Dask (numpy) arrays

As mentioned before, there are other solutions to perform perform parallel computing in Python. However Dask offers an crucial feature not present in other libraries: a built-in parallelized implemtation 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:

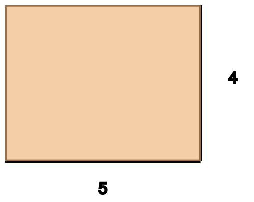
In [27]:
```
nprand
```

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 `delayed` 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.

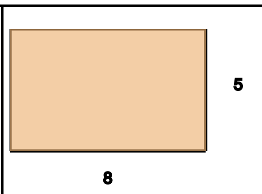
However one can also create arrays directly from a numpy array:

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:

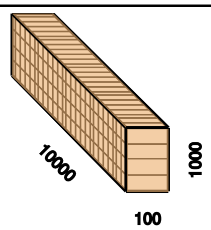
In [78]:
```
large_nparray = np.random.randint(0,100,(10000,1000,100))
```

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

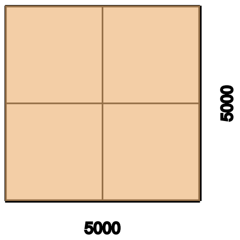
One of the main feature of numpy array is the possibility to slice and index them. Great news: dask arrays behave exactly in the same way for most "regular" cases (e.g. it doesn't implement slicing with multiple lists). Let's see how it works:

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



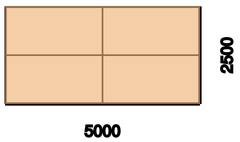
For example we can slice the array:

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



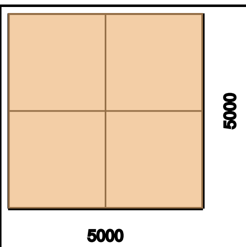
Or we can use logical indexing. First we create a logical array:

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!

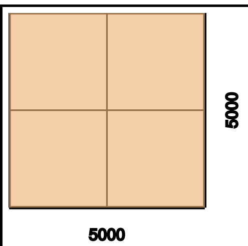
An extremely useful features of Dask is that whenever you are handling a dask-array you can apply most of the Numpy funtions to it and it remains a dask-array, **i.e. it gets integrated in the task graph**. For example:

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

In [46]:
```
cos_array
```

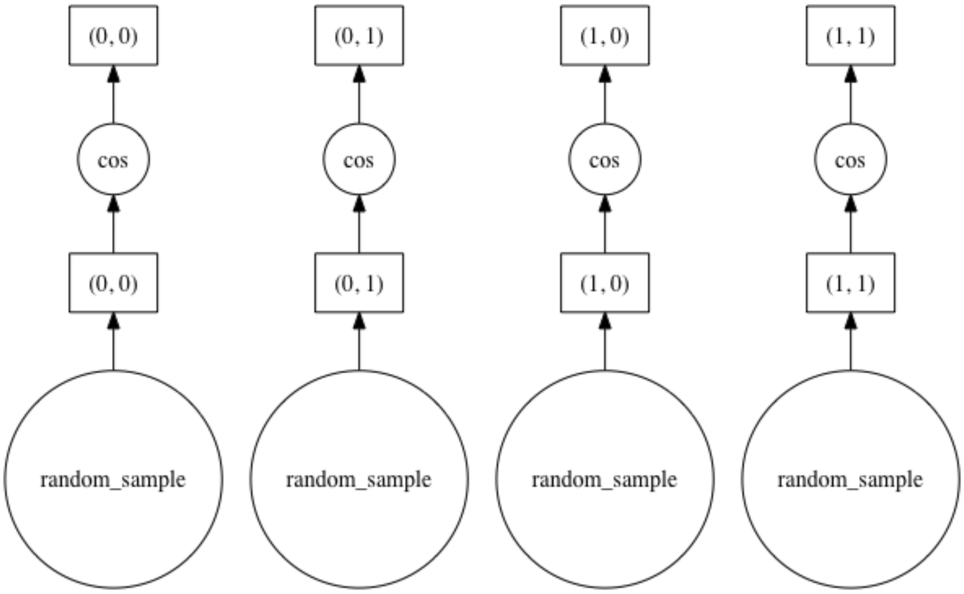
Out[46]:

	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]:



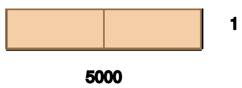
Dask also re-implements many numpy functions internally so that they are accessible as methods of the dask-arrays:

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

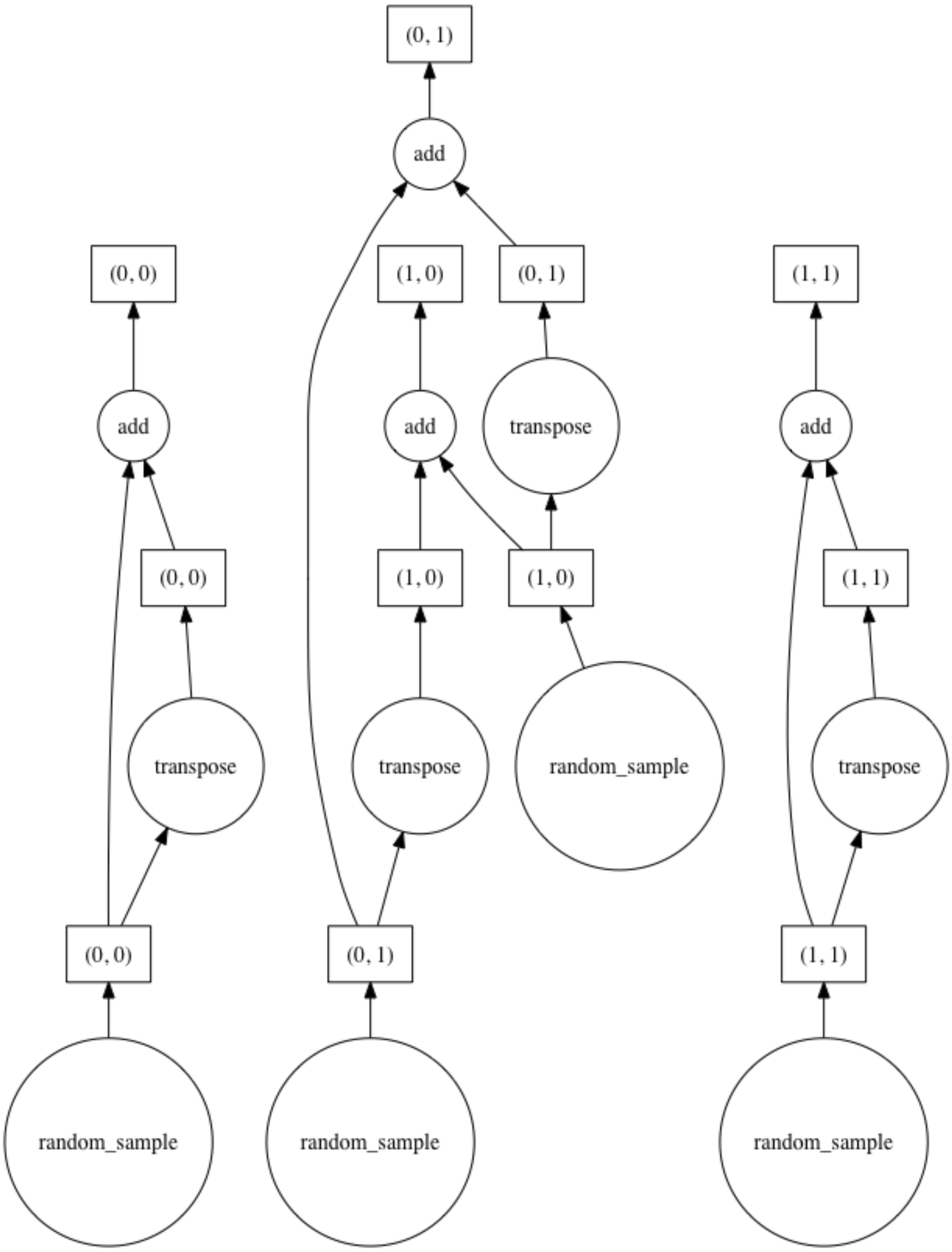


The great advantage of dask-arrays is that functions have been optimized in order to make the task-graph very efficient. For example this simple calculation produces already a quite complex task graph. If handling large "out-of-RAM" array with numpy, one would have to break up the large array and be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

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

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

Out[62]:

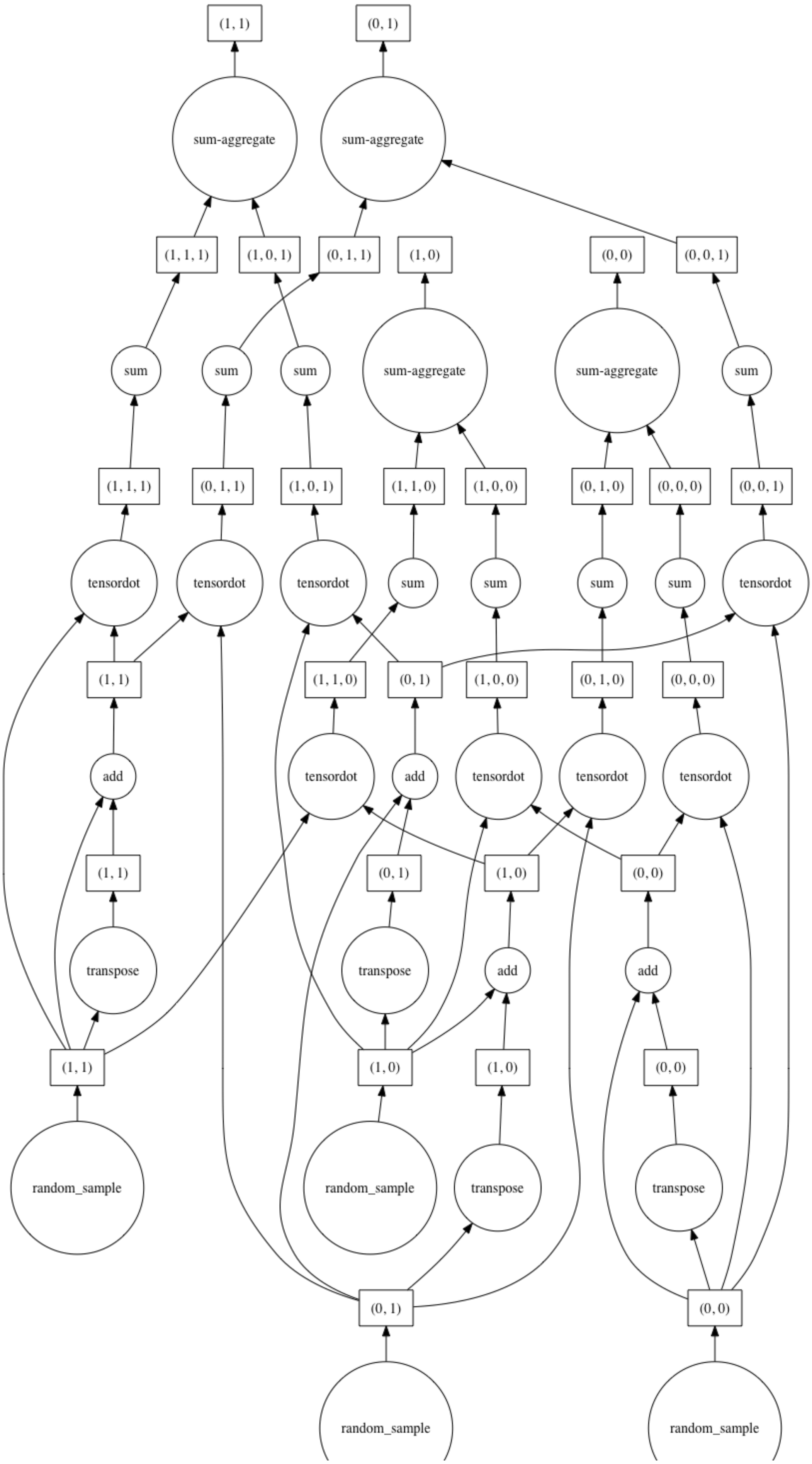


This is already quite complicated, but it can become much more complicated very quickly.

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

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

Out[64]:


(1, 1)
(0, 1)
sum-aggregate
sum-aggregate
(1, 1, 1)
(1, 0, 1)
(0, 1, 1)
(1, 0)
(0, 0)
(0, 0, 1)
sum
sum
sum
sum-aggregate
sum-aggregate
sum
(1, 1, 1)
(0, 1, 1)
(1, 0, 1)
(1, 1, 0)
(1, 0, 0)
(0, 1, 0)
(0, 0, 0)
(0, 0, 1)
tensordot
tensordot
tensordot
sum
sum
sum
sum
tensordot
(1, 1)
(1, 1, 0)
(0, 1)
(1, 0, 0)
(0, 1, 0)
(0, 0, 0)
add
tensordot
add
tensordot
tensordot
tensordot
(1, 1)
(0, 1)
(1, 0)
(0, 0)
transpose
transpose
add
add
(1, 1)
(1, 0)
(1, 0)
(0, 0)
random_sample
random_sample
transpose
transpose
(0, 1)
(0, 0)
random_sample
random_sample

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
```

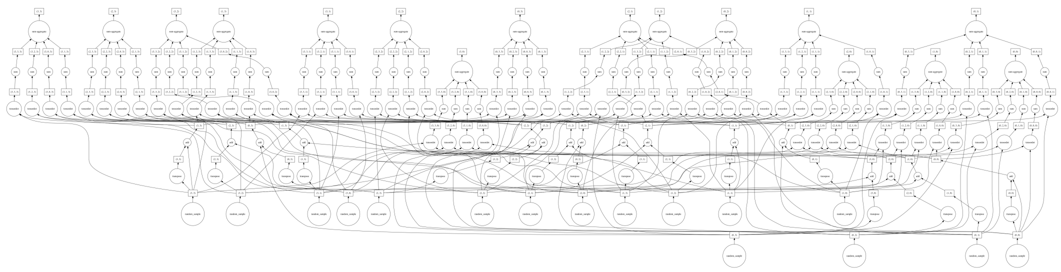
We see here that for a reasonably sized array, the overhead time needed to push data between processes makes Dask slower than basic Numpy, so be careful in what context you use Dask! But Dasks scales nicely:

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

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

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

Out[70]:



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        ])
```

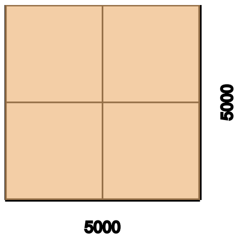
Also some operations such as those reshaping arrays may pose difficulties to Dasks as they require reshuffling array chunks. For example:

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



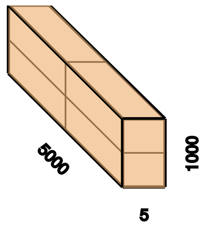
This works because it's easy to reshuffle some chunks:

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

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

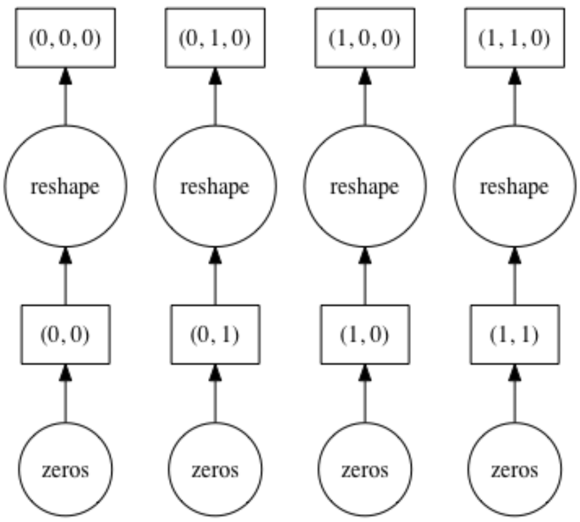
Out[21]:

	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()
```

Out[23]:



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.chunks)
    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 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 to 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()
```