Introduction to workflows using GC3Pie

Riccardo Murri <riccardo.murri@uzh.ch>
S3IT: Services and Support for Science IT
University of Zurich

June 8, 2017

What is GC3Pie?

GC3Pie is . . .

1. An *opinionated* Python framework for defining and running computational workflows;
2. A *rapid development toolkit* for running user applications on clusters and IaaS cloud resources;
3. The worst name ever given to a middleware piece. . .

As *users*, you're mostly interested in this part.

What is GC3Pie?

GC3Pie is . . .

1. An *opinionated* Python framework for defining and running computational workflows;
2. A *rapid development toolkit* for running user applications on clusters and IaaS cloud resources;
3. The worst name ever given to a middleware piece. . .

As *users*, you're mostly interested in this part.

Uses of GC3Pie

Uses of GC3Pie: parameter sweep

You have a simulation code that is dependent on a number of parameters.

Run the code for all possible combinations of parameters.

Then collect all the outputs and post-process to get a statistical overview.

Uses of GC3Pie: model calibration

You have a simulation code that is dependent on a number of parameters.

Run the code for all possible combinations of parameters, and find the ones that “best” approximate a given experimental result.

Uses of GC3Pie: parallel processing

Run the same program over and over again, feeding it different input files each time.

Then collect all the outputs and post-process to get a statistical overview.

Uses of GC3Pie: parallel processing

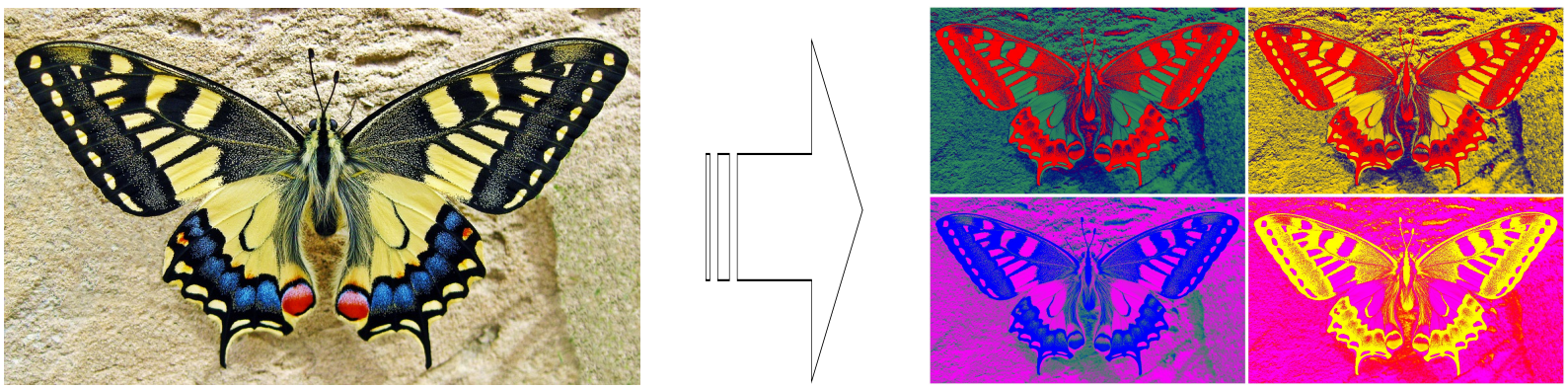
(At times, you chop a large input file into pieces and process each one separately instead.)

> *"For example, say we have a de novo assembly of 100,000 contigs. If we run 1 BLAST job against NR it could take as long as 50,000 minutes/35 days!! (30sec/query sequence), however if we split this job into subsets of 5,000 sequences and ran 20 jobs in "parallel" on a cluster, our total run-time is reduced to only 41 hours."*

