Application requirements

GC3: Grid Computing Competence Center,
University of Zurich

Oct. 1, 2012

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,
    ['mpiexec', '-n', '4', 'simulator'],
    # ...
    @\HL{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).

Example:

```
class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    ['./codeml.bin'],
    inputs = ['/home/rmurri/selectome/codeml.bin',
    # ...
    @\HL{requested\_architecture=gc3libs.Run.Arch.X
```

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.

Example:

```
@\HL{\textbf{from} gc3libs.quantity \textbackslash}
@\HL{\textbf{import} days, hours, minutes, seconds}
class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    # ...
    @\HL{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
```

CPU time vs wall-clock time

"CPU time" is the total time spent by all CPUs in the system actually executing code from our job.

The "wall-clock time" (abbr. "walltime") is the time that passes on a clock from the moment the system starts executing a job until the end of that job.

Requesting memory

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

Example:

```
@\HL{\textbf{from} gc3libs.quantity \textbf{import}}
class CodemlApplication(Application):
  # ...
  Application.__init__(self,
    # ...
    @\HL{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, `core.submit(app)` fails and the job stays in state *NEW*.

Exercise A: Modify the `SquareApplication` in square.py so that it requests an impossibly high amount of memory. Re-run the script and watch it fail: the application should not be submitted and remain in state *NEW*.

Can you achieve the same result by using other requirement specifiers?