Computational workflows with GC3Pie

http://gc3pie.googlecode.com gc3pie@googlegroups.com

What is GC3Pie?

GC3Pie is a library of Python classes to execute and control applications on distributed computing resources (e.g., SGE clusters and ARC-based grids).

GC3Pie is object-oriented: basic classes abstract the generic and repetitive part of application scripting, and let you focus on coding what is specific to your use case. Generic services provided by GC3Pie include: asynchronous job execution, programmatic generation of template files, checkpoint/restart workflow execution.

GC3Pie is a toolkit: it provides the building blocks to write Python scripts to run large computational campaigns (e.g., to analyze a vast dataset or explore a parameter space), and to combine several tasks into a dynamic workflow.

How is GC3Pie different?

Most execution engines represent workflows as data (e.g., some XML format). GC3Pie lets you write Python code instead: you write your workflow as a set of Python classes so the entire workflow logic is expressed in a plain programming language. This means that it is easy to create loops and conditionally branch execution, for example.

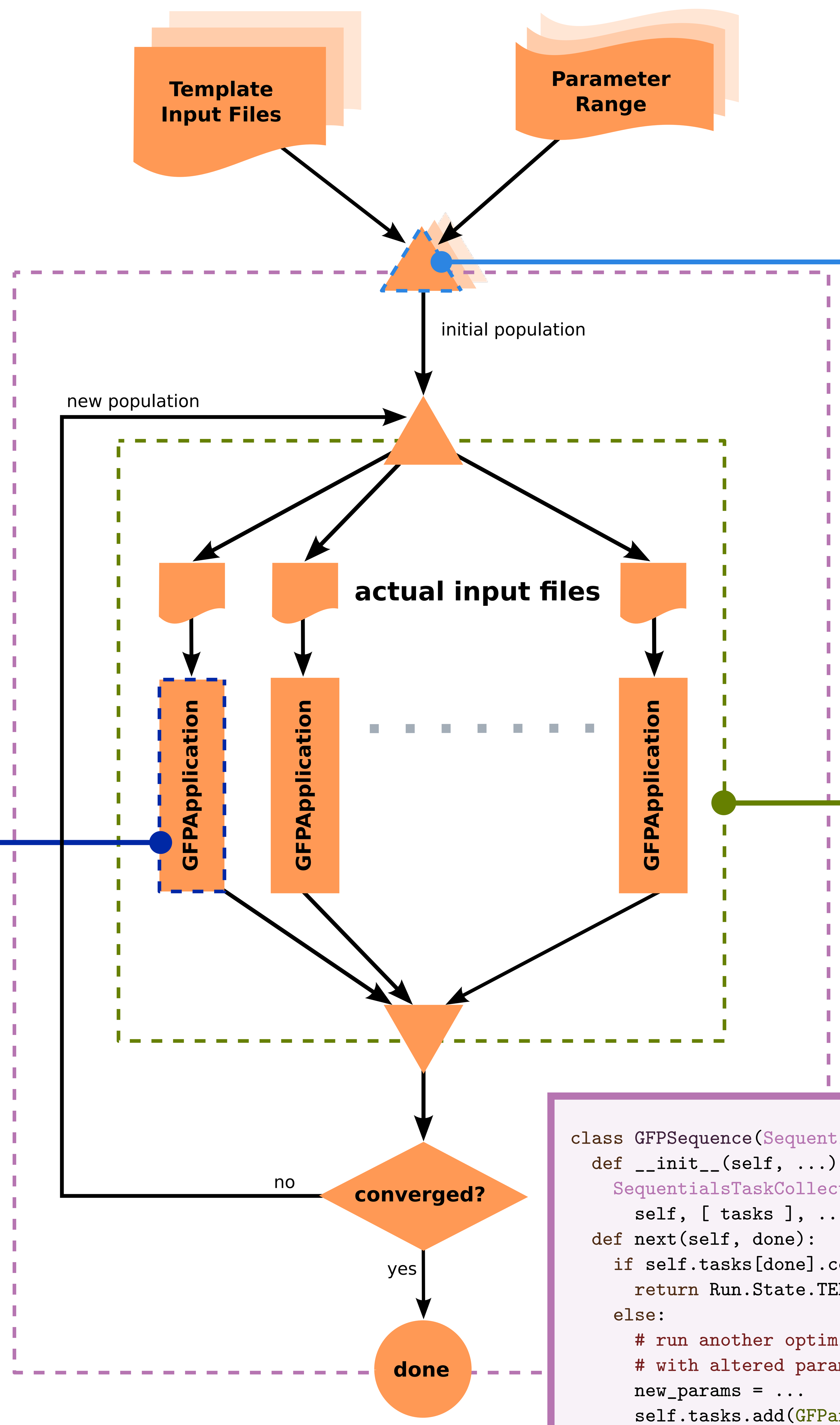
Unlike other Python frameworks for distributing computation, e.g., Celery or Pyro, GC3Pie is designed to coordinate the execution of independent Applications (often pre-existing and written in another language): with GC3Pie you write Python code to steer the computation, not to perform it.

