Cameron, J. CernÃąk et al., Annals of Telecommunications "Towards market-oriented Clouds", vol. 65, numbers 11-12 (2010) pp. 771-776. http://www.nordugrid.org/documents/next-gen-ARC.pdf

[3] Jonen, B. und Scheuring, S.: “Computational workflow in GC3PieâĂIJ (Poster Presentation), EuroScipy, Paris, 25. âĂŞ 28. August 2011, Link: http://www.euroscipy.org/conference/euroscipy2011.

[4] Thorsten Kleinjung, Joppe W. Bos, Arjen K. Lenstra, Dag Arne Osvik, Kazumaro Aoki, Scott Contini, Jens Franke, Emmanuel ThomÃl', Pascal Jermini, Michela ThiÃl'mard, Paul C. Leyland, Peter L. Montgomery, Andrey Timofeev, Heinz Stockinger: A heterogeneous computing environment to solve the 768-bit RSA challenge. Cluster Computing 15(1): 53-68 (2012).

[5] http://www.smscg.ch

[6] CREAM Computing Element. https://wiki.italiangrid.it/CREAM

[7] GC3Pie. http://code.google.com/p/gc3pie/

[8] Ganga. http://ganga.web.cern.ch/ganga/