

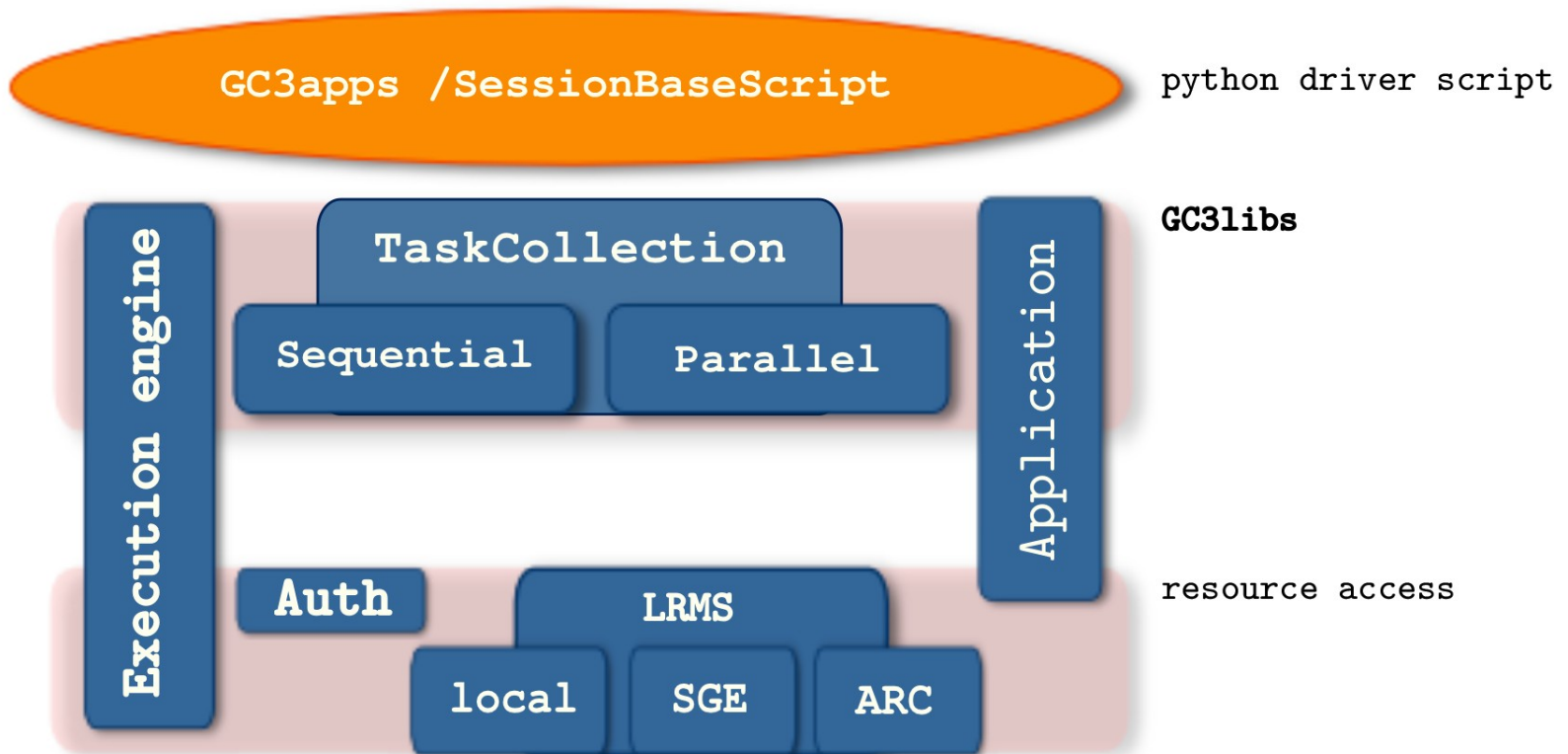
## GC3Pie: a Python framework for high-throughput computing

Sergio Maffioletti  
Grid Computing Competence Center (GC3)  
University of Zürich



# What is GC3Pie?

GC3Pie is a **Python 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**.



GC3Pie provides data structures for **controlling** the **execution** of Applications on computing resources.