Reference: http://sfg.stanford.edu/BLAST.html

Uses of GC3Pie: workflows

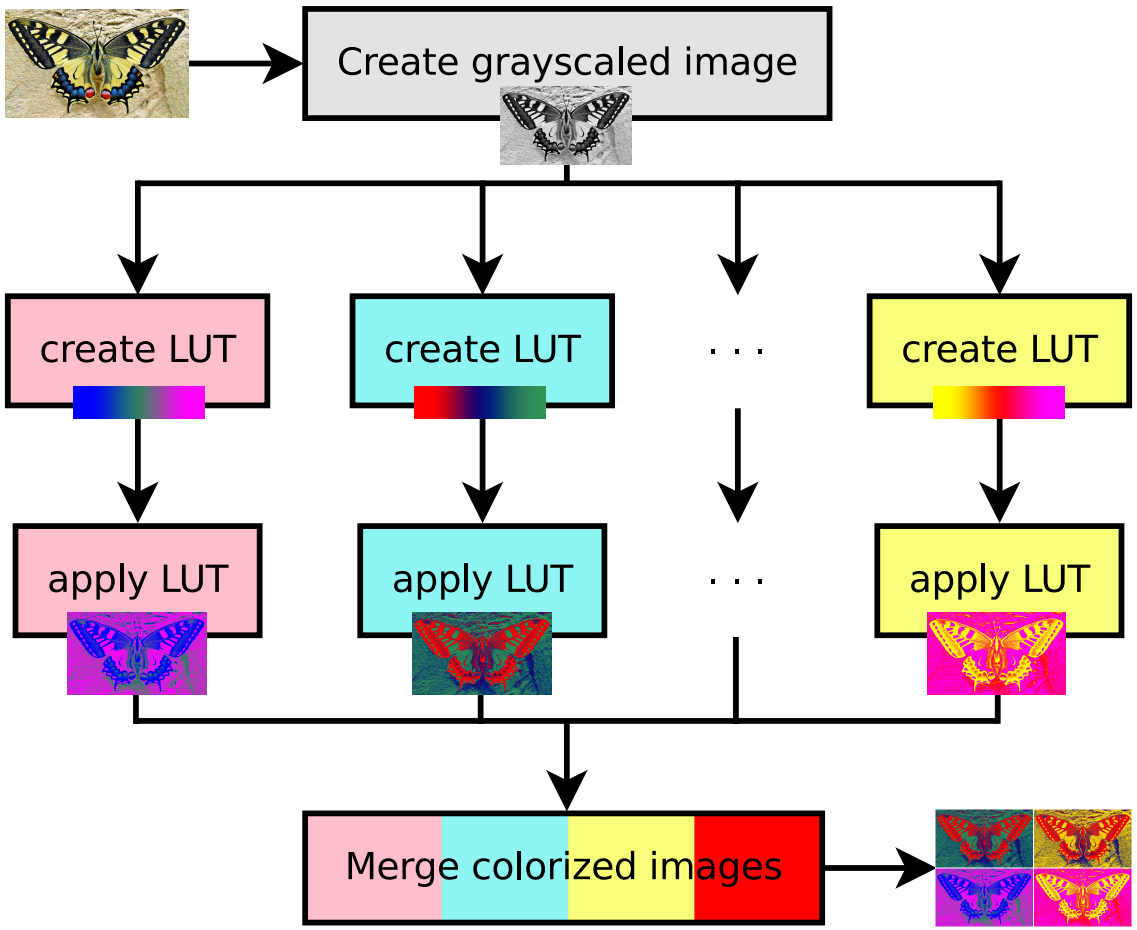
Orchestrate execution of several applications: some steps may run in parallel, some might need to be sequenced.



