Jug Documentation

Release 1.4.0+git

Luis Pedro Coelho

June 25, 2017

Contents

CHAPTER 1

What is Jug?

Jug allows you to write code that is broken up into tasks and run different tasks on different processors.

It currently has two backends. The first uses the filesystem to communicate between processes and works correctly over NFS, so you can coordinate processes on different machines. The second is based on redis so the processes only need the capability to connect to a common redis server.

Jug also takes care of saving all the intermediate results to the backend in a way that allows them to be retrieved later.

CHAPTER 2

Examples

Short Example

Here is a one minute example. Save the following to a file called `primes.py`:

```
from time import sleep

from jug import TaskGenerator


@TaskGenerator
def is_prime(n):
    sleep(1.)
    for j in range(2, n - 1):
        if (n % j) == 0:
            return False
    return True


primes100 = [is_prime(n) for n in range(2, 101)]
```

Of course, this is only for didactical purposes, normally you would use a better method. Similarly, the `sleep` function is so that it does not run too fast.

Now type `jug status primes.py` to get:

```
Task name                      Waiting       Ready    Finished     Running
-------------------------------------------------------------------------
primes.is_prime                      0          99           0           0
.........................................................................
Total:                               0          99           0           0
```

This tells you that you have 99 tasks called `primes.is_prime` ready to run. So run `jug execute primes.py &`. You can even run multiple instances in the background (if you have multiple cores, for example). After starting 4 instances and waiting a few seconds, you can check the status again (with `jug status primes.py`):

```
Task name                      Waiting        Ready     Finished      Running
-------------------------------------------------------------------------
primes.is_prime                      0           63           32            4
.........................................................................
Total:                               0           63           32            4
```

Now you have 32 tasks finished, 4 running, and 63 still ready. Eventually, they will all finish and you can inspect the results with `jug shell primes.py`. This will give you an `ipython` shell. The *primes100* variable is available, but it is an ugly list of *jug.Task* objects. To get the actual value, you call the *value* function:

```
In [1]: primes100 = value(primes100)

In [2]: primes100[:10]
Out[2]: [True, True, False, True, False, True, False, False, False, True]
```

More Examples

There is a worked out example in the tutorial, and another, fully functioning in the *examples/* directory.

CHAPTER 3

Links

- Mailing list
- Github
- Freshmeat
- Documentation
- Homepage

CHAPTER 4

How do I get Jug?

The simplest is using pip:

pip install jug

You can either get the git repository at

http://github.com/luispedro/jug

Or download the package from PyPI

CHAPTER 5

Testimonials

"I've been using jug with great success to distribute the running of a reasonably large set of parameter combinations"
- Andreas Longva

CHAPTER 6

Documentation Contents

Jug Tutorial

What is jug?

Jug is a simple way to write easily parallelisable programs in Python. It also handles intermediate results for you.

Example

This is a simple worked-through example which illustrates what jug does.

Problem

Assume that I want to do the following to a collection of images:

1. for each image, compute some features
2. cluster these features using k-means. In order to find out the number of clusters, I try several values and pick the best result. For each value of k, because of the random initialisation, I run the clustering 10 times.

I could write the following simple code:

```
imgs = glob('*.png')
features = [computefeatures(img,parameter=2) for img in imgs]
clusters = []
bics = []
for k in range(2, 200):
    for repeat in range(10):
        clusters.append(kmeans(features,k=k,random_seed=repeat))
        bics.append(compute_bic(clusters[-1]))
Nr_clusters = argmin(bics) // 10
```

Very simple and solves the problem. However, if I want to take advantage of the obvious parallelisation of the problem, then I need to write much more complicated code. My traditional approach is to break this down into smaller scripts. I'd have one to compute features for some images, I'd have another to merge all the results together and do some of the clustering, and, finally, one to merge all the results of the different clusterings. These would need to be called with different parameters to explore different areas of the parameter space, so I'd have a couple of scripts just for calling the main computation scripts. Intermediate results would be saved and loaded by the different processes.

This has several problems. The biggest are

1. The need to manage intermediate files. These are normally files with long names like *features_for_img_0_with_parameter_P.pp*.
2. The code gets much more complex.

There are minor issues with having to issue several jobs (and having the cluster be idle in the meanwhile), or deciding on how to partition the jobs so that they take roughly the same amount of time, but the two above are the main ones.

Jug solves all these problems!

Tasks

The main unit of jug is a Task. Any function can be used to generate a Task. A Task can depend on the results of other Tasks.

The original idea for jug was a Makefile-like environment for declaring Tasks. I have moved beyond that, but it might help you think about what Tasks are.

You create a Task by giving it a function which performs the work and its arguments. The arguments can be either literal values or other tasks (in which case, the function will be called with the *result* of those tasks!). Jug also understands lists of tasks and dictionaries with tasks. For example, the following code declares the necessary tasks for our problem:

```
imgs = glob('*.png')
feature_tasks = [Task(computefeatures,img,parameter=2) for img in imgs]
cluster_tasks = []
bic_tasks = []
for k in range(2, 200):
    for repeat in range(10):
        cluster_tasks.append(Task(kmeans,feature_tasks,k=k,random_seed=repeat))
        bic_tasks.append(Task(compute_bic,cluster_tasks[-1]))
Nr_clusters = Task(argmin,bic_tasks)
```

Task Generators

In the code above, there is a lot of code of the form *Task(function,args)*, so maybe it should read *function(args)*. A simple helper function aids this process:

```
from jug import TaskGenerator

computefeatures = TaskGenerator(computefeatures)
kmeans = TaskGenerator(kmeans)
compute_bic = TaskGenerator(compute_bic)

@TaskGenerator
def Nr_Clusters(bics):
    return argmin(bics) // 10
```

```
imgs = glob('*.png')
features = [computefeatures(img,parameter=2) for img in imgs]
clusters = []
bics = []
for k in range(2, 200):
    for repeat in range(10):
        clusters.append(kmeans(features,k=k,random_seed=repeat))
        bics.append(compute_bic(clusters[-1]))
Nr_clusters(bics)
```

You can see that this code is almost identical to our original sequential code, except for the decorators at the top and the fact that *Nr_clusters* is now a function (actually a TaskGenerator, look at the use of a decorators).

This file is called the jugfile (you should name it *jugfile.py* on the filesystem) and specifies your problem.

Jug

So far, we have achieved seemingly little. We have turned a simple piece of sequential code into something that generates Task objects, but does not actually perform any work. The final piece is jug. Jug takes these Task objects and runs them. Its main loop is basically

```
while len(tasks) > 0:
    for t in tasks:
        if can_run(t): # ensures that all dependencies have been run
            if need_to_run(t) and not is_running(t):
                t.run()
            tasks.remove(t)
```

If you run jug on the script above, you will simply have reproduced the original code with the added benefit of having all the intermediate results saved.

The interesting is what happens when you run several instances of jug at the same time. They will start running Tasks, but each instance will run its own tasks. This allows you to take advantage of multiple processors in a way that keeps the processors all occupied as long as there is work to be done, handles the implicit dependencies, and passes functions the right values. Note also that, unlike more traditional parallel processing frameworks (like MPI), jug has no problems with the number of participating processors varying throughout the job.

Behind the scenes, jug is using the filesystem to both save intermediate results (which get passed around) and to lock running tasks so that each task is only run once (the actual main loop is thus a bit more complex than shown above).

Intermediate and Final Results

You can obtain the final results of your computation by setting up a task that saves them to disk and loading them from there. If the results of your computation are simple enough, this might be the simplest way.

Another way, which is also the way to access the intermediate results if you want them, is to run the jug script and then access the *result* property of the Task object. For example,

```
img = glob('*.png')
features = [computefeatures(img,parameter=2) for img in imgs]
...

feature_values = [feat.result for feat in features]
```

If the values are not accessible, this raises an exception.

Advantages

jug is an attempt to get something that works in the setting that I have found myself in: code that is *embarrassingly parallel* with a couple of points where all the results of previous processing are merged, often in a simple way. It is also a way for me to manage either the explosion of temporary files that plagued my code and the brittleness of making sure that all results from separate processors are merged correctly in my *ad hoc* scripts.

Limitations

This is not an attempt to replace libraries such as MPI in any way. For code that has many more merge points (i.e., code locations which all threads must reach at the same time), this won't do. It also won't do if the individual tasks are so small that the over-head of managing them swamps out the performance gains of parallelisation. In my code, most of the times, each task takes 20 seconds to a few minutes. Just enough to make the managing time irrelevant, but fast enough that the main job can be broken into thousands of tiny pieces. As a rule of thumb, tasks that last less than 5 seconds should probably be merged together.

The system makes it too easy to save all intermediate results and run out of disk space.

This is still Python, not a true parallel programming language. The abstraction will sometimes leak through, for example, if you try to pass a Task to a function which expects a real value. Recall how we had to re-write the line *Nr_clusters = argmin(bics) // 10* above.

Worked-Out Example 0

Decrypting a Password

Problem: crack an encrypted file by brute force. Assume that the password is a five-letter lower-case word and that you know that the plain text contains my name.

(The complete code for this example and a secret message comes with the jug source)

This is the ultimate parallel problem: try very many keys (26**5 ~ 11M), but there is no interaction between the different tasks.

The brute force version is very simple:

```
for p in product(letters, repeat=5):
    text = decode(ciphertext, p)
    if isgood(text):
        passwd = "".join(map(chr, p))
        print('%s:%s' % (passwd, text))
```

However, if we have more than one processor, we'd like to be able to tell `jug` to use multiple processors.

We cannot simply have each password be its own task: 11M tasks would be too much!

So, we are going to iterate over the first letter and a task will consist of trying every possibility *starting* with that letter:

```
@TaskGenerator
def decrypt(prefix, suffix_size):
    res = []
    for p in product(letters, repeat=suffix_size):
        text = decode(ciphertext, np.concatenate([prefix, p]))
        if isgood(text):
            passwd = "".join(map(chr, p))
            res.append((passwd, text))
```

```
    return res

@TaskGenerator
def join(partials):
    return list(chain(*partials))

fullresults = join([decrypt([let], 4) for let in letters])
```

Here, the `decrypt` function returns a list of all the good passwords it found. To simplify things, we call the `join` function which concatenates all the partial results into a single list for convenience.

Now, run `jug`:

```
$ jug execute jugfile.py &
$ jug execute jugfile.py &
$ jug execute jugfile.py &
$ jug execute jugfile.py &
```

You can run as many simultaneous processes as you have processors. To see what is happening, type:

```
$ jug status jugfile.py
```

And you will get an output such as:

```
Task name                                    Waiting        Ready     Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.join                                       1            0            0
↪ 0
jugfile.decrypt                                    0           14            8
↪ 4
...............................................................................
↪..
Total:                                             1           14            8
↪ 4
```

There are two task functions: `decrypt`, of which 8 are finished, 14 are ready to run, and 4 are currently running; and `join` which has a single instance, which is `waiting`: it cannot run until all the `decrypt` tasks have finished.

Eventually, everything will be finished and your results will be saved in directory `jugdata` in files with names such as `jugdata/5/4/a1266debc307df7c741cb7b997004f` The name is simply a hash of the task description (function and its arguments).

In order to make sense of all of this, we write a final script, which loads the results and prints them on stdout:

```
import jug
jug.init('jugfile.py', 'jugdata')
import jugfile

results = jug.task.value(jugfile.fullresults)
for p, t in results:
    print("%s\n\n    Password was '%s'" % (t, p))
```

`jug.init` takes the jugfile name (which happens to be `jugfile.py`) and the data directory name.

