More on workflows

Riccardo Murri <riccardo.murri@uzh.ch>

S3IT: Services and Support for Science IT

University of Zurich

July 11–14, 2016

Automatic arrangement of tasks

Want to avoid arranging tasks in parallel- and sequential- task collections? Use a `DepedentTaskCollection`!

```
from gc3libs.workflow \
  import DependentTaskCollection

class MyWorkflow(DependentTaskCollection):
  # ...
  def __init__(self, ...):
    DependentTaskCollection.__init__(self)
    app1 = AnApp(...)
    app2 = AnotherApp(...)
    app3 = AThirdApp(...)
    self.add(app1)
    self.add(app2)
    self.add(app3, after=[app1, app2])
```

Usage of DependentTaskCollection

```python
from gc3libs.workflow \
  import DependentTaskCollection

class MyWorkflow(DependentTaskCollection):
  # ...
  def __init__(self, ...):
    DependentTaskCollection.__init__(self)
    app1 = AnApp(...)
    app2 = AnotherApp(...)
    app3 = AThirdApp(...)
    self.add(app1)
    self.add(app2)
    self.add(app3, after=[app1, app2])
```

Initialize the base class.

Usage of DependentTaskCollection

```
from gc3libs.workflow \
  import DependentTaskCollection

class MyWorkflow(DependentTaskCollection):
  # ...
  def __init__(self, ...):
    DependentTaskCollection.__init__(self)
    app1 = AnApp(...)
    app2 = AnotherApp(...)
    app3 = AThirdApp(...)
    self.add(app1)
    self.add(app2)
    self.add(app3, after=[app1, app2])
```

. . . then initialize tasks that you want to run . . .

Usage of DependentTaskCollection

```
from gc3libs.workflow \
  import DependentTaskCollection

class MyWorkflow(DependentTaskCollection):
  # ...
  def __init__(self, ...):
    DependentTaskCollection.__init__(self)
    app1 = AnApp(...)
    app2 = AnotherApp(...)
    app3 = AThirdApp(...)
    self.add(app1)
    self.add(app2)
    self.add(app3, after=[app1, app2])
```

. . . then add tasks to the collection, one by one. . .

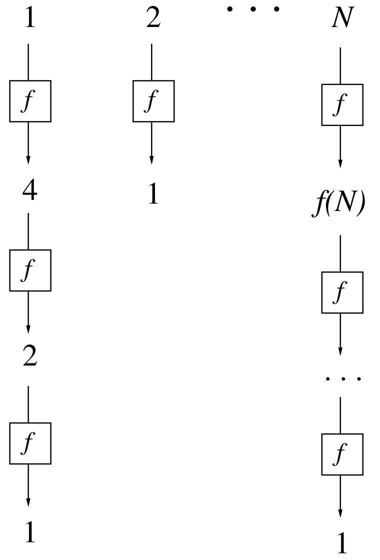
Usage of DependentTaskCollection

```
from gc3libs.workflow \
  import DependentTaskCollection

class MyWorkflow(DependentTaskCollection):
  # ...
  def __init__(self, ...):
    DependentTaskCollection.__init__(self)
    app1 = AnApp(...)
    app2 = AnotherApp(...)
    app3 = AThirdApp(...)
    self.add(app1)
    self.add(app2)
    self.add(app3, after=[app1, app2] )
```

. . . specifying dependencies among them.

The *3n+1* conjecture, a fictitious use case



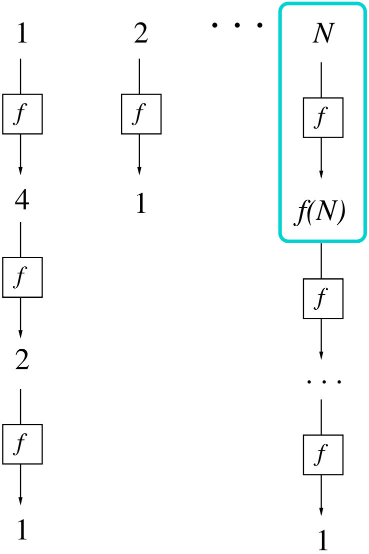
Define a function f, for n positive integer:

- if n is even, then $f(n) = n/2$,
- if n is odd, then $f(n) = 3n + 1$,

For every positive integer n, form the sequence $S(n)$: $n \to f(n) \to f(f(n)) \to f(f(f(n))) \to \ldots$

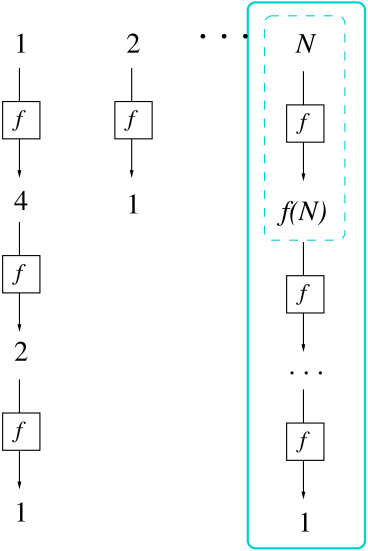
Conjecture: For every positive integer n, the sequence $S(n)$ eventually hits 1.