(Butterfly image originally from pixabay.com)

Uses of GC3Pie: workflows

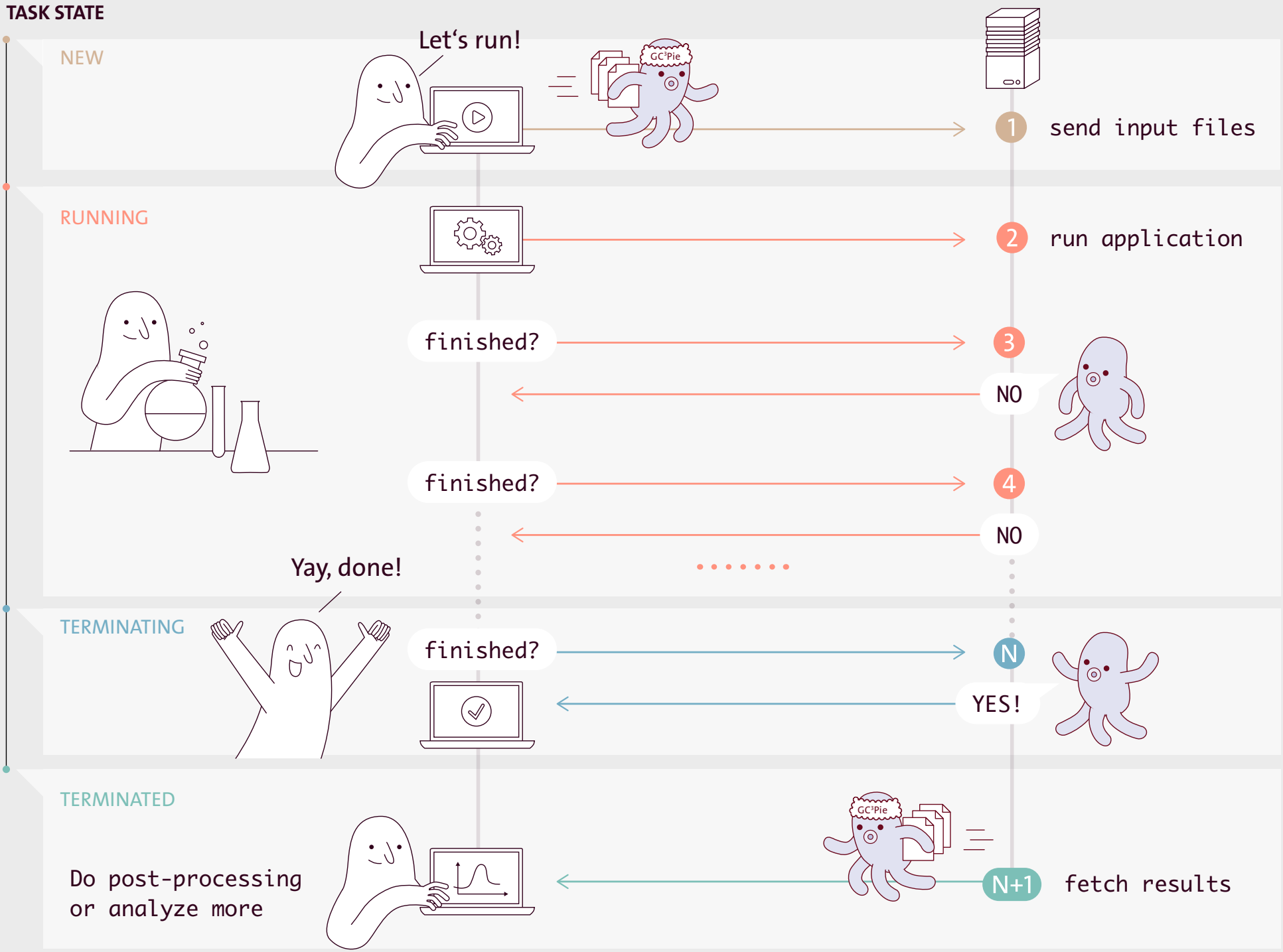
Orchestrate execution of several applications: some steps may run in parallel, some might need to be sequenced.