`jug.task.value` takes a `jug.Task` and loads its result. It handles more complex cases too, such as a list of tasks (and returns a list of their results).

Worked-Out Example 1

Processing text

Problem: Take a list of the British members of Parliament (MPs) in the last decade and characterise each by a couple of *meaningful* word from their wikipedia pages. Meaningful words are those that appear in the article for the the particular MP but not everywhere else.

(The complete code for this example and a list of MPs [valid in 2010] with the jug source)

The algorithm looks like this:

```
allcounts = []
for mp in MPs:
    article = get_data(mp)
    words = count_words(mp, article)
    allcounts.append(words)

global_counts = add_counts(allcounts) # Here all processes must sync

for mp, mp_count in zip(MPs, counts):
    meaningful = []
    for w, c in mp_count:
        if c > global_counts[w] // 100:
            meaningful.append(w)
    meaningful.sort(key=mp_count.get)
    meaningful.reverse()
    print(mp, meaningful[:8])
```

Very simple. It's also *embarassingly parallel*, except for the line which computes `global_counts`, because it uses the results from everyone.

To use `jug`, we write the above, including the functions, to a file (in this case, the file is `jugfile.py`). Now, I can call `jug status jugfile.py` to see the state of the computation:

```
Task name                                   Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.get_data                                  0         657           0
↪ 0
jugfile.count_words                             657           0           0
↪ 0
jugfile.divergence                              657           0           0
↪ 0
jugfile.add_counts                                1           0           0
↪ 0
...............................................................................
↪..
Total:                                         1315         657           0
↪ 0
```

Unsurprisingly, no task is finished and only the `get_data` task is ready to run. No nodes are running. So, let's start a couple of processes[1]:

[1] For this tutorial, all nodes are on the same machine. In real life, they could be on different computers as long as they can communicate with each other.

```
$ jug execute jugfile.py &
$ jug execute jugfile.py &
$ jug execute jugfile.py &
$ jug execute jugfile.py &
$ sleep 4
$ jug status jugfile.py
$ sleep 48
$ jug status jugfile.py
```

This prints out first:

```
Task name                                    Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.get_data                                   0         653           0
↪ 4
jugfile.count_words                              657           0           0
↪ 0
jugfile.divergence                               657           0           0
↪ 0
jugfile.add_counts                                 1           0           0
↪ 0
...............................................................................
↪..
Total:                                          1315         653           0
↪ 4

$ sleep 48
$ jug status jugfile.py

Task name                                    Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.get_data                                   0         635          20
↪ 2
jugfile.count_words                              637           2          16
↪ 2
jugfile.divergence                               657           0           0
↪ 0
jugfile.add_counts                                 1           0           0
↪ 0
...............................................................................
↪..
Total:                                          1295         637          36
↪ 4
```

So, we can see that almost immediately after the four background processes were started, 4 of them were working on the `get_data` task[2].

Forty-eight seconds later, some of the `get_data` calls are finished, which makes some `count_words` tasks be callable and some have been executed. The order in which tasks are executed is decided by `jug` itself.

At this point, we can add a couple more nodes to the process if we want for no other reason than to demonstrate this capability (maybe you have a dynamic clustering system and a whole lot more nodes have become available). The

[2] In order to make this a more realistic example, tasks all call the `sleep()` function to simulate long running processes. This example, without the `sleep()` calls, takes four seconds to run, so it wouldn't be worth the effort to run multiple processors. Check `jugfile.py` for details.

nodes will happily chug along until we get to the following situation:

```
Task name                                    Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.get_data                                   0           0         657
↪ 0
jugfile.count_words                                0           0         657
↪ 0
jugfile.divergence                               657           0           0
↪ 0
jugfile.add_counts                                 0           0           0
↪ 1
...............................................................................
↪..
Total:                                           657           0        1314
↪ 1
```

This is the bottleneck in the programme: Notice how there is only one node running, it is computing `add_counts()`. Everyone else is waiting (there are no *ready* tasks)[3]. Fortunately, once that node finishes, everyone else can get to work computing `divergence`:

```
Task name                                    Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jugfile.get_data                                   0           0         657
↪ 0
jugfile.count_words                                0           0         657
↪ 0
jugfile.divergence                                 0         653           0
↪ 4
jugfile.add_counts                                 0           0           1
↪ 0
...............................................................................
↪..
Total:                                             0         653        1315
↪ 4
```

Eventually, all the nodes finish and we are done. All the results are now left inside `jugdata`. To access it, we can write a little script:

```
import jug
import jug.task

jug.init('jugfile.py', 'jugdata')
import jugfile

results = jug.task.value(jugfile.results)
for mp, r in zip(file('MPs.txt'), results):
    mp = mp.strip()
    print(mp, ":    ", " ".join(r[:8]))
```

[3] There is a limit to how long the nodes will wait before giving up to avoid having one bad task keep every node in active-wait mode, which is very unfriendly if you are sharing a cluster. By default, the maximum wait time is set to roughly half an hour. You can set this with the `--nr-wait-cycles` (how many times jug will check for tasks) and `--wait-cycle-time` (the number of seconds to wait between each check).

The `jug.init()` call takes the *jugfile* (which does not need to be called *jugfile.py*) and the storage backend (at the simplest, just a directory path like here). Internally, `jug.init` imports the module, but we need to import it here too to make the names available (**it is important that you use this interface.** For example, running the jugfile directly on the interpreter might result in different task names and weirdness all around). `jug.task.value` looks up the value computed and then we can process the results into a nicer output format.

Besides serving to demonstrate, `jug`'s abilities, this is actually a very convenient format for organising computations:

1. Have a master jugfile.py that does all the computations that take a long time.
2. Have a secondary outputresult.py that loads the results and does the pretty printing. This should run fast and not do much computation.

The reason why it's good to have the second step as a separate process is that you often want fast iteration on the output or even interactive use (if you are outputting a graph, for example; you want to be able to fiddle with the colours and axes and have immediate feedback). Otherwise, you could have had everything in the main `jugfile.py`, with a final function writing to an output file.

Image Segmentation Tutorial

This was originally material for a presentation and blog post. You can get the slides online.

Let us imagine you are trying to compare two image segmentation algorithms based on human-segmented images. This is a completely real-world example as it was one of the projects where I first used jug[1].

It depends on mahotas for image processing.

We are going to build this up piece by piece.

First a few imports:

```
import mahotas as mh
from jug import TaskGenerator
from glob import glob
```

Here, we test two thresholding-based segmentation method, called `method1` and `method2`. They both (i) read the image, (ii) blur it with a Gaussian, and (iii) threshold it[2]:

```
@TaskGenerator
def method1(image):
    # Read the image
    image = mh.imread(image)[:, :, 0]
    image  = mh.gaussian_filter(image, 2)
    binimage = (image > image.mean())
    labeled, _ = mh.label(binimage)
    return labeled

@TaskGenerator
def method2(image):
    image = mh.imread(image)[:, :, 0]
    image  = mh.gaussian_filter(image, 4)
    image = mh.stretch(image)
    binimage = (image > mh.otsu(image))
    labeled, _ = mh.label(binimage)
    return labeled
```

[1] The code in that repository still uses a pretty old version of jug, this was 2009, after all. `TaskGenerator` had not been invented yet.
[2] This is for demonstration purposes; the paper had better methods, of course.

We need a way to compare these. We will use the Adjusted Rand Index[3]:

```
@TaskGenerator
def compare(labeled, ref):
    from milk.measures.cluster_agreement import rand_arand_jaccard
    ref = mh.imread(ref)
    return rand_arand_jaccard(labeled.ravel(), ref.ravel())[1]
```

Running over all the images **looks exactly like Python**:

```
results = []
for im in glob('images/*.jpg'):
    m1 = method1(im)
    m2 = method2(im)
    ref = im.replace('images', 'references').replace('jpg', 'png')
    v1 = compare(m1, ref)
    v2 = compare(m2, ref)
    results.append( (v1,v2) )
```

But how do we get the results out?

A simple solution is to write a function which writes to an output file:

```
@TaskGenerator
def print_results(results):
    import numpy as np
    r1, r2 = np.mean(results, 0)
    with open('output.txt', 'w') as out:
        out.write('Result method1: {}\nResult method2: {}\n'.format(r1,
                                                                   r2))
print_results(results)
```

§

Except for the ‘‘TaskGenerator‘‘ this would be a pure Python file!

With `TaskGenerator`, we get jugginess!

We can call:

```
jug execute &
jug execute &
jug execute &
jug execute &
```

to get 4 processes going at once.

§

Note also the line:

```
print_results(results)
```

`results` is a list of `Task` objects. This is *how you define a dependency*. Jug picks up that to call `print_results`, it needs all the `results` values and behaves accordingly.

Easy as Py.

§

[3] Again, you can do better than Adjusted Rand, as we show in the paper; but **this is a demo**. This way, we can just call a function in milk

The full script above including data is available from github

Subcommands

Jug is organised as a series of subcommands. They are called by `jug subcommand jugfile.py [OPTIONS]`. This is similar to applications such as version control systems.

Major Subcommands

execute

The main subcommand of jug is *execute*. Execute executes all your tasks. If multiple jug processes are running at the same time, they will synchronise so that each will run different tasks and combine the results.

It works in the following loop:

```
while not all_done():
    t = next_task()
    if t.lock():
        t.run()
        t.unlock()
```

The actual code is much more complex, of course.

status

You can check the status of your computation at any time with status.

shell

Shell drops you into an ipython shell where your jugfile has been loaded. You can look at the results of any Tasks that have already run. It works even if other tasks are running in the background.

IPython needs to be installed for `shell` to work.

More information about jug shell

Minor Subcommands

check

Check is simple: it exits with status 0 if all tasks have run, 1 otherwise. Useful for shell scripting.

sleep-until

This subcommand will simply wait until all tasks are finished before exiting. It is useful for monitoring a computation (especially if your terminal has an option to display a pop-up or bell when it detects activity). It **does not** monitor whether errors occur!

invalidate

You can invalidate a group of tasks (by name). It deletes all results from those tasks and from any tasks that (directly or indirectly) depend on them. You need to give the subcommand the name with the `--invalid` option.

cleanup

Removes all elements in the store that are not used by your jugfile.

Extending Subcommands

Note: This feature is still experimental

Subcommands are now implemented using a modular design that allows extending jug with additional commands. This API is currently in experimental stage and may change in the future.

The following serves as an example of how to extend jug's commands.

Lets assume you wanted to create a custom report and have it available as:

```
$ jug my-fancy-report
```

One way to achieve this is to add the following code to `~/.config/jug/jug_user_commands.py`:

```
from jug.subcommands import SubCommand

class FancyReport(SubCommand):
    "Produces a fancy report of my results"

    name = "my-fancy-report"

    def run(self, *args, **kwargs):
        ...

fancy_report = FancyReport()
```

The first line of the class docstring is important as it will be shown in jug's usage help page. The name attribute is also required and should be the name of your subcommand on the command-line.

The body of the method `run()` defines what should happen when you call the subcommand `jug my-fancy-report`.

The `run` function will receive the following objects:

```
* ``options``  - object representing command-line and user options
* ``store``    - backend object reponsible for handling jobs
* ``jugspace`` - a namespace of jug internal variables (better not touch)
```

additional objects may be introduced in the future so make sure your function uses `*args, **kwargs` to maintain compatibility.

