Table of Contents
1. Parallel Processing using Python's Dask package
2. 1. Overview of Dask
3. 2. Overview of parallel schedulers
4. 3. Implementing operations in parallel "by hand"
5. 3.1. Using a 'future' via delayed
6. 3.2. Parallel maps
7. 3.3. Delayed evaluation and task graphs
8. 3.4. The Futures interface
9. 4. Dask distributed datastructures and "automatic" parallel operations on them
10. 4.1. Arrays (numpy)
11. 4.2. Dataframes (pandas)
12. 4.3. Bags
13. 5. Using different schedulers
14. 5.1. Using threads (no copying)
15. 5.2. Multi-process parallelization via Dask Multiprocessing
16. 5.3. Multi-process parallelization via Dask Distributed (local)
17. 5.4. Distributed processing across multiple machines via an ad hoc cluster
18. 5.5. Distributed processing using multiple machines within a SLURM scheduler job
19. 6. Effective parallelization and common issues
20. 6.1. Nested parallelization and pipelines
21. 6.2. Load-balancing and static vs. dynamic task allocation
22. 6.3. Avoid repeated calculations by embedding tasks within one call to compute
23. 6.4. Copies are usually made
24. 6.5. Parallel I/O
25. 6.6. Adaptive scaling
26. 7. Monitoring jobs
27. 8. Random number generation (RNG)
28. 9. Submitting SLURM jobs from Dask
29. 9.1. Adaptive scaling

Parallel Processing using Python's Dask package

Chris Paciorek, Department of Statistics, UC Berkeley

1. Overview of Dask

The Dask package provides a variety of tools for managing parallel computations.

In particular, some of the key ideas/features of Dask are:

- Separate what to parallelize from how and where the parallelization is actually carried out.
- Different users can run the same code on different computational resources (without touching the actual code that does the computation).
- Dask provides distributed data structures that can be treated as a single data structures when runnig operations on them (like Spark and pbdR).

The idea of a 'future' or 'delayed' operation is to tag operations such that they run lazily. Multiple operations can then be pipelined together and Dask can figure out how best to compute them in parallel on the computational resources available to a given user (which may be different than the resources available to a different user).

Let's import dask to get started.

```
import dask
```

2. Overview of parallel schedulers

One specifies a "scheduler" to control how parallelization is done, including what machine(s) to use and how many cores on each machine to use.

For example,

```
import dask.multiprocessing
# spread work across multiple cores, one worker per core
dask.config.set(scheduler='processes', num_workers = 4)
```

This table gives an overview of the different scheduler.

Type	Description	Multi-node	Copies of objects made?
synchronous	not in parallel	no	no
threaded	threads within current Python session	no	no
processes	background Python sessions	no	yes
distributed	Python sessions across multiple nodes	yes or no	yes

Note that because of Python's Global Interpreter Lock (GIL), many computations done in pure Python code won't be parallelized using the 'threaded' scheduler; however computations on numeric data in numpy arrays, Pandas dataframes and other C/C++/Cython-based

code will parallelize.

For the next section (Section 3), we'll just assume use of the 'processes' schduler and will provide more details on the other schedulers in the following section (Section 4).

3. Implementing operations in parallel "by hand"

Dask has a large variety of patterns for how you might parallelize a computation.

We'll simply parallelize computation of the mean of a large number of random numbers across multiple replicates as can be seen in `calc_mean.py`.

```
from calc_mean import *
```

(Note the code in calc_mean.py is not safe in terms of parallel random number generation - see Section 8 later in this document.)

3.1. Using a 'future' via `delayed`

The basic pattern for setting up a parallel loop is:

for loop

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)

futures = []
n = 10000000
p = 10
for i in range(p):
    futures.append(dask.delayed(calc_mean)(i, n))  # add lazy task

futures
results = dask.compute(*futures)  # compute all in parallel
```

list comprehension

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)

n = 10000000
p = 10
futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]
futures
results = dask.compute(*futures)
```

You could set the scheduler in the `compute` call:

```
results = dask.compute(*future, scheduler = 'processes')
```

but it is best practice to separate what is parallelized from where the parallelization is done, specifying the scheduler at the start of your code.

3.2. Parallel maps

We can do parallel map operations (i.e., a map in the map-reduce or functional programming sense, akin to `lapply` in R).

For this we need to use the distributed scheduler, which we'll discuss more later. But note that the distributed scheduler can work on one or more nodes.

```
# ignore this setup for now; we'll see it again later
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(n_workers = 4)
c = Client(cluster)

# set up and execute the parallel map
inputs = [(i, n) for i in range(p)]
# execute the function across the array of input values
future = c.map(calc_mean_vargs, inputs)
results = c.gather(future)
results
```

The map operation appears to cache results. If you rerun the above with the same inputs, you get the same result back essentially instantaneously. HOWEVER, that means that if there is randomness in the results of your function for a given input, Dask will just continue to return the original output.

3.3. Delayed evaluation and task graphs

You can use `delayed` in more complicated situations than the simple iterations shown above.

```
def inc(x):
    return x + 1

def add(x, y):
    return x + y

x = dask.delayed(inc)(1)
y = dask.delayed(inc)(2)
z = dask.delayed(add)(x, y)
z.compute()

z.visualize(filename = 'task_graph.svg')
```

`visualize()` uses the `graphviz` package to illustrate the task graph (similar to a directed acyclic graph in a statistical model and to how Tensorflow organizes its computations).

One can also tell Dask to always delay evaluation of a given function:

```
@dask.delayed
def inc(x):
    return x + 1

@dask.delayed
def add(x, y):
    return x + y

x = inc(1)
y = inc(2)
z = add(x, y)
z.compute()
```

3.4. The Futures interface

You can also control evaluation of tasks using the Futures interface for managing tasks. Unlike use of `delayed`, the evaluation occurs immediately.

4. Dask distributed datastructures and "automatic" parallel operations on them

Dask provides the ability to work on data structures that are split (sharded/chunked) across workers. There are two big advantages of this:

- You can do calculations in parallel because each worker will work on a piece of the data.
- When the data is split across machines, you can use the memory of multiple machines to handle much larger datasets than would be possible in memory on one machine. That said, Dask processes the data in chunks, so one often doesn't need a lot of memory, even

just on one machine.

Because computations are done in external compiled code (e.g., via numpy) it's effective to use the threaded scheduler when operating on one node to avoid having to copy and move the data.

4.1. Arrays (numpy)

Dask arrays are numpy-like arrays where each array is split up by both rows and columns into smaller numpy arrays.

One can do a lot of the kinds of computations that you would do on a numpy array on a Dask array, but many operations are not possible. See here.

By default arrays are handled via the threads scheduler.

Non-distributed arrays

Let's first see operations on a single node, using a single 13 GB 2-d array. Note that Dask uses lazy evaluation, so creation of the array doesn't happen until an operation requiring output is done.

```
import dask
dask.config.set(scheduler = 'threads', num_workers = 4)
import dask.array as da
x = da.random.normal(0, 1, size=(40000,40000), chunks=(10000, 10000))
# square 10k x 10k chunks
mycalc = da.mean(x, axis = 1)  # by row
import time
t0 = time.time()
rs = mycalc.compute()
time.time() - t0  # 41 sec.
```

For a row-based operation, we would presumably only want to chunk things up by row, but this doesn't seem to actually make a difference, presumably because the mean calculation can be done in pieces and only a small number of summary statistics moved between workers.

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
# x = da.from_array(x, chunks=(2500, 40000))  # adjust chunk size of existing array
x = da.random.normal(0, 1, size=(40000,40000), chunks=(2500, 40000))
mycalc = da.mean(x, axis = 1)  # row means
import time
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 42 sec.
```

Of course, given the lazy evaluation, this timing comparison is not just timing the actual row mean calculations.

But this doesn't really clarify the story...

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
import numpy as np
import time
t0 = time.time()
x = np.random.normal(0, 1, size=(40000,40000))
time.time() - t0   # 110 sec.
# for some reason the from_array and da.mean calculations are not done lazily here
t0 = time.time()
dx = da.from_array(x, chunks=(2500, 40000))
time.time() - t0   # 27 sec.
t0 = time.time()
mycalc = da.mean(x, axis = 1)  # what is this doing given .compute() also takes time?
time.time() - t0   # 28 sec.
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 21 sec.
```

Dask will avoid storing all the chunks in memory. (It appears to just generate them on the fly.) Here we have an 80 GB array but we never use more than a few GB of memory (based on `top` or `free -h`).

