

Parallel computing

This notebook is heavily based on [this \(https://github.com/dask/dask-tutorial/blob/master/01_dask.delayed.ipynb\)](https://github.com/dask/dask-tutorial/blob/master/01_dask.delayed.ipynb) official dask tutorial.

We first "reserve" some CPU. We will see in the next chapter exactly how this is done.

```
In [1]: from time import sleep
        from dask.distributed import Client
```

```
In [ ]: client = Client()
```

There are two simple situations where we can benefit from parallelization: we can have a series of **independent** functions e.g. in a data processing pipeline, or we can have multiple **independent** calls to a given function in a for loop. Let's see the first case, and learn how Dask deals with it.

Delaying computation

For the purpose of illustration, we imagine that we have two functions that are slow to execute. To simulate the slowness we just pause execution within the functions for a few seconds using `time.sleep`, let's say here 1s.

```
In [4]: def inc(x):
        sleep(1)
        return x + 1

        def add(x, y):
            sleep(1)
            return x + y
```

If we use the timing magic function we can check that the following "script" takes 3s:

```
In [8]: %%time
x = inc(1)
y = inc(2)
z = add(x, y)

CPU times: user 176 ms, sys: 26.7 ms, total: 203 ms
Wall time: 3.01 s
```

In principle we could execute the two first lines in parallel as the variables are independent. The solution implemented by Dask is the following:

- "Take notes" on what the program should be doing
- With that information, smartly split the problem into subproblems
- Send each problem to an individual process (e.g. two cores)

So how do we tell Dask what tasks it should take into account when "taking notes"? The solution is to use the function `delayed()` which takes as input another function. Every function "decorated" with the delay function will be included in the Dask flow. Let's try it:

```
In [9]: from dask import delayed
```

```
In [32]: %%time
x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

```
CPU times: user 686 µs, sys: 540 µs, total: 1.23 ms
Wall time: 820 µs
```

We see that the time consumed is minimal. But has `z` been really calculated ?

```
In [33]: z
```

```
Out[33]: Delayed('add-901f6d69-1c83-47a7-b38c-5d1ea3a03e1a')
```

No! `z` is also a delayed object now. Dask *knows* how to calculate it but *hasn't done it yet*. To effectively get the result of `z`, we have to use the `compute()` function or method:

```
In [38]: %%time
z.compute()
```

```
CPU times: user 146 ms, sys: 20.9 ms, total: 167 ms
Wall time: 2.03 s
```

```
Out[38]: 5
```

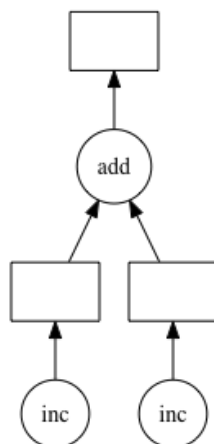
The computation time is now only 2s instead of 3. Indeed the two first calls to `inc` could be done in parallel, making us gain 1s in the process.

What is Dask actually doing: Task graph

Dask offers a very useful way to visualize how the task are split through the `visualize()` method that one can use on any delayed variable. Let's check e.g. what is `z`:

```
In [40]: z.visualize()
```

```
Out[40]:
```



Each "leaf" of that tree starts an independent calculation that can be sent to an independent process if available. Here the two `inc()` calls start separately and are then combined in the `add()` call.

Generating this **Task graph** and handling the flow of information between processes is **the main task of Dask**.

We can make our calculation slightly more complex and see what happens:

```
In [65]: %%time
x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)

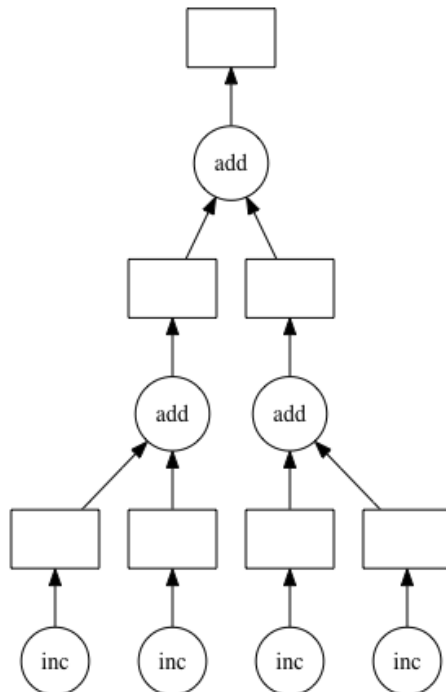
x2 = delayed(inc)(1)
y2 = delayed(inc)(2)
z2 = delayed(add)(x2, y2)

total = delayed(add)(z, z2)

CPU times: user 1.86 ms, sys: 1.16 ms, total: 3.02 ms
Wall time: 2.52 ms
```

```
In [66]: total.visualize()
```

Out[66]:

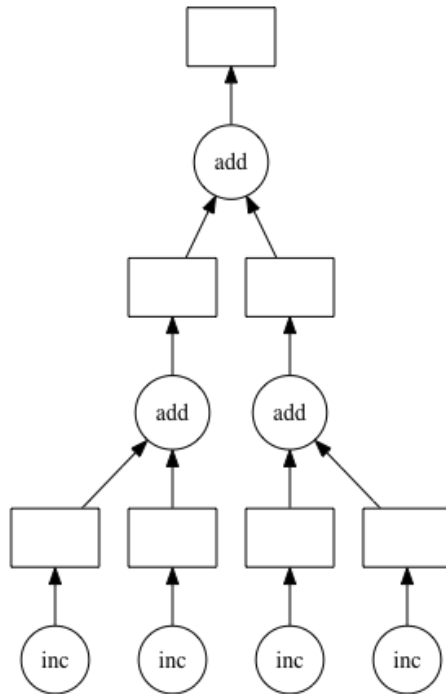


Let's however look at alternative ways of calculating this last sum. First, what happens if we just use a regular `+` sign?

```
In [67]: total_plus = z + z2
```

```
In [68]: total_plus.visualize()
```

```
Out[68]:
```

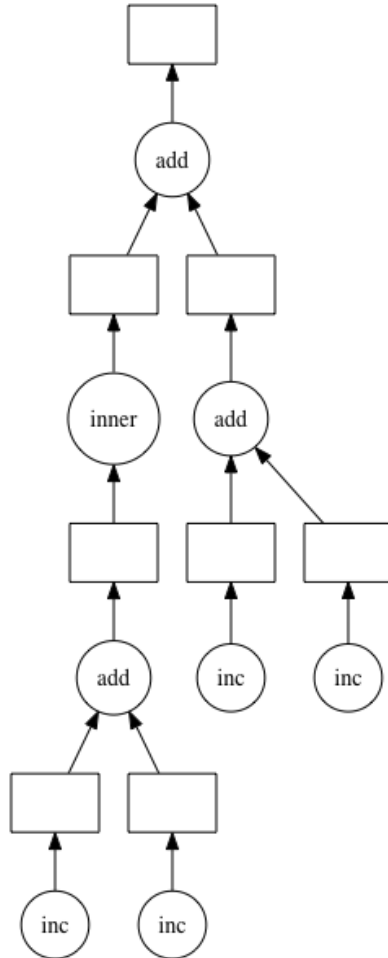


We obtain exactly the same graph. Dask knows about standard operations and automatically includes them in the task graph. Let's see if we use the standard Python `sum()` function:

```
In [69]: total_sum = sum([z, z2])
```

```
In [70]: total_sum.visualize()
```

```
Out[70]:
```



We now see an **important** difference! z and $z2$ are calculated in **sequence** and not in parallel (except for their "internal definitions"). What happens here is that `sum()` not being included in the task graph, triggers the computation first of z and **then** of $z2$.

This shows that one should be careful when using `delay()` for parallelization. Other recommendations are: not delaying a delayed function, breaking long code into multiple delayed functions etc. (see [here \(https://docs.dask.org/en/latest/delayed-best-practices.html\)](https://docs.dask.org/en/latest/delayed-best-practices.html) for more details).

Exercises

If you haven't executed the notebook until now, import the packages and start the client.

```
In [ ]: from dask.distributed import Client
        from dask import delayed
        client = Client()
```

Based on what we just learned apply the `inc()` function on the list data (using a for loop or a comprehension list) and then multiply all elements using the numpy function `np.prod()`. Check the task graph to make sure everything happens as expected:

```
In [103]: data = [3,8,1,4,2,9,4,6]
```

```
In [ ]: newdata = [delayed(inc)(x) for x in data]
newdata
import numpy as np
prod = np.prod(newdata)
prod
prod.visualize()
# prod is a delayed object
# -> bad, it is doing everything sequential, not parallel
prod = delay(np.prod)(newdata)
prod.visualize()
# -> ok, now everything is run in parallel

# NB: make sense to use parallell if you have huge of data,
# otherwise it can even take longer time ...
```

Applied problem: Sequence of operations on images

```
In [1]: import glob, os
        from dask.distributed import Client
        from dask import delayed
        import skimage.io
        import skimage.filters
        import numpy as np
        import matplotlib.pyplot as plt
```

A very common problem when dealing with image processing, is to have a set of images in a folder and having to apply a time-consuming operation on all of them.

Let's first get the names of all images:

```
In [2]: filenames = glob.glob('../Data/BBBC032_v1_dataset/*.tif')
        filenames

Out[2]: ['../PyImageCourse/Data/BBBC032_v1_dataset/BMP4blastocystC2.tif',
        '../PyImageCourse/Data/BBBC032_v1_dataset/BMP4blastocystC3.tif',
        '../PyImageCourse/Data/BBBC032_v1_dataset/BMP4blastocystC1.tif',
        '../PyImageCourse/Data/BBBC032_v1_dataset/BMP4blastocystC0.tif']
```

Dask is not good at parsing filenames so we transform those into absolute paths:

```
In [ ]: filenames = [os.path.abspath(f) for f in filenames]
```

We can import a single image using the `io` module of scikit-image:

```
In [3]: image = skimage.io.imread(filenames[0])
```

```
In [4]: image.shape
```

```
Out[4]: (172, 1344, 1024)
```

It is a quite large image representing volume data. Typical image filtering functions could be relatively slow on this especially with large kernels. We are going to do a gaussian filtering on only part of the image and then measure the mean value of the array:

```
In [24]: %%time
        image = skimage.io.imread(filenames[0])
        filtered = skimage.filters.gaussian(image[0:40,:,:),0.1)
        mean_val = np.mean(im)

        CPU times: user 1.59 s, sys: 725 ms, total: 2.31 s
        Wall time: 2.23 s
```

If we execute that function on all images we are obviously going to spend about 1min on this. Let's try to make it faster using Dask:

```
In [9]: client = Client()
```

In [11]: client

Out[11]:

Client	Cluster
Scheduler: tcp://127.0.0.1:62025	Workers: 4
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 4
	Memory: 17.18 GB

```
In [29]: %%time
all_vals = []
for f in filenames:
    im = skimage.io.imread(f)
    im = skimage.filters.gaussian(im[0:40,:,:),0.1)
    mean_val = np.mean(im)
    all_vals.append(mean_val)
np.max(all_vals)

CPU times: user 6.91 s, sys: 2.61 s, total: 9.52 s
Wall time: 9.15 s
```

Out[29]: 0.00791776016748083

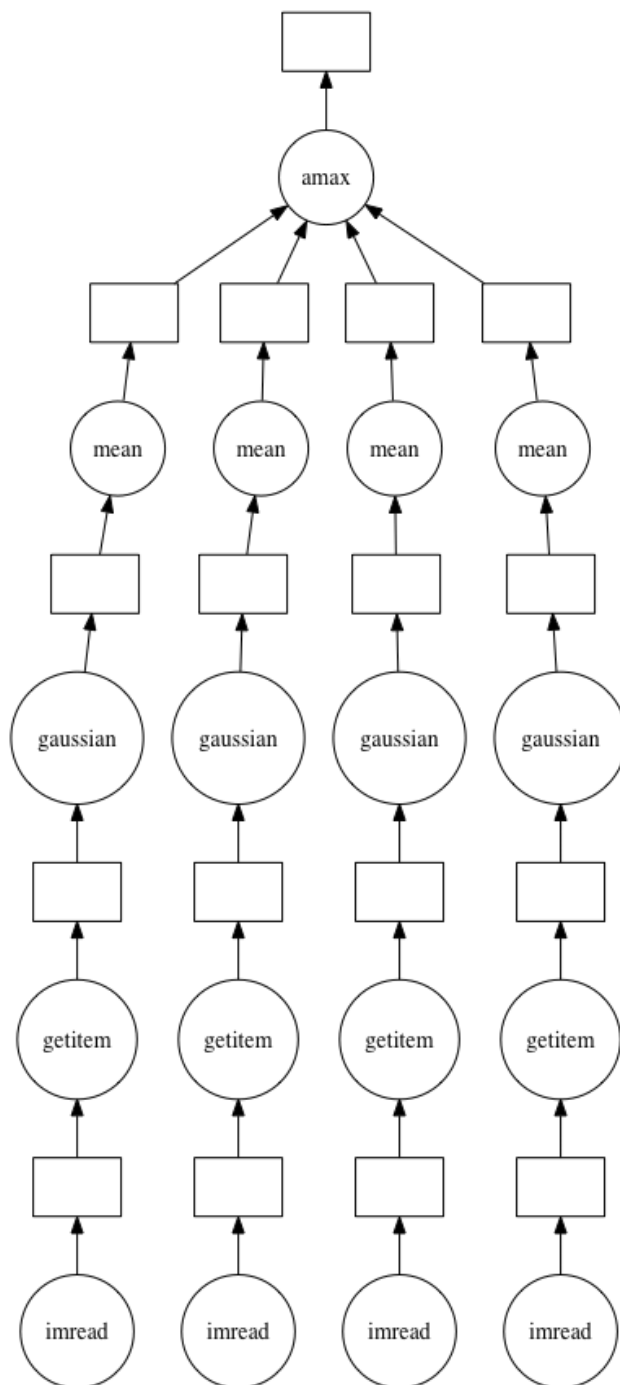
```
In [30]: all_vals = []
for f in filenames:
    im = delayed(skimage.io.imread)(f)
    im = delayed(skimage.filters.gaussian)(im[0:40,:,:),0.1)
    mean_val = delayed(np.mean)(im)
    all_vals.append(mean_val)
```

```
In [31]: max_mean = delayed(np.max)(all_vals)
```



```
In [32]: max_mean.visualize()
```

```
Out[32]:
```



```
In [33]: %%time  
max_mean.compute()
```

```
CPU times: user 301 ms, sys: 24.4 ms, total: 325 ms  
Wall time: 3.98 s
```

```
Out[33]: 0.00791776016748083
```

Creating a cluster and observing it

There are two ways of allocating computing resources in Dask. `dask.distributed` offers more control and can be run both on a local machine (laptop) or on a cluster, and we therefore focus on this.

The allocation and access to resources happens through two objects: a cluster and a client to access it.

Cluster creation

First we create a cluster, here a local cluster on this machine, but it could also be a cluster e.g. on an HPC facility. We can specify many options here, in particular the number of workers (separate Python processes), the threads per worker etc.:

```
In [2]: from dask.distributed import LocalCluster
```

```
In [3]: cluster = LocalCluster(n_workers=1, threads_per_worker=1)
```

Connecting to cluster via client

Now we can connect a client to our cluster to be able to actually use it and submit computations. This is so to say our interface to the cluster:

```
In [4]: from dask.distributed import Client
```

```
In [5]: client = Client(cluster)
```

Here is our client:

```
In [6]: client
```

```
Out[6]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:64240	Workers: 1
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 1
	Memory: 17.18 GB

We see on the right a summary of the status of our cluster and on the left, two important addresses:

- The dashboard address (of the type <http://127.0.0.1:8787/status> (<http://127.0.0.1:8787/status>)) leads us to a dashboard where we can monitor the activity of the cluster. The Dask extension for Jupyterlab allows us to have access to the monitoring dashboard directly within Jupyterlab. We can just choose the Dask icon on the left and use the Dashboard address to access all panels. **Note that when using a Jupyterhub you have to change the address to <https://address-of-hub.ch/user/your-user-name/proxy/8787> (<https://address-of-hub.ch/user/your-user-name/proxy/8787>)**
- The scheduler address allows us to create additional workers, e.g. directly in the terminal, by pointing them to the correct scheduler and typing:

```
dask-worker "tcp://127.0.0.1:55323"
```

On top of many methods and attributes, the `client` also offers an interactive interface to the cluster:

```
In [7]: client.cluster
```

Here we can interactively select the number of workers and say if we want it to be automatically adjusted.

Using Dask extension

An alternative to create a cluster, is to use the Dask extension. If you click on "+NEW", a cluster is automatically created. To use it in your notebook simply click on the "<>" button which will add the client to your current notebook.

```
In [28]: from IPython.display import HTML

HTML("""
<video width="800" controls>
  <source src="images/cluster_extension.mp4" type="video/mp4">
</video>
""")
```

```
Out[28]:
```

No video with supported format and MIME type found.

Example

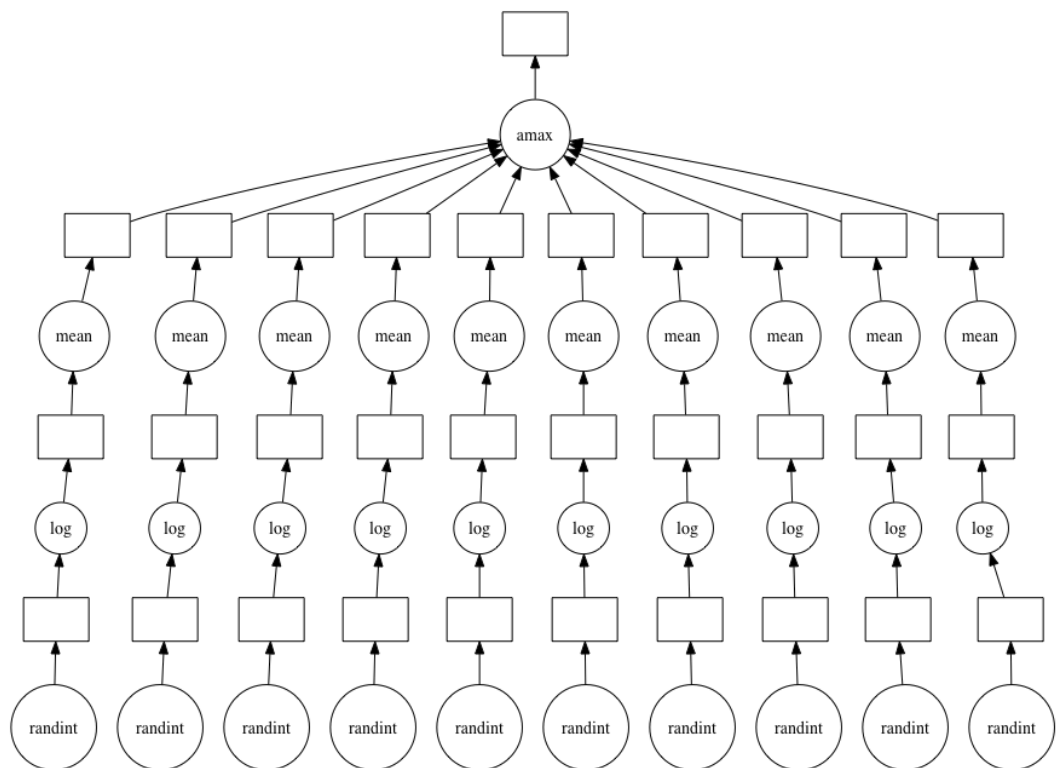
We are going to see all these pieces in action now using a simple example handling numpy arrays.

```
In [8]: from dask import delayed
import numpy as np
```

```
In [16]: all_vals = []
for i in range(10):
    ar = delayed(np.random.randint)(1,100,(1000,1000,10))
    ar = delayed(np.log)(ar)#[:,::2,::2,0]
    mean_val = delayed(np.mean)(ar)
    all_vals.append(mean_val)
maxval = delayed(np.max)(all_vals)
```

```
In [17]: maxval.visualize()
```

```
Out[17]:
```



```
In [11]: maxval.compute()
```

```
Out[11]: 3.6278351697511226
```

Now we do the same operation for a lot more iterations:

```
In [12]: all_vals = []
for i in range(1000):
    ar = delayed(np.random.randint)(1,100,(1000,1000,10))
    ar = delayed(np.log)(ar)#[:,::2,::2,0]
    mean_val = delayed(np.mean)(ar)
    all_vals.append(mean_val)
maxval = delayed(np.max)(all_vals)
```

```
In [13]: cluster
```

```
In [ ]: maxval.compute()
```

Since this is very slow, we can now stop the operation, add a worker and restart.

Alternatively, as described above, to avoid stopping the calculation, we can dynamically assing new workers to the scheduler:

```
In [50]: client
```

```
Out[50]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:55323	Workers: 4
Dashboard: http://127.0.0.1:55324/status (http://127.0.0.1:55324/status)	Cores: 7
	Memory: 68.72 GB

Asynchronous calculation

We have just seen how to add new workers to a running computation. We had to do that through the command line, because the computation is blocking the execution of other tasks until completion. We can go around this by executing the tasks asynchronously using the `client.compute()` function instead of the regular `compute()` method.

Asynchronous calculation is a complex topic about which you can learn more e.g. [here](https://distributed.dask.org/en/latest/manage-computation.html) (<https://distributed.dask.org/en/latest/manage-computation.html>).

```
In [18]: value = client.compute(maxval)
```

```
In [22]: value.result()
```

```
Out[22]: 3.6278351697511226
```

Exercise

Open a new notebook and just try to create a cluster, a client, a dashboard, and run some code on it.

Dask (numpy) arrays

As mentioned before, there are other solutions to perform parallel computing in Python. However Dask offers an crucial feature not present in other libraries: a built-in parallelized implementation of large parts of the popular libraries Numpy and Pandas. In other terms, no need to systematically use `delayed` or think how to optimize a function, Dask has already done it for you!

Here we will first explore possibilities offered by `dask-arrays`, the equivalent of numpy arrays. As usual, we first create our cluster:

```
In [1]: from dask.distributed import Client

client = Client("tcp://127.0.0.1:63517")
client
```

```
Out[1]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:63517	Workers: 4
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 4
	Memory: 17.18 GB

Dask-arrays are numpy-delayed arrays

The equivalent of the `numpy` import is the `dask.array` import:

```
In [2]: import dask.array as da
import numpy as np
```

A great feature of `dask.array` is that it mirror very closely the Numpy API, so if you are familiar with the latter, you should have no problem with dask.