Finally, in order to register the subcommand, you must instanciate the subcommand.

If your subcommand needs configurable options you can expose them via command-line by defining two additional methods:

```
class FancyReport(SubCommand):
    ...

    def parse(self, parser):
        parser.add_argument('--tofile', action='store',
                            dest='report_tofile',
                            help='Name of file to use for report')

    def parse_defaults(self):
        return {
            "report_tofile": "report.txt",
        }

fancy_report = FancyReport()
```

The first method configures argparse arguments that will be available as `jug my-fancy-report --tofile myreport.txt`. These will also be avaiable to the `run()` method as part of the `options` object.

The second defines default values in case of omission. The `key` should match the `dest=` attribute of `add_argument()` and the `value` should be any object to be used by your `run()` method. Note that the value received in the command-line will be automatically converted to the same type as this default (i.e. if your default is `True` any `--tofile john` would result in `bool("john") -> True`).

For more information on parser configuration refer to `argparse`'s documentation.

NOTE: A few words of caution, we cannot rely on `argparse`'s `default=` option since it doesn't allow distinguishing between user supplied and built-in (default) values. For the same reason, don't use `action=` with `store_true` or `store_false` instead use `store_const` and `const=` with `True` or `False`. Failing to do so will cause any matching setting on `jugrc` to not have any effect.

Status

Simple Status

The `status` subcommand displays a summary of the computation status, for example:

```
$ jug status
     Waiting       Ready    Finished     Running  Task name
------------------------------------------------------------------------
           0           2           0           0  jugfile.compfeats
          10           0           0           0  jugfile.nfold
........................................................................
          10           2           0           0  Total
```

Short Status

The same status as above, now in a `short` version:

```
$ jug status --short
12 tasks to be run, 0 finished, (none running).
```

Cached Status

If you have many tasks, then `jug status` can become pretty slow. One way to speed it up is to use a cache:

```
$ jug status --cache
```

or:

```
$ jug status --cache --short
```

The first time you run it, it will be as slow as usual as it will parse the jugfile and interrogate the store for every possible task. However, then it will save a record which will enable it to speed up the next few times.

Note: This is a fragile system, which should be used with care as the cache can easily become out of sync with the jugfile used (`jugfile.py` in the examples above). It is kept in jug as the speed gains can be quite spectacular (from many seconds to instantaneous).

Jug Shell

The `jug shell` subcommand opens up a shell within the environment of the Jugfile. It can be used for debugging and exploration of the task structure.

Inside the environment, you can use the `value(task)` function to load results (if available; otherwise, an exception is thrown).

New in version 1.5: `invalidate(task)` was only added to the shell in version 1.5. Before that, only `task.invalidate()` (non-recursive) was available.

The `invalidate(task)` function invalidates the results of its argument and all dependents, **recursively**, much like the `jug invalidate` subcommand.

You can call methods on `Task` objects directly as well:

- `run()`: run the task and return the result **even if the task has run before**.
- `can_run()`: whether all dependencies are available
- `can_load()`: whether the task has already run
- `invalidate()`: remove the result of this task (**not recursive**, unlike the `invalidate` subcommand, unlike the invalidate function).

You can access the original function with the `.f` attribute as well.

Note

Jug shell requires IPython to be installed.

Types in Jug

Any type that can be `pickle()d` can be used with jug tasks. However, it might sometimes be wiser to break up your tasks in ways that minimise the communication necessary. For example, consider the following image processing code:

```
from glob import glob
from mahotas import imread

def process(img):
    # complex image processing

files = glob('inputs/*.png')
imgs = [Task(imread, f) for f in files]
props = [Task(process, img) for img in imgs]
```

This will work just fine, but it will save too much intermediate data. Consider rewriting this to:

```
from glob import glob

def process(f):
    from mahotas import imread
    img = imread(f)
    # complex image processing

files = glob('inputs/*.png')
props = [Task(process, f) for f in files]
```

I have also moved the import of `mahotas.imread` to inside the `process` function. This is another micro-optimisation: it makes `jug status` and friends just that little bit faster (as they do not need to perform this import to do their jobs).

Structuring Your Tasks

Tips & Tricks

These are some general guidelines for programming with jug.

What can be a task?

Almost any named function can be a task.

What should be a task?

The trade-off is between tasks that are too small (you have too many of them and the overhead of `jug` will overwhelm your process) or too big (and then you have too few tasks per processor.

As a rule of thumb, each task should take at least a few seconds, but you should have enough tasks that your processors are not idle.

Task size control

Certain mechanisms in jug, for example, `jug.mapreduce.map` and `jug.mapreduce.mapreduce` allow the user to tweak the task breakup with a couple of parameters

In `map` for example, `jug` does, by default, issue a task for each element in the sequence. It rather issues one for each 4 elements. This expects tasks to not take that long so that grouping them gives you a better trade-off between the

throughput and latency. You might quibble with the default, but the principle is sound and it is only a default: the setting is there to give you more control.

Identifying tasks

In the module `jug.hash`, jug attempts to construct a unique identifier, called a hash, for each of your tasks. For doing that, the name of the function involved invoked in the task together with the parameters that it receives are used. This makes jug easy to use but has some drawbacks:

- If your functions take long/big arguments, the hash process will potentially be costly. That's a common situation when you are processing arrays for example, or if you are using sets/dictionaries, in which case the default handling needs to get a sorted list from the elements of the set/dictionary.
- Jug might not know how to handle the types of your arguments,
- Arguments might be equivalent, and thus the tasks should be identified in the same way, without jug knowing. As a very contrived example, suppose that a task uses an argument which is an angle and for the purpose of your program all the values are equivalent modulo 2*pi.

If you control the types of your arguments, you can add a `__jug_hash__` method to your type directly. This method should return a string:

```
class MySpecialThing(object):
    def __jug_hash__(self):
        return some_string
```

Alternatively, you can use `jug.utils.CustomHash` in the case where you cannot (or rather, would not) change the types:

```
from jug.utils import CustomHash
def my_hash_function(x):
    return some_string_based_on_x

complex = ...
value = CustomHash(complex, my_hash_function)
```

Now, `value` behaves exactly like `complex`, but its hash is computed by calling `my_hash_function`.

Tasklets

New in version 0.8: Tasklets were added in version 0.8, starting with the betas (named 0.7.9..)

A Tasklet is a light-weight task. It looks very similar to a Task *except that it does not save its results to disk*. Every time you need its output, it is recomputed. Other than that, you can pass it around, just like a Task.

They also get sometimes automatically generated.

Before Tasklets

Before Tasklets were a part of jug, often there was a need to write some connector Tasks:

```
def select_first(t):
    return t[0]

result = Task(...)
```

```
result0 = Task(select_first, result)

next = Task(next_step,result0)
```

This obviously works, but it has two drawbacks:

1. It is not natural Python
2. It is space inefficient on disk. You are saving `result` and then `result0`.

With Tasklets

First version:

```
def select_first(t):
    return t[0]
result = Task(...)
result0 = Tasklet(select_first, result)
next = Task(next_step,result0)
```

If you look closely, you will see that we are now using a *Tasklet*. This jugfile will work exactly the same, but it will not save the `result0` to disk. So you have a performance gain.

It is still not natural Python. However, the Task can *generate Tasklets automatically*:

Second version:

```
result = Task(...)
next = Task(next_function,result[0])
```

Now, we have a version that is both idiomatic Python and efficient.

Invalidating Jug Task Results (invalidate subcommand)

When you invalidate results of a task, you are telling jug that everything that was computed using that function should no longer be used. Thus, any result which depended on that function needs to be recomputed.

Invalidation is manual

To invalidate a task, you need to use the `invalidate` command. Jug does not detect that you fixed a bug in your code automatically.

Invalidation is dependency aware

When you invalidate a task, all results that depend on the invalid results will also be invalidated. This is what makes the invalidate subcommand so powerful.

Example

Consider the British parliament example we used elsewhere in this guide:

```
allcounts = []
for mp in MPs:
    article = get_data(mp)
    words = count_words(mp, article)
    allcounts.append(words)

global_counts = add_counts(allcounts) # Here all processes must sync
```

Its dependency graph looks like this:

```
M0 -> get_data -> count_words -> C0
M1 -> get_data -> count_words -> C1
M2 -> get_data -> count_words -> C2
...
C0 C1 C2 -> add_counts -> avgs
C0 + avgs -> divergence
C1 + avgs -> divergence
C2 + avgs -> divergence
...
```

This is a typical fan-in-fan-out structure. After you have run the code, `jug status` will give you this output:

```
\      Waiting       Ready    Finished     Running  Task name
-------------------------------------------------------------------------------
           0           0           1           0  jugfile.add_counts
           0           0         656           0  jugfile.count_words
           0           0         656           0  jugfile.divergence
           0           0         656           0  jugfile.get_data
...............................................................................
           0           0        1969           0  Total
```

Now assume that `add_counts` has a bug. Now you must:

1. Fix the bug (well, of course)
2. Rerun everything that could have been affected by the bug.

Jug invalidation helps you with the second task.

```
$ jug invalidate --target add_counts
Invalidated  Task name
-------------------------------------------------------
          1  jugfile.add_counts
        656  jugfile.divergence
.......................................................
        657  Total
```

will remove results for all the `add_counts` tasks, *and all the ''divergence'' tasks* because those results depended on results from `add_counts`. Now, `jug status` gives us:

```
\     Waiting       Ready    Finished     Running  Task name
-------------------------------------------------------------------------------
           0           1           0           0  jugfile.add_counts
           0           0         656           0  jugfile.count_words
         656           0           0           0  jugfile.divergence
           0           0         656           0  jugfile.get_data
...............................................................................
         656           1        1312           0  Total
```

So, now when you run `jug execute`, `add_counts` will be re-run as will everything that could possibly have changed as well.

Idioms

There are certain types of computation that show up again and again in parallel computation. This section shows how to perform them with jug.

Map/Reduce

Currently, the dominant paradigm for large scale distributed computing is the map/reduce paradigm. Originally made prominent by Google's proprietary implementation, it is now available in many implementations, including open-source ones such as Hadoop.

Jug is not a direct competitor to Hadoop as it focuses on medium sized problems. Jug does not implement a distributed file system of any sort, but assumes that all compute nodes have access to a central repository of information (such as a shared filesystem or a redis server).

On the other hand, jug supports much more complex computations than map-reduce. If, however, your problem is naturally described as a map/reduce computation, then *jug* has some helper functions.

`jug.mapreduce.mapreduce`

The `jug.mapreduce.mapreduce` function implements mapreduce:

```
jug.mapreduce(reducer, mapper, inputs)
```

is roughly equivalent to:

```
Task{ reduce(reducer, map(mapper, inputs)) }
```

If the syntax of Python supported such a thing.

An issue that might come up is that your *map* function can be **too fast**. A good task should take at least a few seconds (otherwise, the overhead of scheduling and loading the data overwhelms the performance advantages of parallelism. Analogously for the *reduce* function.

Therefore, `jug` groups your inputs so that a mapping task actually consists of mapping and reducing more than one input. How many is controlled by the `map_step` parameter. By default, it is set to 4. Similarly, the `reduce_step` parameter controls how many reduction steps to perform in a single task (by default, 8; reflecting the fact that reduce operations tend to be lighter than map operations).