The *3n+1* conjecture, *(I)*



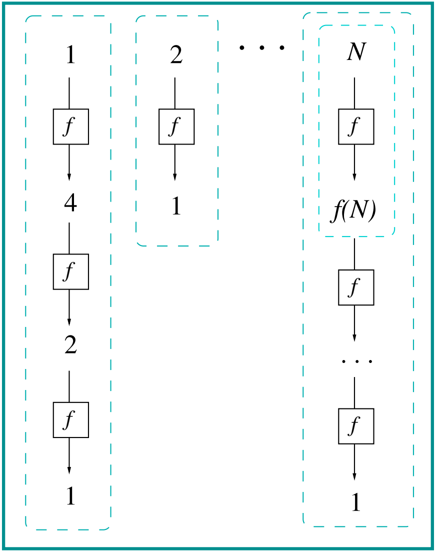
A computational job $J(n, k)$, applies function f to the result of $J(n, k)$.

The *3n+1* conjecture, *(II)*



A sequence $H(n)$ of jobs computes the chain $n \to f(n) \to \ldots \to 1$.

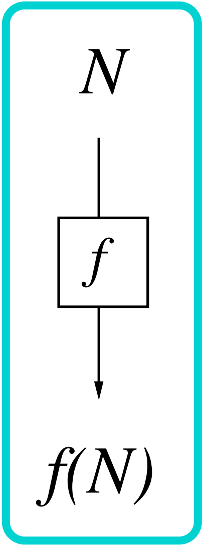
The *3n+1* conjecture, *(III)*



Run one sequence $H(n)$ per each $n = 1, \dots, N$.

The can all run in **parallel**.

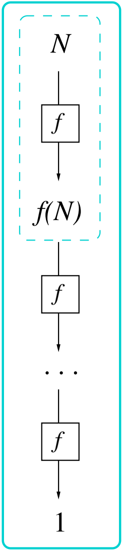
The *3n+1* conjecture (IV)



Let’s define the simple application that computes f:

```
class HotpoApplication(Application):
  def __init__(self, n):
    Application.__init__(
      self,
      executable = ’/usr/bin/expr’,
      arguments = (
          # run ‘expr n / 2‘ if n
is even
          [n, ’/’, n] if n % 2 == 0
          # run ‘expr 1 + 3 * n‘
if n is odd
          else [1, ’+’, 3, ’*’, n]),
      stdout = "stdout.txt",
    )
```

The *3n+1* conjecture (V)



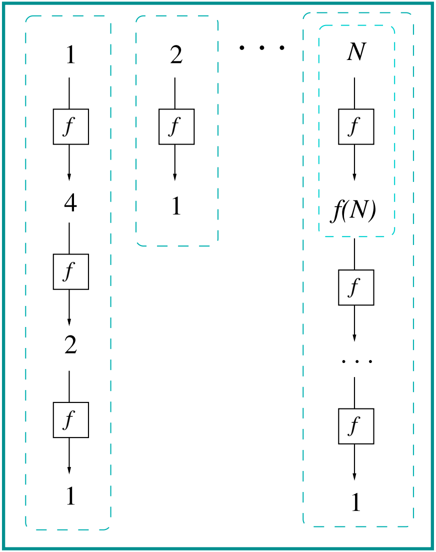
Now string together applications to compute a single sequence:

```
class HotpoSequence(SequentialTask):

  def __init__(self, n):
    # compute first iteration of f
    self.tasks = [ HotpoApplication(n) ]
    SequentialTask.__init__(self, self.task

  def next(self, k):
    last = self.tasks[k].result
    if last == 1:
      return TERMINATED
    else:
      self.tasks.append(MyApplication(last)
       return RUNNING
```

The *3n+1* conjecture (VI)



Parallel tasks are independent by definition, so it’s even easier to create a collection:

```
tasks =
  ParallelTaskCollection([
    HotpoSequence(n)
      for n in range(1, N) ]
```

We can run such a collection like any other `Task`.