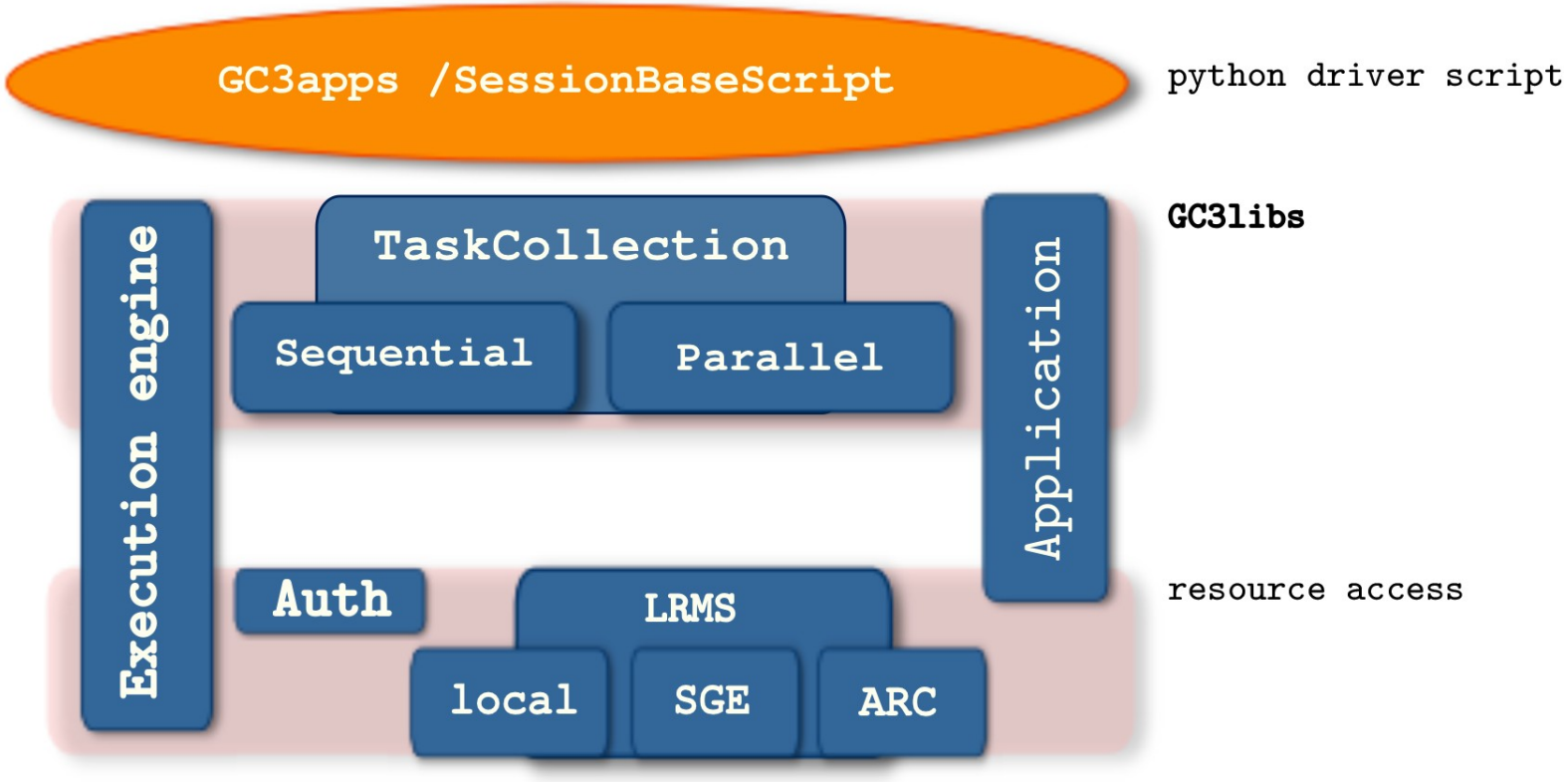
GC3Pie: a Python framework for high-throughput computing

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's components architecture



GC3Pie execution model

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

A typical high-throughput use case?

