Dask presentation

This notebook is a quick demonstration of the DASK library. DASK is designed for distributed computing in Python. The architecture of such a system possesses a Scheduler, several Workers and several clients (only one client is used here). The user who wants to run distributed computing connects its client to the scheduler, then the scheduler divides data into 'partitions' and assignes them to the different workers. In this notebook, Client() is configured in distributed mode but runs locally the scheduler and the workers by itself.

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

In [2]:
```
client = Client(n_workers=5,threads_per_worker=1)
client
```

Out[2]:

Client
Scheduler: tcp://127.0.0.1:37563
Dashboard: http://127.0.0.1:8787/status (http://127.0.0.1:8787/status)

Cluster
Workers: 5
Cores: 5
Memory: 8.22 GB

Dask provides differents user interfaces, which are:

- Bag: A dask.bag is an unordered collection allowing repeats.
- Array: A dask.array is an array of the same element.
- DataFrame: A dask.dataFrame is a data structure like a dictionary that contains heterogeneous data accessible by 'labels'.
- Delayed: A dask.delayed is a low-level way to define one or several tasks
- Futures: A dask.future is a low-level way to execute real-time computing

Dask.bag

Dask.bag allows operations usually used in functional programming like map, filter and fold. The computation is lazy until the method '.compute()' is called. Let's start with an example of list of string (here a string contains a number). The argument npartitions=5 is optionnal, it is set to 5 here for an in-depth understanding of the system behavior. Given the short lenght of the list, if this optional argument was not set explicitly, it would automaticaly be set to 1, hence without any parallelization. The method .from_sequence() loads data from the client to the scheduler so we usually set a small amount of data in the list (ex: filenames, references, indexes, etc...). The result returned is a dask.bag, an object on which it is possible to apply lazy computation.

In [3]:
```
rdd = db.from_sequence(['0','1','2','3','4','5','6','7','8',
                        '9','10','11','12'],npartitions=5)
```

The map() applies the function passed through the argument to each element. Here, our function convert each 'string' to 'int'. If we have had filenames, url, .. , we would have load/download the content the same way. The result would be:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

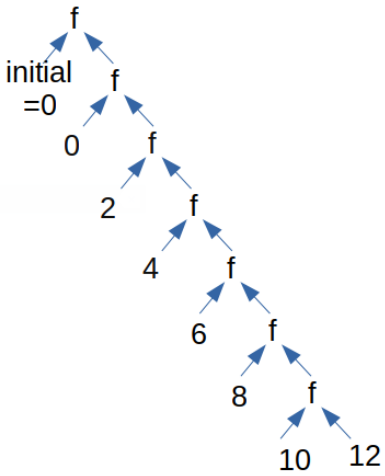
In [4]:
```
rdd = rdd.map(lambda e: int(e))
```

The filter() applies a boolean function passed through the argument to each element and allows to make a selection onto the elements we want to keep. Here the function returns True if the element 'e' is even and False else. The result would be: [0, 2, 4, 6, 8, 10, 12]

In [5]:
```
rdd = rdd.filter(lambda e: e%2==0)
```

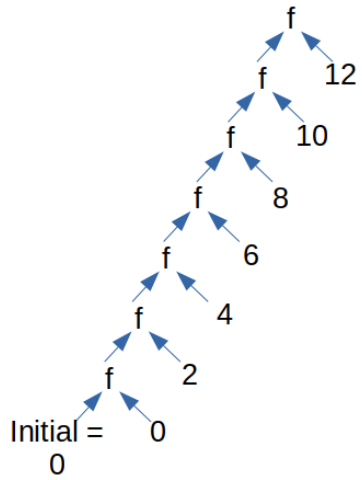
The fold() method is a 'Higher order function' that iterates binary operation over a list in a recursive way. The Dask's fold function parrallelizes automatically this operation. The restriction with dask.fold lies in that the binary function (in argument) must be associative in order to have a predictive result. Let's see why, in theory, two kinds of fold operation exist: fold left and fold right, one for each associative side. See the figure below.