The compound task section has a worked out example of using map/reduce.

Parameter Sweep

This is a standard problem in many fields, for example, in machine learning. You have an algorithm that takes a few parameters (let's call them `p0` and `p1`) and a function which takes your input data (`data`) and the parameters and outputs the score of this parameter combination.

In pure Python, we'd write something like:

```
best = None
best_val = float("-Inf")
for p0 in range(100):
    for p1 in range(-20, 20):
        cur = score(data, p0, p1)
        if cur > best_val:
            best = p0,p1
            best_val = cur
print('Best parameter pair', best)
```

This is, obviously, an **embarassingly parallel** problem and we want *jug* to handle it.

First note: we can, of course, perform this with a *map/reduce*:

```
def mapper(data, p0, p1):
    return (p0, p1, score(data, p0, p1))

def reducer(a, b):
    _, _, av = a
    _, _, bv = b
    if av > bv: return a
    return b

best = jug.mapreduce.mapreduce(
                reducer,
                mapper,
                [(p0, p1)
                        for p0 in range(101)
                        for p1 in range(-20, 21)])
```

However, if you want to look at the whole parameter space instead of just the best score, this will not work. Instead, you can do:

```
from jug import TaskGenerator

score = TaskGenerator(score)
results = {}
for p0 in range(100):
    for p1 in range(-20, 20):
        result[p0,p1] = value(data, p0, p1)
```

Now, *after you've run ''jug execute''*, you can use `jug shell` and load the result dictionary to look at all the results.

```
result = value(result)
print(result[0, -2])
# Look for the maximum score
print(max(result.values()))
# Look at maximum score *and* the parameters that generated it:
print(max((v, k) for k, v in result.iteritems()))
```

Barriers

Often part of your control structure depends on previous computations. For example, you might load all the data, and filter some of them out based on a long computation:

```
from jug import Task
inputs = load_data()

def keep(datum):
    # A long running computation which decides whether datum should be kept
    ...

keeps = [Task(keep, i) for i in inputs]

# Now I want to throw out data
# This will NOT work:
inputs = [i for i,k in zip(inputs,keeps) if k]
results = [Task(long_computation, i) for i in inputs]
```

This will not work: the `keeps` list is a list of `Tasks` not its results.

The solution is to use a `barrier()`:

```
from jug import Task, barrier, value
inputs = load_data()

def keep(datum):
    # A long running computation which decides whether datum should be kept
    ...

keeps = [Task(keep, i) for i in inputs]

barrier() # <-------- this will divide the jugfile in two!
inputs = [i for i,k in zip(inputs, value(keeps)) if k]
results = [Task(long_computation, i) for i in inputs]
```

This effectively divides the jugfile in two or more blocks: up to the barrier call and after the barrier call. When a barrier call is reached, if there are any tasks that have not run, then the jugfile is not loaded any further. This ensures that **after the call** you can load the results of previous tasks.

`barrier()` is also useful when there are dependencies that Jug cannot see (e.g., one task which writes a file which later tasks rely on). An alternative solution (not always applicable) is to add these dependencies through unused parameters.

bvalue

New in version 0.10: bvalue() was added in version 0.10. Before this version, you needed to call barrier() & value() separately.

`bvalue` is a more targeted version of `barrier`, which combines the effect of `value()` as well. The above example could also be written as:

```
from jug import Task, bvalue
inputs = load_data()

def keep(datum):
    # A long running computation which decides whether datum should be kept
    ...

keeps = [Task(keep, i) for i in inputs]
```

```
inputs = [i for i,k in zip(inputs, bvalue(keeps)) if k]
results = [Task(long_computation, i) for i in inputs]
```

Note, however, that if there are additional tasks which are not loaded by the `bvalue()` call, the processing can continue processing them.

Compound Tasks

This is a fairly advanced topic, which you should only tackle once you have mastered the basics of Jug.

A compound task is a function that builds up a potentially-complex task. Because this function might generate many intermediate tasks which are not very meaningful, using a compound task construct allows us to throw them away once they have run.

For example:

```
@TaskGenerator
def poly(a,b,c):
    return a*b+c

@TaskGenerator
def max3(values):
    'Return the maximum 3 values'
    values.sort()
    return tuple(values[-3:])

def buildup(numbers):
    results = []
    for n in numbers:
        results.append(poly(n, 2*n, poly(n,n,n)))
    return max3(results)

# Work in Python2 & Python3
numbers = list(range(192))
intermediate = buildup(numbers)
...
```

We have a fairly complex function `buildup`, which takes the list of `N` numbers and generated `2*N+1` Tasks. These are a lot of Tasks and may make jug run slower. On the other hand, you may not want to simply have a single Task do everything:

```
def poly(a,b,c):
    return a*b+c

def max3(values):
    'Return the maximum 3 values'
    values.sort()
    return tuple(values[-3:])

@TaskGenerator
def buildup(numbers):
    results = []
    for n in numbers:
        results.append(poly(n, 2*n, poly(n,n,n)))
    return max3(results)
```

Because, this way, you will have lost any ability to have the different calls to `poly` be run in parallel.

A compound task behaves like the first example until all necessary tasks have been computed:

```
@TaskGenerator
def poly(a,b,c):
    return a*b+c

@TaskGenerator
def max3(values):
    'Return the maximum 3 values'
    values.sort()
    return tuple(values[-3:])

@CompoundTaskGenerator
def buildup(numbers):
    results = []
    for n in numbers:
        results.append(poly(n, 2*n, poly(n,n,n)))
    return max3(results)

# Work in Python2 & Python3
numbers = list(range(192))
intermediate = buildup(numbers)
...
```

Basically, when Jug sees the `CompoundTask`, it asks *is the result of all of this already available?* If yes, then it is just the current result; otherwise, the function is called immediately (**not** at execution time, but every time the jugfile is loaded).

See Also

See also the section on mapreduce.

Utilities

The module `jug.utils` has a few functions which are meant to be used in writing jugfiles.

Identity

This is simply implemented as:

```
@TaskGenerator
def identity(x):
    return x
```

This might seem like the most pointless function, but it can be helpful in speeding things up. Consider the following case:

```
from glob import glob

def load(fname):
   return open(fname).readlines()
```

```
@TaskGenerator
def process(inputs, parameter):
    ...

inputs = []
for f in glob('*.data'):
    inputs.extend(load(f))
# inputs is a large list

results = {}
for p in range(1000):
    results[p] = process(inputs, p)
```

How is this processed? Every time `process` is called, a new `jug.Task` is generated. This task has two arguments: `inputs` and an integer. When the hash of the task is computed, both its arguments are analysed. `inputs` is a large list of strings. Therefore, it is going to take a very long time to process all of the hashes.

Consider the variation:

```
from jug.utils import identity

# ...
# same as above

inputs = identity(inputs)
results = {}
for p in range(1000):
    results[p] = process(inputs, p)
```

Now, the long list is only hashed once! It is transformed into a `Task` (we reuse the name `inputs` to keep things clear) and each `process` call can now compute its hash very fast.

Using `identity` to induce dependencies

`identity` can also be used to introduce dependencies. One can define a helper function:

```
def value_after(val, token):
    from jug.utils import identity
    return identity( [val, token] )[0]
```

Now, this function, will always return its first argument, but will only run once its second argument is available. Here is a typical use case:

1. Function `process` takes an output file name
2. Function `postprocess` takes as input the output filename of `process`

Now, you want to run `process` and **then** `postprocess`, but since communication is done with files, Jug does not see that these functions depend on each other. `value_after` is the solution:

```
token = process(input, ofile='output.txt')
postprocess(value_after('output.txt', token))
```

This works independently of whatever `process` returns (even if it is `None`).

jug_execute

This is a simple wrapper around `subprocess.call()`. It adds two important pieces of functionality:

1. it checks the exit code and raises an exception if not zero (this can be disabled by passing `check_exit=False`).
2. It takes an argument called `run_after` which is ignored but can be used to declare dependencies between tasks. Thus, it can be used to ensure that a specific process only runs after something else has run:

```
from jug.utils import jug_execute
from jug import TaskGenerator

@TaskGenerator
def my_computation(input, ouput_filename):
    ...

token = my_computation(input, 'output.txt')
# We want to run gzip, but **only after** `my_computation` has run:
jug_execute(['gzip', 'output.txt'], run_after=token)
```

`jug.utils.`**`timed_path`**(*path*)

Returns a Task object that simply returns *path* with the exception that it uses the paths mtime (modification time) and the file size in the hash. Thus, if the file is touched or changes size, this triggers an invalidation of the results (which propagates to all dependent tasks).

Parameters **ipath** : str

A filesystem path

Returns **opath** : str

A task equivalent to `(lambda: ipath)`.

`jug.utils.`**`identity`**(*x*)

identity implements the identity function as a Task (i.e., `value(identity(x)) == x`)

This seems pointless, but if `x` is, for example, a very large list, then using this function might speed up some computations. Consider:

```
large = list(range(100000))
large = jug.utils.identity(large)
for i in range(100):
    Task(process, large, i)
```

This way the list `large` is going to get hashed just once. Without the call to `jug.utils.identity`, it would get hashed at each loop iteration.

https://jug.readthedocs.io/en/latest/utilities.html#identity

Parameters **x** : any object

Returns **x** : x

class `jug.utils.`**`CustomHash`**(*obj*, *hash_function*)

Set a custom hash function

This is an advanced feature and you can shoot yourself in the foot with it. Make sure you know what you are doing. In particular, hash_function should be a strong hash: `hash_function(obj0) == hash_function(obj1)` is taken to imply that `obj0 == obj1`

You can use the helpers in the `jug.hash` module (in particular `hash_one`) to help you. The implementation of `timed_path` is a good example of how to use a CustomHash:

```
def hash_with_mtime_size(path):
    from .hash import hash_one
    st = os.stat_result(os.stat(path))
    mtime = st.st_mtime
    size = st.st_size
    return hash_one((path, mtime, size))

def timed_path(path):
    return CustomHash(path, hash_with_mtime_size)
```

The `path` object (a string or bytes) is wrapped with a hashing function which checks the file value.

Parameters **obj** : any object

hash_function : function

This should take your object and return a str

`jug.utils.`**`sync_move`**(*src*, *dst*)

Sync the file and move it

This ensures that the move is truly atomic

Parameters **src** : filename

Source file

dst: filename

Destination file

Map/Reduce

Mapping, reducing and all that Jazz

Jug is **not** a map/reduce framework. However, it is still useful to sometimes frame problems in that framework. And applying the same function to a large collection of elements (in a word, *mapping*) is exactly the absurdly parallel problem that Jug excels at.

Naïve Solution

Let's say you want to double all numbers between 0 and 1023. You could do this:

```
from jug import TaskGenerator

@TaskGenerator
def double(x):
    return 2*x

numbers = range(1024)
result = map(double, numbers)
```

This might work well for this problem. However, if instead of 1,024 numbers, you had 1 million and each computation was very fast, then this would actually be very inefficient: you are generating one task per computation. As a rule of thumb, you want your computations to last at least a few seconds, otherwise, the overhead of maintaining the infrastructure becomes too large.

Grouping computations

You can use `jug.mapreduce.map` to achieve a better result:

```
from jug import mapreduce

result = mapreduce.map(double, numbers, map_step=32)
```

The `map_step` argument defines how many calls to `double` will be performed in a single Task.

You can also include a reduce step:

```
@TaskGenerator
def add(a, b):
    return a + b

final = mapreduce.map(add, double, numbers, map_step=32)
```

this is sort of equivalent to:

```
final = reduce(add, map(double, numbers))
```

except that **the order in which the reduction is done is not from left to right**! In fact, this only works well if the reduction function is associative.

Curried mapping

The above is fine, but sometimes you need to pass multiple arguments to the function you want to loop over:

```
@TaskGenerator
def distance_to(x, ref):
    return abs(x - ref)

ref = 34.34
inputs = range(1024)

result = [distance_to(x, ref) for x in inputs]
```

This works, but we are back to where we were: too many small Tasks!

`currymap` to the rescue:

```
result = mapreduce.currymap(distance_to, [(x,ref) for x in inputs])
```

Arguably this function should have been called `uncurrymap` (as it is equivalent to the Haskell expression `map . uncurry`), but that just doesn't sound right (I also like to think it's the programming equivalent to the Currywurst, a culinary concept which almost makes me chuckle).

Example

The canonical example for map/reduce is counting words in files. Here, we will do the same with some very small files:

```
inputs = [
        "banana apple apple banana",
        "apple pear football",
```

```
        "pear",
        "banana apple apple",
        "football banana",
        "apple pear",
        "waldorf salad",
        ]
```

The `mapper` function will output a dictionary of counts:

```
def count1(str):
    from collections import defaultdict
    counts = defaultdict(int)
    for word in str.split():
        counts[word] += 1
    return counts
```

(We used the very useful `collections.defaultdict`).

While the `reducer` adds two dictionaries together:

```
def merge_dicts(rhs, lhs):
    # Note that we SHOULDN'T modify arguments, so we will create a copy
    rhs = rhs.copy()
    for k,v in lhs.iteritems():
        rhs[k] += v
    return rhs
```

We can now use `jug.mapreduce.mapreduce` to put these together:

```
final_counts = jug.mapreduce.mapreduce(
                        merge_dicts,
                        count1,
                        inputs,
                        map_step=1)
```

Running `jug status` shows up the structure of our problem:

```
Task name                                    Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jug.mapreduce._jug_map_reduce                      0           6           0
↪ 0
jug.mapreduce._jug_reduce                          1           0           0
↪ 0
...............................................................................
↪..
Total:                                             1           6           0
↪ 0
```

If we had more than just 6 "files", the values in the table would be much larger. Let's also assume that this is part of some much larger programme that computes counts and then does some further processing with them.

Once that task is done, we might not care anymore about the break up into 6 units. So, we can wrap the whole thing into a **compound task**:

```
final_counts = CompoundTask(jug.mapreduce.mapreduce,
                        merge_dicts,
```

```
                        count1,
                        inputs,
                        map_step=1)
```

At first, this does not do much. The status is the same:

```
Task name                                   Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jug.compound.compound_task_execute                1           0           0
↪ 0
jug.mapreduce._jug_map_reduce                     0           6           0
↪ 0
jug.mapreduce._jug_reduce                         1           0           0
↪ 0
...............................................................................
↪..
Total:                                            2           6           0
↪ 0
```

But if we *execute* the tasks and re-check the status:

```
Task name                                   Waiting       Ready    Finished
↪Running
-------------------------------------------------------------------------------
↪--
jug.mapreduce.mapreduce                           0           0           1
↪ 0
...............................................................................
↪..
Total:                                            0           0           1
↪ 0
```

Now, `jug status` reports a single task (the mapreduce task) and it is *Finished*.

Compound tasks not only lower the cognitive load, but they also make operations such as `jug status` much faster.

Full example source code

We left out the imports above, but other than that, it is a fully functional example:

```
import jug.mapreduce
from jug.compound import CompoundTask

inputs = [
        "banana apple apple banana",
        "apple pear football",
        "pear",
        "banana apple apple",
        "football banana",
        "apple pear",
        "waldorf salad",
        ]


def count1(str):
```

```
    from collections import defaultdict
    counts = defaultdict(int)
    for word in str.split():
        counts[word] += 1
    return counts


def merge_dicts(rhs, lhs):
    # Note that we SHOULDN'T modify arguments, so we will create a copy
    rhs = rhs.copy()
    for k,v in lhs.iteritems():
        rhs[k] += v
    return rhs

#final_counts = jug.mapreduce.mapreduce(
#                        merge_dicts,
#                        count1,
#                        inputs,
#                        map_step=1)

final_counts = CompoundTask(jug.mapreduce.mapreduce,
                        merge_dicts,
                        count1,
                        inputs,
                        map_step=1)
```

Backends

What is a Jug Backend?

A jug backend serves two tasks: it saves intermediate results and it synchronises the running processes when needed.

What Backends Are Available?

There are three backend available: one is based on the filesystem, the other is a redis backend and a simple in-memory backend which does not allow sharing across processes.

Filesystem

By default, jug will save its results in a directory called `jugdata`. This is done in a way that works across **NFS** if you are using a cluster.

Redis

Redis is a non-relational database system. I assume you have already installed it (it is easy to install from source, but it is now a part of Ubuntu, so that is even easier).

1. Run a redis server (see its docs for how to control it, but simply calling `redis-server` should work).
2. Now start your jug jobs with the `--jugdir=redis://127.0.0.1/`.

In Memory Store

If you just want an in-memory store, use `--jugdir=dict_store:filename` and the results will be loaded and saved into `filename` (use just `--jugdir=dict_store` to get a run where results are *not* saved to file.

This is only appropriate for small projects, but has the lowest maintenance of any system.

Which Backend Should I Use?

If all your nodes share a filesystem and you don't want to set anything up, just use the default filesystem backend. If your computations are non-trivial (in general, you should avoid breaking up your algorithm so much that each tasks takes less than a second), then this will be fast enough and very robust.

Do note that `jug` **works well over NFS**.

If your nodes do not share a filesystem then you are going to have to use redis. For some cases (if you have many outputs of computations that do not take very long), it is also faster and, if your results are small, takes up significantly less space.

The tradeoffs are speed and space vs. convenience.

Configuration

Configuration file

New in version 1.3: In previous versions, the configuration file was called `~/.jug/configrc`. Since version 1.3, the path `~/.config/jugrc` is used (for compatibility, `~/.jug/configrc` is read if `~/.config/jugrc` is missing).

On startup, `jug` reads the file `~/.config/jugrc` (if it exists). It expects an *.ini* format file. It can have the following fields:

```
[main]
jugdir=%(jugfile)s.jugdata
jugfile=jugfile.py

[status]
cache=off

[execute]
aggressive-unload=False
pdb=False
wait-cycle-time=12
nr-wait-cycles=150
```

These have the same meaning as the analogous command line options. If both are given, the command line takes priority.

Bash Shell Helpers

Who is this document for?

This is for someone using Jug on a compute cluster where there is a dedicated head node. These scripts help launch, monitor, and terminate jug on all of the compute nodes. They should be used with caution, as they are not heavily tested and may need to be modified for your setup.

Terms used in this document

head node Computer that does not perform heavy computation locally. Instead it is in charge of managing and monitoring jug on other computers

compute node/client Interchangable terms for this doc. Computers that are connected to the head node (typically via an SSH connection) and will be performing computation.

jug instance A single jug process, typically running on a compute node

What scripts are available?

jug The normal jug executable

jugrun Starts jug executables on non-head machines. The number of executables started on each client machine should be configured to match the hardware of each client machine e.g. for 12 hardware threads, it may be reasonable to run 10 instances of jug

jugstatus Monitors jug execution by occasionally calling 'jug status'

jughalt Stops all jug executors on all compute node computers

juglog Outputs a log of jug run/halt/stop commands

jugoutput If your jug tasks print to stdout or stderr, this collects all of that content from each remote machine and prints it out. Note that this command will likely need to be configured for your computer

Installing these scripts

You need to

1. create an alias for all of the jug* commands in your `.bash_profile`
2. create a `workers.iplist` file
3. create a `.waitonjug` executable
4. create a *run_workers.sh'* script.

All of these are covered here. Also note, these scripts require `pssh`

Creating command aliases

```
local ~$ ssh myserver
myserver ~$ nano .bash_profile
# Nano opens a text editor and I paste the following at
# the top (minus the hash signs)
#
#alias pssh='pssh -i -h workers.iplist '
```

```
#alias jugstatus='watch -n 10 -d jug status --cache benchmark_jug.py'
#alias jugrun='echo "`date`: Start" >> .juglog; screen -d -m -S jugwatcher sh ~/.
↪waitonjug; pssh screen -d -S jug -m sh run_workers.sh'
#alias jugoutput='pssh cat /mnt/localhd/.jug*'
#alias jughalt='echo "`date`: Halt ">> .juglog; screen -S jugwatcher -X quit;  pssh␣
↪pkill jug;'
#alias juglog='cat ~/.juglog'
#
# Now we need to tell the server to 'reload' .bash_profile
myserver ~$ source .bash_profile
# Now type 'jug' and hit tab a few times to see the following
myserver ~$ jug
jug     jughalt     juglog      jugoutput   jugrun      jugstatus
```

Creating workers.iplist

This is pretty simple, just create a file called workers.iplist and insert something like this to identify all of your compute nodes:

```
10.0.2.2
10.0.2.4
10.0.2.5
10.0.2.6
10.0.2.7
```

Creating .waitonjug executable Create a new file called .waitonjug and insert the following

```
#!/bin/sh

echo "Waiting..."
jug sleep-until benchmark_jug.py
echo "`date`: Completed" >> ~/.juglog
echo "Complete, returning"
```

Then save the file and make it executable by typing `chmod a+x .waitonjug`.

Creating run_workers.sh

Create a new file called `run_workers.sh` and paste the following, but *be sure to modify the script!* There are a few things to modify: the number of workers, the name of your jug python code, and where to send the output.

Number of Workers: This script assumes that all compute nodes can run the same number of jug processes without being overloaded. I would generally recommend setting the number of jug processes to be slightly lower than the total number of hardware threads your compute node can support. For example, each of my compute nodes has 12 hardware threads (6 cores, 2 threads each), so I've set to run 10 jug processes per compute node.

Name of script: Below, the name of my jug script is benchmark_jug.py. Yours is likely different, so please update

Output redirection: I'm outputing stdout and stderr to `/mnt/localhd`. If your jug tasks do not use stdout or stderr, then perhaps just do `jug execute <my_jug>.py &> /dev/null &` to redirect everything to /dev/null. If you actually want output, make sure that the directory you're using for output (in my case /mnt/localhd) is NOT shared by NFS, or your workers on different machines will be overwriting the same file. Or be a boss and upgrade this script to read in the hostname ;-)

```
#!/bin/sh

JUG_PROCESSES_PER_WORKER=10

rm /mnt/localhd/.jug*
```

```
for i in {1..${JUG_PROCESSES_PER_WORKER}}
do
    jug execute benchmark_jug.py > /mnt/localhd/.jug$i.out 2> /mnt/localhd/.jug$i.err␣
↪&
done
wait
```

After creating run_workers.sh, don't forget to make it executable using `chmod a+x run_workers.sh`

Understanding the scripts

pssh Pssh is required for all of these scripts. It allows me to broadcast one command to multiple computers and receive the reply. It makes an ssh connection to each computer, executes the command, and aggregates the replies. Pssh is what reads in the `workers.iplist`

jugstatus This uses the watch command to call 'jug status' every ten seconds

jugrun *This will likely need minor modifications for your use. See the 'installation' section above.* This first posts log entry, then sets up what I call a 'watcher', which is a tiny executable that runs in the background on the host computer (actually, it runs inside of a detached screen session) and waits on jug to complete the 'jugrun' command. If you peek inside the `.waitonjug` code you will see that this 'wait on jug' logic is nothing more than 1) use jug's sleep-until 2) create log message indicating that the job is complete.