- 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.

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.
- Possible to 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.)

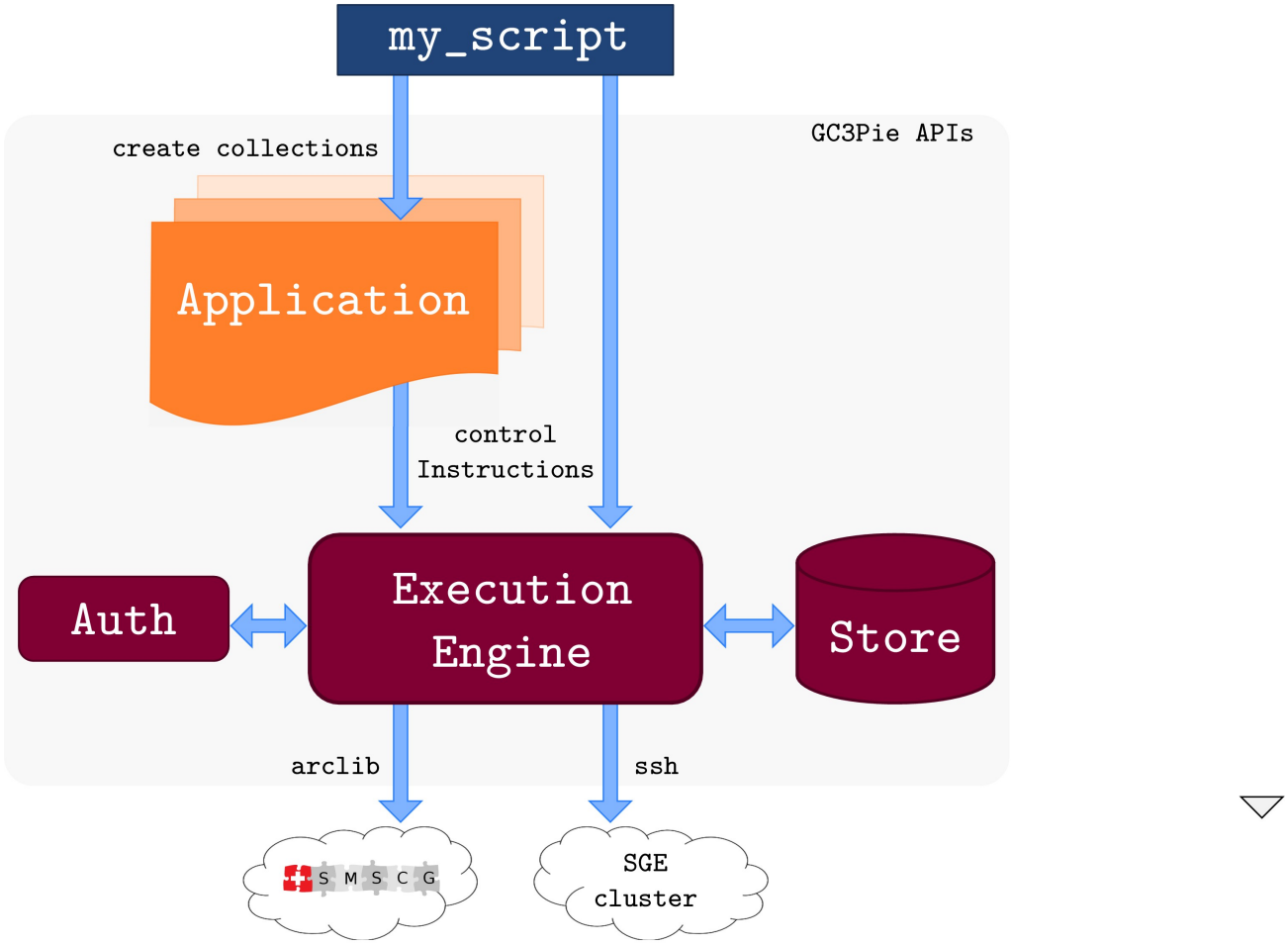
GamessApplication

```
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)
```

GamessApplication

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

GC3Pie's execution model



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 ?