Workflows with GC3Pie

GC3Pie encourages a compositional approach for building workflows: the basic unit in a workflow is called a **Task**; tasks can be grouped into collections, which prescribe the order in which tasks are executed.

The classes SequentialTaskCollection and ParallelTaskCollection are the basic compositions of **Tasks**; by subclassing them you define how to coordinate the execution of **Tasks**. For example, retry the execution of a certain step in a sequence, or stop a parallel parameter sweep when a certain percentage of the tasks in it are successfully done.

TaskCollections are mutable objects, so Python code can alter them on the fly, while a composition is running. This allows the creation of dynamic workflows, whose structure is not fixed in advance, rather built in response to external events.



For each optimization, a workflow is started.

A driver script class SessionBasedScript provides support for running several **Tasks** in parallel:

```
class GFPScript(SessionBasedScript):
  def new_tasks(self):
    for ctry1, ctry2 in self.country_pairs:
      # add tasks to the session
      yield (jobname,# unique identifier
             GFPSequence  # task class
             [ args ],  # creation arguments
             { kwargs })# creation keywords
```

The SessionBasedScript class implements a command line interface to manage a session consisting of the tasks so created.

Several simulations are run in parallel, one for each combination of input parameters; input files for each simulations are created using a templating mechanism.

The ParallelTaskCollection class is used to manage concurrent **Tasks**; subclass it to create specialized collections:

```
class GFPParallel(ParallelTaskCollection):
  def __init__(self, params..., **kwargs):
    # create Task collection from parameters
    tasks = [ GFPApplication(...) ]
    ParallelTaskCollection.__init__(
      self, tasks, **kwargs)
```

Applications are the basic **Tasks** that comprise a workflow in GC3Pie. An **Application** is just a UNIX process, i.e., any command that can be run from the shell command line. The **Application** class should be subclassed to specify error-checking policies and post-processing of output files:

```
class GFPApplication(Application):
  def __init__(self, ...):
    Application.__init__(
      executable="./forwardPremiumOut",
      arguments=[ "1", "2", "3" ],
      inputs=[ "input.file.name" ],
      outputs=[ "out.file", "out.directory" ])

  def terminated(self):
    # this gets called once the Task is done
    if "simulation.out" in self.outputs:
      self.execution.returncode = 0 # success
    else:
      self.execution.returncode = 1 # fail!
```

```
class GFPSequence(SequentialTaskCollection):
  def __init__(self, ...):
    SequentialsTaskCollection.__init__(
      self, [ tasks ], ...)
  def next(self, done):
    if self.tasks[done].converged == True:
      return Run.State.TERMINATED
    else:
      # run another optimization step,
      # with altered parameters
      new_params = ...
      self.tasks.add(GFParallel(new_params))
```

A succession of **Tasks** is implemented through a SequentialTaskCollection. Tasks in the collection are executed one after the other; after one of them completes, the `next()` method is called to determine what step to take: continue execution with another task, or stop.

The `next()` method can modify the collection, or re-run tasks already run. It is thus possible to implement indefinite loops, that repeat until a certain condition is met.

A real-world example: (Economic) Model calibration using Global Optimization

This workflow shows how a differential evolution optimizer is implemented with the GC3Pie library to support the analysis of a computationally intense economic model. The paper *Time-Varying International Diversification and the Forward Premium* seeks to understand the comovement of interest rates and exchange rates. The economic model is calibrated with data for five countries with major currencies.

To illustrate the explanatory power of the model, the countries' preference parameters are chosen to bring the simulated economies close to the real world. The resulting 10-dimensional optimization problem of a nonconvex function is undertaken with the help of the GC3Pie library.

In the above example GFPSequence, a SequentialTaskCollection, implements the differential evolution optimizer: each optimization iteration is one sequential task. When initialized, the optimizer generates an initial population of size n for the N-dimensional parameters (N=10).

The whole population is evaluated in parallel as a ParallelTaskCollection (see the box). Each of the n tasks within the collection is an Application instance (**GFPApplication**, see the box), a C++ implementation of the economic model that simulates the interaction of two economies.

After the GFPParallel has completed, the GFPSequence.next() method checks for convergence, otherwise generates a new population of size n and evaluates it (see the box)

The SessionBasedScript class provides a command line interface and allows running several optimizations in parallel (see the box).

References

Jonen B., Scheuring S. *Time-Varying International Diversification and the Forward Premium* (working paper). Institut für Banking und Finance, University of Zurich, 2011.

Price K.V., Storn R.M., Lampinen J.A. *Differential evolution: a practical approach to global optimization*. Springer, 2005.

Universität Zürich UZH

Riccardo Murri, Sergio Maffioletti,
Grid Computing Competence Center,
University of Zurich
http://www.gc3.uzh.ch/

Benjamin Jonen, Simon Scheuring,
Institut für Banking und Finance,
University of Zurich
http://www.ibf.uzh.ch/