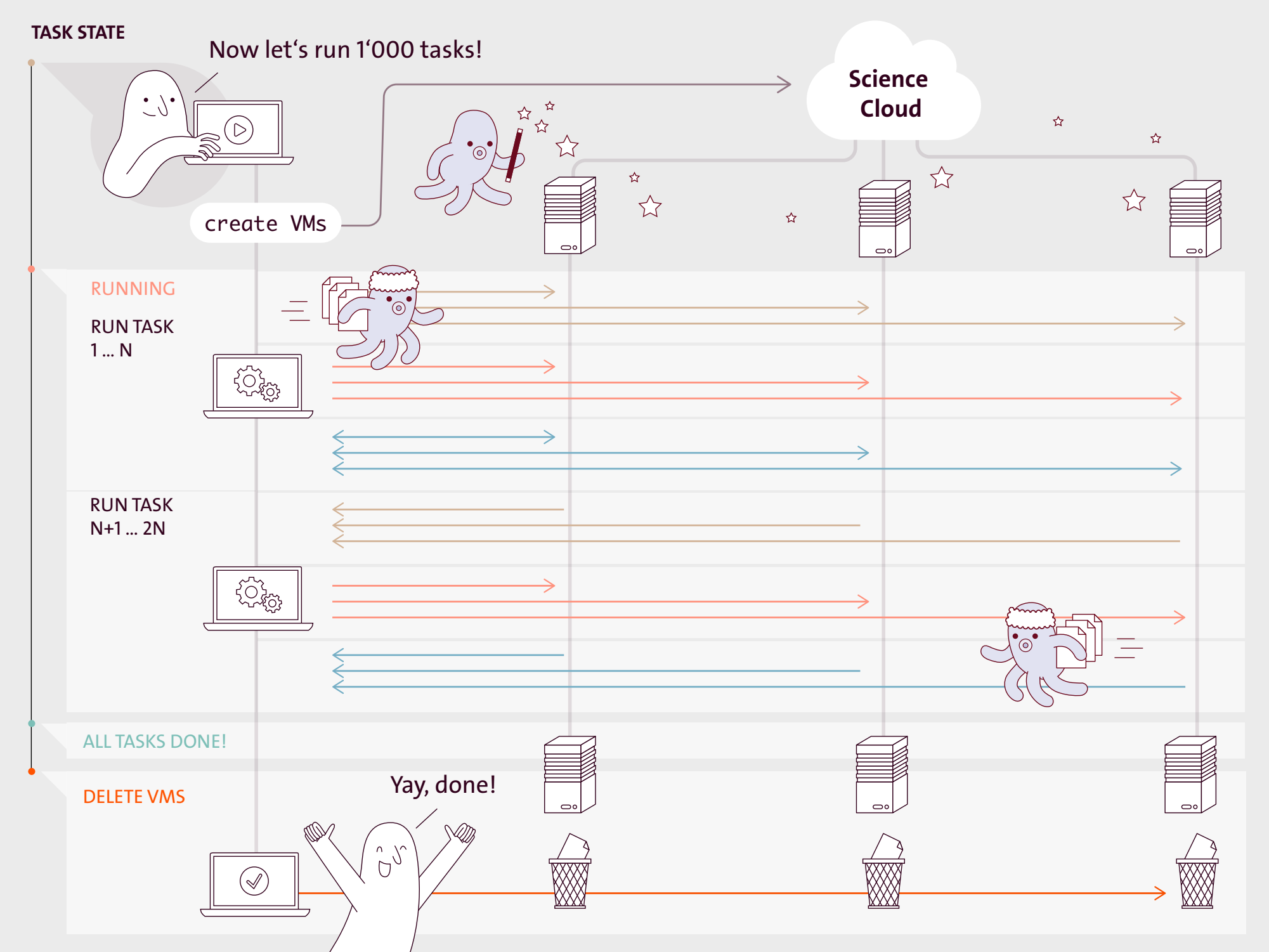


How GC3Pie works

A typical high-throughput script structure

1. Initialize computational resources
2. Prepare programs and inputs for submission
3. Submit tasks
4. Monitor task status (loop)
5. Retrieve results
6. Postprocess and display


TASK STATE
NEW
Let's run!
GC³Pie
1
send input files
RUNNING
2
run application
finished?
3
NO
finished?
4
NO
Yay, done!
TERMINATING
finished?
N
YES!
TERMINATED
Do post-processing
or analyze more
GC³Pie
N+1
fetch results


TASK STATE
Now let‘s run 1‘000 tasks!
Science Cloud
create VMs
RUNNING
RUN TASK
1 ... N
RUN TASK
N+1 ... 2N
ALL TASKS DONE!
DELETE VMS
Yay, done!

What GC3Pie handles for you

1. Resource allocation (e.g. starting new instances on ScienceCloud)
2. Selection of resources for each application in the session
3. Data transfer (e.g. copying input files in the new instances)
4. Remote execution of the application
5. Retrieval of results (e.g. copying output files from the running instance)
6. De-allocation of resources

Concepts and glossary

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

If you can run it in the terminal,
you can run it in GC3Pie.

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

A single execution of an `Application`
is indeed called a `Run`.

(Other systems might call this a "job".)

GC3Pie glossary: Task

GC3Pie runs user applications
on clusters and IaaS cloud resources

More generally, GC3Pie runs `Task`s.

`Task`s are a superset of applications,
in that they include workflows.

GC3Pie glossary: Resources

GC3Pie runs user applications
on clusters and IaaS cloud resources

`Resource`s are the computing infrastructures where GC3Pie executes applications.

Resources may include: your laptop, the "Ubelix" cluster, SWITCHengines, Amazon EC2.

Workflow scaffolding

Let's start coding!

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex10a
  ex10a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    apps_to_run = [ ]
    return apps_to_run
```

Download this code into a file named `ex10a.py`

Open it in your favorite text editor.

Exercise 10.A:

Download this code into a file named `ex10a.py`

1. Run the following command:

   ```
   $ python ex10a.py --help
   ```

   Where does the program description in the help text come from? Is there anything weird in other parts of the help text?

2. Run the following command:

   ```
   $ python ex10a.py
   ```

   What happens?

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
   import ex10a
   ex10a.AScript().run()

class AScript(SessionBasedScript):
   """
   Minimal workflow scaffolding.
   """
   def __init__(self):
     super(AScript, self).__init__(
         version='1.0')
   def new_tasks(self, extra):
     apps_to_run = [ ]
     return apps_to_run
```

These lines are needed in every session-based script.

See issue 95 for details.

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex10a
  ex10a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    apps_to_run = [ ]
    return apps_to_run
```

For this to work, it is **needed** that this is the actual file name.

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex10a
  ex10a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    apps_to_run = [ ]
    return apps_to_run
```

This is the program's help text!

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex10a
  ex10a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
      version='1.0' )
  def new_tasks(self, extra):
    apps_to_run = [ ]
    return apps_to_run
```

A version number is **mandatory**.

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex10a
  ex10a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    apps_to_run = [ ]
    return apps_to_run
```

This is the core of the script.

Return a list of `Application` objects, that GC3Pie will execute.

The `Application` object

Specifying commands to run, I

You need to "describe" an application to GC3Pie, in order for GC3Pie to use it.

This "description" is a blueprint from which many actual command instances can be created.

(A few such "descriptions" are already part of the core library.)