For example let's create an array of random numbers and check that they behave the same way:

```
In [20]: nprand = np.random.randint(0,100, (4,5))
```

```
In [21]: darand = da.random.randint(0,100, (4,5))
```

```
In [22]: nprand.shape
```

```
Out[22]: (4, 5)
```

```
In [23]: darand.shape
```

```
Out[23]: (4, 5)
```

Let's look that the arrays directly:

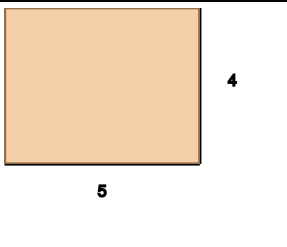
```
In [27]: nprand
```

```
Out[27]: array([[14, 98, 63,  6, 62],
                [ 7,  7, 16, 53, 85],
                [90, 87, 60, 32, 90],
                [92, 83, 90, 57, 23]])
```

```
In [28]: darand
```

```
Out[28]:
```

	Array	Chunk
Bytes	160 B	160 B
Shape	(4, 5)	(4, 5)
Count	1 Tasks	1 Chunks
Type	int64	numpy.ndarray



Here we see already a difference. Numpy just shows the matrix, while dask shows us a much richer output, including size, type, dimensionality etc.

But do the darand values exist anywhere ? Let's check that we can find the maximum in the array:

```
In [33]: darand.max()
```

```
Out[33]:
```

	Array	Chunk
Bytes	8 B	8 B
Shape	()	()
Count	3 Tasks	1 Chunks
Type	int64	numpy.ndarray

Again, we get some info but no values. In fact, as with `de\ayed` before, the values have not been computed yet!

The logic is the same as with `delayed`. Any time we actually want a result we can call the `compute` method:

```
In [35]: darand.max().compute()
```

```
Out[35]: 98
```

There could also be intermediate steps:

```
In [37]: myval = 10*darand.max()
```

```
In [40]: myval.compute()
```

```
Out[40]: 980
```

Dask re-implements many standard array creation functions, including `zeros()`, `ones()` and many of the `np.random` module.

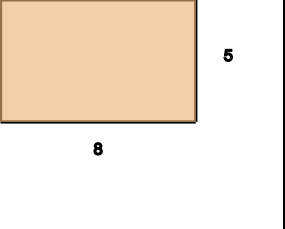
However one can also create arrays directly from a numpy array:

```
In [29]: da_array = da.from_array(np.ones((5,8)))
```

```
In [30]: da_array
```

Out[30]:

	Array	Chunk
Bytes	320 B	320 B
Shape	(5, 8)	(5, 8)
Count	1 Tasks	1 Chunks
Type	float64	numpy.ndarray



Dask-arrays are distributed

Let's create a larger array and see how it is handled by Dask and compare it with Numpy:

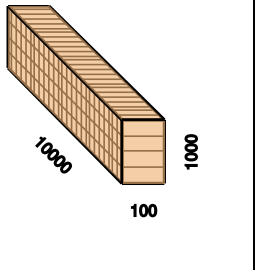
```
In [78]: large_narray = np.random.randint(0,100,(10000,1000,100))
```

```
In [75]: myarray = da.random.randint(0,100,(10000,1000,100))
```

```
In [77]: myarray
```

Out[77]:

	Array	Chunk
Bytes	8.00 GB	80.00 MB
Shape	(10000, 1000, 100)	(400, 250, 100)
Count	100 Tasks	100 Chunks
Type	int64	numpy.ndarray



First, notice how the array visualisation is helpful! Second, note that we have information about "chunks". When handling larger objects, Dasks automatically breaks them into chunks that can be generated or operated on by different workers in a parallel way. We can compute the mean of this array and observe what happens:

```
In [66]: mean = myarray.mean()
```

```
In [67]: mean.visualize()
```

```
Out[67]:
```

```
In [68]: mean.compute()
```

```
Out[68]: 49.4997707582
```

Slicing like in Numpy

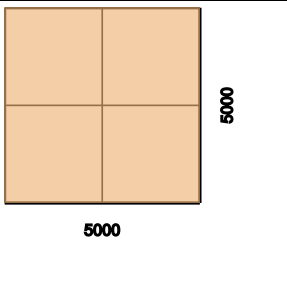
One of the main feature of numpy array is the possibility to slice and index them. Great news: dask arrays behave exactly in the same way for most "regular" cases (e.g. it doesn't implement slicing with multiple lists). Let's see how it works:


```
In [5]: myarray = da.random.random((5000,5000))
```

```
In [6]: myarray
```

Out[6]:

	Array	Chunk
Bytes	200.00 MB	50.00 MB
Shape	(5000, 5000)	(2500, 2500)
Count	4 Tasks	4 Chunks
Type	float64	numpy.ndarray



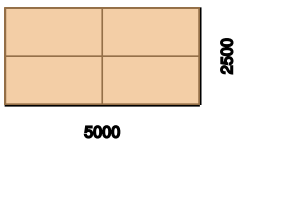
For example we can slice the array:

```
In [8]: sliced_array = myarray[::2,:]
```

```
In [9]: sliced_array
```

Out[9]:

	Array	Chunk
Bytes	100.00 MB	25.00 MB
Shape	(2500, 5000)	(1250, 2500)
Count	8 Tasks	4 Chunks
Type	float64	numpy.ndarray



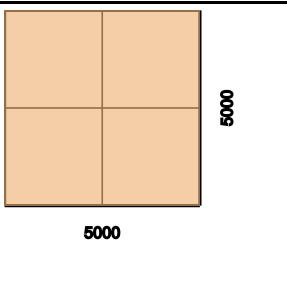
Or we can use logical indexing. First we create a logical array:

```
In [11]: logical_array = myarray > 0.5
```

```
In [12]: logical_array
```

Out[12]:

	Array	Chunk
Bytes	25.00 MB	6.25 MB
Shape	(5000, 5000)	(2500, 2500)
Count	8 Tasks	4 Chunks
Type	bool	numpy.ndarray



And then use it for logical indexing:

```
In [13]: extracted_values = myarray[logical_array]
```

In [14]: `extracted_values`

Out[14]:

	Array	Chunk
Bytes	unknown	unknown
Shape	(nan,)	(nan,)
Count	48 Tasks	4 Chunks
Type	float64	numpy.ndarray

Of course here for example we don't know the size of the resulting length. This is a typical case where any downstream parallelization becomes difficult as chunks of the array cannot be distributed. However we can get the result:

In [15]: `values = extracted_values.compute()`

In [16]: `values`

Out[16]: `array([0.65084936, 0.51052718, 0.90765229, ..., 0.86515662, 0.68235459, 0.56506943])`

Numpy functions just work!

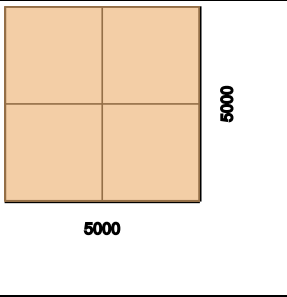
An extremely useful features of Dask is that whenever you are handling a dask-array you can apply most of the Numpy funtions to it and it remains a dask-array, **i.e. it gets integrated in the task graph**. For example:

In [45]: `cos_array = np.cos(myarray)`

In [46]: `cos_array`

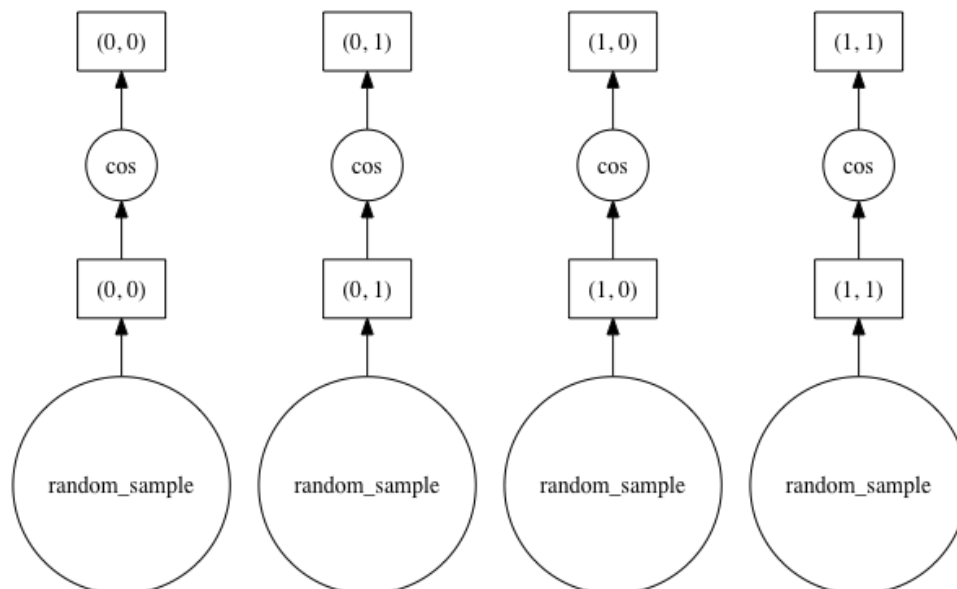
Out[46]:

	Array	Chunk
Bytes	200.00 MB	50.00 MB
Shape	(5000, 5000)	(2500, 2500)
Count	8 Tasks	4 Chunks
Type	float64	numpy.ndarray



```
In [47]: cos_array.visualize()
```

```
Out[47]:
```

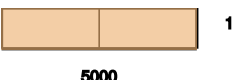


Dask also re-implements many numpy functions internally so that they are accessible as methods of the dask-arrays:

```
In [58]: proj = myarray.sum(axis = 0)
```

```
In [59]: proj
```

```
Out[59]:
```

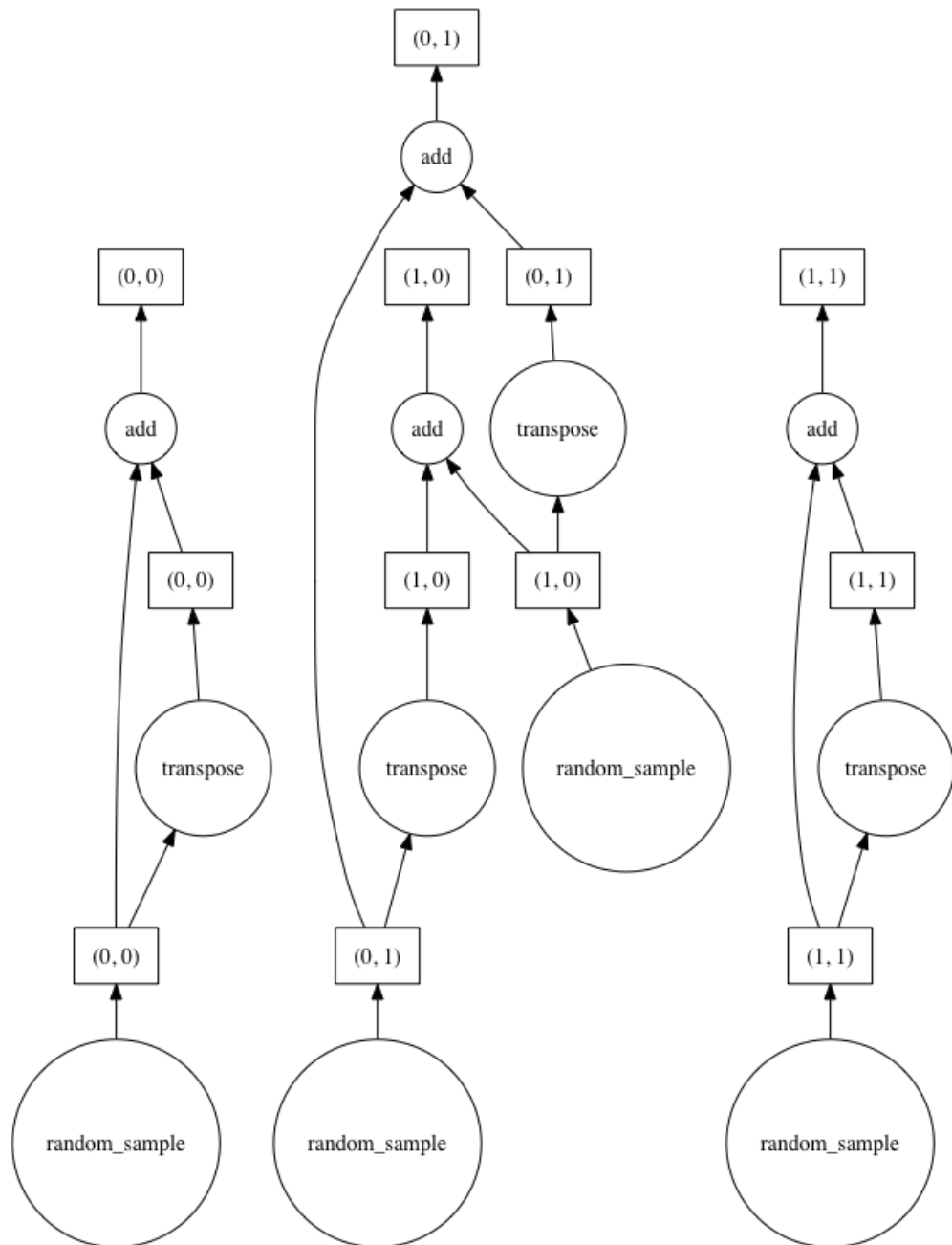
	Array	Chunk	
Bytes	40.00 kB	20.00 kB	
Shape	(5000,)	(2500,)	
Count	10 Tasks	2 Chunks	
Type	float64	numpy.ndarray	

