Compmake: Minimal Effort Parallelization and Job Management for Python Applications

Andrea Censi

Abstract—Compmake is a jobs-oriented Python library that provides Make-like functionality to Python applications. The user minimally modifies its program, so that now it describes the computation as a jobs graph. Once that is done, Compmake provides for free a series of features including parallelization, caching, failure tolerance, dependencies handling, job scheduling, and a console interface to inspect the results.

I. Introduction

Compmake is a Python library that provides Make [1]-like facilities for Python applications. To use Compmake, the user minimally modifies their Python script to turn function calls into “jobs” definition, so that Compmake can understand the processing layout and the opportunities for parallelization (Fig. 1).

Once this is done, Compmake provides several features:

- **Parallelization**: Jobs can be executed in parallel, both on a single host, on a cluster using SGE, and, experimentally, in the cloud using Multyvac.
- **Caching**: The computation can be interrupted (CTRL-C) and restarted without losing the results of the jobs already executed.
- **Selective computation**: The user can choose to execute only part of the jobs.
- **Failure tolerance**: If a job fails by throwing an exception, the error is recorded, and processing continues for jobs that do not depend on the failed one.
- **Console**: Familiar commands (make, clean, ls, etc.) are available for scheduling job processing and inspecting completions, failures, and resources usage.

Compmake’s design goal is to have the most functionality while imposing little burden on the programmer. Parallelization is the most obvious benefit, but not the only one. Compmake helps immensely the development process.

Compmake allows creating applications that are structured in “jobs” which create a “job graph” (Fig. 1b). The jobs graph is dynamic. Jobs that can depend on other jobs, and spawn other jobs recursively. Once the jobs are defined, there are several options for running them. They can be run serially or in parallel, either all in the same process or spawning a different process. Several backends for parallel processing are available, including single-machine multiprocessing, and cluster processing using the Sun Grid Engine (SGE) interface [2].

The author is with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, USA. E-mail: censi@mit.edu. This work was supported by the National Science Foundation’s National Robotics Initiative under grant #1405259.

Figure 1. The user minimally modifies the program, so that Compmake can understand its processing layout and the opportunities for parallelization. The same program can be run serially, in parallel using single-host multiprocessing, or on a cluster.
II. Why using Compmake

There are many reasons to use Compmake. Perhaps the most evident is the support for parallelization. This sections explains the other less obvious benefits of Compmake, especially during development.

Suppose that you are working on the simple program below:

```python
for param1 in [1, 2, 3]:
    for param2 in [10, 11, 12]:
        res1 = funcA(param1)
        res2 = funcB(res1, param2)
        draw(res2)
```

For every value of the parameters, there is an invocation of the functions `funcA()`, `funcB()`, followed by `draw()`, which depends on the results of the other two. The following are probable scenarios that occur during development in which Compmake can help.

Suppose you are working on improving the function `draw()` . You find yourself running the whole program again and again to check the results of that function. This is a waste because `funcA()` and `funcB()` did not change. It’s obvious that some caching mechanism is needed. You can implement one in different ways. You can explicitly dump the intermediate results to a file using pickling [3], then write code to read that file if it exists. But now your 5 lines program has become a buggy 30 lines program.

**Compmake helps by implementing automatic caching of the results.**

Suppose there is a bug for one combination of values of the parameters, say `param1=2` and `param2=11` . Therefore you want to re-run processing only for that combination of the parameters. Without Compmake the best way to do this is perhaps to comment out the two `for` loops, which is easy enough...Eventually, the source code becomes a mess of commented and decommented lines.

**Compmake helps by implementing selective execution.** The user can choose which jobs to execute.

You left the computation running for the night. When you check it out in the morning, you discover that there is one combination of `param1` and `param2` that makes `funcB()` throw an exception. Your program terminated around 1am. The fix is easy enough and you have to start again from the beginning.

**Compmake helps by working around the failure of part of your jobs.**

What about parallelization? Yes, the `multiprocessing` module [4] seem quite easy to use. You just need to add a few lines of code. But wait, there is a nested loop. You probably have to write different functions... and, where exactly can you parallelize?

**Compmake helps by automatically discovering the possible parallelization opportunities in your program.**

No, really, what you want is running your computation on the large university cluster. You do have an account. Of course, the code using the multiprocessing module cannot be directly adapted to run on the cluster.

**Compmake helps by separating the frontend (job definition) and the backend (job execution).** There are several backends available: you can try your program on your computer with multiprocessing, and then run the same program, unchanged, on an SGE cluster.

In short, writing computationally intensive batch processes presents a series of common problems. In isolation, each of them could be overcome by writing ad hoc code. Compmake helps by solving each of these problems (and more) in a robust way, once and for all.

III. When not to use Compmake

Compmake has been designed for relatively long computational expensive batch jobs. Currently, Compmake is not well suited for:

- Applications consisting of very small jobs (say < 0.1 s of CPU time per job), because of the overhead.
- Applications containing more than ~50,000 jobs. That is the level for which you can have a responsive console with the filesystem storage backend.
- Applications for which all the intermediate results are too big to be stored on disk.

See also Section VIII, which describes Compmake’s limitations from a more technical point of view.

A. Alternatives

Depending on what you need to do, there are many other products that might be used:

- There are libraries that focus on performance instead of generality and are optimized for specific dataflows. One example is Spark [5].
- Celery [8] is a popular distributed task queue.
- Scoop [9] is a recent Python library that shares some of Compmake’s philosophy.