GC3Pie application model

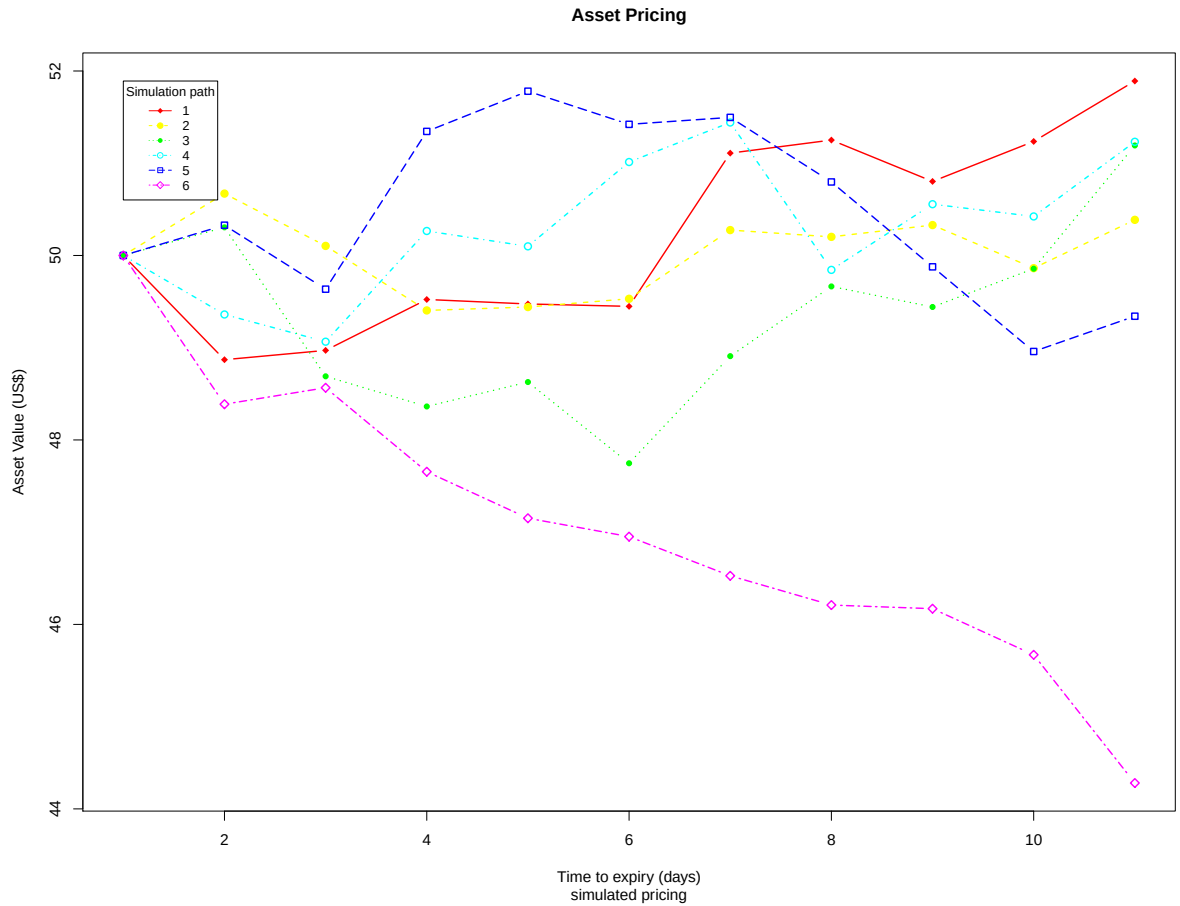
In GC3Pie, an application "description" is an object of the `gc3libs.Application` class (or subclasses thereof).

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.)

Detour: asset pricing, I

The script `simAsset.R` simulates asset pricing over a certain amount of time. Different pricing paths are generated using a 1D Brownian motion, all starting from the same initial price.



Detour: asset pricing, II

You can run the `simAsset.R` script by giving it one single input file in CSV format. For example:

```
$ Rscript simAsset.R simAsset.dat
```

Each run of `simAsset.R` script produces two output files *per each input row*:

result*K*.csv	table of generated data: each column is a simulation path, each row is a time step;
result*K*.pdf	plot of the above.

(You can download a sample "`simAsset1.dat`" input file from this URL.)

Detour: asset pricing, III

Each line of the input file for `simAsset.R` contains the following fields, separated by a comma (‘,’):

- S_0 stock price today (e.g., 50)
- μ expected return (e.g., 0.04)
- σ volatility (e.g., 0.1)
- δ size of time steps (e.g., 0.273)
- e days to expiry (e.g., 1000)
- N number of simulation paths to generate

(You can download a sample “`simAsset1.dat`” input file from this URL.)