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

How do I manage authentication ?

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

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

[resource/nor]
type = arc1
auth = smscg
...
```

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 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.

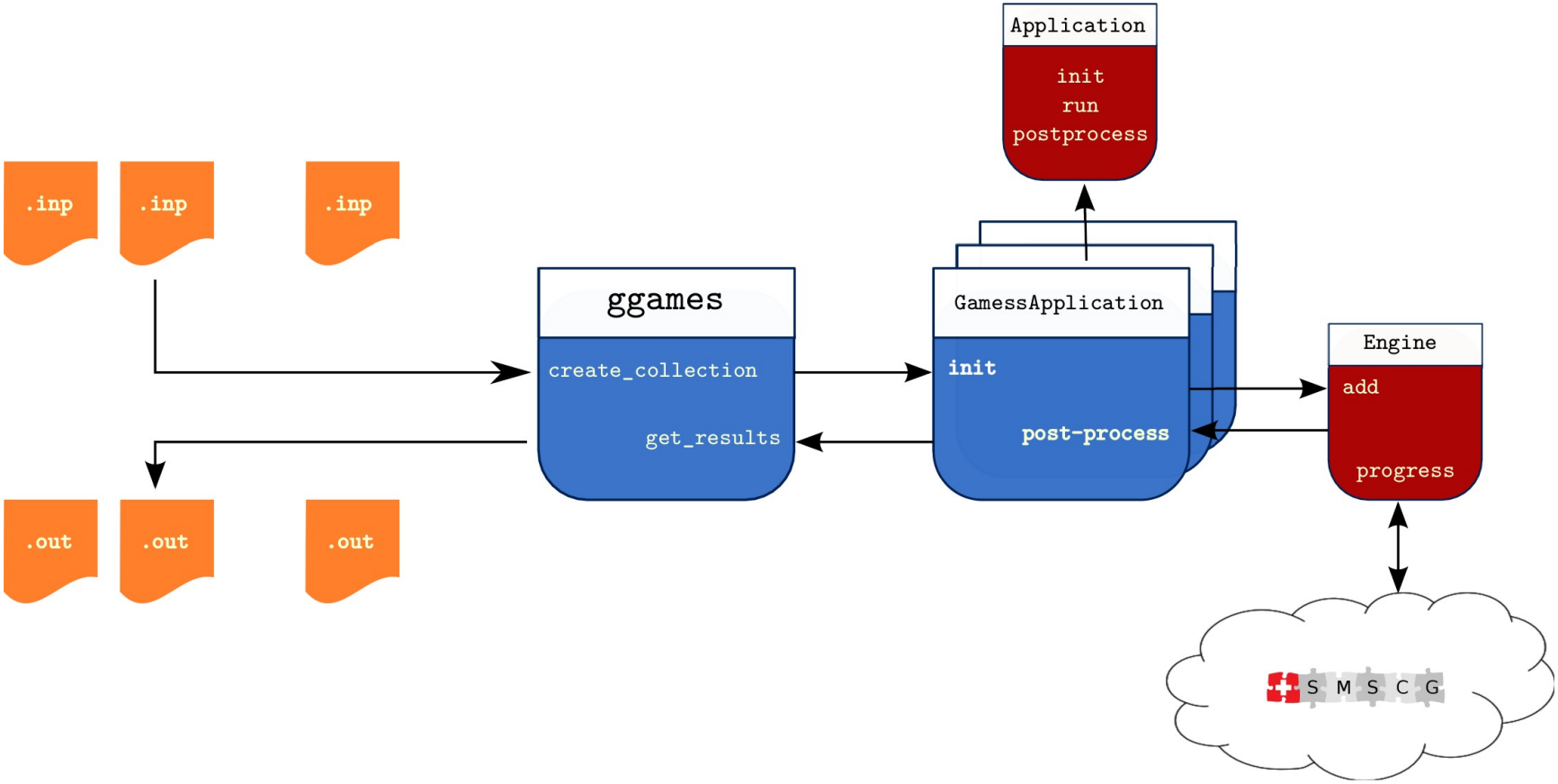
High-throughput GAMESS analysis: ggamess

ggamess mainly has been designed for a method to predict reaction pathways for molecules without usage of chemical knowledge (hypersphere method[1])

ggamess 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 ggamess session has generated DFT-energy calculation for **17713** molecules on the ARC-based SMSCG infrastructure (failure rate: 0.05%)

ggamess


Application
init
run
postprocess
.inp
.inp
.inp
ggames
create_collection