FoldRight:



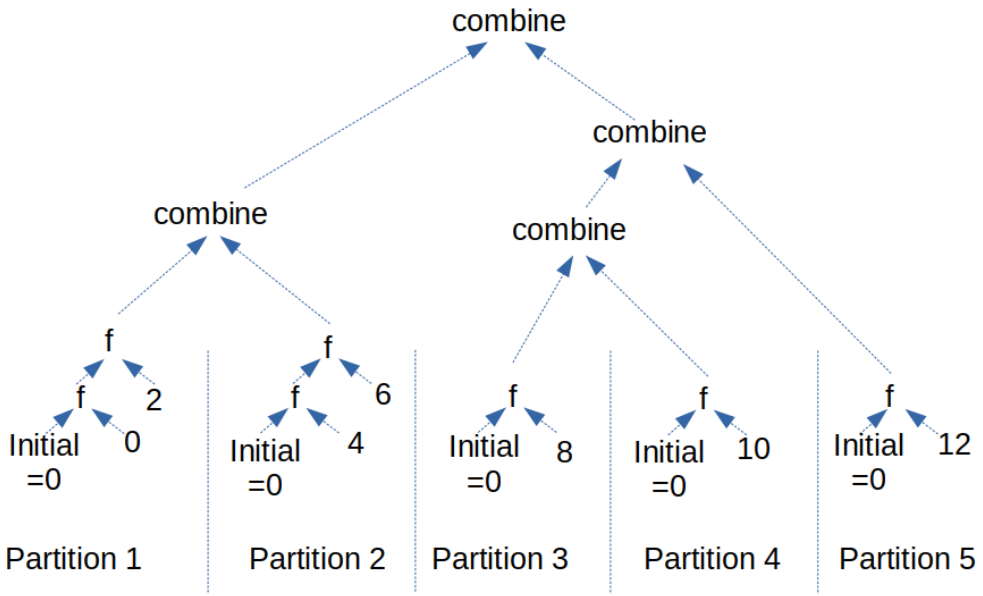
0+(0+(2+(4+(6+(8+(10+12))))))

FoldLeft:



(((((0 + 0)+ 2)+ 4)+ 6)+ 8)+ 10)+ 12

In distributed/parallelized way, the process is quite different; several parts are executed alongside each other in foldleft way and the result of each partitions is then aggregated to each other. Let's see below one example of the execution tree could be:



((0 + 0) + (0 + 4)) + (((0 + 8) + (0 + 10)) + (0 + 12))

Combine function is the function that combines the results from two partitions. This function is the same as 'binop'(=f in pictures) as long as no 'combine' function is not explicitly defined.

In [6]:
```
rdd = rdd.fold(binop=lambda acc, e: acc+e, initial=0)
```

All the functions described above do not compute. They build a graph of dependances of the different

steps of the calculation. The method '.compute()' is the function that triggers computation thanks to this graph. The result computed on workers is then brought back to client.

In [7]:
```
rdd.compute()
```

Out[7]:
```
42
```

Another interesting method provided by DASK is '.foldby()'. The foldby has the same ability of '.fold()' but it acts over different subgroups that we define by a criteria; the 'key'. Let's have an example below: We have a collection built from cities and their associated zip code. We want to group cities by state departement (The first two digits of the zip code identify the state departement).

In [8]:
```
rdd = db.from_sequence([
('Balma',31130),
('Blagnac',31700),
('Ramonville-Saint-Agne',31520),
('Vieille-Toulouse',31320),
('L\'Union',31240),
('Portet-sur-Garonne',31120),
('Tournefeuille',31170),
('Suresnes', 92150),
('Nanterre', 92000),
('La Garenne-Colombes', 92250),
('Bobigny', 93000),
('Montreuil', 93100)],npartitions=5)
```

The key function takes the first two digits of the zip code and the binop function takes the current city and add it to an accumulator 'acc[0]+b[0]'. The binop function needs to have the same data structure for each argument and for the return, hence the (acc[0]+b[0],).