Running the asset pricing example, I

Here is how you would tell GC3Pie
to run that command-line.

```
from gc3libs import Application

class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

Always inherit from Application

Your application class must inherit from class `gc3libs.Application`

```
from gc3libs import Application

class PricingApp1 (Application) :
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `arguments` parameter, I

The `arguments=` parameter is the actual command-line to be invoked.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        "Rscript", "simAsset.R", inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `arguments` parameter, II

The first item in the `arguments` list is the name or path to the command to run.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `arguments` parameter, III

The rest of the list are arguments to the program, as you would type them at the shell prompt.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `inputs` parameter, I

The `inputs` parameter holds a list of files that you want to *copy* to the location where the command is executed. (Remember: this might be a remote computer!)

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', inp],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `inputs` parameter, II

Input files retain their name during the copy,
but not the entire path.

For example:

```
inputs = [
  '/home/rmurri/db01/values.dat',
  '/home/rmurri/stats.csv',
]
```

will make files *values.dat* and *stats.csv* available in the command execution directory.

The `inputs` parameter, III

You need to pass the full path name into the `inputs` list, but use only the "base name" in the command invocation.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `outputs` parameter, I

The `outputs` argument list files that should be copied from the command execution directory back to your computer.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The `outputs` parameter, II

Output file names are *relative to the execution directory*. For example:

```
outputs = ['result.dat', 'program.log']
```

(Contrast with input files, which must be specified by *absolute path*, e.g., `/home/rmurri/values.dat`)

Any file with the given name that is found in the execution directory will be copied back. (*Where?* See next slides!)

If an output file is *not* found, this is *not* an error. In other words, **output files are optional**.

The `output_dir` parameter, I

The `output_dir` parameter specifies where output filess will be downloaded.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
```

The *output_dir* parameter, II

By default, GC3Pie does not overwrite an existing output directory: it will move the existing one to a backup name.

So, if `pricing1.d` already exists, GC3Pie will:

1. rename it to `pricing1.d.~1~`
2. create a new directory `pricing1.d`
3. download output files into the new directory

The `stdout` parameter

This specifies that the command's *standard output* should be saved into a file named `stdout.txt` and retrieved along with the other output files.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout="stdout.txt" )
```

(The `stderr` parameter)

There's a corresponding `stderr` option for the command's *standard error* stream.

```
class PricingApp1(Application):
  """Simulate asset pricing."""
  def __init__(self, infile):
    inp = basename(infile)
    Application.__init__(
      self,
      arguments=[
        'Rscript', 'simAsset.R', inp],
      inputs=['simAsset.R', infile],
      outputs=['result1.csv', 'result1.pdf'],
      output_dir='pricing1.d',
      stdout='stdout.txt')
      stderr="stderr.txt" )
```

Mixing `stdout` and `stderr` capture

You can specify **either one** of the `stdout` and `stderr` parameters, **or both**.

If you give both, and they have the same value, then `stdout` and `stderr` will be intermixed just as they are in normal screen output.

Let's run!

In order for a session-based script to execute something, its `new_tasks()` method must return a list of `Application` objects to run.

```
class AScript(SessionBasedScript):
  # ...
  def new_tasks(self, extra):
    # 'self.param.args' is the list
    # of command-line arguments
    input_file = self.params.args[0]
    app = PricingApp1(input_file)
    return [app]
```

Exercise 10.B:

Edit the `ex10a.py` file: insert the code to define the `PricingApp1` application, and modify the `new_tasks()` method to return one instance of it (as in the previous slide).

Can you process the `simAsset1.dat` file using this GC3Pie script?

(You can download the code for `PricingApp1` and the "`simAsset1.dat`" file from this URL.)

Exercise 10.C: What happens if you omit the `result1.pdf` file from the `outputs=[...]` line? And if you omit the `result1.csv` one?

Exercise 10.D:
Edit the script from Exercise 10.B above and add the ability to process multiple input lines: for each row of the file given on the command line, an instance of `ProcessingApp1` should be run. Test this with the `simAsset2.dat`

Exercise 10.E: Edit the script from Exercise 10.B above and add the ability to process multiple input files: for each file name given on the command line, an instance of `ProcessingApp1` should be run.

Resource definition

The `gservers` command

The `gservers` command is used to see configured and available resources.

```
$ gservers
+--------------------+--------------------------+-----------+
|                    | localhost                |           |
+--------------------+--------------------------+-----------+
|            frontend | ( Frontend host name )   | localhost |
|                type | ( Access mode )          | shellcmd  |
|             updated | ( Accessible? )          | True      |
|              queued | ( Total queued jobs )    | 0         |
|         user_queued | ( Own queued jobs )      | 0         |
|            user_run | ( Own running jobs )     | 6         |
|   max_cores_per_job | ( Max cores per job )    | 4         |
| max_memory_per_core | ( Max memory per core )  | 8GiB      |
|        max_walltime | ( Max walltime per job ) | 8hour     |
+--------------------+--------------------------+-----------+
```