The actual business logic of jugrun is to use pssh to tell each compute node to execute the `run_workers.sh` script. The run_workers script runs on each compute node, and launches all the instances of jug. It also waits on them to be terminated (e.g. killed by either jug completing or a call to jughalt). It waits because if the script terminates before the child processes (e.g. the instances of jug) then bad things will happen

jughalt Creates a log message about halting, terminates the .waitonjug detached screen so that we don't have anyone waiting for the job to complete, and then uses pssh to issue a command to all compute node to kill all jug processes. The pkill command is used to automatically find and kill and processes names *jug*. Once the jug processes die then the run_worker.sh scripts will automatically terminate

juglog Outputs the contents of the log file from the run/halt/complete. Simple file, can be used with other options e.g. `tail -f ~/.juglog`

jugoutput *This will likely need to be modified for your use.* In my setup, all files under /home/myuser/ are shared via NFS. This means that any output files placed in my home directory can have problems as multiple jug processes are writing to the same file and NFS is trying to share that file across multiple machines. My solution was to output jug-process-specific files into a directory that is not shared by NFS, specifically /mnt/localhd on each computer. The jugoutput command uses pssh to collect all of these log files and print them to me on the head node. Useful for monitoring progress of individual jug tasks e.g. a particularly long running method call.

Frequently Asked Questions

Why the name jug?

The cluster I was using when I first started developing jug was called "juggernaut". That is too long and there is a Unix tradition of 3-character programme names, so I abbreviated it to jug.

How to work with multiple computers?

The typical setting is that all the computers have access to a networked filesystem like NFS. This means that they all "see" the same files. In this case, the default file-based backend will work nicely.

You need to start separate `jug execute` processes on each node.

See also the answer to the next question if you are using a batch system or the bash utilities page if you are not.

Will jug work on batch cluster systems (like SGE/LFS/PBS)?

Yes, it was built for it.

The simplest way to do it is to use a job array.

On LFS, it would be run like this:

```
bsub -o output.txt -J "jug[1-100]" jug execute myscript.py
```

For SGE, you often need to write a script. For example:

```
cat >>jug1.sh <<EOF
#!/bin/bash

exec jug execute myscript.py
EOF

chmod +x jug1.sh
```

Now, you can run a job array:

```
qsub -t 1-100 ./jug1.sh
```

Alternatively, depending on your set up, you can pass in the script on STDIN:

```
echo jug execute myscript.py | qsub -t 1-100
```

In any case, 100 jobs would start running with jug synchronizing their outputs.

Given that jobs can join the computation at any time and all of the communication is through the backend (file system by default), jug is especially suited for these environments.

The project gridjug integrates jug with gridmap to help run jug on SGE clusters (this is an external project).

How do I clean up locks if jug processes are killed?

Jug will attempt to clean up when exiting, including if it receives a SIGTERM signal on Unix. However, there is nothing it can do if it receives a SIGKILL (or if the computer crashes).

The solution is to run `jug cleanup` to remove all the locks.

In some cases, you can avoid the problem in the first place by making sure that SIGTERM is being properly delivered to the jug process.

For example, if you executing a script that only runs jug (like in the previous question), then use `exec` to replace the script by the jug process.

Alternatively, in bash you can set a `trap` to catch and propagate the `SIGTERM`:

```
#!/bin/bash
N_JOBS=10

pids=""
for i in $(seq $N_JOBS); do
    jug execute &
    pids="$! $pids"
done
trap "kill -TERM $pids; exit 1" TERM
wait
```

It doesn't work with random input!

Normally the problem boils down to the following:

```
from jug import Task
from random import random

def f(x):
    return x*2

result = Task(f, random())
```

Now, if you check `jug status`, you will see that you have one task, an `f` task. If you run `jug execute`, jug will execute your one task. But, now, if you check `jug status` again, there is still one task that needs to be run!

While this may be surprising, it is actually correct. Everytime you run the script, you build a task that consists of calling `f` with a different number (because it's a randomly generated number). Given that tasks are defined as the combination of a Python function and its arguments, every time you run jug, you build a different task (unless you, by chance, draw twice the same number).

My solution is typically **to set the random seed at the start of the computation explicitly**:

```
from jug import Task
from random import random, seed

def f(x):
    return x*2

seed(123) # <- set the random seed
result = Task(f, random())
```

Now, everything will work as expected.

(As an aside: given that jug was developed in a context where it is important to be able to reproduce your results, it is generally a good idea that if your computation depends on pseudo-random numbers, you be explicit about the seeds. So, *this is a feature not a bug*.)

Why does jug not check for code changes?

1) It is very hard to get this right. You can easily check Python code (with dependencies), but checking into compiled C is harder. If the system runs any command line programmes you need to check for them (including libraries) as well as any configuration/datafiles they touch.

You can do this by monitoring the programmes, but it is no longer portable (I could probably figure out how to do it on Linux, but not other operating systems) and it is a lot of work.

It would also slow things down. Even if it checked only the Python code: it would need to check the function code & all dependencies + global variables at the time of task generation.

I believe sumatra accomplishes this. Consider using it if you desire all this functionality.

2) I was also afraid that this would make people wary of refactoring their code. If improving your code even in ways which would not change the results (refactoring) makes jug recompute 2 hours of results, then you don't do it.

3) Jug supports explicit invalidation with jug invalidate. This checks your dependencies. It is not automatic, but often you need a person to understand the code changes in any case.

Can jug handle non-pickle() objects?

Short answer: No.

Long answer: Yes, with a little bit of special code. If you have another way to get them from one machine to another, you could write a special backend for that. Right now, only `numpy` arrays are treated as a special case (they are not pickled, but rather saved in their native format), but you could extend this. Ask on the mailing list if you want to learn more.

Is jug based on a background server?

No. Jug processes do not need a server running. They need a shared *backend*. This may be the filesystem or a *redis* database. But **jug does not need any sort of jug server**.

Can I pass command line arguments to a Jugfile?

Yes. They will be available using `sys.argv` as usual.

If you need to pass arguments starting with a dash, you can use `--` (double dash) to terminate option processing. For example, if your jugfile contains:

```
import sys
print(sys.argv)
```

Now you can call it as:

```
# Argv[0] is the name of the script
$ jug execute
['jugfile.py']

$ jug execute jugfile.py
['jugfile.py']

# Using a jug option does not change ``sys.argv``
$ jug execute --verbose=info jugfile.py
['jugfile.py']

$ jug execute --verbose=info jugfile.py argument
['jugfile.py', 'argument']

# Use -- to terminate argument processing
$ jug execute --verbose=info jugfile.py argument -- --arg --arg2=yes
['jugfile.py', 'argument', '--arg', '--arg2=yes']
```

The Why of Jug

This explains the *philosophy* behind **jug** and some of its design decisions.

Why jug? A Bit of History

Jug was designed to solve two intertwined problems:

1. Writing parallel code.
2. Managing intermediate files.

Up until that point, I had been writing code that saved intermediate results to files with complex filenames (e.g., `r3_d2_p0_p3_p22_v9.pp` for the results of stage 3, dataset 2, with parameters (0,3,22), running version 9 of the code). This becomes taxing on the mind that needs to keep track of things and extremely brittle. You constantly run the risk of having results that you are not sure how to reproduce again.

Therefore, I decided I was going to write a solution for this.

The initial idea was something like an enhanced Makefile language. This evolved into something similiar to scons. Very rapidily it became apparent that a good solution involved *Tasks* and saving results to files based on a hash of the inputs. This is still the basis of jug's architecture. All of this was at the paper napkin stage, written in some off time I had before I wrote some actual code.

The motivating applications were scientific applications and some of that is probably part of jug's DNA in ways that are most apparent to those outside of science.

Design Criteria

Jug was meant to *run in a queuing batch system*. Therefore, it was a good idea to have the possibility of *just adding processes* to a running process without any explicit synchronisation. This explains why **there is no central manager** handing out tasks. Tasks coordinate based on a central store such as the filesystem. This also required jug to play nice with NFS and not rely on intra-memory communication.

Another goal (not one of the original goals, but it became a feature that we felt we needed to keep) is that, as much as possible, **code should look normal**. Many scripts can be *jugified* by adding `@TaskGenerator` to a few function declarations. Part of this involves *making it unintrusive* (not necessarily light-weight or low on features, but the user code should not need much work).

jug was also meant to be used in an *exploratory development environment*. This is why we have features such as `jug shell`, `jug invalidate` (a much better alternative than attempting to selectively update "all of the affected files" after some code change), and, to a certain extent, `barrier()`. Much of the optimisation work that has been put into jug has been to support interactive work better.

As for more down to earth goals, there should **never be any known bugs** in jug. Any bug report has a promise to fix it ASAP. Fixing bugs takes priority over new features, always. To attempt to keep quality high, when a bug is found, a regression test is always written.

Writing a Backend

What is a backend

A backend is simply a store for objects. It needs to support four operations:

1. Saving Python objects associated with a key (a key is a string)

2. Loading Python objects by their key
3. Create a lock (identified by a key)
4. Release a lock (identified by a key)

There are a few other operations, like deleting an entry, that are useful, but not strictly necessary.

Backends are identified by a URI type string, generically called `jugdir`. For example, to connect to a redis backend, use `redis:host:port/database`. Your backend should support a similar scheme.

How to write a backend

You can start with the file `jug/backends/base.py` which provides a template with documentation. Implement the functions in there. This module details all the operations that are necessary to implement a jug backend.

It can be used as a starting point (template) for writing new backends.

Magic Jug Methods

This is an advanced use of jug and you can shoot yourself in the foot doing this. If you cannot figure out why this functionality could be useful, then you probably should not be using it.

Custom hash functions

Sometimes, you may want to give your objects a special hash function, either to add functionality or for efficiency. There are two ways to do it: (1) use a `CustomHash` object for simple cases or (2) add a `__jug_hash__` method for more complex ones.

Using the CustomHash wrapper

For example, here is how `timed_path` is implemented (minus the comments in the real code):

```
def hash_with_mtime_size(path):
    from .hash import hash_one
    st = os.stat_result(os.stat(path))
    mtime = st.st_mtime
    size = st.st_size
    return hash_one((path, mtime, size))

def timed_path(path):
    return CustomHash(path, hash_with_mtime_size)
```

The return value from `timed_path` is an object which behaves exactly like `path` (i.e., as a file path), but when jug needs to hash it, it calls the function `hash_with_mtime_size`.

Implementing a __jug_hash__ method

When jug wants to hash an object, first it checks whether the object has a `__jug_hash__` method. If so, it will call that and it is done. Next, it checks whether the object is one of its known types (dict, list, tuple, numpy array, ...). If so, it will use optimized code. Otherwise; it resorts to calling pickle on the object and then hashing the pickled representation.

This fallback can be very inefficient. For example, let's say you have an object which is basically just a numpy array loaded from disk, which remembers its initial location. The standard pickling method would be very inefficient compared to the optimized numpy code.