In [9]:
```
rdd = rdd.foldby(key=lambda e:int(e[1]/1000),
                 binop=lambda acc,b:(acc[0]+[ b[0] ],), initial=([],),
                 combine=lambda a,b:(a[0]+b[0],))
rdd.compute()
```

Out[9]:
```
[(31,
  (['Balma',
    'Blagnac',
    'Ramonville-Saint-Agne',
    'Vieille-Toulouse',
    "L'Union",
    'Portet-sur-Garonne',
    'Tournefeuille'],)),
 (92, (['Suresnes', 'Nanterre', 'La Garenne-Colombes'],)),
 (93, (['Bobigny', 'Montreuil'],))]
```

Limitations

https://docs.dask.org/en/latest/bag.html#known-limitations (https://docs.dask.org/en/latest/bag.html#known-limitations)

Bags provide very general computation (any Python function). This generality comes at cost. Bags have the following known limitations:

1. By default, they rely on the multiprocessing scheduler, which has its own set of known limitations (see Shared Memory)
2. Bags are immutable and so you can not change individual elements
3. Bag operations tend to be slower than array/DataFrame computations in the same way that standard Python containers tend to be slower than NumPy arrays and Pandas DataFrames
4. Bag's groupby is slow. You should try to use Bag's foldby if possible. Using foldby requires more thought though

Dask.array

A dask array is an array of Numpy array. The dask.array API is very, very similar to the NumPy.ndarray API, but all the methods from NumPy are not implemented in DASK.

In [10]:
```
import dask.array as da
a = da.random.random(10000, chunks=100)
```

Here, for example, we compute the distance between two points a and b along the 10000 dimensions:

In [11]:
```
dist = da.linalg.norm(a-b)
```

Out[11]:
```
41.051551430721
```

Dask.dataframe

As Dask.array copies NumPy.ndarray, Dask.dataframe copies dataframe and dataSerie from Pandas. A dataFrame is a list of dataseries, a structure like a dictionary that contains heterogeneous data reachable by labels. In the following example, we load csv that contains information about car sales. What we decide to do is to extract a list of the 10 top car price lower than €40000. The main interest of dataseries is that they are indexed and they can be sorted. Such a process is not possible with the dask.bag and the dask.array (by default).

In [12]:
```
import dask.dataframe as dd
```

In [13]:
```
df = df[df.price <= 40000]
df = df.nlargest(10, 'price')
```

Out[13]:

	brand	model	year	km	power	price
LEXUS	NX	NX 300h 4WD	2015	39 816	Essence / Courant Électrique	31980
RENAULT	TALISMAN	Talisman dCi 130 Energy	2015	24416	Diesel	19490
RENAULT	ESPACE	Espace dCi 160 Energy Twin Turbo	2015	51800	Diesel	18990
BMW	SERIE 2	Active Tourer 218d 150 ch	2015	49819	Diesel	18900
RENAULT	KOLEOS	Koleos 2.0 dCi 150	2015	62731	Diesel	15990
RENAULT	KOLEOS	Koleos 2.0 dCi 175	2015	67263	Diesel	15990
NISSAN	QASHQAI	Qashqai 1.2 DIG-T 115	2015	58998	Essence Sans Plomb	15989
RENAULT	KADJAR	Kadjar dCi 110 Energy eco²	2015	54525	Diesel	15979
NISSAN	QASHQAI	Qashqai 1.6 dCi 130 Stop/Start	2015	75120	Diesel	15760
RENAULT	TALISMAN	Talisman dCi 130 Energy	2015	65724	Diesel	14990

Dask.futures

Dask.delayed

The lazy computation is possible thanks to the computation graph, it is a DAG (a Directed Acyclic Graph) which allows dask to know the different dependencies of the step computing, and, hence the parallelization. Every lazy dask function add one or more steps to this graph automaticaly. However, it is possible to build this graph by hand with the dask.delayed() function. This one is very usefull to parallelize a loop in the code. Just bellow we have two examples with a 'for' loop.

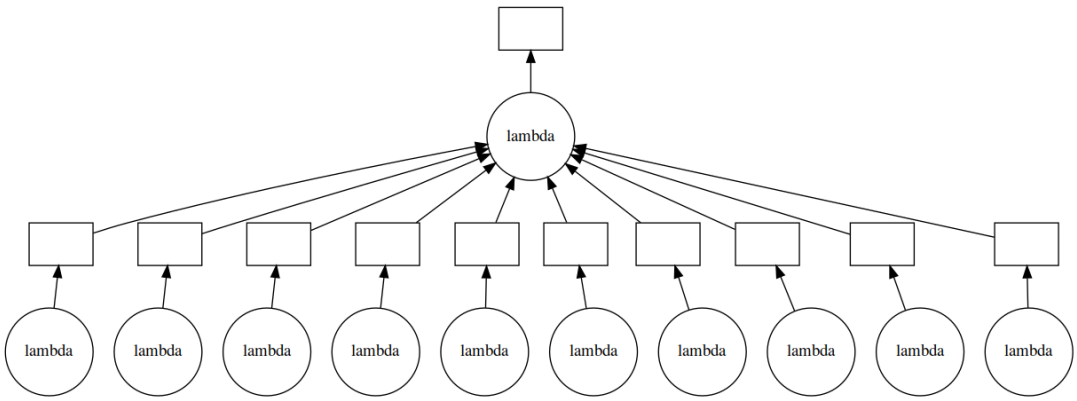
In [14]:
```
from dask import delayed
```