Resources are defined in file `$HOME/.gc3/gc3pie.conf`

The `gservers` command

The `gservers` command is used to see configured and available resources.

```
$ gservers
+---------------------+--------------------------+-----------+
|                     | localhost                |           |
+---------------------+--------------------------+-----------+
|            frontend | ( Frontend host name )   | localhost |
|                type | ( Access mode )          | shellcmd  |
|             updated | ( Accessible? )          | True      |
|              queued | ( Total queued jobs )    | 0         |
|         user_queued | ( Own queued jobs )      | 0         |
|            user_run | ( Own running jobs )     | 6         |
|   max_cores_per_job | ( Max cores per job )    | 4         |
| max_memory_per_core | ( Max memory per core )  | 8GiB      |
|        max_walltime | ( Max walltime per job ) | 8hour     |
+---------------------+--------------------------+-----------+
```

Resources are defined in file `$HOME/.gc3/gc3pie.conf`

Example execution resources: local host

Allow GC3Pie to run tasks on the local computer.

This is the default installed by GC3Pie into `$HOME/.gc3/gc3pie.conf`

```
[resource/localhost]
enabled = yes
type = shellcmd
frontend = localhost
transport = local
max_cores_per_job = 2
max_memory_per_core = 2GiB
max_walltime = 8 hours
max_cores = 2
architecture = x86_64
auth = none
override = no
```

Example execution resources: SLURM

Allow submission of jobs to the “Hydra” cluster.

```
[resource/hydra]
enabled = no
type = slurm
frontend = login.s3it.uzh.ch
transport = ssh
auth = ssh_user_rmurri
max_walltime = 1 day
max_cores = 96
max_cores_per_job = 64
max_memory_per_core = 1 TiB
architecture = x86_64
prologue_content =
  module load cluster/largemem

[auth/ssh_user_rmurri]
type=ssh
username=rmurri
```

Example execution resources: OpenStack

```
[resource/sciencecloud]
enabled=no
type=openstack+shellcmd
auth=openstack

vm_pool_max_size = 32
security_group_name=default
security_group_rules=
  tcp:22:22:0.0.0.0/0,
  icmp:-1:-1:0.0.0.0/0
network_ids=
  c86b320c-9542-4032-a951-c8a068894cc2

# definition of a single execution VM
instance_type=1cpu-4ram-hpc
image_id=2b227d15-8f6a-42b0-b744-ede52ebe59f7

max_cores_per_job = 8
max_memory_per_core = 4 GiB
max_walltime = 90 days
max_cores = 32
architecture = x86_64

# how to connect
vm_auth=ssh_user_ubuntu
keypair_name=rmurri
public_key=~/.ssh/id_dsa.pub
```

```
[auth/ssh_user_ubuntu]
# default user on Ubuntu VM images
type=ssh
username=ubuntu


[auth/openstack]
# only need to set the `type` here;
# any other value will be taken from
# the `OS_*` environment variables
type = openstack
```

Allow running tasks on the "ScienceCloud" VM infrastructure.

Select execution resource

When multiple resources are available, you can select where applications will be run with option `-resource/-r`:

```
$ ./script.py -r localhost
```

The resource name must exists in the configuration file (i.e., check `gservers`' output).

Stopping a script and re-starting it with a different resource will likely result in an error: old tasks can no longer be found.

Post-processing

Post-processing features, I

When the remote computation is done, the `terminated` method of the application instance is called.

The path to the output directory is available as `self.output_dir`.

If `stdout` and `stderr` have been captured, the **relative** paths to the capture files are available as `self.stdout` and `self.stderr`.

Post-processing features, II

For example, the following code logs a warning message if the standard error output is non-empty:

```
class MyApp(Application):
  # ...
  def terminated(self):
    stderr_file = self.output_dir+"/"+self.stderr
    stderr_size = os.stat(stderr_file).st_size
    if stderr_size > 0:
      gc3libs.log.warn(
        "Application %s reported errors!", self)
```

Useful in post-processing

These attributes are available in the `terminated()` method:

`self.inputs`
Python dictionary, mapping local (absolute) paths to remote paths (relative to execution directory)

`self.outputs`
Python dictionary, mapping remote paths (relative to execution directory) to *URLs* where they have been copied. In particular, `self.outputs.keys()` is the list of output file names.

`self.output_dir`
Path to the local directory where output files have been downloaded.

Exercise 10.F: Modify class `PricingApp1` so that, when the task is done, the `result1.csv` file is read and the average final price is computed and printed to the screen together with the initial price.

You might need to check for the existence of the output file, and that it contains the correct output: not all tasks are successful!

Exercise 10.G: Bonus points if you can, instead of printing the initial and final prices, save them to a file `pricing_totals.csv`.