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
x = da.random.normal(0, 1, size=(100000,100000), chunks=(10000, 10000))
mycalc = da.mean(x, axis = 1)  # row means
import time
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 205 sec.
rs[0:5]
```

Distributed arrays

This should be straightforward based on using Dask distributed. However, one would want to be careful about creating arrays by distributing the data from a single Python process as that would involve copying between machines.

4.2. Dataframes (pandas)

Dask dataframes are Pandas-like dataframes where each dataframe is split into groups of rows, stored as smaller Pandas dataframes.

One can do a lot of the kinds of computations that you would do on a Pandas dataframe on a Dask dataframe, but many operations are not possible. See here.

By default dataframes are handled by the threads scheduler.

Here's an example of reading from a dataset of flight delays (about 11 GB data). You can get the data here.

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.dataframe as ddf
air = ddf.read_csv('/scratch/users/paciorek/243/AirlineData/csvs/*.csv.bz2',
      compression = 'bz2',
      encoding = 'latin1',   # (unexpected) latin1 value(s) 2001 file TailNum field
      dtype = {'Distance': 'float64', 'CRSElapsedTime': 'float64',
      'TailNum': 'object', 'CancellationCode': 'object'})
# specify dtypes so Pandas doesn't complain about column type heterogeneity

air.DepDelay.max().compute()   # this takes a while
sub = air[(air.UniqueCarrier == 'UA') & (air.Origin == 'SFO')]
byDest = sub.groupby('Dest').DepDelay.mean()
byDest.compute()               # this takes a while too
```

You should see this:

```
Dest
ACV    26.200000
BFL     1.000000
BOI    12.855069
BOS     9.316795
CLE     4.000000
...
```

4.3. Bags

Bags are like lists but there is no particular ordering, so it doesn't make sense to ask for the i'th element.

You can think of operations on Dask bags as being like parallel map operations on lists in Python or R.

By default bags are handled via the multiprocessing scheduler.

Let's see some basic operations on a large dataset of Wikipedia log files. You can get a subset of the Wikipedia data here.

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)  # multiprocessing is the default
import dask.bag as db
wiki = db.read_text('/scratch/users/paciorek/wikistats/dated_2017/part-0000*gz')
import time
t0 = time.time()
wiki.count().compute()
time.time() - t0   # 136 sec.

import re
def find(line, regex = "Obama", language = "en"):
    vals = line.split(' ')
    if len(vals) < 6:
        return(False)
    tmp = re.search(regex, vals[3])
    if tmp is None or (language != None and vals[2] != language):
        return(False)
    else:
        return(True)


wiki.filter(find).count().compute()
obama = wiki.filter(find).compute()
obama[0:5]
```

Note that it is quite inefficient to do the `find()` (and implicitly reading the data in) and then compute on top of that intermediate result in two separate calls to `compute()`. More in Section 6.

5. Using different schedulers

5.1. Using threads (no copying)

```
dask.config.set(scheduler='threads', num_workers = 4)
n = 100000000
p = 4
futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]

t0 = time.time()
results = dask.compute(*futures)
time.time() - t0    # 20 sec.
```

In this case the parallelization is not very effective, as computation for a single iteration is about 5 seconds.

Note that because of Python's Global Interpreter Lock (GIL), many computations done in pure Python code cannot be parallelized using the 'threaded' scheduler; however computations on numeric data in numpy arrays, pandas dataframes and other C/C++/Cython-based code will parallelize.

5.2. Multi-process parallelization via Dask Multiprocessing

We can often more effectively parallelize by using multiple Python processes.

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)
n = 100000000
p = 4
futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]


t0 = time.time()
results = dask.compute(*futures)
time.time() - t0  # 5 sec.
```

5.3. Multi-process parallelization via Dask Distributed (local)

According to the Dask documentation, using Distributed on a local machine has advantages over multiprocessing, including the diagnostic dashboard (see Section 7) and better handling of when copies need to be made. As we saw previously using Distributed also allows us to use the handy `map()` operation.

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(n_workers = 4)
c = Client(cluster)

futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]
t0 = time.time()
results = dask.compute(*futures)
time.time() - t0   # 7 sec.
```

5.4. Distributed processing across multiple machines via an ad hoc cluster

We need to set up a scheduler on one machine (possibly the machine we are on) and workers on whatever machines we want to do the computation on.

One option is to use the `dask-ssh` command to start up the scheduler and workers. (Note that for this to work we need to have password-less SSH working to connect to the various machines.)

```
export SCHED=$(hostname)
dask-ssh --scheduler ${SCHED} radagast.berkeley.edu radagast.berkeley.edu arwen.berkeley.edu arwen.berkeley.edu &
## or:
## echo -e "radagast.berkeley.edu radagast.berkeley.edu arwen.berkeley.edu arwen.berkeley.edu" > .hosts
```

```
## dask-ssh --scheduler ${SCHED} --hostfile .hosts
```

Then in Python, connect to the cluster via the scheduler.