The way to solve this is to define a `__jug_hash__` method. Inside it, we can rely on the jug hashing machinery to access the optimized version!

Here is how we'd do it:

```
import numpy as np
class NamedNumpy(object):
    def __init__(self, ifile):
        self.data = np.load(ifile)
        self.name = ifile

    def transform(self, x):
        self.data *= x

    def __jug_hash__(self):
        from jug.hash import hash_one

        return hash_one({
            'type': 'NumpyPair',
            'data': self.data,
            'name': self.name,
            })
```

The function `hash_one` takes one object and hashes it using the jug machinery. Because we are passing it a dictionary, it recursively build a hash for it. Thus, our `NamedNumpy` object now has a very fast hash function.

In fact, the `CustomHash` object we saw above, just defined its `__jug_hash__` function to call whatever you pass it in.

Overriding the `value` function

Similarly to overriding the hashing, we can override the `value` call which jug used internally to load objects.

Again, `value(x)` works in the following way:

1. Does the `x.__jug_value__` member exist? If so, call it.
2. Is `x` one of the composite types it knows about (dict, list,...). If so, use special code to recursively get all the sub objects. For a list `value([x,y]) == [value(x), value(y)]`.
3. Is it a `Task` or a `Tasklet`? If so, load it from the store.
4. Otherwise, `value(x) == x`.

What if you have your own sequence object? Then you can set a `__jug_value__` method, which will be called whenever `value(self)` is needed. This is a pretty advanced use case: if you cannot figure out why this may be useful, then you probably don't need to use it.

API Documentation

JUG: Coarse Level Parallelisation for Python

The main use of jug is from the command line:

```
jug status jugfile.py
jug execute jugfile.py
```

Where `jugfile.py` is a Python script using the `jug` library.

class `jug.`**`Task`**(*f*, *dep0*, *dep1*, ..., *kw_arg0=kw_val0*, *kw_arg1=kw_val1*, ...)

Defines a task, which will call:

```
f(dep0, dep1,..., kw_arg0=kw_val0, kw_arg1=kw_val1, ...)
```

See also:

TaskGenerator function

Attributes

result	Result value

store	

Methods

can_load()	Returns whether result is available.
can_run()	Returns true if all the dependencies have their results available.
dependencies()	for dep in task.dependencies():
hash()	Returns the hash for this task.
invalidate()	Equivalent to `t.store.remove(t.hash())`.
is_loaded()	Returns True if the task is already loaded
is_locked()	Note that only calling lock() and checking the result atomically checks for the lock().
load()	Loads the results from the storage backend.
lock()	Tries to lock the task for the current process.
run([force, save])	Performs the task.
unload()	Unload results (can be useful for saving memory).
unload_recursive()	Equivalent to:
unlock()	Releases the lock.
value()	

`can_load`()

Returns whether result is available.

`can_run`()

Returns true if all the dependencies have their results available.

`dependencies`()

for dep in task.dependencies(): ...

Iterates over all the first-level dependencies of task *t*

Parameters **self** : Task

Returns **deps** : generator

A generator over all of *self* 's dependencies

See also:

recursive_dependencies retrieve dependencies recursively

hash()
Returns the hash for this task.

The results are cached, so the first call can be much slower than subsequent calls.

invalidate()
Equivalent to `t.store.remove(t.hash())`. Useful for interactive use (i.e., in `jug shell` mode).

is_loaded()
Returns True if the task is already loaded

is_locked()
Note that only calling lock() and checking the result atomically checks for the lock(). This function can be much faster, though, and, therefore is sometimes useful.

Returns **is_locked** : boolean

Whether the task **appears** to be locked.

See also:

lock create lock

unlock destroy lock

load()
Loads the results from the storage backend.

This function *always* loads from the backend even if the task is already loaded. You can use *is_loaded* as a check if you want to avoid this behaviour.

Returns Nothing

lock()
Tries to lock the task for the current process.

Returns True if the lock was acquired. The correct usage pattern is:

```
locked = task.lock()
if locked:
    task.run()
else:
    # someone else is already running this task!
```

Not that using can_lock() can lead to race conditions. The above is the only fully correct method.

Returns **locked** : boolean

Whether the lock was obtained.

result
Result value

run(*force=False*, *save=True*)
Performs the task.

Parameters **force** : boolean, optional

if true, always run the task (even if it ran before) (default: False)

save : boolean, optional

if true, save the result to the store (default: True)

unload()
Unload results (can be useful for saving memory).

unload_recursive()
Equivalent to:

```
for tt in recursive_dependencies(t): tt.unload()
```

unlock()
Releases the lock.

If the lock was not held, this may remove another thread's lock!

class jug.**Tasklet**(*base*, *f*)
A Tasklet is a light-weight Task.

It looks like a Task, behaves like a Task, but its results are not saved in the backend.

It is useful for very simple functions and is automatically generated on subscripting a Task object:

```
t = Task(f, 1)
tlet = t[0]
```

`tlet` will be a `Tasklet`

See also:

Task

Methods

can_load()
dependencies()
unload()
unload_recursive()
value()

class jug.**TaskGenerator**(*f*)
@TaskGenerator def f(arg0, arg1, ...)

...

Turns f from a function into a task generator.

This means that calling `f(arg0, arg1)` results in: `Task(f, arg0, arg1)`. This can make your jug-based code feel very similar to what you do with traditional Python.

Methods

__call__(*args, **kwargs)

class jug.**iteratetask**(*base*, *n*)

Examples:

```
a,b = iteratetask(task, 2)
for a in iteratetask(task, n):
    ...
```

This creates an iterator that over the sequence `task[0], task[1], ..., task[n-1]`.

Parameters **task** : Task(let)

n : integer

Returns iterator

jug.**value**(*obj*)

Loads a task object recursively. This correcly handles lists, dictonaries and eny other type handled by the tasks themselves.

Parameters **obj** : object

Anything that can be pickled or a Task

Returns **value** : object

The result of the task `obj`

jug.**CachedFunction**(*f*, **args*, ***kwargs*)

is equivalent to:

```
task = Task(f, *args, **kwargs)
if not task.can_load():
    task.run()
value = task.value()
```

That is, it calls the function if the value is available, but caches the result for the future.

You can often use `bvalue` to achieve similar results:

```
task = Task(f, *args, **kwargs)
value = bvalues(task)
```

This alternative method is more flexible, but will only be execute lazily. In particular, a `jug status` will not see past the `bvalue` call until `jug execute` is called to execute `f`, while a `CachedFunction` object will always execute.

Parameters **f** : function

Any function except unnamed (lambda) functions

Returns **value** : result

Result of calling `f(*args,**kwargs)`

See also:

bvalue function An alternative way to achieve similar results to `CachedFunction(f)` is using `bvalue`.

jug.**CompoundTask**(*f*, **args*, ***kwargs*)
f should be such that it returns a *Task*, which can depend on other Tasks (even recursively).

If *f* cannot been loaded, then this becomes equivalent to:

```
f(*args, **kwargs)
```

However, if it can, then we get a pseudo-task which returns the same value without *f* ever being executed.

Parameters **f** : function returning a `jug.Task`

Returns **task** : jug.Task

jug.**CompoundTaskGenerator**(*f*)
@CompoundTaskGenerator def f(arg0, arg1, ...)

...

Turns f from a function into a compound task generator.

This means that calling `f(arg0, arg1)` results in: `CompoundTask(f, arg0, arg1)`

See also:

TaskGenerator

jug.**barrier**()
In a jug file, it assures that all tasks defined up to now have been completed. If not, parsing will (temporarily) stop at that point.

This ensures that, after calling `barrier()` you are free to call `value()` to get any needed results.

See also:

bvalue function Restricted version of this function. Often faster

jug.**bvalue**(*t*)
Named after `barrier``+``value`, `value = bvalue(t)` is similar to:

```
barrier()
value = value(t)
```

except that it only checks that *t* is complete (and not all tasks) and thus can be much faster than a full `barrier()` call.

Thus, `bvalue` stops interpreting the Jugfile if its argument has not run yet. When it has run, then it returns its value.

See also:

barrier Checks that **all** tasks have results available.

jug.**set_jugdir**(*jugdir*)
Sets the jugdir. This is the programmatic equivalent of passing `--jugdir=...` on the command line.

Parameters **jugdir** : str

Returns **store** : a jug backend

jug.**init**(*jugfile={'jugfile'}*, *jugdir={'jugdata'}*, *on_error='exit'*, *store=None*)
Initializes jug (create backend connection, ...). Imports jugfile

Parameters **jugfile** : str, optional

jugfile to import (default: 'jugfile')

jugdir : str, optional

jugdir to use (could be a path)

on_error : str, optional

What to do if import fails (default: exit)

store : storage object, optional

If used, this is returned as `store` again.

Returns **store** : storage object

jugspace : dictionary

`jug.`**`is_jug_running`**`()`
Returns True if this script is being executed by jug instead of regular Python

Task: contains the Task class.

This is the main class for using jug.

There are two main alternatives:

- Use the `Task` class directly to build up tasks, such as `Task(function, arg0, ...)`.
- Rely on the `TaskGenerator` decorator as a shortcut for this.

class `jug.task.`**`Task`**(*f*, *dep0*, *dep1*, ..., *kw_arg0=kw_val0*, *kw_arg1=kw_val1*, ...)
Defines a task, which will call:

```
f(dep0, dep1,..., kw_arg0=kw_val0, kw_arg1=kw_val1, ...)
```

See also:

TaskGenerator function

Attributes

result	Result value

store	

Methods

can_load()	Returns whether result is available.
can_run()	Returns true if all the dependencies have their results available.
dependencies()	for dep in task.dependencies():
hash()	Returns the hash for this task.
invalidate()	Equivalent to `t.store.remove(t.hash())`.
is_loaded()	Returns True if the task is already loaded
is_locked()	Note that only calling lock() and checking the result atomically checks for the lock().

Continued on next page

Table 6.6 – continued from previous page

load()	Loads the results from the storage backend.
lock()	Tries to lock the task for the current process.
run([force, save])	Performs the task.
unload()	Unload results (can be useful for saving memory).
unload_recursive()	Equivalent to:
unlock()	Releases the lock.
value()	

can_load()
Returns whether result is available.

can_run()
Returns true if all the dependencies have their results available.

dependencies()

for dep in task.dependencies(): ...

Iterates over all the first-level dependencies of task *t*

Parameters **self** : Task

Returns **deps** : generator

A generator over all of *self*'s dependencies

See also:

recursive_dependencies retrieve dependencies recursively

hash()
Returns the hash for this task.

The results are cached, so the first call can be much slower than subsequent calls.

invalidate()
Equivalent to `t.store.remove(t.hash())`. Useful for interactive use (i.e., in `jug shell` mode).

is_loaded()
Returns True if the task is already loaded

is_locked()
Note that only calling lock() and checking the result atomically checks for the lock(). This function can be much faster, though, and, therefore is sometimes useful.

Returns **is_locked** : boolean

Whether the task **appears** to be locked.

See also:

lock create lock

unlock destroy lock

load()
Loads the results from the storage backend.

This function *always* loads from the backend even if the task is already loaded. You can use *is_loaded* as a check if you want to avoid this behaviour.

Returns Nothing

lock()
Tries to lock the task for the current process.

Returns True if the lock was acquired. The correct usage pattern is:

```
locked = task.lock()
if locked:
    task.run()
else:
    # someone else is already running this task!
```

Not that using can_lock() can lead to race conditions. The above is the only fully correct method.

Returns **locked** : boolean

Whether the lock was obtained.

result
Result value

run(*force=False*, *save=True*)
Performs the task.

Parameters **force** : boolean, optional

if true, always run the task (even if it ran before) (default: False)

save : boolean, optional

if true, save the result to the store (default: True)

unload()
Unload results (can be useful for saving memory).

unload_recursive()
Equivalent to:

```
for tt in recursive_dependencies(t): tt.unload()
```

unlock()
Releases the lock.

If the lock was not held, this may remove another thread's lock!

class jug.task.**Tasklet**(*base*, *f*)
A Tasklet is a light-weight Task.

It looks like a Task, behaves like a Task, but its results are not saved in the backend.

It is useful for very simple functions and is automatically generated on subscripting a Task object:

```
t = Task(f, 1)
tlet = t[0]
```

`tlet` will be a `Tasklet`

See also:

Task

Methods

can_load()
dependencies()
unload()
unload_recursive()
value()

jug.task.**recursive_dependencies**(*t*, *max_level=-1*)

for dep in recursive_dependencies(t, max_level=-1): ...

Returns a generator that lists all recursive dependencies of task

Parameters **t** : Task

input task

max_level : integer, optional

Maximum recursion depth. Set to -1 or None for no recursion limit.

Returns **deps** : generator

A generator over all dependencies

class jug.task.**TaskGenerator**(*f*)

@TaskGenerator def f(arg0, arg1, ...)

...

Turns f from a function into a task generator.

This means that calling `f(arg0, arg1)` results in: `Task(f, arg0, arg1)`. This can make your jug-based code feel very similar to what you do with traditional Python.

Methods

__call__(*args, **kwargs)

class jug.task.**iteratetask**(*base*, *n*)

Examples:

```
a,b = iteratetask(task, 2)
for a in iteratetask(task, n):
    ...
```

This creates an iterator that over the sequence `task[0], task[1], ..., task[n-1]`.

Parameters **task** : Task(let)

n : integer

Returns iterator

jug.task.**value**(*obj*)

Loads a task object recursively. This correcly handles lists, dictonaries and eny other type handled by the tasks themselves.

Parameters **obj** : object

Anything that can be pickled or a Task

Returns **value** : object

The result of the task `obj`

mapreduce: Build tasks that follow a map-reduce pattern.

jug.mapreduce.**mapreduce**(*reducer*, *mapper*, *inputs*, *map_step=4*, *reduce_step=8*)

Create a task that does roughly the following:

```
reduce(reducer, map(mapper, inputs))
```

Roughly because the order of operations might be different. In particular, *reducer* should be a true *reducer* functions (i.e., commutative and associative).

Parameters **reducer** : associative, commutative function

This should map Y_0,Y_1 -> Y'

mapper : function from X -> Y

inputs : list of X

map_step : integer, optional

Number of mapping operations to do in one go. This is what defines an inner task. (default: 4)

reduce_step : integer, optional

Number of reduce operations to do in one go. (default: 8)

Returns **task** : jug.Task object

jug.mapreduce.**map**(*mapper*, *sequence*, *map_step=4*)

sequence' = map(mapper, sequence, map_step=4)

Roughly equivalent to:

```
sequence' = [Task(mapper,s) for s in sequence]
```

except that the tasks are grouped in groups of *map_step*

Parameters **mapper** : function

function from A -> B

sequence : list of A

map_step : integer, optional

nr of elements to process per task. This should be set so that each task takes the right amount of time.

Returns **sequence'** : list of B

sequence'[i] = mapper(sequence[i])

See also:

mapreduce

currymap function Curried version of this function

jug.mapreduce.**reduce**(*reducer*, *inputs*, *reduce_step=8*)

Parameters **reducer** : associative, commutative function

This should map Y_0,Y_1 -> Y'

inputs : list of X

reduce_step : integer, optional

Number of reduce operations to do in one go. (default: 8)

Returns **task** : jug.Task object

See also:

mapreduce

jug.compound.**CompoundTask**(*f*, **args*, ***kwargs*)
f should be such that it returns a *Task*, which can depend on other Tasks (even recursively).

If *f* cannot been loaded, then this becomes equivalent to:

```
f(*args, **kwargs)
```

However, if it can, then we get a pseudo-task which returns the same value without *f* ever being executed.

Parameters **f** : function returning a `jug.Task`

Returns **task** : jug.Task

jug.compound.**CompoundTaskGenerator**(*f*)
@CompoundTaskGenerator def f(arg0, arg1, ...)

...

Turns f from a function into a compound task generator.

This means that calling `f(arg0, arg1)` results in: `CompoundTask(f, arg0, arg1)`

See also:

`TaskGenerator`

jug.compound.**compound_task_execute**(*x*, *h*)
This is an internal function. Do **not** use directly.

jug.utils.**timed_path**(*path*)
Returns a Task object that simply returns *path* with the exception that it uses the paths mtime (modification time) and the file size in the hash. Thus, if the file is touched or changes size, this triggers an invalidation of the results (which propagates to all dependent tasks).

Parameters **ipath** : str

A filesystem path

Returns **opath** : str

A task equivalent to `(lambda: ipath)`.

jug.utils.**identity**(*x*)
identity implements the identity function as a Task (i.e., `value(identity(x)) == x`)

This seems pointless, but if `x` is, for example, a very large list, then using this function might speed up some computations. Consider:

```
large = list(range(100000))
large = jug.utils.identity(large)
for i in range(100):
    Task(process, large, i)
```

This way the list `large` is going to get hashed just once. Without the call to `jug.utils.identity`, it would get hashed at each loop iteration.

https://jug.readthedocs.io/en/latest/utilities.html#identity

Parameters **x** : any object

Returns **x** : x

class `jug.utils.`**`CustomHash`**(*obj*, *hash_function*)
Set a custom hash function

This is an advanced feature and you can shoot yourself in the foot with it. Make sure you know what you are doing. In particular, hash_function should be a strong hash: `hash_function(obj0) == hash_function(obj1)` is taken to imply that `obj0 == obj1`

You can use the helpers in the `jug.hash` module (in particular `hash_one`) to help you. The implementation of `timed_path` is a good example of how to use a CustomHash:

```
def hash_with_mtime_size(path):
    from .hash import hash_one
    st = os.stat_result(os.stat(path))
    mtime = st.st_mtime
    size = st.st_size
    return hash_one((path, mtime, size))

def timed_path(path):
    return CustomHash(path, hash_with_mtime_size)
```

The `path` object (a string or bytes) is wrapped with a hashing function which checks the file value.

Parameters **obj** : any object

hash_function : function

This should take your object and return a str

`jug.utils.`**`sync_move`**(*src*, *dst*)
Sync the file and move it

This ensures that the move is truly atomic

Parameters **src** : filename

Source file

dst: filename

Destination file

History

version **1.4.0** (Tue Jan 3 2017) - Fix bug with writing very large objects to disk - Smarter handling of –aggressive-unload (do not unload what will be

immediately necessary)

- Work around corner case in *jug shell* command
- Add test-jug subcommand
- Add return_tuple decorator

version **1.3.0** (Tue Nov 1 2016) - Update *shell* subcommand to IPython 5 - Use ~/.config/jugrc as configuration file - Cleanup usage string - Use *bottle* instead of *web.py* for webstatus subcommand - Add *jug_execute* function - Add timing functionality

version **1.2.2** (Sat Jun 25 2016) - Fix bugs in shell subcommand and a few corner cases in encoding/decoding

results

version **1.2.1** (Mon Feb 15 2016) - Changed execution loop to ensure that all tasks are checked (issue #33 on

github)

- Fixed bug that made 'check' or 'sleep-until' slower than necessary
- Fixed jug on Windows (which does not support fsync on directories)
- Made Tasklets use slightly less memory

version **1.2** (Thu Aug 20 2015) - Use HIGHEST_PROTOCOL when pickle()ing - Add compress_numpy option to file_store - Add register_hook_once function - Optimize case when most (or all) tasks are already run - Add –short option to 'jug status' and 'jug execute' - Fix bug with dictionary order in kwargs (fix by Andreas Sorge) - Fix ipython colors (fix by Andreas Sorge) - Sort tasks in 'jug status'

version **1.1** (Tue Mar 3 2015) - Python 3 compatibility fixes - fsync(directory) in file backend - Jug hooks (still mostly undocumented, but already enabling internal code simplification)

version **1.0** (Tue May 20 2014) - Adapt status output to terminal width (by Alex Ford) - Add a newline at the end of lockfiles for file backend - Add –cache-file option to specify file for `status --cache`

version **0.9.7** (Tue Feb 18 2014)

- Fix use of numpy subclasses
- Fix redis URL parsing
- Fix `shell` for newer versions of IPython
- Correctly fall back on non-sqlite `status`
- Allow user to call set_jugdir() inside jugfile

version **0.9.6** (Tue Aug 6 2013)

- Faster decoding
- Add jug-execute script
- Add describe() function
- Add write_task_out() function

version **0.9.5** (May 27 2013)

- Added debug mode
- Even better map.reduce.map using blocked access
- Python 3 support
- Documentation improvements

version **0.9.4** (Apr 15 2013)

- Add CustomHash wrapper to set __jug_hash__
- Print traceback on import error
- Exit when no progress is made even with barrier

- Use Tasklets for better jug.mapreduce.map
- Use Ipython debugger if available (patch by Alex Ford)
- Faster –aggressive-unload
- Add currymap() function

version **0.9.3** (Dec 2 2012)

- Fix parsing of ports on redis URL (patch by Alcides Viamontes)
- Make hashing robust to different orders when using randomized hashing (patch by Alcides Viamontes)
- Allow regex in invalidate command (patch by Alcides Viamontes)
- Add `--cache --clear` suboption to status
- Allow builtin functions for tasks
- Fix status –cache‘‘ (a general bug which seems to be triggered mainly by `bvalue()` usage).
- Fix `CompoundTask` (broken by earlier `__jug_hash__` hook introduction)
- Make `Tasklets` more flexible by allowing slicing with `Tasks` (previously, slicing with tasks was **not** allowed)

version **0.9.2** (Nov 4 2012):

- More flexible mapreduce()/map() functions
- Make TaskGenerator pickle()able and hash()able
- Add invalidate() method to Task
- Add –keep-going option to execute
- Better help messsage

version **0.9.1** (Jun 11 2012):

- Add –locks-only option to cleanup subcommand
- Make cache file (for `status` subcommand) configurable
- Add `webstatus` subcommand
- Add bvalue() function
- Fix bug in `shell` subcommand (`value` was not in global namespace)
- Improve identity()
- Fix bug in using Tasklets and –aggressive-unload
- Fix bug with Tasklets and sleep-until/check

version **0.9**:

- In the presence of a barrier(), rerun the jugfile. This makes barrier much easier to use.
- Add set_jugdir to public API
- Added CompoundTaskGenerator
- Support subclassing of Task
- Avoid creating directories in file backend unless it is necessary
- Add jug.mapreduce.reduce (which mimicks the builtin reduce)

For older version see `ChangeLog` file.

CHAPTER 7

What do I need to run Jug?

It is a Python only package. Jug is continuously tested with Python 2.6 and up (including Python 3.3 and up).

CHAPTER 8

How does it work?

Read the tutorial.

CHAPTER 9

What's the status of the project?

Since version 1.0, jug should be considered stable.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

U

V