GC3Pie allows:

- Uniform access** to distributed computing resources

- Automatic** authentication control

- Creation of **large collections** of Applications as composite unites

- Direct** and/or **delegated control** of Application execution process  
(**core and engine**)

- Combine** different collections of Applications into a **dynamic** workflow

- Run a generic Application on a range of different inputs; where 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.

```
class GameSSApplication(gc3libs.Application):  
    def __init__(self, inp_file_path,  
*other_input_files, **kw):  
        gc3libs.Application.__init__(self,  
            executable = "$GAMESS_LOCATION/nggms",  
            arguments = arguments,  
            inputs = (inp_file_path, input_file_name),  
            outputs = [ output_file_name ],  
            join = True,  
            **kw)
```

```
def terminated(self):  
    ...  
    if match.group('ddikick_outcome') == 'unexpectedly':  
        self.execution.exitcode = 2  
    elif match.group('ddikick_outcome') == 'gracefully':  
        self.execution.exitcode = 0  
    ...
```

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.)



You don't.

- GC3Libs will **checks** that there is always a valid proxy and certificate when attempting Grid operations, and if necessary, **renews** it.
- The SSH authentication (when required) is assumed **granted** by an external entity (e.g ssh-agent)
- Each computing resource has its **own authentication profile**. In `gc3pie.conf` is possible to link an authentication profile with one or more resources

```
[auth/cluster]  
type = ssh  
username = sergio
```

```
[auth/smscg]  
type = voms-proxy  
cert_renewal_method = slcs  
idp = uzh.ch  
vo = smscg
```

```
[resource/nor]  
type = arcl  
auth = smscg  
...
```

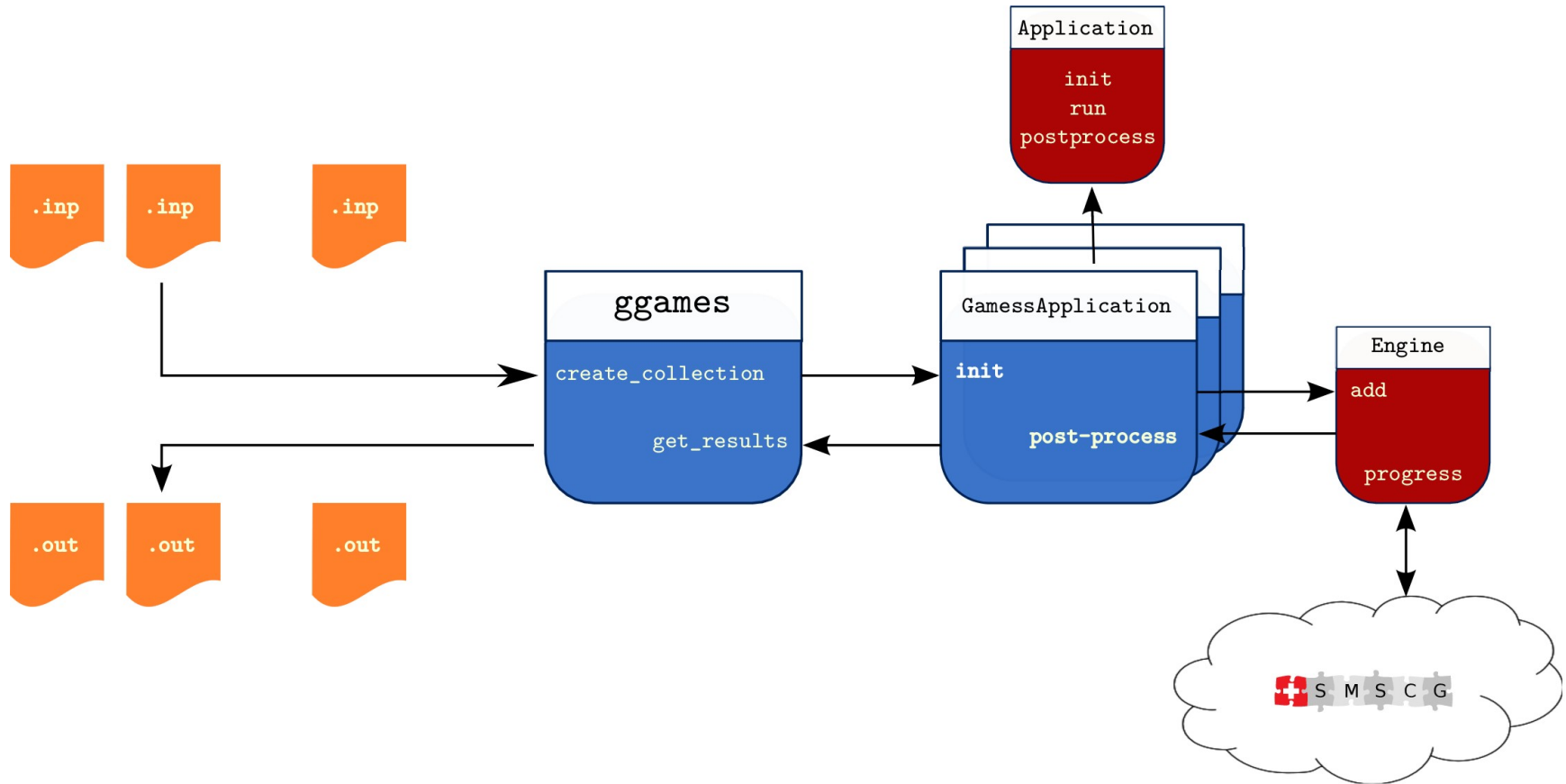
1. Create a `gc3libs.core.Engine` instance and load saved jobs into it
2. Create new instance(s) of the specialized `Application` class
3. Let engine `manage` jobs until all are done
4. Retrieve results (`the Engine does it`)
5. `Postprocess` and display