Both Celery and Scoop could be used to serve as “backend” for Compmake (just like Multyvac and SGE).
IV. A Compmake Tutorial

(See also the screencast at purl.org/censi/compmake-basics)

A. Installation

The simplest way to install Compmake is using easy_install:

```
$ easy_install -U compmake
```

Alternatively, you can use pip:

```
$ pip install -U compmake
```

You can also clone the Git repository:

```
$ git clone https://github.com/AndreaCensi/compmake.git
$ cd compmake
$ python setup.py develop
```

If you plan to make your own changes to Compmake, first fork the repository at github.com/AndreaCensi/compmake so that you can easily do pull requests.

B. Creating jobs

Compmake makes a Python program parallelizable through a simple change in the code. A function invocation like

```
function(param)
```

is modified in an invocation of the kind

```
context.comp(function, param)
```

where `context` is a `compmake.Context` object.

This and a couple of lines of initialization are all that is needed to make a program parallelizable.

The invocation `context.comp(function, param)` does not actually run the computation `function(param)`, but rather it creates a corresponding “job” in the job database. It returns, immediately, a “promise” (or “future”) representing the result of calling that function. This value can be used successive calls to `context.comp()`. In this way, Compmake learns the computational structure of your program.

For example, suppose this is the original code:

```
res = function(param)
function2(param2, res)
```

In Compmake you would write:

```
p = context.comp(function, param)  # p is a placeholder for function(param)
assert isinstance(p, compmake.Promise)  # use p as you would use the result
context.comp(function2, param2, p)
```

When the snippet is executed two jobs are created. The result of the first function will be passed to the second function.

C. A complete example

To write a complete example, there are a couple of lines of initialization needed to initialize Compmake’s “context”.

Fig. 2 gives a complete example of a Compmake program (`example.py`). Above the source code there is a download link.

If you run this program:

```
$ python example.py
```

You will see a prompt:

```
Compmake 3.3 (27 jobs loaded)
@:
```

This prompt, starting with “@:” is the prompt for Compmake’s console.

Run `make` at the prompt:

```
@: make
```

This will execute all jobs serially.
D. Using the console

Compmake’s console is used for listing, executing, cleaning jobs.

1) Listing jobs: The command `ls` gives a list of the jobs together with their status:

```
@: ls
  draw todo
draw-2 todo
draw-3 todo
draw-4 todo
  [...]  
```

The possible states are given in Table I.

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>todo</td>
<td>The job was not executed yet.</td>
</tr>
<tr>
<td>done</td>
<td>The job was successfully executed.</td>
</tr>
<tr>
<td>failed</td>
<td>The job was executed and failed.</td>
</tr>
<tr>
<td>blocked</td>
<td>The job cannot be executed because one of its dependencies failed or is blocked.</td>
</tr>
</tbody>
</table>

2) Making jobs: The command `make [<jobs>]` runs the computations in series:

```
@: make
```

The first time you run this, you will see the names of the jobs being executed scrolling by. However, the second time, the output will be something like:

```
@: make
Nothing to do.
```

because Compmake has cached the results of the computation.

After making, use `ls` to see the results:

```
@: ls
  draw done
draw-2 done
draw-3 done
draw-4 done
draw-5 done
  [...]  
```

3) Cleaning up: Use the command `clean` to clean all jobs:

```
@: clean
```

4) Selective re-make: You can selectively remake part of the computations. For example, suppose that you modify the `draw()` function, and you want to rerun only the last step. You can achieve that by

```
@: remake draw
```

Compmake will reuse the computations (`funcA()` and `funcB()`) but it will redo the last step.

The same effect can be obtained using `clean` and `make`:

```
@: clean draw*
@: make draw*
```

E. Naming jobs

Each invocation of the `context.comp()` function produces one “job”. Each job is described by a unique ID. By default, the ID is generated by the name of the function, with a progressive number postponed. You can use the command `ls` to obtain a list of the jobs. For this example, the output would be:

```
@: ls
draw todo
draw-2 todo
draw-3 todo
draw-4 todo
draw-5 todo
  [...]  
```

As you can see, the jobs are named `funcA-n`, `funcB-n`, `draw-n`.

It is very useful to have distinctive names for the jobs. Compmake provides two mechanisms to that effect. The first is the function `context.comp_prefix()` which takes a string used as a prefix for the job ids generated. That command is particularly useful in scenarios like the example where one want to group the functions by the parameters:

```
from compmake2.py

using_compmake2.py
```

```
from mycomputations import funcA, funcB, draw
if __name__ == '__main__':
    from compmake import Context
    context = Context()
    for param_a in [1, 2, 3]:
        for param_b in [10, 11, 12]:
            # Add a prefix to the job ids for easy reference
            prefix = 'a%s-b%s' % (param_a, param_b)
            context.comp_prefix(prefix)
            res1 = context.comp(funcA, param_a)
            res2 = context.comp(funcB, res1, param_b)
            context(comp(draw, res2)
        context.compmake_console()
```

Now the `ls` command gives:

```
@: ls
  a1-b10-draw todo
  a1-b10-funcA todo
  a1-b10-funcB todo
  a1-b11-draw todo
  a1-b11-funcA todo
  a1-b11-funcB todo
  [...]  
```

Another method is using the `job_id` keyword argument to `context.comp()`:

```
from compmake3.py

using_compmake3.py
```

```python
from mycomputations import funcA, funcB, draw
if __name__ == '__main__':
    from compmake import Context
    context = Context()
    for param_a in [1, 2, 3]:
        for param_b in [10, 11, 12]:
            # Add a prefix to the job ids for easy reference
            prefix = 'a%s-b%s' % (param_a, param_b)
            context.comp_prefix(prefix)
            res1 = context.comp(funcA, param_a)
            res2 = context.comp(funcB, res1, param_b)
            context(comp(draw, res2)
        context.compmake_console()
```
from mycomputations import funcA, funcB, draw

if __name__ == '__main__':
    from compmake import Context
    context = Context()
    for param_a in [1, 2, 3]:
        for param_b in [10, 11, 12]:
            prefix = 'a%s-b%s' % (param_a, param_b)
            context.comp_prefix(prefix)
            # use job_id to override default naming
            res1 = context.comp(funcA, param_a, job_id='preparing')
            res2 = context.comp(funcB, res1, param_b, job_id='computing')
            context.comp(draw, res2, job_id='drawing')
    context.compmake_console()

Now the ls command gives:

@: ls
    a1-b1-b10-computing todo
    a1-b1-b10-drawing todo
    a1-b1-b10-preparing todo
    a1-b11-computing todo
    a1-b11-drawing todo
    a1-b11-preparing todo
    a1-b12-computing todo
    a1-b12-drawing todo
    [...]

F. Tolerance to job failure

If a some of the jobs fail (e.g., they throw an exception),
Compmake will go forward with the rest of the jobs.

To try this behavior, see the example example_fail.py. In this
example some jobs will launch an exception.

When Compmake runs this application, it will tell which jobs
failed:

@: make
Job 'a2-b11-computing' failed:
| Exception raised for b = 11.
Job 'a3-b11-computing' failed:
| Exception raised for b = 11.
Job 'a1-b11-computing' failed:
| Exception raised for b = 11.
Make failed (3 failed, 3 blocked)

The console displays the exception for the job that failed. To
see further details about the errors, including the backtrace,
use the command details.

Some jobs will have a status of failed. The jobs that depend
on them will have status of blocked:

@: ls *a-
    a1-b1-computing done
    a1-b1-drawing done
    a1-b1-preparing done
    a1-b1-computing failed
    a1-b1-drawing blocked
    a1-b1-preparing done
    a1-b12-computing done
    a1-b12-drawing done

If you run make again, Compmake will try again to run only
the jobs that failed previously.

G. Cleaning and remaking

Now that you know how to give names to your jobs, you can
use them for referring to them. For example

@: make a1-b11-drawing

You can use the "*" wildcard. This is very useful to refer
only to part of the jobs. In the example, you can write

@: remake -*

to re-do only the subset of computations with a certain value
of the parameters. Or, you can remake the last stage of the
computation:

@: remake *-drawing

H. Single-host multiprocessing

To use single-host multiprocessing, instead of using make,
simply use parmake:

@: parmake

Optionally, you can specify the number of processes to
spawn:

@: parmake n=11

If you don’t specify a number, Compmake will use the number
of detected processors. If your jobs are IO-bound rather than
CPU-bound, you should specify a larger number.

What's happening under the hood is that Compmake
spawns n workers processes using the multiprocessing modu-
le. So be aware that each job will run in a different process.

Another backend supported is SGE, described later in Sec-
"ion IX.

I. The compmake executable

Compmake comes with a command-line program called, unsur-
prisingly, compmake. This program can be used to start
a session with a pre-existing job database and to run com-
mants in batch mode. The syntax is:

$ compmake <job_db> [-c <command>]

For example, suppose the script example.py was executed
like this:

$ python example.py

By default the jobs DB is created in the directory
out-<script>. In this case, the jobs database is in
out-example:
Now by using `compmake` one can start a session just by doing:

```
$ compmake out-example
```

This will load the existing jobs DB:

```
Loading existing jobs DB 'out-example'.
Compmake 3.3 (27 jobs loaded)
```

By using the `-c` switch one can run commands in batch mode. For example, this command cleans the previous computation and runs it again:

```
$ compmake out-example -c "clean; parmake; ls"
```
V. In detail: defining the jobs graph

Compmake provides much flexibility about the definition of jobs. Jobs can depend on other jobs, they can create other jobs, and they can “redirect” to other jobs.

A. The basic job definition

Fig. 3 illustrates the graphical convention used to describe the job graph. One job is created using the invocation

```python
context.comp(job1, param1)
```

Here `job1()` is a function, or any callable and pickable object, and `param1` is any (pickable) parameter.

The resulting job graph is illustrated below. It has one node, `job1`, indicating the job, and two other nodes, pictured with circles, indicating the parameter and the result of the job.

![Job Graph](image1)

Figure 3. Graphic language for describing jobs and jobs dependencies.

The listing in Fig. 4 shows a complete Compmake program. This programs defines three jobs, each being an application of the function `f` with a different parameter. Thus these three jobs can be executed in parallel.

```python
def f(x):
    print('processing %s' % x)
if __name__ == '__main__':
    from compmake import Context
    c = Context()
    for p in [42, 43, 44]:
        c.comp(f, x=p):
    c.batch_command('clean;parmake echo=1')
```

(a) Source code

![Job Graph](image2)

(b) Resulting job graph

Figure 4. Python/Compmake program that creates 3 independent jobs and executes them in parallel. The inset shows the `jobs graph`, using boxes to represent jobs, and circles for parameters.

B. Jobs with dependencies

A job can depend on another, in the sense that the result of the second job becomes a parameter of the first. The basic case is shown in Fig. 5.

The first job is created as follows:

```python
r = context.comp(job1, param1)
```

The value `r` is a placeholder, a “promise”, that represents the result of the job. This value can be used as a parameter to another job definition. For example:

```python
context.comp(job2, r, param2)
```

The second job now depends on the first job. Graphically, this is represented by making the output node of the first job be the input node of the second job (Fig. 5).

![Job Graph](image3)

Figure 5. Graphic language for describing jobs and jobs dependencies.

The listing in Fig. 6 shows a typical program that creates jobs with dependencies. The `report` function depends on the output of `statistics`, which depends on three calls of the function `f`.

```python
comp = compcontext
parmake
f
42
f
43
f
44

(a) Source code

![Job Graph](image4)

(b) Jobs graph

Figure 6. The typical scientific data application has three kinds of jobs: 1) the jobs that perform the ”meat” of the computation (like the function `f` in this example); 2) Jobs that compute statistics from the computation (like the function `statistics`); and 3) Jobs that visualize the results (like the function `report`). The job graph is a tree structure. In the inset, the dependence between jobs is represented by the little circles representing both the output of a job and the input of another.
C. “Dynamic” jobs defining other jobs

Compmake supports design patterns in which jobs can define other jobs recursively. A job can be marked as “dynamic” if it is created using “context.comp_dynamic” rather than “context.comp”. For example:

```python
context.comp_dynamic(djob)
```

The function `djob()` must take as first argument a `context`, which is a reference that can be used to define further jobs. For example:

```python
def djob(context):
    assert isinstance(context, compmake.Context)
    context.comp(child1)
    context.comp(child2)
```

When the job is executed two new jobs are created. If the job is cleaned and recomputed, the same jobs are overwritten. In general, Compmake is smart about keeping track of what job generated what.

The graphical representation of this is shown in Fig. 7. A dashed brown arrow indicates that the job generated the others in the dashed rectangle.

![Graphical representation of jobs and job dependencies](image)

Figure 7. Graphic language for describing jobs and job dependencies.

A complete example is shown in Fig. 8. The parameter `reurse=1` to parmake instructs Compmake to schedule the current jobs as well as any other jobs that might be created. Without that option, the three jobs are defined but not executed.

D. Jobs redirecting to other jobs

So far we have seen two possible relations between jobs:

1) A job depends on another job
2) A job defines another job.

There is another relation between jobs: when a job redirects its output to be the output of another. This is one of those cases where the difference is very little from the point of view of the user, but the implementation became quite complicated.

For example, consider the code

```python
def djob(context):
    return context.comp(child)
```

The function `djob` is a dynamic job that defines another job `child`. In addition, the return value of the job itself is the return value of the child. This means that a job that depends on `djob` will be having as input the output `child`, and Compmake will take care of everything in the background for this to happen smoothly. This is a complete example:

```python
def f(x):
    print(f"f({x}r) \times {x}")
    return x * 2

def schedule(context, params):
    print(f"schedule(context, params={params})\times {params.__repr__()}")
    for p in [42, 43, 44]:
        context.comp(f, x*p)

if __name__ == '__main__':
    from compmake import Context
    c = Context()
    c.comp_dynamic(schedule, params=[42, 43, 44])
    c.batch_command('clean;parmake echo=1;reurse=1')
```

The output of this will be “output-of-child”. Note that here the job `another` becomes dependent on `child`, so that if you remake `child` also `another` will be re-done.

Graphically, this relation between jobs is represented by a black arrow from the output of `djob` to the output of `child` (Fig. 9).

![Diagram showing job dependency](image)

Figure 9. A job that defines a child and redirects to its result.

A complete example is reported in Fig. 10. Here, at definition time, the job graph looks as in Fig. 10b: the job `report` depends only on the job `schedule`. However, when the job `schedule` is executed, it creates three `f` jobs and the job `statistics` whose promise is returned as the result.
Therefore, after the execution of `schedule` the job graph looks as in Fig. 10c. Now `report` depends on all the other jobs. If one of the `f` jobs fails, then `report` will have a state of `blocked` at the end of execution.

```python
def f(x):
    print('f(x=%r)' % x)
    return x * 2

def statistics(res):
    print('statistics(res=%s)' % res.__repr__())
    return sum(res)

def schedule(context, params):
    print('schedule(context, params=%s)' % params.__repr__())
    jobs = [context.comp(f, x=p) for p in params]
    summary = context.comp(statistics, jobs)
    # returns a job "promise", not a value!
    return summary

def report(summary):
    print('report(summary=%r)' % summary)

if __name__ == '__main__':
    from compmake import Context
    c = Context()
    summary = c.comp_dynamic(schedule, [42, 43, 44])
    c.comp(report, summary)
    c.batch_command('clean;parmake echo=1 recurse=1')
```

Figure 10. Returning the value of another job
VI. Console Commands Reference

A. Job aliases

All commands that take as argument a list of jobs understand wildcards, a few predetermined aliases and some logical operators to describe the list of jobs on which they need to operate. The list of aliases is shown in Table II.

1) By job ID: The console supports the use of the wildcard “*”. For example, this matches all jobs that have “drawing” in the job ID:

@: ls *drawing*

2) By function name: Compmake also keeps track of the function name, which is different from the job ID. You refer to the function name using the syntax “funcname()”.

For example:

```python
def funcA():
    pass
context.comp(funcA, job_id='preparation')
```

The resulting job will have the id “preparation” but the name of the function “funcA” will be associated to it as well.

So it is possible to refer to that job using:

@: make preparation

as well as using

@: make funcA()

3) By job status: There are several aliases that refer to the status of the job: all (all jobs), failed, blocked, done, todo.

The alias uptodate refers to the jobs that are done and for which all dependencies (recursively) are done as well.

4) By definition depth: The alias root refers to all jobs that are defined in the main context. The alias level1 refers to the jobs defined by root jobs. The alias level2 refers to the jobs defined by those in level1 and so forth.

5) By type: The alias dynamic refers to the jobs that are marked as dynamic—those that were defined using context.comp_dynamic and that take a context argument.

6) By dependencies relations: The alias bottom refers to jobs that have no dependencies.

The alias top refers to all those jobs that are the “top” of the graph, in the sense that no other jobs depends on them.

B. Logical operations on jobs lists

The console also supports some logical operations on jobs lists: and, except, not. If two lists are specified next to each other, “or” is implied.

1) Implied or: For example, this lists all jobs matching “drawing” or “computing”:

@: ls *drawing* *computing*

2) And: This lists all failed jobs that match “drawing”:

@: ls failed and *drawing*

3) Except: This lists all failed jobs except those matching “drawing”:

@: ls failed except *drawing*

4) Not: The conjunction “not list” is equivalent to “all except list”. For example this:

@: ls not *drawing*

is equivalent to

@: ls all except *drawing*

---

Table II

<table>
<thead>
<tr>
<th>CompMake Console Job Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>by name</strong></td>
</tr>
<tr>
<td>name*</td>
</tr>
<tr>
<td>funcname()</td>
</tr>
<tr>
<td><strong>by job status</strong></td>
</tr>
<tr>
<td>all or *</td>
</tr>
<tr>
<td>failed</td>
</tr>
<tr>
<td>done</td>
</tr>
<tr>
<td>todo</td>
</tr>
<tr>
<td>blocked</td>
</tr>
<tr>
<td>uptodate</td>
</tr>
<tr>
<td><strong>by definition depth</strong></td>
</tr>
<tr>
<td>root</td>
</tr>
<tr>
<td>level d</td>
</tr>
<tr>
<td><strong>by type and dependencies relations</strong></td>
</tr>
<tr>
<td>dynamic</td>
</tr>
<tr>
<td>bottom</td>
</tr>
<tr>
<td>top</td>
</tr>
</tbody>
</table>
C. The command `ls`

We have already seen the command `ls [jobs]`. It lists the jobs or lists of jobs given on the command line. Apart from the ID, it also gives much more information.

A possible output for `ls` for an advanced example is shown in Fig. 11.

![Figure 11. Example output for command ls.](image)

From left to right, the fields that are displayed are:

- A flag “d” if this is a dynamic job.
- The level if greater than 0.
- The job ID.
- The job status.
- The size of the job result, if exceeding 1 MB. This is useful to keep the disk usage in check.
- The length of the computation, if exceeding 1 s.
- When the job was executed.

D. The command `stats`

The command `stats` groups functions by their name.

![Figure 12. Example output for command stats.](image)

E. The command `details`

The command `details` shows much more information about the job, including the dependencies relations, as well as the stdout and stderr recorded during the execution.

A typical example of the output is shown in Fig. 13.

![Figure 13. Example output for command details.](image)

The information displayed is the following:

**Job ID**

“Defined by” This is the chain of jobs that defined this one. For a job defined in the main context this is just “[root]”.

**Command** This is the name of the function/callable associated to the job, as reported by the method `__str__()`.

**Dependencies (direct)/(other):** These two fields list the job’s dependencies, both the direct ones, and all dependencies of those recursively.

**Depending on this (direct)/(other):** Similarly, these two fields show the jobs that depend on this, both directly and recursively.

**Status** One of those reported in Table I.

**Uptodate** A job can be `done` but still need to be remade because its dependencies have been updated. This field clarifies this situation.

**Wall/CPU time** This is the wall time and CPU time associated to the job. Because of the nature of parallel computations, the wall time is not reliable at all.

**Host** This is the name of the host that processed this job.

**Args/cache/userobject size** This is the size in bytes of the data structures in Compmake’s DB associated to this job. The size “args” is the size of the pickled command and its arguments. The size “cache” is mainly the size of captured stdout/stderr plus other fixed fields. The size “userobject” is the size of the pickled return value.

**Captured stdout/stderr** These are the captured stdout and stderr during the execution of the job.
F. Commands for making: `make`, `parmake`, `sgemake`

The commands `make`, `parmake`, `sgemake` start the job processing, each on a different backend.

1) The option `recurse`: The option `recurse=1` instructs any of the `*make` commands that if a target job defines other jobs, those jobs should be executed as well.

For example, suppose the job graph is defined as follows:

```python
def f():
    pass

def djob(context):
    """Context.comp(f)"
    ... 
    context.comp_dynamic(djob)
```

In this case, using just `make` will execute `djob` but not the newly created job `f`:

```
@: ls
djob todo
@: make djob
@: ls
djob done
f todo
```

Using the `recurse=1` option will have both jobs done:

```
@: ls
djob todo
@: make recurse=1 djob
@: ls
djob done
f done
```

2) The option `new_process`: `make` and `parmake` also have an option `new_process=1` that makes them start an additional process from scratch. So in total there are 5 different backends:

```
make    Uses serial processing.
make new_process=1 Uses serial processing, but each job is run in a separate Python interpreter by calling the `compmake` executable.
parmake Uses parallel processing. In particular this is achieved by using one new Python Process per job, which is forked from the original process.
parmake new_process=1 runs each parallel job in a separate Python interpreter.
sgemake Uses SGE for cluster processing. See Section IX for details.
```

The `new_process` switch is very useful as a workaround instead of “reloading” the code—see Section VIII-B for details.

3) The option `n`: The option `n` gives the number of parallel processes for the commands `make` and `parmake`:

```
@: parmake n=8    # 8 concurrent jobs
```

If it is not specified it defaults to the value of the config `max_parallel_jobs`, which is default the number of CPUs in the system.

4) The option `echo`: Usually the stdout/stderr from the jobs is suppressed. The commands `make` and `parmake` have an option `echo=1` to display the output as it is produced. This option is not compatible with `new_process=1`—see Section VIII-C for details.

G. Commands for cleaning and remaking: `clean`, `remake`, `parremake`

The command `clean` is used to clean the result of the computation. Any job is returned to the state of `todo`:

```
@: ls
job1 failed
job2 done
@: clean job1 job2; ls
job1 todo
job2 todo
```

The commands `remake` and `parremake` are just shortcuts for `clean; make` and `clean; parmake`, respectively.

H. Other useful commands

1) `exit`: Terminates the console session.

2) `help`: Use `help <command>` to see a brief help about the command.

3) `config`: The command `config` is used for setting configuration variables. It is discussed later in Section VII-A.
I. Other advanced commands used for debugging

There are a few other commands that are useful for the advanced user or developer.

1) **dump**: Usage:

   ```
   @: dump directory=<directory> <jobs>
   ```
   Dumps the result of jobs as pickle files in the given directory.

2) **dump_stdout**: Usage:

   ```
   @: dump_stdout directory=<directory> <jobs>
   ```
   Dumps the result of jobs on the stdout using `print`.

3) **delete**: Completely deletes a job from the DB. Use with caution, as it might compromise the job consistency.

   To reset everything use:

   ```
   @: delete not root; clean
   ```

4) **graph**: Creates and render a GraphViz graph of the given targets and dependencies using the `gvgen` module.

5) **reload**: Reloads a module. See Section VIII-B for a discussion on the limitations of this command.

6) **check_consistency**: Checks the consistency of the DB.

7) **debug_priority**: Shows the computed priority of the jobs.

8) **make_single**: Used internally for SGE backend.

9) **commands_html**: Writes an HTML output describing the commands.

10) **config_config**: Writes an HTML output describing the configuration.
VII. Configuration reference

Compmake has many configuration options. They can be set with the command `config` (Section VII-A) or by reading an initialization file `compmake.rc` (Section VII-B) or with the command line (Section VII-C).

A. Setting configurations using the command `config`

The command `config` is used to get/set the configuration options.

To display the current configuration use

```
@: config
```

To set a configuration option, use:

```
@: config <option> <value>
```

For example, use

```
@: config colorize False
```

B. Using an initialization file

When the executable `compmake` starts it looks for an initialization file in various locations. These are, in order:

```
~/.compmake/compmake.rc
~/.config/compmake.rc
~/.compmake.rc
~/.compmake.rc
compmake.rc
.compmake.rc
```

The contents of each file is interpreted as a console command. In particular you can write `config` commands:

```
$ cat compmake.rc
config colorize False
```

C. Setting configurations using the command line

The executable `compmake` takes all options as switches. For example:

```
@: compmake --colorize False <directory>
```

Is equivalent to

```
@: config colorize False
```

D. Configuration options for processing

1) `recurse` (default False): Default value for the `recurse` option to the `*make` command. See Section VI-F1.

2) `new_process` (default False): Default value for the `new_process` option to the `*make` command. See Section VI-F2.

3) `max_parallel_jobs` (default = number of CPUs): This option gives the default value for the `n` option to `sgemake` and `parmake`. See Section VI-F3.

E. Configuration options for visualization

1) `echo` (default False): Default value for the `echo` option to the `*make` command. See Section VI-F4.

2) `echo_stderr` (default True): If `echo` is True, stderr can be suppressed by setting `echo_stderr` to False.

3) `echo_stdout` (default True): If `echo` is True, stdout can be suppressed by setting `echo_stdout` to False.

4) `status_line_enabled` (default True): If False, suppress the animation showing the state of the computation.

5) `colorize` (default True): If False, suppress colorization of the output.

6) `readline` (default True): If True, Compmake tries to use the `readline` or `pyreadline` libraries.

7) `set_proc_title` (default True): If True, Compmake changes the process title to reflect the job currently executing.

8) `verbose_definition` (default True): If True, Compmake is verbose in displaying the names of the jobs being defined.
VIII. Limitations of Compmake

Compmake has a series of limitations that need to be taken into account. Some of these are intrinsic limitations of Python. Some other are design choices to keep the design simple. Some other might be implemented in the future.

A. All functions and parameters need to be pickable

All callables and their parameters need to be pickable. In general, Python's pickle system is quite flexible, but there are a few well-known limitations.

1) Cannot use lambda functions: It's not possible to use lambda functions in jobs definitions. For example, this will not work:

```python
f = lambda x: x * 2
context.comp(f, 3)
```

This will throw an exception.

The workaround is often to define a class that can be pickled. For example this works:

```python
class F():
    def __call__(self, x):
        return x * 2
context.comp(F(), 3)
```

Of course, the class `F` must be defined in the module and must be itself pickable.

Also look at the module `functools` which provides many related facilities such as partial function application ("currying") which creates objects that can be pickled.

2) Using functions defined directly in `__main__` might prevent some functionality: Some of the examples throughout have used functions defined directly in the `__main__` module. For example:

```python
def func():
    pass
if __name__ == '__main__':
    context = Context()
    context.comp(func)
    context.compmake_console()
```

This works fine for basic cases, but it fails in more complex case, such as using the `new_process=1` option:

```
context.batch_command('make new_process=1')
```

This will fail citing a pickling error. This behavior was extremely tricky to debug. In short, this is a known bug of Python. For more information see [11] and the relevant bug report [12].

The workaround is to always define functions in another module:

```python
from othermodule import func
if __name__ == '__main__':
    context = Context()
    context.comp(func)
    context.compmake_console()
```

B. Reloading modules

A functionality that is always requested is the ability to reload the code of a module, so that when a job fail, the user can edit the code, and immediately the job executed the updated code.

Python does have the ability to "reload" a module, and actually Compmake has a command `reload` but in practice its use is not advised because it never does what one expects. There are two main problems:

1) It is difficult for the user to keep track of which modules need to be reloaded. For example, one might be editing many files at once.
2) Reloading a module out of order might have very weird results regarding the initialization of data structures.

The workaround is to use the option `new_process=1` to the commands `make` and `parmake`:

```
@: make new_process=1 failed
```

In this way, the failed jobs are executed in a new process, different than the one that is displaying the console. The consequence is that the code that the user is editing is immediately reloaded.

C. No stdout/stderr visualization when using `new_process`

There is an option `echo=1` for `make` and `parmake` that displays the output of the jobs as they are executed. This functionality is not implemented when also the option `new_process=1` is specified. Compmake will warn about this and ignore the `echo` option.

It was preferred not to implement inter-process communication for things that are not strictly necessary, like visualizing the output of a process. The reason is that if a job is very verbose it might saturate a queue or the bandwidth of the user’s terminal. So at that point there should be some logic to discard data to maintain responsiveness of the manager.

Note that you can examine the output of a process using the command `details <job>` after the execution.

D. Some libraries are just picky

Some libraries just don’t like to run in child processes and so will fail when using `parmake`. Sometimes the errors mention the function `exec()`. For example:

```
The process has forked and you cannot use this Core Foundation functionality safely. You MUST exec().
```

The workaround is to use `parmake` for most jobs and then use `make` for the picky ones. For example:

```
@: parmake most*; make picky; parmake *
```
E. Jobs cannot communicate with each other

Jobs cannot communicate with each other. Think of them as little units of computation that can be executed each on a different computer.
IX. The SGE Backend

Compmake uses SGE (Sun Grid Engine) to run jobs on a cluster. Zero configuration is needed on the part of Compmake once SGE is configured, but setting up SGE can take up a few minutes.

(See also the screencast at purl.org/censi/compmake-sge)

A. What is SGE?

The Sun Grid Engine became the “Oracle Grid Engine” when Sun was acquired by Oracle. Last year Oracle pulled out of the market, but the code has been open source for years. Currently there are several forks, free and commercial. The Open Grid Scheduler is the main one and the one that is available in Linux distributions. The company Univa provides a commercial fork that runs on OS X as well.

B. Using SGE from Compmake

Once SGE is configured the user experience is very easy. Instead of using `make` or `parmake`, just use the command `sgemake` to run the computation using SGE:

```
@: sgemake
```

The command takes the usual parameters. In particular the option `n` gives the number of concurrent jobs that are submitted to the queue:

```
@: sgemake n=100
```

C. Setting up SGE on a local machine with Ubuntu

There are several guides for this. See for example [14].

1) Setting up DNS: An important note: the reason why SGE installation might fail incomprehensibly is because DNS is not set up correctly. Before installing SGE, make sure that the server has a fully-qualified hostname. To check this, type

```
$ hostname
```

The result must be a fully-qualified hostname which can be pinged from other hosts in the network. So `mycomputer` does not work, but `mycomputer.mycompany.com` works. An address such as `mycomputer.local` works.

On Ubuntu 14.04, a permanent change can be done by editing `/etc/hosts` and `/etc/hostname`. It is advised to reboot to check that the change is permanent. Reboot before installing SGE with the commands below so that the automatic configuration script will pick up the correct address.

2) Installation using `apt-get`: These are the packages needed in Ubuntu 14.04:

```
$ sudo apt-get install gridengine-master gridengine-client gridengine-common gridengine-qmon gridengine-exec
```

Make sure you give correct information for the configuration dialogs pop up.

It is needed to install some fonts for the QMon GUI to work. Just install all xfonts:

```
$ sudo apt-get install xfonts-\*
```

3) Configuration: The following steps are setting up a queue and give yourself permissions to submit to it. These steps can be done using the `qconf` GUI or with the command-line program `qconf`.

To add an admin user `<user>`:

```
$ sudo qconf -ao <user>  
$ sudo qconf -am <user>
```

Next, add a submit host:

```
$ sudo qconf -as <mycomputer>
```

Here `<mycomputer>` must be a fully-qualified hostname as discussed above. Finally, add a queue, using

```
$ sudo qconf -aq queue1
```

By default each queue executes one job at a time. On Ubuntu the command above starts an editor; change `slots` to the number of CPUs in your system. If you want to have multiple jobs executed concurrently, you can either add more queues, or change the capacity of this queue using the `qmon` interface.

The next two steps need to be done using the GUI `qmon`:

- In the “User Configuration” tab, add your user to one of the user groups.
- In the “Job Control” tab, add that user group to the queue.

4) Testing SGE: Here's a simple script `hello.sh` to try:

```
#!/bin/bash

echo "Hello" > /tmp/hello.txt
```

To submit this job, use:

```
$ qsub hello.sh
```

If everything is set up correctly after a few seconds the file `~/tmp/hello.txt` will be created.

If you get a message like:

```
Unable to run job: warning: \ <user> your job is not allowed to run in any queue
```

It means that the last two steps were not executed correctly.

D. Useful SGE commands

1) Monitoring the queue: The command `qstat` can be used to monitor the status of the queues:

```
$ watch qstat
```

$ qstat
2) Deleting jobs: If Compmake receives a KeyboardInterrupt exception (CTRL-C) then it will delete the current jobs from the SGE queues. However if it is interrupted again then some jobs might be remaining in the queue. To delete all current jobs in the queue, use the command:

```
$ qdel -u <user>
```

E. Using Starcluster to set up a cluster in the cloud

This guide only looked at setting up SGE on a single machine. For setting up a cluster in the cloud, one possibility is using the software developed by the StarCluster project [15]. Using StarCluster it is very very easy to run a full-fledged cluster on the Amazon Elastic Compute Cloud [16].

Please refer to the StarCluster documentation for setting up your AWS account and authentication.

1) Starcluster configuration of a Compmake-compatible cluster: You probably want to modify the configuration file in `./starcluster/config` to define your cluster type with the appropriate dependencies for your project.

The following is the basic definition of a cluster “class” for compmakecluster to use Compmake with:

```
[cluster compmakecluster]
KEYNAME = mykey
CLUSTER_SIZE = 4
CLUSTER_USER = sgeadmin
CLUSTER_SHELL = bash
NODE_INSTANCE_TYPE = cc1.4xlarge
NODE_IMAGE_ID = ami-52a0c53b
# use a plugin to install Compmake on the cluster
PLUGINS = compmake-installer
```

The `compmake-installer` plugin simply installs the Compmake python package:

```
[plugin compmake-installer]
setup_class = starcluster.plugins.pypkginstaller.PyPkgInstaller
packages = compmake
```

F. Limitations for SGE backend

1) SGE workers need to share a filesystem: So far there is only one storage backend implemented which stores the data on disk. Therefore all SGE workers need to be able to see the storage directory under the same name.

A backend for Compmake is just a key-value store. In the past I experimented with a Redis [17] backend. Other backends such as MongoDB [18] could be implemented easily.
X. The Multyvac backend (experimental)

Compmake has a Multyvac [19] backend. This backend is currently experimental.

(See also the screencast at purl.org/censi/compmake-multyvac)

A. What is Multyvac?

Multyvac is the successor of PiCloud after PiCloud has been bought by Dropbox in November 2013. It allows to schedule remote execution of Python functions, taking care of all code serialization.

B. Using Multyvac from Compmake

Once Multyvac is configured, instead of using make or parmake, just use the command cloudmake to run your computation using Multyvac. For example, this command runs the computation using 50 cloud workers:

@: cloudmake n=50

Sometimes there is some latency in the cloud job execution. To check the status of your jobs, login to your Multyvac account and use the interface at https://www.multyvac.com/account/jobs/

C. Configuration

1) Sign up for an account: First, you need to sign up for a Multyvac account at the URL:

https://www.multyvac.com/account/register/

2) Install Multyvac: To install Multyvac, you can use pip:

$ pip install multyvac

3) Set up the API key: To create an API key, use:

$ python -m multyvac.setup

4) Optional: creating “layers”: Multyvac will serialize your Python code automatically. However some libraries need to be installed on the workers.

Multyvac calls “layer” a configuration of the system. Suppose your code needs to use Numpy and PIL. These libraries have a C implementation that must be compiled, so they cannot be simply serialized like pure Python code.

Proceeds as follow. First, open a Python console:

$ python

In Python, use the following commands to create a layer called “myconfiguration”:

>>> multyvac.layer.create('myconfiguration')
>>> layer = multyvac.layer.get('myconfiguration')
>>> modify_job = layer.modify()   >>> modify_job.open_ssh_console()

The last command will give you a SSH shell on Multyvac’s servers, which currently run Ubuntu 12.04. On that shell you can install any software. For example, to install Numpy, Scipy, and PIL, use

$ sudo apt-get install python-numpy python-imaging python-scipy

$ exit

Once you exit the SSH shell, you are still in Python. Use this command to save the configuration:

>>> modify_job.snapshot()

At this point, you have available a layer called “myconfiguration” that has Numpy and PIL installed. To tell Compmake to use this layer, set the config multyvac_layer:

@: config multyvac_layer myconfiguration

It is probably useful to create a configuration file /\.compmake/compmake.rc containing this command.

D. Configuration reference for Multyvac backend

These are the configuration option used by the backend.

1) multyvac_debug: If set to True, the logging output of the multyvac module is written on the screen. Otherwise the logging in that module is disabled.

2) multyvac_layer: This option allows to specify which Multyvac “layer” to use (Section X-C4).

3) multyvac_sync_up: Directories to be synchronized up (e.g. containing datasets).

4) multyvac_sync_down: Directories to be synchronized down (e.g. containing output data).

5) multyvac_max_jobs: Default number of cloud jobs to be instantiated. This can be overridden using the n parameter to cloudmake.

E. Commands reference for Multyvac backend

These are the configuration option used by the backend.

1) cloudclean: Deletes all the content of the remote directories that were specified with multyvac_sync_down.

2) cloud_sync_down: Synchronizes the data down, for the directories specified by multyvac_sync_down.

3) cloud_sync_up: Synchronizes the data up, for the directories specified by multyvac_sync_up.

F. Limitations of the Multyvac backend

1) “Dynamic” jobs are not run remotely: For now, the main limitation of the Multyvac backend compared with the others is that “dynamic” jobs, which are those created with
comp_dynamic rather than comp (V-C). If these jobs are encountered, they will be run locally rather than on the cloud.
XI. Troubleshooting

A. Installation of the `readline` library fails

Compmake uses the `readline` [20] library to provide advanced command line functionality like command history and completion. In recent versions if `readline` is not installed then Compmake tries to import the substitute `pyreadline` [21].

1) Installation on Ubuntu 14: If this library is not already installed, `pip` or `easy_install` will try to install it but fail. On Ubuntu 14, the following steps should be enough.

First install the dependencies:

```bash
$ sudo apt-get install build-essential libreadline-*
lib*curses*
```

And then use `pip` or `easy_install` install the package:

```bash
$ sudo easy_install readline
```

Before trying to install Compmake again, check that this command succeeds:

```bash
$ python -c "import readline"
```

References


[21] pyreadline 2.0: A python implementation of GNU readline. URL: https://pypi.python.org/pypi/pyreadline/2.0.