get_results
GamessApplication
init
post-process
Engine
add
progress
.out
.out
.out
S M S C G

ggamess.py

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


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

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

Supported Applications

System biology:

grosetta – 10000 jobs/run

Ecomony models:

gprepium – 40000 – 80000 jobs/run

Compchem:

cchem_gfit_abc_workflow – UML based

ggamess (Gamess) 17000 jobs/run

gricomp (Turbomole) not extensively tested yet

Institute of Research and Operation:

george – not extensively tested yet

Crypto:

gcrypto – 1000 jobs/run

Evolutionary Biology and Environmental Studies:

gcodeml – 10000 jobs/run

gmhc_coev (Matlab) – UML based

Geography

ggeotop – 1000 jobs/run

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.

Composition of tasks (I)

- 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.

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 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.

Conclusions

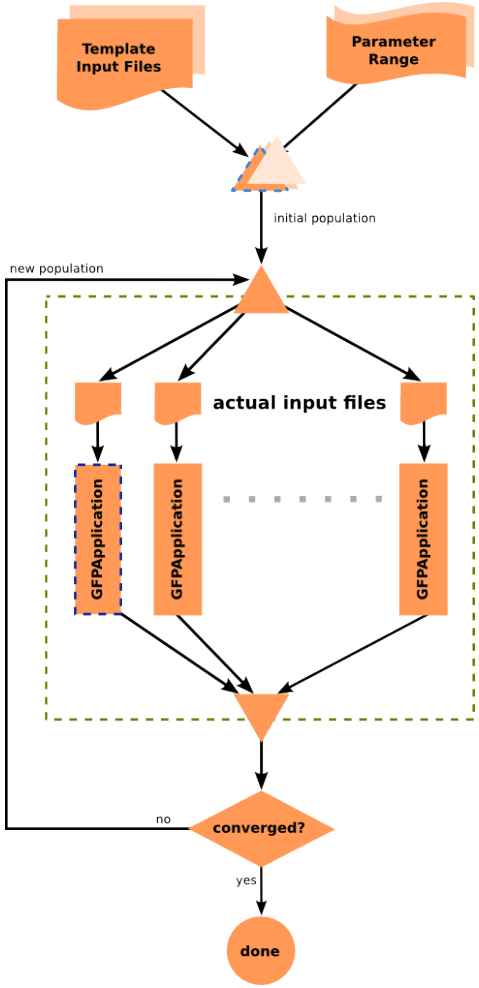
GC3Pie helps manage the logistics of running thousands of jobs and collecting the results.

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/

A workflow example

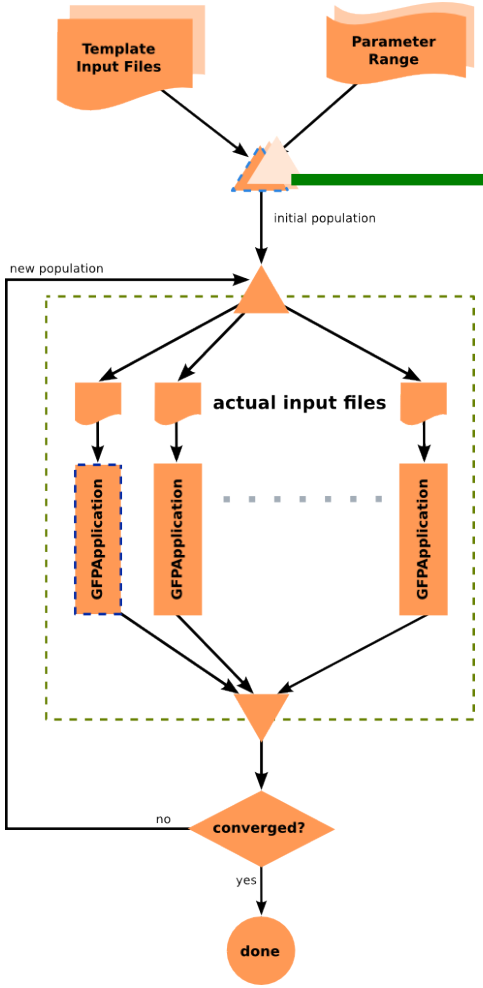


This workflow shows how a differential evolution optimizer is implemented with the GC3Pie library to support the analysis of a computationally intense economic model.

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

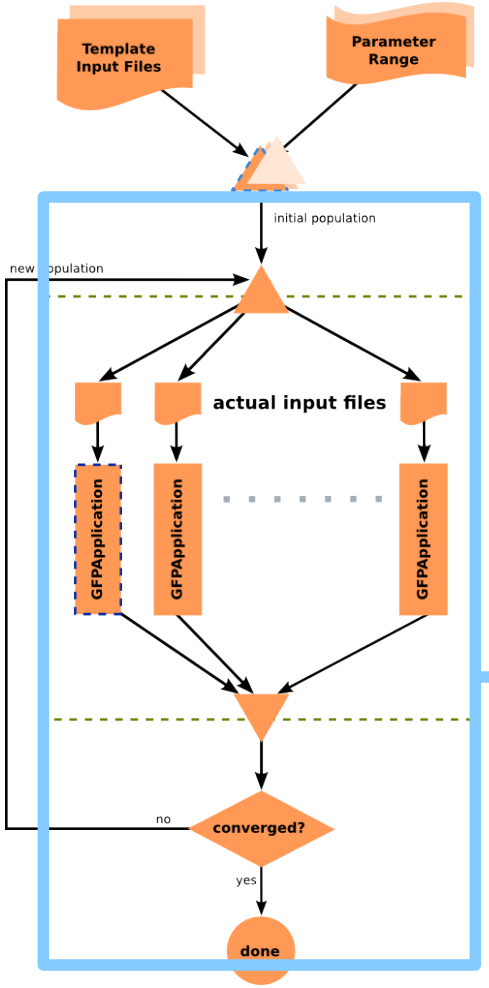
Benjamin Jonen, Simon Scheuring,
Institut für Banking und Finance,
University of Zurich
http://www.ibf.uzh.ch/

a workflow example - step 1



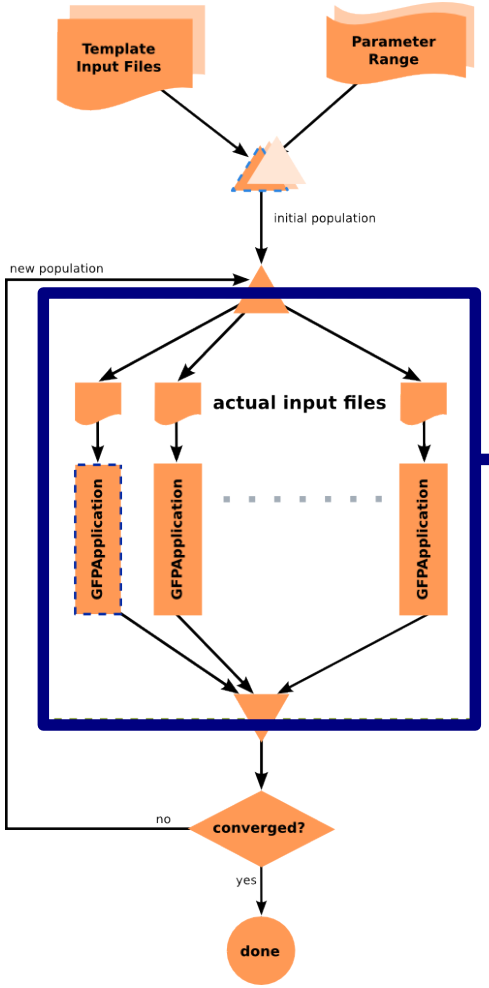