The great advantage of dask-arrays is that functions have been optimized in order to make the task-graph very efficient. For example this simple calculation produces already a quite complex task graph. If handling large "out-of-RAM" array with numpy, one would have to break up the large array and be very smart about how to process each task.

```
In [61]: newda = myarray + da.transpose(myarray)
```

```
In [62]: newda.visualize()
```

```
Out[62]:
```

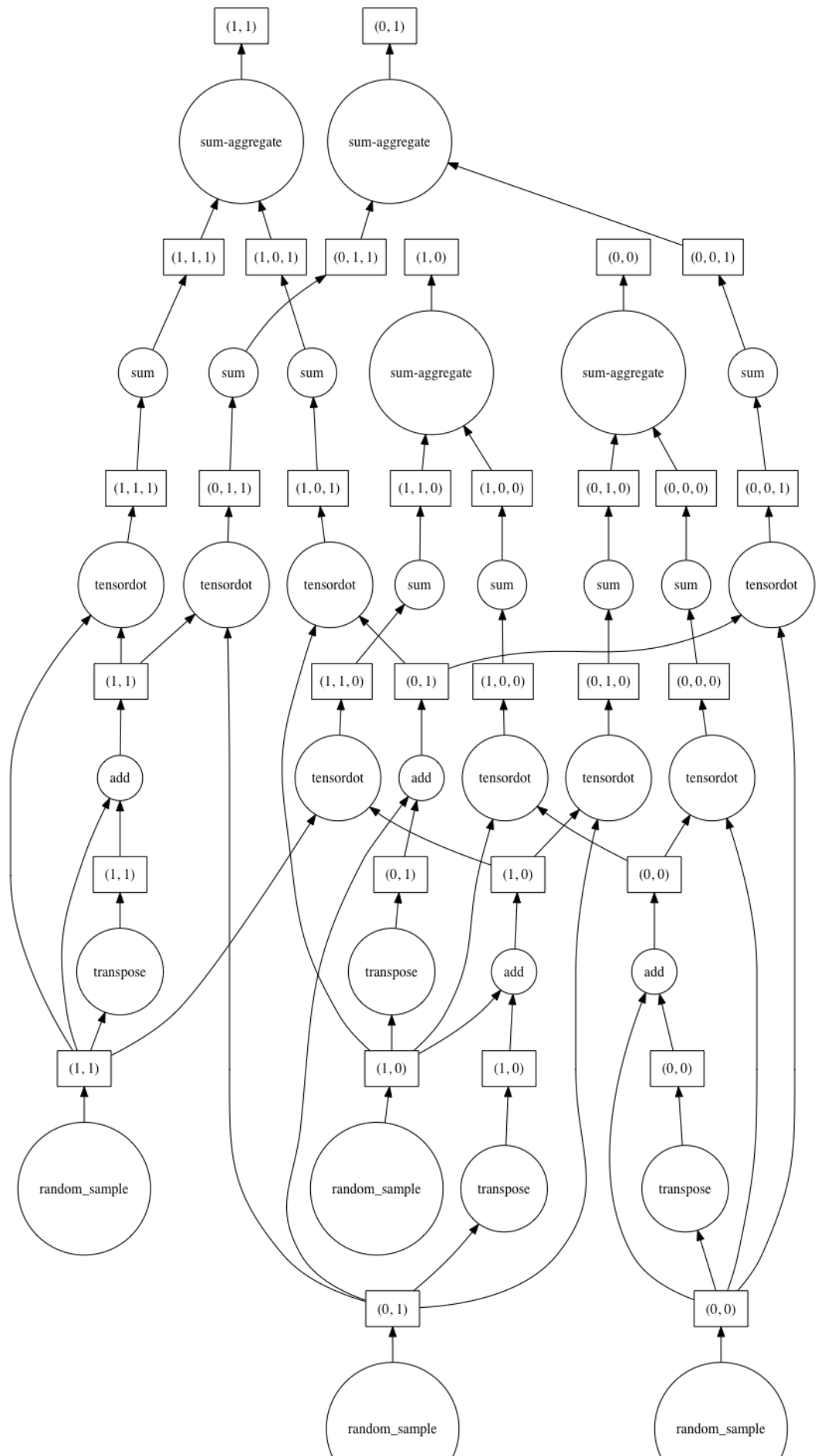


This is already quite complicated, but it can become much more complicated very quickly.

```
In [63]: newda = da.dot(myarray, myarray + da.transpose(myarray))
```

```
In [64]: newda.visualize()
```

Out[64]:



```
In [65]: %%time
         computed_array = newda.compute();

CPU times: user 161 ms, sys: 190 ms, total: 351 ms
Wall time: 5.42 s
```

```
In [66]: myarray2 = np.random.random((5000,5000))
```

```
In [67]: %%time
         newnp = np.dot(myarray2, myarray2 + np.transpose(myarray2))

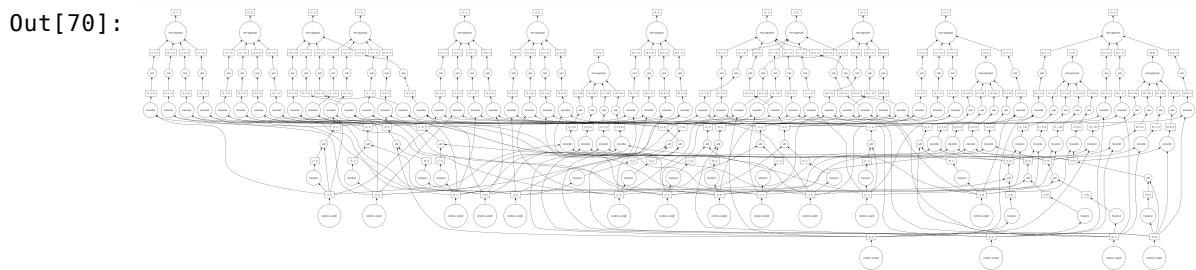
CPU times: user 10.7 s, sys: 212 ms, total: 10.9 s
Wall time: 3.62 s
```

We see here that for a reasonably sized array, the overhead time needed to push data between processes makes Dask slower than basic Numpy, so be careful in what context you use Dask! But Dask scales nicely:

```
In [68]: myarray = da.random.random((10000,10000))
```

```
In [69]: newda = da.dot(myarray, myarray + da.transpose(myarray))
```

```
In [70]: newda.visualize()
```



Limitations

Of course there are limitations to what one can do. For example, most linear algebra functions are not dask compatible:

```
In [12]: myarray = da.random.random((10,10))

         eigenval, eigenvect = np.linalg.eig(myarray);

/Users/gw18g940/miniconda3/envs/dask-tutorial/lib/python3.7/site-packages
/dask/array/core.py:1333: FutureWarning: The `numpy.linalg.eig` function
is not implemented by Dask array. You may want to use the da.map_blocks f
unction or something similar to silence this warning. Your code may stop
working in a future release.
  FutureWarning,
```

The result is not a dask array:

```
In [15]: eigenval

Out[15]: array([[ 4.72657183+0.j,  0.25040593+0.91024704j,
                  0.25040593-0.91024704j,  0.79482289+0.j,
                  0.74611566+0.j,  0.43071796+0.j,
                 -0.89630506+0.j,  -0.52818014+0.18462008j,
                 -0.52818014-0.18462008j, -0.39702164+0.j])
```

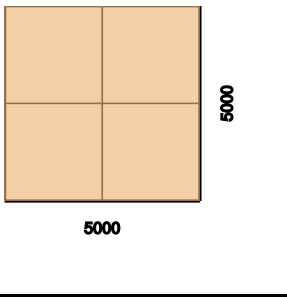
Also some operations such as those reshaping arrays may pose difficulties to Dasks as they require reshuffling array chunks. For example:

```
In [18]: myarray = da.zeros((5000,5000))
```

```
In [19]: myarray
```

Out[19]:

	Array	Chunk
Bytes	200.00 MB	50.00 MB
Shape	(5000, 5000)	(2500, 2500)
Count	4 Tasks	4 Chunks
Type	float64	numpy.ndarray



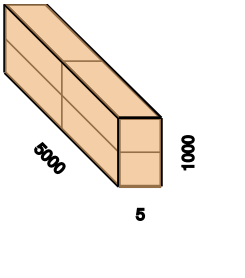
This works because it's easy to reshuffle some chunks:

```
In [20]: reshaped = np.reshape(myarray, (5000,1000,5))
```

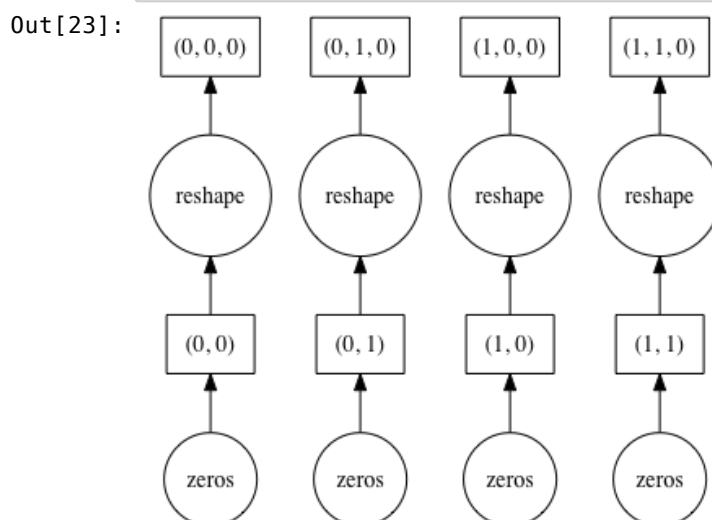
```
In [21]: reshaped
```

Out[21]:

	Array	Chunk
Bytes	200.00 MB	50.00 MB
Shape	(5000, 1000, 5)	(2500, 500, 5)
Count	8 Tasks	4 Chunks
Type	float64	numpy.ndarray



```
In [23]: reshaped.visualize()
```



But this doesn't:


```

In [22]: reshaped = np.reshape(myarray, (1000, 5000, 5))

-----
ValueError                                Traceback (most recent call last)
<ipython-input-22-a69696d87bee> in <module>
----> 1 reshaped = np.reshape(myarray, (1000, 5000, 5))

<__array_function__ internals> in reshape(*args, **kwargs)

~/miniconda3/envs/dask-tutorial/lib/python3.7/site-packages/dask/array/core.py in __array_function__(self, func, types, args, kwargs)
    1357         if da_func is func:
    1358             return handle_nonmatching_names(func, args, kwargs)
--> 1359         return da_func(*args, **kwargs)
    1360
    1361     @property

~/miniconda3/envs/dask-tutorial/lib/python3.7/site-packages/dask/array/reshape.py in reshape(x, shape)
    193
    194     # Logic for how to rechunk
--> 195     inchunks, outchunks = reshape_rechunk(x.shape, shape, x.chunk
s)
    196     x2 = x.rechunk(inchunks)
    197

~/miniconda3/envs/dask-tutorial/lib/python3.7/site-packages/dask/array/reshape.py in reshape_rechunk(inshape, outshape, inchunks)
     62         oleft -= 1
     63         if reduce(mul, outshape[oleft : oi + 1]) != din:
--> 64             raise ValueError("Shapes not compatible")
     65
     66         # TODO: don't coalesce shapes unnecessarily

ValueError: Shapes not compatible

```

While it actually works in numpy:

```

In [26]: numpy_array = np.zeros((5000, 5000))

In [27]: reshaped = np.reshape(numpy_array, (1000, 5000, 5))

In [28]: reshaped.shape
Out[28]: (1000, 5000, 5)

```

Exercise

Try to solve this exercise. Regularly check the visual representation of arrays and of the task-graph to understand what is going on.

1. Create a dask-array of normally distributed values with mean=9, and sigma = 1 of size 5000x5000
2. Add to it a **numpy** array of the same size and filled with ones. What kind of array do you obtain ?
3. Use numpy-style indexing to recover only the values smaller than 10
4. Can you find how to create a dask-histogram of those values?
5. Compute the histogram and try to plot the result using matplotlib

```
In [ ]: import dask.array as da
import numpy as np

da_array = da.random.normal(loc=9, scale=1, size=(5000, 5000))

np_array = np.ones((5000,5000))

added = da_array + np_array

added
# output is still a dask array
```

```
In [ ]: masked = added[added < 10]
```

```
In [ ]: # in numpy, no need to specify bins and range (automatically chosen if n
ot specified)
# here it requires to specify bins and range
# (dask has to know what you want to now how to distribute things)
myhist, bins = da.histogram(masked, bins=100, range=[-9,11])
myhist.visualize()
```

```
In [ ]: myhist.compute()
```

```
In [1]: from dask.distributed import Client

client = Client("tcp://127.0.0.1:56643")
client
```

```
Out[1]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:56643	Workers: 4
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 4
	Memory: 17.18 GB

Dask dataframes

Just like numpy arrays, Dask implements an equivalent of the Pandas dataframe. Let's briefly remember what a dataframe is by loading some tabular data:

```
In [2]: import pandas as pd
```

```
In [3]: births = pd.read_csv('../Data/Birthdays.csv')
```

```
In [3]: births
```

```
Out[3]:
```

	Unnamed: 0	state	year	month	day	date	wday	births
0	1	AK	1969	1	1	1969-01-01	Wed	14
1	2	AL	1969	1	1	1969-01-01	Wed	174
2	3	AR	1969	1	1	1969-01-01	Wed	78
3	4	AZ	1969	1	1	1969-01-01	Wed	84
4	5	CA	1969	1	1	1969-01-01	Wed	824
...
372859	372860	VT	1988	12	31	1988-12-31	Sat	21
372860	372861	WA	1988	12	31	1988-12-31	Sat	157
372861	372862	WI	1988	12	31	1988-12-31	Sat	167
372862	372863	WV	1988	12	31	1988-12-31	Sat	45
372863	372864	WY	1988	12	31	1988-12-31	Sat	18

372864 rows × 8 columns

A dataframe is a table where each line represents an observation and that can contain numerical values or categorical values (it can contain lists or other complex objects, but rather shouldn't). Each line has an index that can be used to recover that line:

```
In [4]: births.loc[0]
```

```
Out[4]: Unnamed: 0      1  
state      AK  
year      1969  
month      1  
day        1  
date      1969-01-01  
wday      Wed  
births     14  
Name: 0, dtype: object
```

One can recover each variable either for a specific index:

```
In [5]: births.loc[0].state
```

```
Out[5]: 'AK'
```

or for the entire dataframe:

```
In [6]: births.state
```

```
Out[6]: 0      AK  
1      AL  
2      AR  
3      AZ  
4      CA  
      ..  
372859 VT  
372860 WA  
372861 WI  
372862 WV  
372863 WY  
Name: state, Length: 372864, dtype: object
```

Dataframes also support numpy-like indexing:

```
In [7]: births.state == 'AK'
```

```
Out[7]: 0      True  
1     False  
2     False  
3     False  
4     False  
      ...  
372859 False  
372860 False  
372861 False  
372862 False  
372863 False  
Name: state, Length: 372864, dtype: bool
```

```
In [8]: births[births.state == 'AK']
```

```
Out[8]:
```

	Unnamed: 0	state	year	month	day	date	wday	births
0	1	AK	1969	1	1	1969-01-01	Wed	14
51	52	AK	1969	1	2	1969-01-02	Thurs	20
102	103	AK	1969	1	3	1969-01-03	Fri	20
153	154	AK	1969	1	4	1969-01-04	Sat	16
204	205	AK	1969	1	5	1969-01-05	Sun	18
...
372609	372610	AK	1988	12	27	1988-12-27	Tues	38
372660	372661	AK	1988	12	28	1988-12-28	Wed	40
372711	372712	AK	1988	12	29	1988-12-29	Thurs	31
372762	372763	AK	1988	12	30	1988-12-30	Fri	28
372813	372814	AK	1988	12	31	1988-12-31	Sat	29

7306 rows × 8 columns

```
In [ ]: births.mean()
```

Pandas is a huge library that offers all necessary tools for advanced data science and is used in many other packages such as the plotting library seaborn or the machine learning packages scikit-learn (you can learn a bit more about Pandas e.g. [here](https://github.com/guiwitz/Pandas_course) (https://github.com/guiwitz/Pandas_course)).

Import as dask-dataframe

We import now the same csv file, but now as a dask-dataframe and not a pandas-dataframe:

```
In [4]: from dask import dataframe as dd
```

```
In [5]: births_da = dd.read_csv('../Data/Birthdays.csv')
```

```
In [8]: len(births_da.compute())
```

```
Out[8]: 372864
```

```
In [6]: births_da
```

```
Out[6]: Dask DataFrame Structure:
```

	Unnamed: 0	state	year	month	day	date	wday	births
npartitions=1								
	int64	object	int64	int64	int64	object	object	int64

Dask Name: from-delayed, 3 tasks

Again, we see that there are no actual data there. All Dask did was to read the first lines to figure out the columns and types. If we want to have a clearer idea of the file content we can use `head()`:

```
In [7]: births_da.head()
```

Out[7]:

	Unnamed: 0	state	year	month	day	date	wday	births
0	1	AK	1969	1	1	1969-01-01	Wed	14
1	2	AL	1969	1	1	1969-01-01	Wed	174
2	3	AR	1969	1	1	1969-01-01	Wed	78
3	4	AZ	1969	1	1	1969-01-01	Wed	84
4	5	CA	1969	1	1	1969-01-01	Wed	824

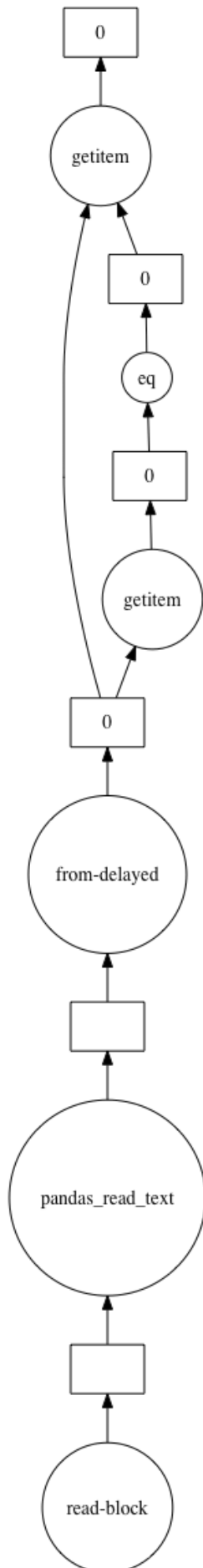
Now we can now do the same fancy indexing that we did before:

```
In [8]: subtable = births_da[births_da.state == 'AK']
```

and see that there are still no data there. Let's look at the task graph:

```
In [9]: subtable.visualize()
```

Out[9]:



As the file is small, no tasks are parallelized. But we can do this artificially by forcing dask to break the file into smaller chunks:

```
In [10]: births_da = dd.read_csv('../Data/Birthdays.csv',  
                                blocksizes=5e6)
```