Global post-processing, I

In order to do "global" post-processing (i.e., across `Application` objects), you need to hook into the GC3Pie script's main loop:

`after_main_loop(self)`
to execute some code *after* the main loop, i.e., before the script quits. A list of all Application objects is available in the `self.session.tasks.values()` list.

Global post-processing, II

Example: compute statistical distribution of termination statuses:

```
def after_main_loop(self):
  # check that all tasks are terminated
  can_postprocess = True
  for task in self.session.tasks.values():
    if task.execution.state != 'TERMINATED':
      can_postprocess = False
      break
  if can_postprocess:
    # do stuff...  (see next slide)
```

Global post-processing, III

Example: compute statistical distribution of termination statuses (cont'd):

```
def after_main_loop(self):
  # ...  (see prev slide)
  if can_postprocess:
    status_counts = defaultdict(int)
    for app in self.session.tasks.values():
      termstatus = app.execution.returncode
      status_counts[termstatus] += 1
```

Variable `self.session.tasks` holds a mapping `JobID` ⇒ `Application`; thus `self.session.tasks.values()` is a list of all the `Application` instances returned by `new_tasks`

Exercise 10.H: Modify the script from the last exercise to produce a scatter plot of the initial and average final prices as a final post-processing step when everything else is done.

Application requirements

Applications need to allocate computing resources.
For instance: request 4 processors for 8 hours.

GC3Pie allows requesting:

- the number of processors that a job can use,
- the architecture (32-bit or 64-bit) of these processors,
- the guaranteed duration of a job,
- the amount of memory that a job can use (per processor).

More fine-grained matching is possible, but outside the scope of this introductory training.

Resources are requested using additional constructor parameters for `Application` objects.

The allowed parameters are: `requested_cores`, `requested_architecture`, `requested_walltime`, `requested_memory`.

Running parallel jobs

You request allocation of a certain number of processors using the `requested_cores` parameter: set it to the number of CPU cores that you want.

For example, the following runs the command `mpixexec simulator` on 4 processors:

```
class ZodsApplication(Application):
  # ...
  Application.__init__(self,
    arguments=['mpiexec', '-n', '4', 'simulator'],
    # ...
    requested_cores=4 )
```

Note that GC3Pie only guarantees the availability of a certain number of processors; it is your application's responsibility to use them, e.g., by starting a command using MPI or any other parallel processing mechanism.

Requesting processor architecture

If you send the compiled executable along with your application, you need to select only resources that can run that binary file.

The `requested_architecture` parameter provides the choice between `gc3libs.Run.Arch.X86_64` (for 64-bit Intel/AMD computers) and `gc3libs.Run.Arch.X86_32` (for 32-bit ones):

```
from gc3libs import Run
class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    arguments=['./codeml.bin'],
    inputs = ['/usr/local/bin/codeml', ...]
    # ...
    requested_architecture=Run.Arch.X86_64 )
```

Requesting running time

In order to ensure that your job is allotted enough time to run on the remote computing system, use the `requested_walltime` parameter.

```
from gc3libs.quantity \
  import minutes, hours, days

class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    # ...
    requested_walltime=8*hours )
```

You **must** use a `gc3libs.quantity` multiple for the `requested_walltime` parameter; any other value will be rejected with an error.

Units of time

The Python module `gc3libs.quantity` provides units for expressing time requirements in days, hours, minutes, seconds.

Just multiply the unit by the amount you need:

```
>>> an_hour = 1*hours
```

Or sum the amounts:

```
>>> two_days = 1*days + 24*hours
```

GC3Pie will automatically perform the conversions:

```
>>> two_hours = 2*hours
>>> another_two_hours = 7200*seconds
>>> two_hours == another_two_hours
True
```

Requesting memory

In order to secure a certain amount of memory for a job, use the `requested_memory` parameter.

Example:

```
from gc3libs.quantity import GB, MB, kB
class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    # ...
    requested_memory=8*GB )
```

Note that `requested_memory` expresses the total memory used by the job!

Units of memory

The Python module `gc3libs.quantity` provides units for expressing memory requirements in kilo-, Mega- and Giga-bytes.

Just multiply the unit by the amount you need:

```
>>> a_gigabyte = 1*GB
>>> two_megabytes = 2*MB
```

GC3Pie will automatically perform the conversions:

```
>>> two_gigabytes = 2*GB
>>> another_two_gbs = 2000*MB
>>> two_gigabytes == another_two_gbs
True
```

All together now

```
from gc3libs.quantity import GB, MB, kB
from gc3libs.quantity import days, hours, minutes

class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    # ...
    requested_cores=1,
    requested_memory=2*GB,
    requested_walltime=8*hours)
```

When several resource requirements are specified, GC3Pie tries to satisfy *all* of them. **If this is not possible, task submission fails and the task stays in state *NEW*.**