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

**ggames** mainly has been designed for a method to predict reaction pathways for molecules without usage of chemical knowledge (hypersphere method<sup>[1]</sup>)

**ggames** scans the specified **INPUT** directories recursively for '.inp' files, and submit a GAMESS job for **each input** file found; keeps a **record** of jobs (submitted, executed and pending) in a session file; job progress is **monitored** and, when a job is done, its '.out' and '.dat' file are **retrieved** back either to the submission host or to a gridFTP compliant storage service.

For this example, a single **ggames** session has generated DFT-energy calculation for **17713** molecules on the ARC-based SMSCG infrastructure (failure rate: 0.05%)



```
import gc3libs
from gc3libs.application.gamess import GamessApplication
from gc3libs.cmdline import SessionBasedScript

class GGameScript (SessionBasedScript):
    def __init__(self):
        SessionBasedScript.__init__(
            self,
            application = GamessApplication,
            input_filename_pattern = '*.inp'
        )

# run it
if __name__ == '__main__':
    GGameScript().run()
```

A yellow starburst callout with a black outline, containing text.

**Check  
yourself !**

<http://code.google.com/p/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.

- The unit of job composition is called a **Task** in GC3Libs.
- An Application is the primary **instance** of a Task.
- 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 then a task, too.



- 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 of tasks 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.

# How is GC3Pie different?

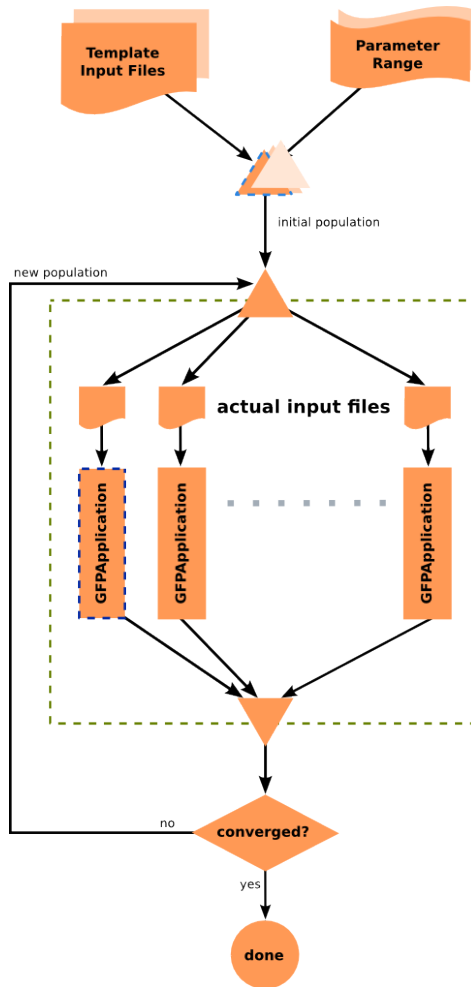
- Most execution engines represent workflows as data (e.g., some XML format). GC3Pie lets you write Python code instead: you write your workflow as a set of Python classes so the entire workflow logic is expressed in a plain programming language. This means that it is easy to create loops and conditionally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