```
class GFPScript(SessionBasedScript):
  def new_tasks(self):
    for ctry1, ctry2 in self.country_pairs:
      # add tasks to the session
      yield (jobname,# unique identifier
             GFPSequence  # task class
             [ args ],  # creation arguments
             { kwargs })# creation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

a workflow example - step 2



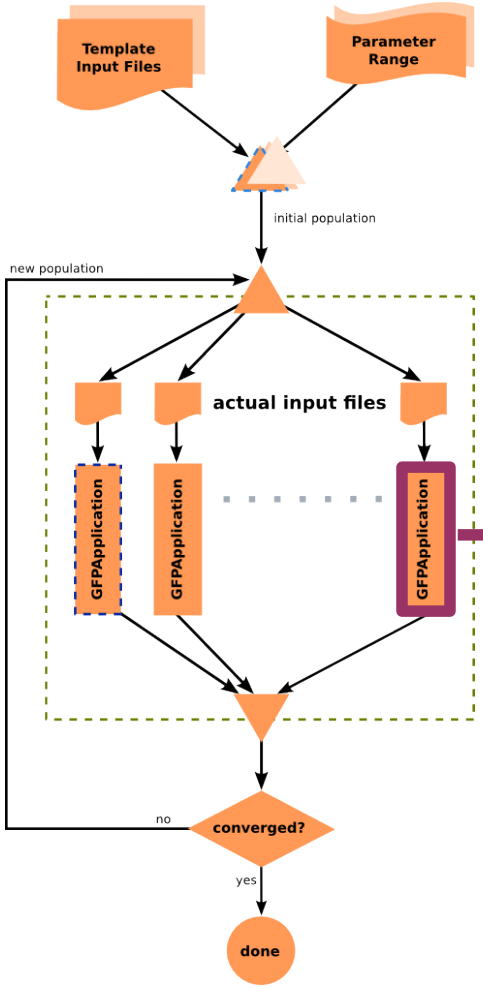

```
class GFPSequence(SequentialTaskCollection):
  def __init__(self, ...):
    SequentialsTaskCollection.__init__(
      self, [ tasks ], ...)
  def next(self, done):
    if self.tasks[done].converged == True:
      return Run.State.TERMINATED
    else:
      # run another optimization step,
      #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

a workflow example - step 3




```
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!
```

Chunked parallel execution

Cryptographic code to search within a large rangespace (800M – 2400M increment 1000)

Need to dilute the submission process to avoid infrastructure flood

Chunked parallel execution

Approach:

ParallelTaskCollection class that divide the overall parameter range into chunks

```
class ChunkedParameterSweep(ParallelTaskCollection):
def __init__(self, jobname, min_value, max_value, step,
   chunk_size, grid=None):
```

```
Every update cycle, the chunk is updated with new jobs
   (thus keeping the chunk size constant)
```

Chunked parallel execution

Approach:

ParallelTaskCollection class that divide the overall parameter range into chunks

```
class ChunkedParameterSweep(ParallelTaskCollection):
def __init__(self, jobname, min_value, max_value, step,
   chunk_size, grid=None):
```

```
Every update cycle, the chunk is updated with new jobs
   (thus keeping the chunk size constant)
```

Chunked parallel execution

Approach:

ParallelTaskCollection class that divide the overall parameter range into chunks

```
def new_task(self, param, **kw):
  """
  Create a new `CryptoApplication` for computing the range
  `param` to `param+self.parameter_count_increment`.
  """
 return CryptoApplication(
  param, self.step, self.gnfs_location,
   self.input_files_archive, self.output_folder, **kw)
```