The first one is a loop in which each iteration is independent of the previous one, this 'for' loop acts like a map. Into this last loop, we are multiplying each element by 10.

In [15]:
```
collection = [0,1,2,3,4,5,6,7,8,9]
result = []
for e in collection:
    r = delayed(lambda e:e*10)(e)
    result.append(r)
result = delayed(lambda x:x)(result)
result.visualize('graph1.png')
```

Out[15]: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

Thanks to the '.visualize()' method wich allows us to display the graph built behind. We easily see that each iteration can be computed separately from each other

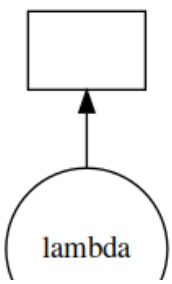


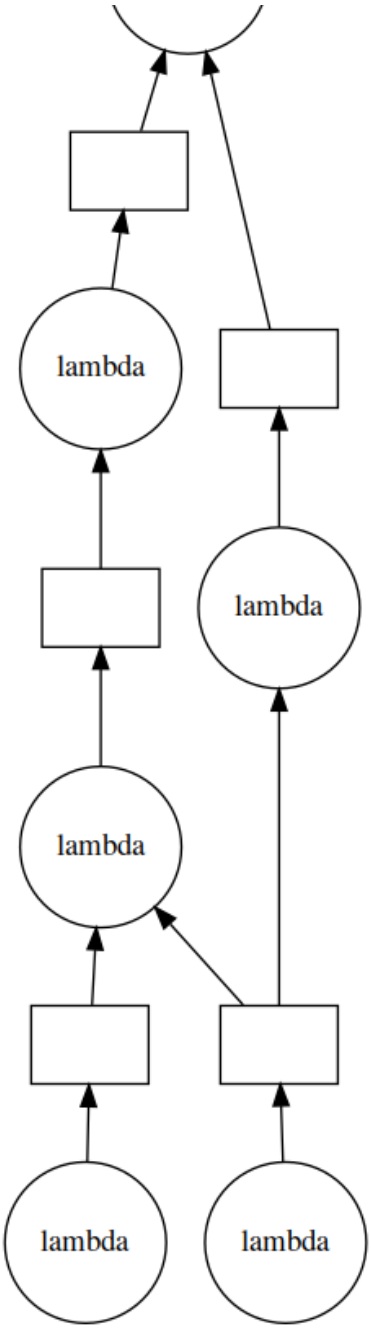
The second example, bellow, also shows a 'for' loop but with a strong dependancy between iterations. Most of the calculations need the result of the previous operation. Let's see the graph below and the code that shows the steps.

In [16]:
```
a = 1
b = 1
add = lambda a,b:a+b
for _ in range(2):
    a = delayed(add)(1,a)
    b = delayed(add)(1,b)
    a = delayed(add)(a,b)

result = a
result.visualize('graph2.png')
```

Out[16]: 8





Another advantage of dask.delayed is the use of the decorator '@delayed'. Put that decorator over a function is a equivalent to the '.delayed(function)(args)'. It's a easy way to parallelize a group of function and it's possible to use it over a recursive function. Dask provides the example below wich computes the Fibonacci number. This function call itself twice, first for 'a = fib(n - 1)' and the second for 'b = fib(n - 2)', those last two calls are computed in parallel.