GC3Pie helps manage the logistics of **running thousands of jobs** and collecting the results.

It abstracts away the small differences in grids & clusters so you can build workflows that utilize all your resources at once.

Created at the University of Zurich's Grid Computing Competency Center (GC3) <http://www.gc3.uzh.ch/>

Open-source, hosted at <http://code.google.com/p/gc3pie/>

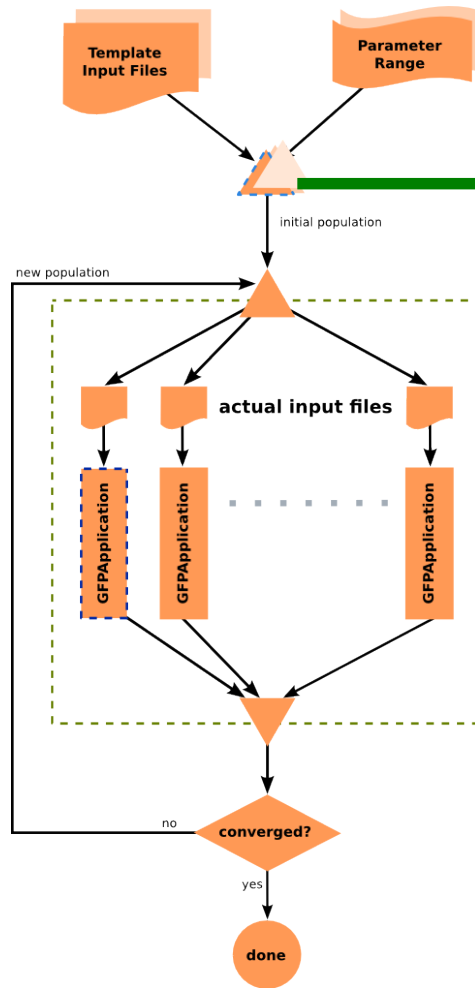


This workflow shows how a **differential evolution optimizer** is implemented with the GC3Pie library to support the analysis of a computationally intense **economic model**.

Each workflow execution spawns between **40000 and 80000 jobs** on the Swiss DCI infrastructure SMSCG (<http://www.smscg.ch>)

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

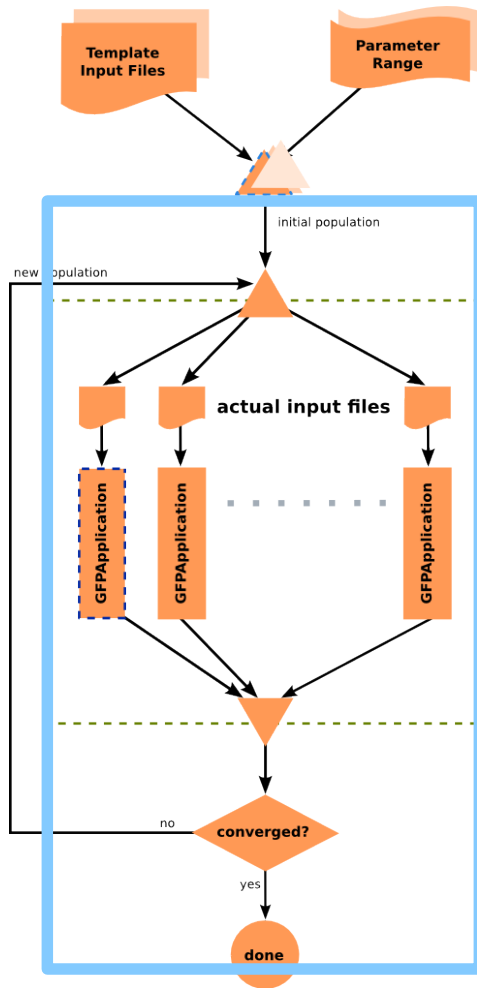
# a workflow example - step 1



```

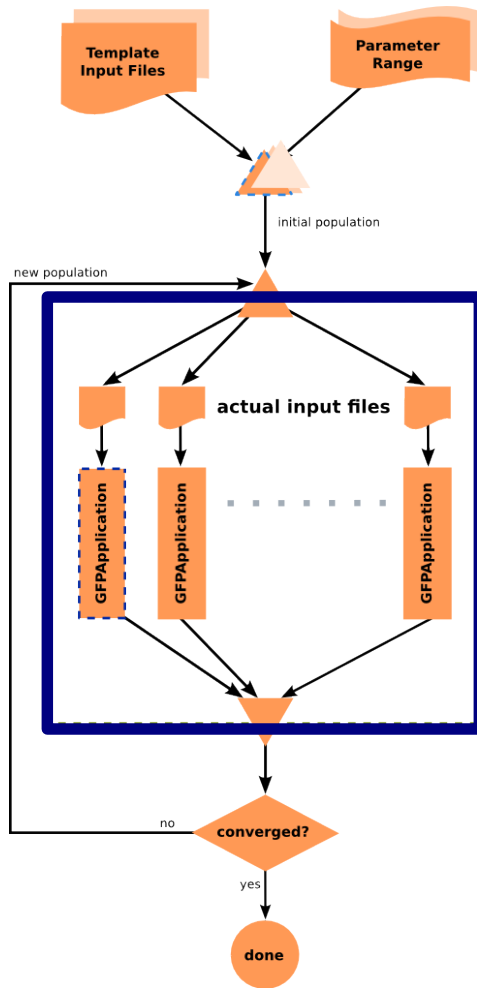
class GFPScript(SessionBasedScript):
    def new_tasks(self):
        for ctry1, ctry2 in self.country_pairs:
            # add tasks to the session
            yield (jobname, # unique identifier
                  GFPSequence # task class
                  [ args ], # creation arguments
           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

# a workflow example - step 2



```

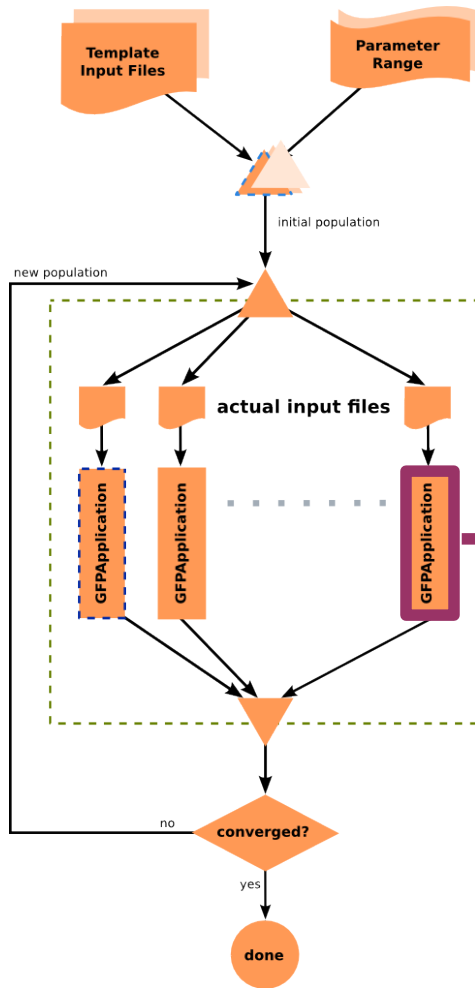
class GFPSequence(SequentialTaskCollection):
    def __init__(self, ...):
        SequentialTaskCollection.__init__(
            self, [ tasks ], ...)
    def next(self, done):
        if self.tasks[done].converged == True:
            return Run.State.TERMINATED
        else:
            # run another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



```

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

# a workflow example - step 4



```

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!
  
```