```
In [12]: len(births_da.compute())
```

```
Out[12]: 372864
```

```
In [11]: births_da
```

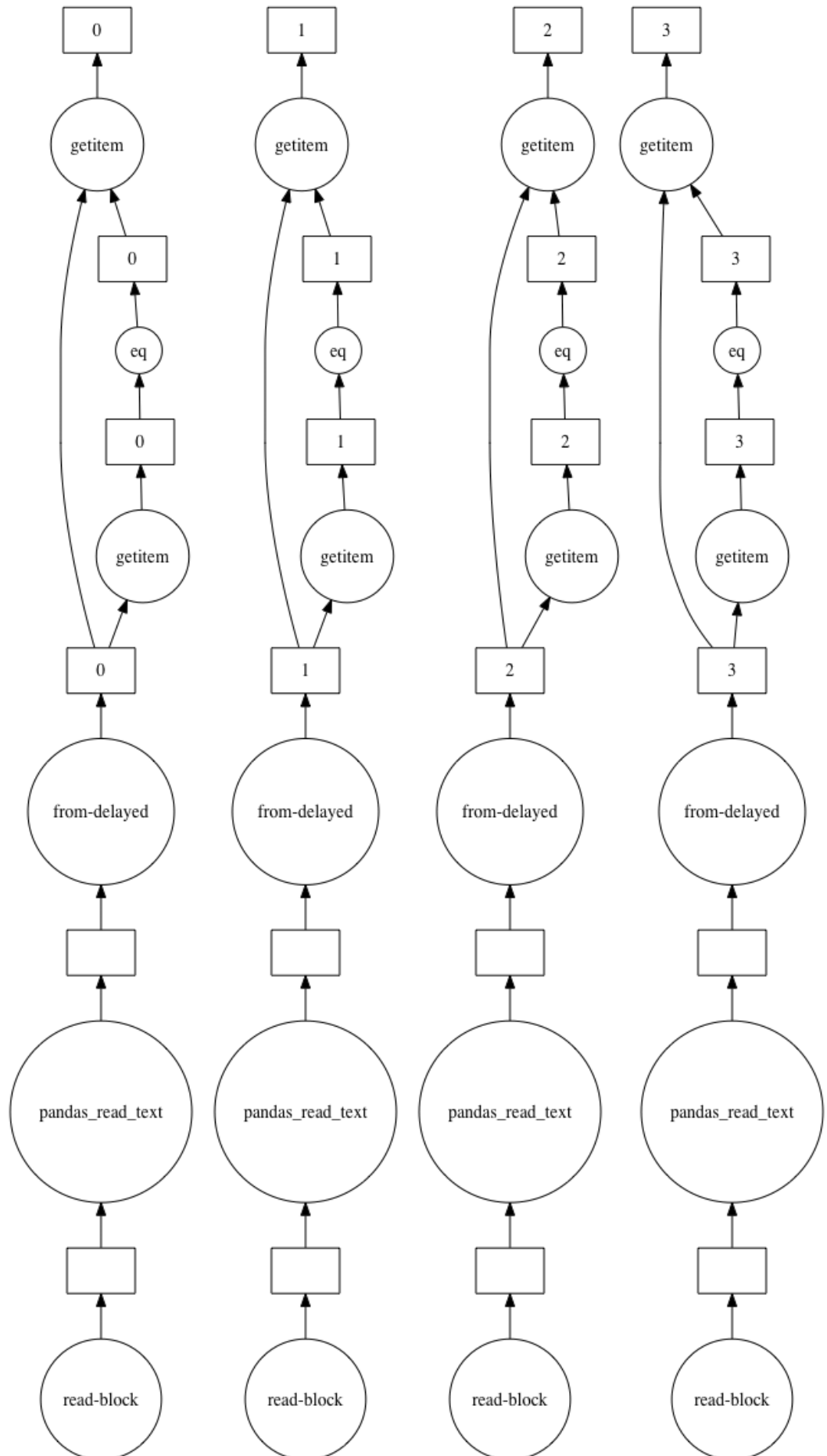
```
Out[11]: Dask DataFrame Structure:
```

	Unnamed: 0	state	year	month	day	date	wday	births
npartitions=4								
	int64	object	int64	int64	int64	object	object	int64

Dask Name: from-delayed, 12 tasks

```
In [13]: subtable = births_da[births_da.state == 'AK']  
         subtable.visualize()
```

Out[13]:



Other classic Pandas applications

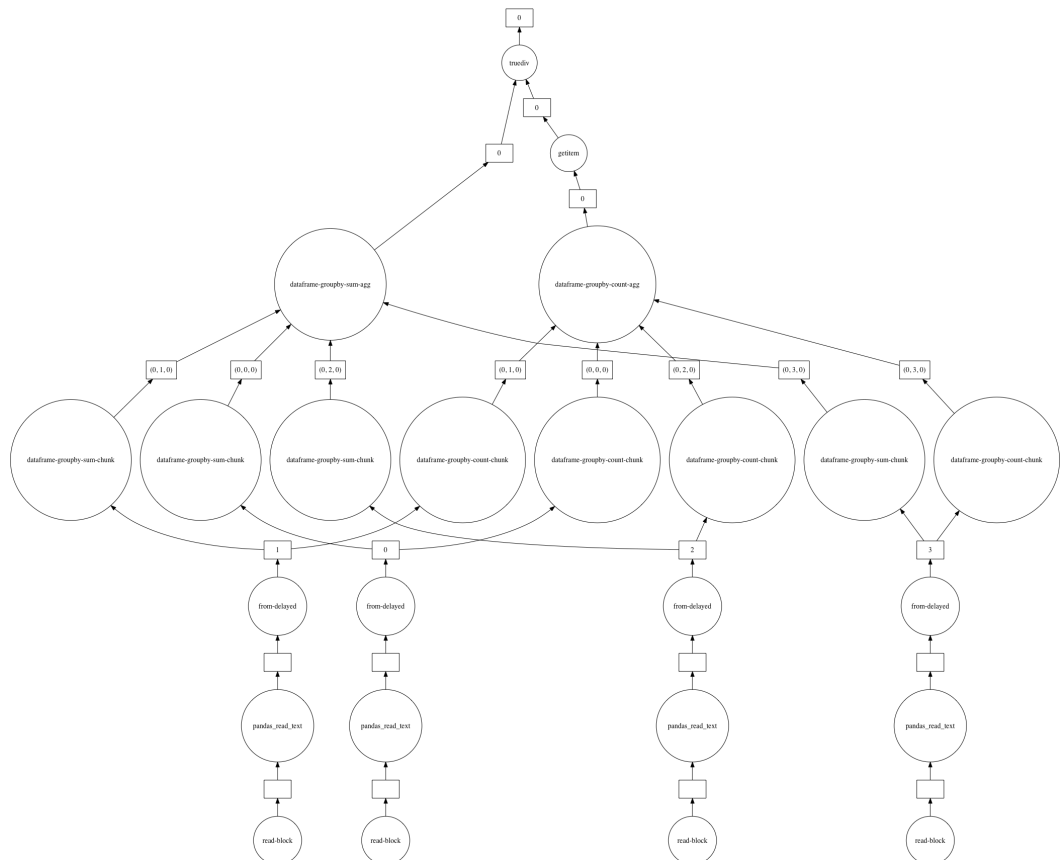
One of the main uses of dataframes is the production of statistics, in particular for specific sub-parts of the dataframe through the `groupby()` function. These operations are supported by Dask as well:

```
In [14]: births_grouped = births_da.groupby('state')
```

```
In [16]: births_group_mean = births_grouped.mean()
```

```
In [17]: births_group_mean.visualize()
```

Out[17]:



```
In [18]: births_group_mean.compute()
```

Out[18]:

	Unnamed: 0	year	month	day	births
state					
AK	186490.363674	1978.499726	6.523542	15.731727	25.374350
AL	186357.881565	1978.492615	6.522566	15.743846	165.039934
AR	186453.473461	1978.497674	6.522845	15.739808	93.600547
AZ	186436.387141	1978.496854	6.521067	15.739672	129.580575
CA	186215.389352	1978.484778	6.522048	15.770375	1067.956997
CO	186518.550034	1978.501027	6.522930	15.729637	129.552088
CT	186394.528583	1978.494530	6.519967	15.743572	112.737691
DC	186399.759130	1978.494597	6.521953	15.741622	58.195732
DE	186472.827699	1978.498426	6.522650	15.733817	25.825236
FL	186477.669358	1978.498563	6.523471	15.733817	359.148898
GA	186349.298605	1978.491523	6.524200	15.747881	251.521739
HI	186501.362852	1978.499726	6.523542	15.731727	47.265125
IA	186505.551328	1978.500000	6.522310	15.731454	119.108815
ID	186521.236929	1978.500684	6.523542	15.731727	45.567068
IL	186317.537326	1978.489882	6.520919	15.748018	488.039923
IN	186447.118999	1978.496786	6.520996	15.741485	233.006976
KS	186398.155382	1978.494050	6.521680	15.741759	98.729449
KY	186367.933133	1978.492274	6.522768	15.745932	153.559415
LA	186465.618279	1978.497469	6.522917	15.737994	207.178137
MA	186341.089542	1978.490636	6.523718	15.750239	216.457553
MD	186426.612228	1978.495281	6.522637	15.741622	150.657366
ME	186436.477904	1978.495827	6.521959	15.737584	43.806950
MI	186289.479448	1978.488051	6.519050	15.765806	387.427694
MN	186346.094053	1978.491046	6.519481	15.749829	173.539029
MO	186444.592831	1978.496101	6.521959	15.737447	210.423998
MS	186422.434610	1978.494802	6.522572	15.740082	121.646101
MT	186539.550034	1978.501027	6.522930	15.729637	34.754825
NC	186514.812509	1978.499658	6.522650	15.733543	240.947448
ND	186541.550034	1978.501027	6.522930	15.729637	32.675702
NE	186542.550034	1978.501027	6.522930	15.729637	69.408077
NH	186468.643678	1978.497126	6.521346	15.735495	36.259442
NJ	186334.745117	1978.489551	6.524519	15.762737	271.122797
NM	186481.319102	1978.497537	6.523262	15.735632	65.893404
NV	186502.294375	1978.498700	6.522239	15.733543	33.046394
NY	186081.586118	1978.475931	6.522296	15.786854	702.754534
OH	186360.525618	1978.491051	6.520700	15.757754	455.844514
OK	186524.860526	1978.499726	6.522584	15.731727	127.695182

```
In [19]: births_group_mean.births.nlargest(10).compute()
```

```
Out[19]: state
CA      1067.956997
TX      718.072715
NY      702.754534
IL      488.039923
OH      455.844514
PA      444.353615
MI      387.427694
FL      359.148898
NJ      271.122797
GA      251.521739
Name: births, dtype: float64
```

```
In [20]: births_da.state.unique().compute()
```

```
Out[20]: 0    AK
          1    AL
          2    AR
          3    AZ
          4    CA
          5    CO
          6    CT
          7    DC
          8    DE
          9    FL
         10    GA
         11    HI
         12    IA
         13    ID
         14    IL
         15    IN
         16    KS
         17    KY
         18    LA
         19    MA
         20    MD
         21    ME
         22    MI
         23    MN
         24    MO
         25    MS
         26    MT
         27    NC
         28    ND
         29    NE
         30    NH
         31    NJ
         32    NM
         33    NV
         34    NY
         35    OH
         36    OK
         37    OR
         38    PA
         39    RI
         40    SC
         41    SD
         42    TN
         43    TX
         44    UT
         45    VA
         46    VT
         47    WA
         48    WI
         49    WV
         50    WY
          Name: state, dtype: object
```


Larger files

The birth dataset is not very large and dask doesn't really help because it fits in RAM and the overhead of communication between processes is too important.

Let's look at a case where files are larger and/or our dataset is split between multiple files. This dataset is taken from [Zenodo](https://zenodo.org/record/834557#.Xj0fMxP0nOS) (<https://zenodo.org/record/834557#.Xj0fMxP0nOS>) and represents an analysis of all edits made to Wikipedia pages from its beginning to 2016.

Data are split among multiple zip files, each containing multiple "largish" (500Mb) CSV files. Let's look at one of them:

```
In [21]: filepath = '../Data/wikipedia/20161101-current_content-part1-12-1728.csv'
```

```
In [22]: wikipedia_changes = dd.read_csv(filepath)
```

```
In [23]: wikipedia_changes
```

Out[23]: **Dask DataFrame Structure:**

	page_id	last_rev_id	token_id	str	origin_rev_id	in	out
npartitions=9							
	int64	int64	int64	object	int64	object	object

...

Dask Name: from-delayed, 27 tasks

We see that here Dask decided by default to split the file into 9 partitions because of its size. Let's look at a few lines:

```
In [24]: wikipedia_changes.head()
```

Out[24]:

	page_id	last_rev_id	token_id	str	origin_rev_id	in	out
0	12	746687538	1623	see	233194	[391426, 988138, 6540619, 6551217, 12116305, 1...	[391368, 407005, 6539886, 6540818, 12116304, 1...
1	12	746687538	1624	also	233194	[391426, 988138, 6540619, 6551217, 12116305, 1...	[391368, 407005, 6539886, 6540818, 12116304, 1...
2	12	746687538	3519	.	178538	[391426, 18309960, 18310083, 47354530, 1328933...	[391381, 871060, 18310026, 18310134, 47417405,...
3	12	746687538	4507	=	320749	[83542729, 160471915]	[367665, 83543709]
4	12	746687538	4508	=	320749	[83542729, 160471915]	[367665, 83543709]

The `page_id` corresponds to a specific Wikipedia topic, the `str` represents a given word that has been added or modified. The `in` and `out` arrays represent a sequence of events (referenced by an ID) of adding and removal, i.e. the longer the list, the most often this word has been edited.

Word of caution: Dask imports each partition as a separate dataframe, meaning that if the index is a default numeric index, it restarts for each dataframe. In other words, when querying `index = 0`, we will here get 9 items:

```
In [25]: wikipedia_changes.loc[0].compute()
```

```
Out[25]:
```

	page_id	last_rev_id	token_id	str	origin_rev_id	in	out
0	12	746687538	1623	see	233194	[391426, 988138, 6540619, 6551217, 12116305, 1...	[391368, 407005, 6539886, 6540818, 12116304, 1...
0	593	744804419	36875	by	155262821	[164630979, 167839234, 183617334, 185043789, 1...	[164630961, 167839008, 183617090, 185043774, 1...
0	700	746750216	1260	check	619139	[61773188, 91845565]	[61773072, 91844748]
0	783	746647937	207927	[655587695	[707531216]	[707530825]
0	864	745162899	76425		262349476	[314394579, 347669693, 348610355, 350408772, 4...	[314394537, 347669682, 348610301, 350408703, 4...
0	991	744928000	3073	important	18972725	[77455083, 87982073, 156235181, 156235404, 163...	[77453607, 87981675, 156235168, 156235397, 163...
0	1175	746229520	33743]]	576608421	[654529644]	[654529496]
0	1347	746716698	26536	jpg	163084252	[294293698]	[294293671]
0	1537	747000036	174476	</	477994012	[489689538, 496207384, 511974564, 602763537, 6...	[489689469, 496207260, 497925062, 579792032, 6...

Hence there is no simple way to "get the first 10 elements of the dataframe". Instead, it's much simpler for example to ask "give me the first 10 elements of `page_id = 593`":

```
In [26]: first_words = wikipedia_changes[wikipedia_changes.page_id==593].loc[0:20].compute()
```

Let's see what strings we have here:

```
In [27]: ' '.join(list(first_words.str.values))
```

```
Out[27]: 'by [[ george ]] , are puppet - animated films which typically use a different version of a puppet for different'
```

Seems to be [this page \(https://www.google.com/search?q=animated+films+which+typically+use+a+different+version+of+a+puppet+for+different&oq=animated+films+which+typically+use+a+different+version+of+a+puppet+for+different&aqs=chrome..69l57.167j0j4&sourceid=chrome&ie=UTF-8\)](https://www.google.com/search?q=animated+films+which+typically+use+a+different+version+of+a+puppet+for+different&oq=animated+films+which+typically+use+a+different+version+of+a+puppet+for+different&aqs=chrome..69l57.167j0j4&sourceid=chrome&ie=UTF-8).

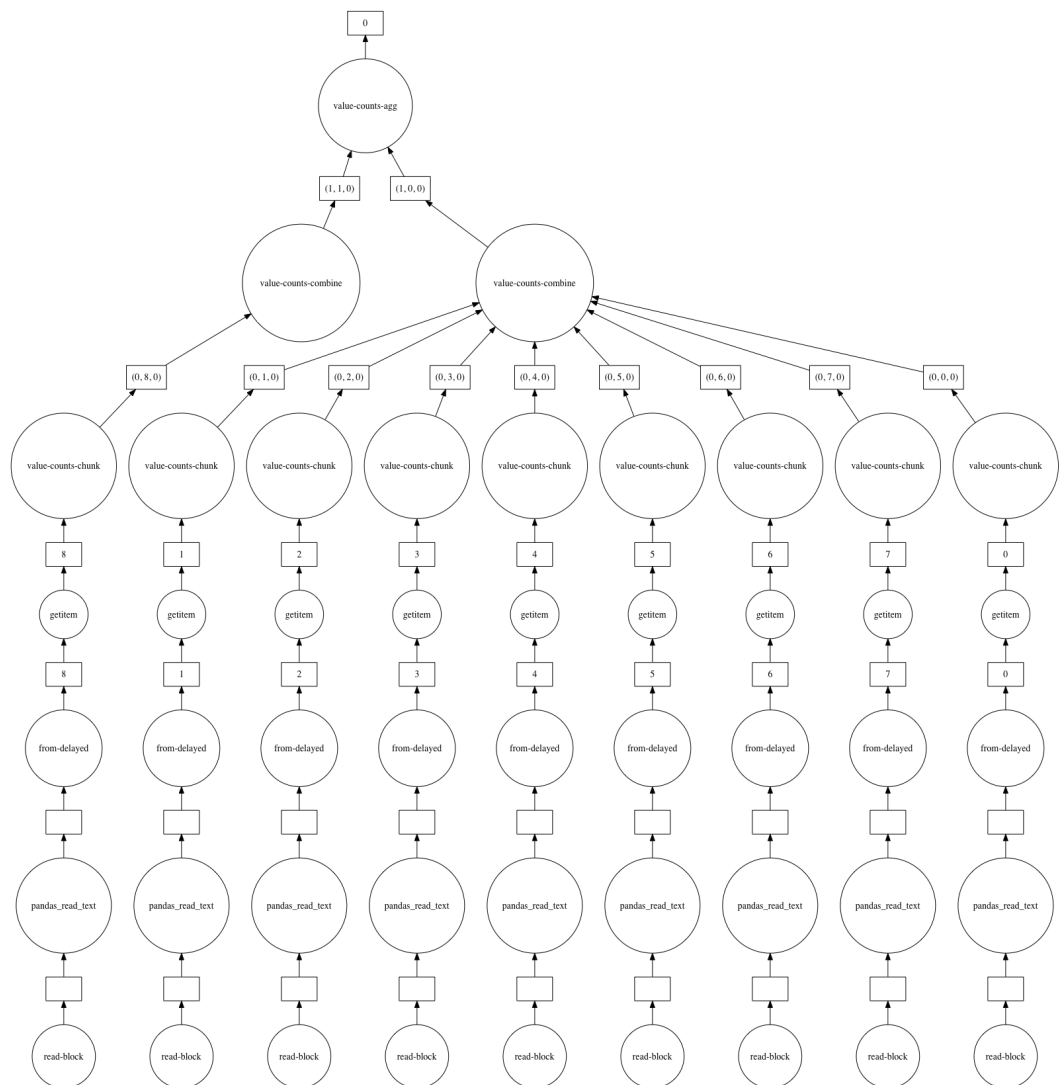
Compare Pandas and Dask

Let's see how Pandas and Dask compare on this single "largish" (500Mb) file. We can for example count occurrences of single words. We can use the same functions as in Pandas (`value_counts`) as Dask implements a very close API:

```
In [28]: count_str = wikipedia_changes.str.value_counts()
```

```
In [29]: count_str.visualize()
```

```
Out[29]:
```



```
In [30]: real_count = count_str.compute()
```

Let's look at the the few most used words or "tokens":

```
In [31]: real_count.head(n = 30)
```

```
Out[31]: |      256476
         =      229648
         .      210243
         ,      181501
         the    157474
         [[      141926
         ]]      141926
         /      106560
         of      105209
         -       90254
         and      74325
         in       59890
         >        57475
         ref      54172
         :        52983
         )        47957
         (        47930
         to       44798
         a        44271
         }}       41837
         {{       41745
         *        39386
         <        38841
         </       28086
         ;        27291
         &        25329
         is       21762
         as       18853
         for      17466
         title    16764
         Name: str, dtype: int64
```

Now we compare the performances of Pandas and Dask:

```
In [32]: %%time
wikipedia_changes = dd.read_csv(filepath)
count_str = wikipedia_changes.str.value_counts()
real_count = count_str.compute()

CPU times: user 106 ms, sys: 17.2 ms, total: 123 ms
Wall time: 4.57 s
```

```
In [33]: %%time
wiki_pd = pd.read_csv(filepath)
count_str = wiki_pd.str.value_counts()

CPU times: user 4.61 s, sys: 466 ms, total: 5.08 s
Wall time: 5.11 s
```

We see that Dask doesn't help much in this case.

Multiple large files

We only looked at a tiny part of the dataset. We will now look at much more of it even though still not at the complete one.

Dask offers the very useful feature of being able to open multiple files as one dask-dataframe by using the wild-card * or generating a file list. For example here, we have multiple CSV files in the folder and we can just say:

```
In [35]: wiki_large = dd.read_csv('../Data/wikipedia/2016*.csv')
```

We see many more partitions, meaning that dask indeed considered all files. If we wanted to import the files with pandas we would have more trouble:

```
In [36]: import glob
all_files = glob.glob('../Data/wikipedia/2016*.csv')
#wiki_large_pd = pd.concat((pd.read_csv(f) for f in all_files))
```

```
In [37]: all_files
```

```
Out[37]: ['../Data/wikipedia/20161101-current_content-part2-1729-3376.csv',
 '../Data/wikipedia/20161101-current_content-part3-3378-4631.csv',
 '../Data/wikipedia/20161101-current_content-part1-12-1728.csv',
 '../Data/wikipedia/20161101-current_content-part4-4633-5902.csv']
```

Let's time again the same tasks as before:

```
In [38]: %%time
wiki_large = dd.read_csv('../Data/wikipedia/2016*.csv')
count_str = wiki_large.str.value_counts()
real_count = count_str.compute()

CPU times: user 292 ms, sys: 48.5 ms, total: 340 ms
Wall time: 16.4 s
```

```
In [39]: %%time
all_files = glob.glob('../Data/wikipedia/2016*.csv')
wiki_large_pd = pd.concat([pd.read_csv(f) for f in all_files])
count_str = wiki_large_pd.str.value_counts()

CPU times: user 20.6 s, sys: 2.91 s, total: 23.5 s
Wall time: 24.1 s
```

Exercise

1. Create a dask-dataframe from the data in the ../Data/Chicago_taxi folder
2. Try to understand the file by looking at the columns
3. People have multiple ways of paying. Can you find out which category gives on average the largest tip (use groupby) ?

```
In [ ]: import dask.dataframe as dd
taxi = dd.read_csv('../Chicago_taxi/chicago.csv')
taxi
taxi = dd.read_csv('../Chicago_taxi/chicago.csv', dtype={'taxi_id': 'float64'})
# if not specified, get an error at mean().compute()
taxi_group = taxi.groupby('payment_type')
mean_val = taxi_group.mean().compute()
mean_val.tips
```

Some more features

We quickly summarize here some more features that might be of interest even for beginners.

```
In [2]: from dask.distributed import Client

client = Client("tcp://127.0.0.1:63517")
client
```

```
Out[2]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:63517	Workers: 4
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 4
	Memory: 17.18 GB

Calculating multiple outputs

Sometimes we need multiple outputs from a computation. However until now all we have seen are series of delayed computations and final `compute()` call. It is however possible to recover **multiple** intermediate results and to do that **without computational penalty***. Let's consider this example:

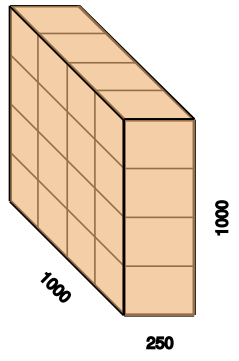
```
In [2]: import dask
import dask.array as da
```

```
In [3]: my_array = da.random.random((1000, 1000, 250))
```

```
In [4]: my_array
```

```
Out[4]:
```

	Array	Chunk
Bytes	2.00 GB	125.00 MB
Shape	(1000, 1000, 250)	(250, 250, 250)
Count	16 Tasks	16 Chunks
Type	float64	numpy.ndarray



We want to calculate the difference between max and min projections along the third axis. But we also want to check the maximum projection.

```
In [5]: maxproj = my_array.min(axis = 2)
meanproj = my_array.max(axis = 2)
difference = maxproj - meanproj
```

If we calculate things separately, the maximum projection is done twice:

Domain specific applications

In addition to the default features (array, dataframe, bag etc.) offered by Dask, there are additional domain-specific modules. We are looking at two of them here: machine learning and image processing.

dask_ml

Just like dask-array is a port of Numpy to Dask, dask_ml is a port from scikit-learn to Dask. Scikit-learn is currently probably the most popular machine-learning package in Python. Dask offers a subset of function available in scikit-learn using the same syntax. Let's see an example. The calculation is **only for the purpose of illustration** and is not realistic.

We look again at the taxi dataset:

```
In [1]: from dask.distributed import Client

client = Client("tcp://127.0.0.1:49550")
client
```

```
Out[1]:
```

Client	Cluster
Scheduler: tcp://127.0.0.1:49550	Workers: 4
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)	Cores: 4
	Memory: 17.18 GB

```
In [2]: import dask.dataframe as dd
```

```
In [24]: taxi = dd.read_csv('../Data/Chicago_taxi/chicago_taxi_trips_2016_01.csv',
dtype={'taxi_id': 'float64'})
```

```
In [25]: taxi
```

```
Out[25]: Dask DataFrame Structure:
```

	taxi_id	trip_start_timestamp	trip_end_timestamp	trip_seconds	trip_miles	picku
npartitions=3						
	float64	object	object	float64	float64	float6

Dask Name: from-delayed, 9 tasks

We are working here on a trivial question and checking the relation between the fare and the trip time (in seconds). We only keep those two variables:

```
In [26]: taxi = taxi[['trip_seconds', 'fare']]
```

```
In [27]: taxi = taxi.dropna()
```

Then we can easily split out dataset into train and test sets:

```
In [28]: train, test = taxi.random_split([0.8, 0.2])
```

We pick a linear regression model from dask_ml:

```
In [56]: from dask_ml.linear_model import LinearRegression
```

We create a linear fit object as we would do it in scikit-learn:

```
In [30]: linfit = LinearRegression(fit_intercept=False)
```

And we call the `fit()` method like for any other scikit-ml method:

```
In [57]: linfit_model = linfit.fit(train[['trip_seconds']].values, train[['fare']].values)
```

We can then predict values for our test set:

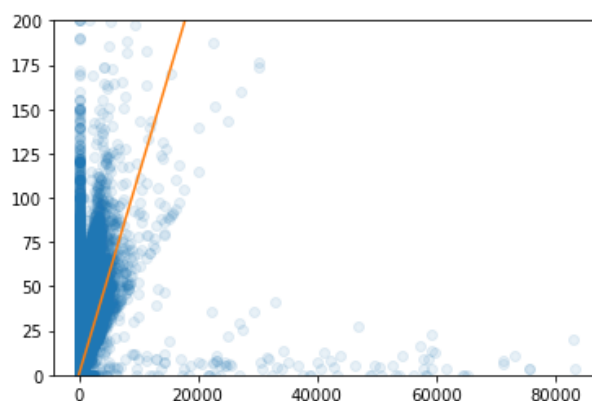
```
In [59]: pred = linfit_model.predict(test[['trip_seconds']].values)
```

And we see that the model takes in dask object and also generates a dask object. Hence we can do prediction for large datasets! Finally we can look at our fit:

```
In [61]: import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
plt.plot(test.trip_seconds.compute().values, test.fare.compute().values, 'o', alpha = 0.1)
plt.plot(np.arange(0, 20000), linfit_model.coef_[0]*np.arange(0, 20000))
ax.set_ylim([0, 200])
```

Out[61]: (0, 200)



dask-image

We have seen before that we could wrap an image importer and other image processing functions into `delay()` calls. However Dask offers a built-in set of functions to deal with images. We are going to illustrate this through an example.

We load a series of images:

```
In [39]: from dask_image import imread, ndfilters
import dask
```

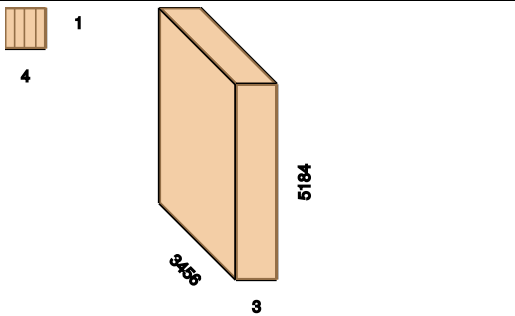
We have a series of images in a folder. We want to analyze all of them and create a large dask array:

```
In [57]: images = imread.imread('/Users/gwl8g940/OneDrive - Universitaet Bern/Cou
rses/DaskCourse/Butterflies/CAM01798*.JPG')
```

```
In [58]: images
```

Out[58]:

	Array	Chunk
Bytes	214.99 MB	53.75 MB
Shape	(4, 3456, 5184, 3)	(1, 3456, 5184, 3)
Count	12 Tasks	4 Chunks
Type	uint8	numpy.ndarray



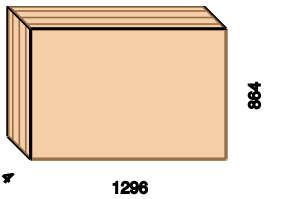
Then we only keep a single channel and downscale the image by slicing the array:

```
In [59]: im_downscale = images[:, ::4, ::4, 0]
```

```
In [61]: im_downscale
```

Out[61]:

	Array	Chunk
Bytes	4.48 MB	1.12 MB
Shape	(4, 864, 1296)	(1, 864, 1296)
Count	16 Tasks	4 Chunks
Type	uint8	numpy.ndarray



Then we filter each image using a gaussian filter implemented in dask-image:

```
In [62]: im_filtered = ndfilters.gaussian_filter(im_downscale, sigma=(0,2,2))
```

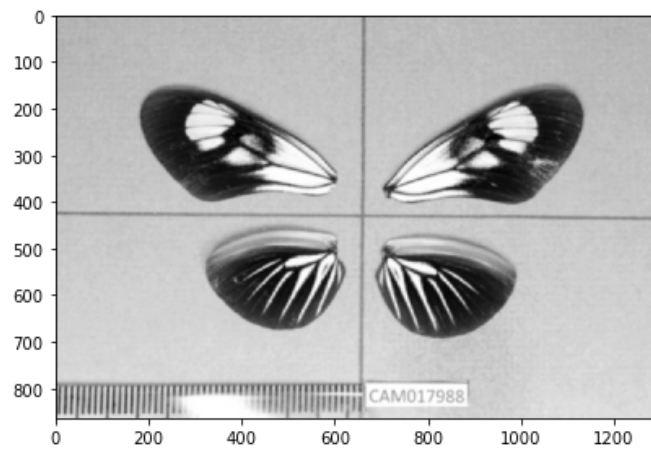
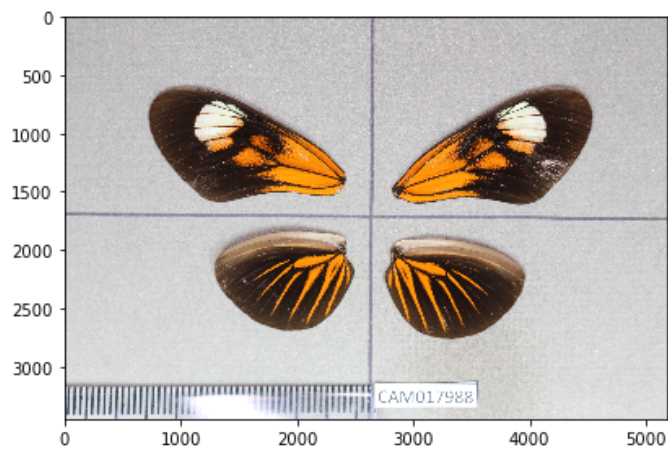
We recover both the original and filtered image for comparison. Note that this is not something that one would typically do as it loads all data into RAM:

```
In [63]: result = dask.compute(images, im_filtered)
```

```
In [64]: import matplotlib.pyplot as plt

fig, ax = plt.subplots(2,1, figsize=(10,10))
ax[0].imshow(result[0][0,:,:],cmap = 'gray')
ax[1].imshow(result[1][0,:,:],cmap = 'gray')
```

Out[64]: <matplotlib.image.AxesImage at 0x12e305390>



Running dask on a cluster

Dask greatly simplifies the work on a HPC cluster where different CPUs do not belong to the *same* machine like on a large station or a Google Cloud/AWS/Azure/SWITCHengine cloud computer.

In particular the dask-jobqueue module helps dealing with various queuing systems typically used on such systems. For example at Unibe, the Ubelix cluster uses the SLURM system. In normal usage, one has to write submission requests to execute jobs, make sure they properly exploit resources if meant to work in parallel etc. As we'll show here Dask massively simplifies the procedure.

```
In [1]: from dask_jobqueue import SLURMcluster
```

First we create a "cluster on the cluster" and use the SLURMcluster in this particular case. We can specify here all parameters that one can commonly specify on SLURM. Here we only say how many CPUs and how much RAM per CPU we need:

```
In [2]: cluster = SLURMcluster(
        cores=1,
        memory="5 GB"
    )
```

With this command, Dask has created (but not submitted) the request to slurm. We can use the `job_script()` method to see how that request looks. It's a standard SBATCH script:

```
In [ ]: print(cluster.job_script())
```

At the top we see specifications for the cluster (including e.g. our RAM request) and on the bottom we see the command executed on the cores so that we can use them with Dask. Note that **this is all done automatically for you**.

Then we proceed as usual and create a client that we connect to the cluster. Unfortunately, it's not yet possible to use the dask dashboard on the cluster.

```
In [4]: from dask.distributed import Client
        client = Client(cluster)
```

However we can adjust the size of our cluster which for the moment has 0 workers and thus 0 CPUs. Any time we scale up, new jobs are sent to the cluster. If we scale down, the jobs are stopped. If we monitor our resource usage on the cluster, we will see jobs appearing and disappearing.

```
In [16]: client.cluster
```

We can also use the simple `scale()` command:

```
In [14]: cluster.scale(jobs=10)
```

Finally we can do what we came to do: calculations !

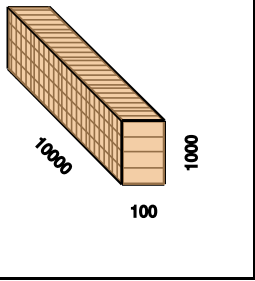
```
In [8]: import dask.array as da
```

```
In [9]: myarray = da.random.randint(0,100,(10000,1000,100))
```

```
In [10]: myarray
```

```
Out[10]:
```

	Array	Chunk
Bytes	8.00 GB	80.00 MB
Shape	(10000, 1000, 100)	(400, 250, 100)
Count	100 Tasks	100 Chunks
Type	int64	numpy.ndarray



```
In [13]: %%time
myarray.mean().compute()
```

CPU times: user 302 ms, sys: 32.4 ms, total: 335 ms
Wall time: 7.14 s

```
Out[13]: 49.498726549
```

```
In [15]: %%time
myarray.mean().compute()
```

CPU times: user 462 ms, sys: 52.9 ms, total: 515 ms
Wall time: 2.83 s

```
Out[15]: 49.498726549
```