In [17]:
```
from dask import delayed, compute
# https://distributed.dask.org/en/latest/task-launch.html

@delayed
def fib(n):
    if n < 2:
        return n
    # We can use dask.delayed and dask.compute to launch
    # computation from within tasks
    a = fib(n - 1)  # these calls are delayed
    b = fib(n - 2)
```

```
    a, b = compute(a, b)  # execute both in parallel
    return a + b

f = fib(8)
```

Out[17]: 21

It's also possible to generate the graph from a Python dictionary, where the key is the variable and the value is the computation to carry out. The method '.get()' triggers the computation.

In [18]:
```
from dask.threaded import get
from operator import add

graph = {'x': 1,
 'y': 2,
 'z': (add, 'x', 'y'),
 'w': (sum, ['x', 'y', 'z']),
 'v': [(sum, ['w', 'z']), 2]}
```

Out[18]: 3

Dask.futures

Dask.future is used to compute in real-time, there is no more lazy computation in this part. dask.future extends the native Python's concurrent.futures. Let's see an example of how it works:

In [19]:
```
```

For this example, we are using the map operation where the function executed in argument is defined just below. We are simulating a complex time-undefined function with the primitive 'time.sleep()' which waits for the process for a while (it waits for the time specified in parameter).

In [20]:
```
def complex_algorithm(e):
    import time
    time.sleep(e)
```

The next piece of code shows how to call the function complex_algorithm(5). The method '.submit()' returns a future object that follows the status of the function complex_algorithm(5) (either 'pending' or 'finish' when the algorithm has returned. The 'wait' is a blocking function that waits for the complex_algorithm to finish.

In [21]:
```
import time
future = client.submit(complex_algorithm,5)
print(future) # <Future: pending, ... >
wait(future, timeout=12)
```

```
<Future: pending, key: complex_algorithm-bdb7c61f884b46bb3acb38dfda348502>
<Future: finished, type: builtins.int, key: complex_algorithm-bdb7c61f884b46bb
3acb38dfda348502>
```

The function as_completed(futures) provides a Python generator that allows to iterate over a collection of 'future'. The as_completed(futures) is a blocking function that awaits a new element to return. We can watch this behavior with the following example:

In [22]:
```
import random
collection = [random.randint(5,30) for _ in range(0,200)]
```

```
[6, 7, 22, 10, 28, 11, 28, 15, 28, 26, 30, 10, 28, 29, 18, 13, 5, 18, 22, 19,
21, 19, 25, 26, 16, 14, 20, 6, 12, 14, 21, 19, 5, 21, 25, 14, 6, 7, 24, 22, 2
6, 27, 22, 8, 17, 22, 14, 10, 30, 23, 13, 30, 16, 26, 27, 7, 9, 7, 8, 23, 10,
29, 6, 18, 30, 12, 9, 20, 7, 14, 26, 6, 30, 24, 18, 17, 8, 11, 13, 26, 6, 6, 1
0, 28, 27, 18, 29, 19, 29, 7, 29, 23, 8, 23, 10, 20, 30, 10, 14, 25, 22, 7, 2
0, 23, 28, 12, 12, 27, 22, 24, 14, 6, 7, 5, 27, 6, 6, 30, 19, 25, 5, 13, 5, 1
1, 10, 6, 20, 9, 10, 27, 24, 18, 8, 24, 8, 11, 16, 13, 10, 13, 8, 18, 12, 16,
6, 16, 5, 12, 19, 5, 23, 10, 15, 15, 16, 14, 15, 20, 14, 24, 13, 21, 20, 25, 1
```

For each integer in the 'collection' the function 'complex_function()' will wait for the time specify by this integer. Then, we trigger all the instance of 'complex_function()' at the same time, so the shortest time it waits for, the fastest it returns (until there are enough partitions, workers and/or threads to observe that phenomenon). An example of output is :
5,5,5,5,5,5,5,5,7,7,7,7,7,7,8,8,8,8,8,8,8,9,9,9,9,9,10,10,10,10,10,10,10,10,10,10,10,10,10,10,12,12,12,12,12

This example was run with (Client(n_workers=5,threads_per_worker=3)), and we can see the (pseudo)order of the result that shows that the fastest algorithms return first (the results at begining of the list).