```
from dask.distributed import Client
c = Client(address = os.getenv('SCHED') + ':8786')
n = 100000000
p = 4

futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]
results = dask.compute(*futures)
c.close()
```

I don't see that `c.close` shuts down the scheduler and worker processes, so do it the hard way...

```
ps aux | grep dask-ssh | head -n 1 | awk -F' ' '{print $2}' | xargs kill
```

5.5. Distributed processing using multiple machines within a SLURM scheduler job

To run within a SLURM job we can use `dask-ssh` or a combination of `dask-scheduler` and `dask-worker`..

Provided that we have used --ntasks or --nodes and --ntasks-per-node to set the number of CPUs desired (and not --cpus-per-task), we can use `srun` to enumerate where the workers should run.

First we'll start the scheduler and the workers. (On Savio make sure to use up-to-date version of dask binaries (dask-scheduler and dask-worker) in user-installed recent version of Dask.)

```
export SCHED=$(hostname)
dask-scheduler&
sleep 10
srun dask-worker tcp://${SCHED}:8786 &   # might need ${SCHED}.berkeley.edu
sleep 20
```

Then in Python, connect to the cluster via the scheduler.

```
import os
from dask.distributed import Client
c = Client(address = os.getenv('SCHED') + ':8786')
n = 100000000
p = 24

futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]
t0 = time.time()
results = dask.compute(*futures)
time.time() - t0
```

Let's process the 500 GB of Wikipedia log data on Savio. (Note that when I tried this on the SCF I had some errors that might be related to the SCF not being set up for fast parallel I/O.

```
import os
from dask.distributed import Client
c = Client(address = os.getenv('SCHED') + ':8786')
import dask.bag as db
wiki = db.read_text('/global/scratch/paciorek/wikistats_full/dated/part*')
import time
t0 = time.time()
wiki.count().compute()
time.time() - t0   # 153 sec. using 96 cores on Savio

import re
def find(line, regex = "Obama", language = "en"):
    vals = line.split(' ')
    if len(vals) < 6:
        return(False)
    tmp = re.search(regex, vals[3])
    if tmp is None or (language != None and vals[2] != language):
        return(False)
    else:
        return(True)


wiki.filter(find).count().compute()
# obama = wiki.filter(find).compute()
# obama[0:5]
```

Alternatively, we can use dask-ssh, but I've had problems sometimes with using SSH to connect between nodes of a SLURM job, so the approach above is likely to be more robust as it relies on SLURM itself to connect between nodes.

```
export SCHED=$(hostname)
srun hostname > .hosts
dask-ssh --scheduler ${SCHED} --hostfile .hosts
```

6. Effective parallelization and common issues

6.1. Nested parallelization and pipelines

We can set up nested parallelization (or an arbitrary set of computations) and just have Dask's delayed functionality figure out how to do the parallelization, provided there is a single call to the compute() method.

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)
```

```
@dask.delayed
def calc_mean_vargs2(inputs, nobs):
    import numpy as np
    data = np.random.normal(inputs[0], inputs[1], nobs)
    return([np.mean(data), np.std(data)])

params = zip([0,0,1,1],[1,2,1,2])
m = 20
n = 10000000
out = list()
for param in params:
    out_single_param = list()
    for i in range(m):
        out_single_param.append(calc_mean_vargs2(param, n))
    out.append(out_single_param)

t0 = time.time()
output = dask.compute(*out)  # 15 sec. on 4 cores
time.time() - t0
```

6.2. Load-balancing and static vs. dynamic task allocation

When using `delayed`, Dask starts up each delayed evaluation separately (i.e., dynamic allocation). This is good for load-balancing, but each task induces some overhead (a few hundred microseconds). If you have many quick tasks, you probably want to break them up into batches manually, to reduce the impact of the overhead.

Even with a distributed `map()` it doesn't appear possible to ask that the tasks be broken up into batches.

6.3. Avoid repeated calculations by embedding tasks within one call to compute

As far as I can tell, Dask avoids keeping all the pieces of a distributed object or computation in memory. However, in many cases this could mean repeating computations or reading data if you need to do multiple operations on a dataset.

For example, if you are create a Dask distributed dataset from data on disk, I think this means that every distinct set of computations (each computational graph) will involve reading the data from disk again.

One implication is that if you can include all computations on a large dataset within a single computational graph (i.e., a call to `compute`) that may be much more efficient than making separate calls.

Here's an example with Dask bag on the Wikipedia data, where we make sure to do all our computations as part of one graph:

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)
import dask.bag as db
wiki = db.read_text('/scratch/users/paciorek/wikistats/dated_2017/part-0000*gz')

import re
def find(line, regex = "Obama", language = "en"):
    vals = line.split(' ')
    if len(vals) < 6:
        return(False)
    tmp = re.search(regex, vals[3])
    if tmp is None or (language != None and vals[2] != language):
        return(False)
    else:
        return(True)

total_cnt =wiki.count()
obama = wiki.filter(find)
obama_cnt = obama.count()
(obama, obama_cnt, total_cnt) = db.compute(obama, obama_cnt, total_cnt)
cnt
obama[0:5]
```

If you time doing the computations above, you'll see that doing it all together is much faster than the time of each of the three operations done separately.

Note that when reading from disk, disk caching by the operating system (saving files that are used repeatedly in memory) can also greatly speed up I/O (and can easily confuse you in terms of timing your code...).

6.4. Copies are usually made

Except for the 'threads' scheduler, copies will be made of all objects passed to the workers.

In general you want to delay the input objects. There are a couple reasons why:

- Dask hashes the object to create a name, and if you pass the same object as an argument multiple times, it will repeat that hashing.
- When using the distributed scheduler, delaying the inputs will prevent sending the data separately for every task (rather it should send the data separately for each worker).

In this example, most of the "computational" time is actually spent transferring the data rather than computing the mean.

```
dask.config.set(scheduler = 'processes', num_workers = 4)

import numpy as np
x = np.random.normal(size = 40000000)
x = dask.delayed(x)   # here we delay the data

def calc(x, i):
    return(np.mean(x))

out = [dask.delayed(calc)(x, i) for i in range(20)]
t0 = time.time()
output = dask.compute(*out)
time.time() - t0   # about 20 sec. so ~4 sec. per task


## Actual computation is much faster than 4 sec. per task
y = np.random.normal(size = 40000000)
t0 = time.time()
calc(y, 1)
time.time() - t0
```

Here we see that if we use the Distributed (local) scheduler, we get much faster performance, likely because Dask avoids making copies of the input for each task.

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(n_workers = 4)
c = Client(cluster)

import numpy as np
x = np.random.normal(size = 40000000)
x = dask.delayed(x)  # here we delay the data

def calc(x, i):
    return(np.mean(x))

out = [dask.delayed(calc)(x, i) for i in range(20)]
t0 = time.time()
output = dask.compute(*out)
time.time() - t0    # 2.5 sec.
```

That took a few seconds if we delay the data but takes ~40 seconds if we don't. Note that Dask does warn us if it detects a situation like this where we haven't delayed the data.

6.5. Parallel I/O

For this to make sense we want to be on a system where we can read multiple files without having the bottleneck of accessing a single disk. For example the Savio filesystem is set up for fast parallel I/O.

(Note that I'm using the 'processes' scheduler here because I ran into issues with the workers starting when using a Distributed (local) cluster.)

```
import dask.multiprocessing
dask.config.set(scheduler = 'processes', num_workers = 24)


## define a function that reads data but doesn't need to return entire
## dataset back to master process to avoid copying cost
def readfun(yr):
    import pandas as pd
    out = pd.read_csv('/global/scratch/paciorek/airline/' + str(yr) + '.csv.bz2',
                      header = 0, encoding = 'latin1')
    return(len(out))

results = []
for yr in range(1988, 2009):
    results.append(dask.delayed(readfun)(yr))


import time
t0 = time.time()
output = dask.compute(*results)  # parallel I/O
time.time() - t0   ## 75 seconds for 21 files
```

```
results2 = []
t0 = time.time()
for yr in range(1988, 1990):
    results2.append(readfun(yr))


time.time() - t0  ## 40 sec. just for the first two
```

6.6. Adaptive scaling

With a resource manager like Kubernetes, Dask can scale the number of workers up and down to adapt to the computational needs of a workflow. Similarly, if submitting jobs to SLURM via Dask, it will scale up and down automatically - see Section 9.

7. Monitoring jobs

By default Dask wants to use port 8787 for a web interface showing status. This port is occupied on my machine, so I'll select a different port.

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(diagnostics_port = 8786, n_workers = 4)
c = Client(cluster)

## open a browser to localhost:8786, then watch the progress
## as the computation proceeds

p = 40

futures = [dask.delayed(calc_mean)(i, n) for i in range(p)]
t0 = time.time()
results = dask.compute(*futures)
time.time() - t0
```

8. Random number generation (RNG)

In the code above, I was cavalier about the seeds for the random number generation in the different parallel computations.

Using the basic numpy RNG, one could simply set different seeds for each task. While these streams of random numbers are unlikely to overlap, it doesn't seem possible to guarantee that. If you'd like more guarantee, one option is the RandomGen package, which you can you as replacement for numpy's RNG. It allows you to "jump" the generator ahead by a large number of values, as well as to use several generators that allow for independent streams (similar to the L'Ecuyer functionality in R).

Here's some template code for using RandomGen and jumping ahead by 2^128 numbers in the Mersenne-Twister

```
n = 5
import randomgen as rg
seed = 1
rng = rg.MT19937(seed)
rng.jump(1)
random = rng.generator
random.normal(0, 1, n)
```

So here's how you can use that in a parallelized context:

```
def calc_mean(i, n, rng):
    import numpy as np
    rng.jump(i)  ## jump in steps of 2^128, one jump per task
    random = rng.generator
    data = random.normal(size = n)
    return([np.mean(data), np.std(data)])

import dask.multiprocessing
dask.config.set(scheduler = 'processes', num_workers = 4)

results = []
n = 10000000
p = 10
import randomgen as rg
seed = 1
rng = rg.MT19937(seed)  ## set an overall seed and pass the MT object to the workers

for i in range(p):
    results.append(dask.delayed(calc_mean)(i, n, rng))  # add lazy task
```

```
output = dask.compute(*results)  # compute all in parallel
```

Apparently you may be able to address this issue in dask.array using the RandomState class but I'm having trouble finding documentation on this.

```
import dask.array as da
state = da.random.RandomState(1234)
help(state)  # chunks arg is not well documented
```

9. Submitting SLURM jobs from Dask

One can submit jobs to a scheduler such as SLURM from within Python. In general I don't recommend this as it requires you to be running Python within a stand-alone server while the SLURM job is running, but here is how one can do it.

Note that the SLURMCluster() call defines the parameters of a single SLURM job, but `scale()` is what starts one or more jobs. If you ask for more workers than there are processes defined in your job definition, more than one SLURM job will be launched.

Be careful to request as many processes as cores; if you leave out `processes`, it will assume only one Python process (i.e., Dask worker) per job. (Also the memory argument is required.)

The `queue` argument is the SLURM partition you want to submit to.

```
import dask_jobqueue
## Each job will include 16 Dask workers and a total of 16 cores (1 per worker)
cluster = dask_jobqueue.SLURMCluster(processes=16, cores=16, memory='24 GB',
                                     queue='low', walltime = '3:00:00')

## This will now start 32 Dask workers, which in this case
## requires two SLURM jobs of 16 workers each.
cluster.scale(32)
from dask.distributed import Client
c = Client(cluster)

## carry out your parallel computations

cluster.close()
```

On a cluster like Savio where you may need to provide an account (-A flag), you pass that via the `project` argument to `SLURMCluster`.

9.1. Adaptive scaling

If you use `cluster.adapt()` in place of `cluster.scale()`, Dask will start and stop SLURM jobs to start and stop workers as needed. Note that on a shared cluster, you will almost certainly want to set a maximum number of workers to run at once so you don't accidentally submit 100s or 1000s of jobs.

I'm still figuring out how this works. It seems to work well when having each SLURM job control one worker on one core - in that case Dask starts a set of workers and uses those workers to iterate through the tasks. However when I try to use 16 workers per SLURM job, Dask submits a series of single 16-core jobs rather than using two 16-core jobs that stay active while working through the tasks.

```
import dask_jobqueue
## Each job will include 16 Dask workers and a total of 16 cores (1 per worker)
## This should now start up to 32 Dask workers, but only one set of 16 workers seems to start
## and SLURM jobs start and stop in quick succession.
## cluster = dask_jobqueue.SLURMCluster(processes=16, cores=16, memory='24 GB')

## In contrast, this seems to work well.
cluster = dask_jobqueue.SLURMCluster(cores = 1, memory = '24 GB')

cluster.adapt(minimum=0, maximum=32)

from dask.distributed import Client
c = Client(cluster)

## carry out your parallel computations
p=200
n=10000000
inputs = [(i, n) for i in range(p)]
# execute the function across the array of input values
future = c.map(calc_mean_vargs, inputs)
results = c.gather(future)

cluster.close()
```