In [23]:
```
futures = client.map(complex_algorithm,collection)
import sys

for future in as_completed(futures):
    r = future.result()
    sys.stdout.write(str(r)+',')
    sys.stdout.flush()
```

```
5,5,5,5,5,5,5,5,5,5,5,13,13,13,13,13,13,13,13,16,16,16,16,16,16,16,18,18,18,1
8,18,18,18,6,6,6,6,6,6,6,6,6,6,6,6,6,25,25,25,25,25,11,11,11,11,11,30,30,30,3
0,30,30,30,8,8,8,8,8,8,8,8,8,9,9,9,9,21,21,21,21,21,10,10,10,10,10,10,10,10,1
0,10,10,10,14,14,14,14,14,14,14,14,14,14,15,15,15,15,15,23,23,23,23,23,23,23,1
9,19,19,19,19,19,19,17,17,17,17,29,29,29,29,29,29,26,26,26,26,26,26,26,26,7,7,
7,7,7,7,7,7,7,7,12,12,12,12,12,12,12,24,24,24,24,24,24,24,27,27,27,27,27,27,2
7,27,27,27,20,20,20,20,20,20,20,20,20,22,22,22,22,22,22,22,22,22,28,28,28,28,2
8,28,28,28,
```

Diagnostics (distributed mode)

At the start, the client provides a dashboard, it is a Gui web interface where all the metrics are displayed in real time. It is also possible to record all that information by the function 'client.profile(filename="dask-profile.html")' that collects statistical profiling information about recent work. This function produces a web html file in the current folder.

In [24]:
```
client.profile(filename="dask-profile.html")
```

Out[24]:
```
({'description': {'filename': '', 'name': '', 'line_number': 0, 'line': ''},
  'children': {'apply;/home/sylvain/.local/lib/python3.7/site-packages/dask/ut
ils.py;28': {'description': {'filename': '/home/sylvain/.local/lib/python3.7/s
ite-packages/dask/utils.py',
     'name': 'apply',
     'line_number': 30,
     'line': 'return func(*args, **kwargs)\n'},
    'children': {'__call__;/home/sylvain/.local/lib/python3.7/site-packages/da
sk/utils.py;879': {'description': {'filename': '/home/sylvain/.local/lib/pytho
n3.7/site-packages/dask/utils.py',
       'name': '__call__',
       'line_number': 880,
       'line': 'return getattr(obj, self.method)(*args, **kwargs)\n'},
      'children': {'nlargest;/home/sylvain/.local/lib/python3.7/site-packages/
pandas/core/frame.py;5018': {'description': {'filename': '/home/sylvain/.local
/lib/python3.7/site-packages/pandas/core/frame.py',
         'name': 'nlargest',
         'line_number': 5125,
         'line': 'return algorithms.SelectNFrame(self, n=n, keep=keep, columns
```

To record a precise scope, we should use the function performance_report that way:

In [26]:
```
from dask.distributed import performance_report

a = da.random.normal(size=(1000, 1000), chunks=(100, 100))
res = a.dot(a.T).mean(axis=0)

with performance_report(filename="dask-report.html"):
    res.compute()
```

```
/home/sylvain/.local/lib/python3.7/site-packages/dask/array/routines.py:272: P
erformanceWarning: Increasing number of chunks by factor of 10
  axes=(left_axes, right_axes),
```

Dask provides a local mode (=without any 'client') with which it is possible profile the code in-depth with:

- 'Profiler': A class used to profile Dask's execution at the task level.
- 'ResourceProfiler': A class used to profile Dask's execution at the resource level.
- 'CacheProfiler': A class used to profile Dask's execution at the scheduler cache level.
- For more information : https://docs.dask.org/en/latest/diagnostics-local.html (https://docs.dask.org/en/latest/diagnostics-local.html)

To make it clean, at the end, we just have to close the connection with the scheduler:

In [27]:
```
client.shutdown()
```

```
distributed.nanny - WARNING - Worker process still alive after 3 seconds, kill
ing
distributed.nanny - WARNING - Worker process still alive after 3 seconds, kill
ing
distributed.nanny - WARNING - Worker process still alive after 3 seconds, kill
ing
distributed.nanny - WARNING - Worker process still alive after 3 seconds, kill
ing
distributed.nanny - WARNING - Worker process still alive after 3 seconds, kill
ing
distributed.client - ERROR - Failed to reconnect to scheduler after 10.00 seco
nds, closing client
_GatheringFuture exception was never retrieved
future: <_GatheringFuture finished exception=CancelledError()>
concurrent.futures._base.CancelledError
```

In [ ]:

In [ ]: