Dask

Dask est une bibliothèque de calcul parallèle flexible pour le calcul analytique. Dask fournit un ordonnancement dynamique des tâches parallèles et des collections de données volumineuses de haut niveau comme `dask.array` et `dask.dataframe` . Pour en savoir plus sur Dask, voir : https://docs.dask.org/en/latest/ (https://docs.dask.org/en/latest/)

Note: Des parties de ce notebook proviennent des sources suivantes:

- https://github.com/rabernat/research_computing (https://github.com/rabernat/research_computing)
- https://github.com/dask/dask-examples (https://github.com/dask/dask-examples)

Démarrer un cluster distribué Dask et un client pour le tableau de bord

Le démarrage du Cluster/Client Dask est généralement facultatif.
Il fournit un tableau de bord qui est utile pour avoir un aperçu du calcul.
L'utilisation de `dask_jobqueue` permet également d'obtenir une plus grande puissance de calcul en mettant à l'échelle Dask sur plusieurs noeuds du HPC.

Le lien vers le tableau de bord sera visible lorsque vous créerez le cluster ou le client ci-dessous. Comme dask-labextension (https://github.com/dask/dask-labextension) est intégré dans l'environnement actuel, il peut être suffisant pour surveiller les tâches Dask (voir les fenêtres Task Stream et Progress sur la droite). Sinon, nous vous recommandons d'avoir le tableau de bord ouvert d'un côté de votre écran tout en utilisant votre ordinateur portable de l'autre côté. Cela peut demander un certain effort pour arranger vos fenêtres, mais les voir toutes les deux en même temps est très utile pour apprendre.

Création du cluster local

Entrée [7]:

```
import dask.distributed
cluster = dask.distributed.LocalCluster()
```

```
/softs/rh7/conda-envs/pangeo_202012/lib/python3.8/site-packages/distributed/
node.py:151: UserWarning: Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the HTTP server on port 37665 instead
  warnings.warn(
```

Connexion d'un client au cluster:

Entrée [8]:

```
import dask.distributed
client = dask.distributed.Client(cluster)
client
```

Out[8]:

Client
Scheduler: tcp://127.0.0.1:37829
Dashboard: http://127.0.0.1:37665/status (http://127.0.0.1:37665/status)

Cluster
Workers: 1
Cores: 1
Memory: 4.29 GB

Dask Arrays

Un tableau dask ressemble beaucoup à un tableau numpy. Cependant, un tableau dask ne contient pas directement de données. Il représente plutôt symboliquement les calculs nécessaires pour générer les données. Rien n'est réellement calculé tant que les valeurs numériques réelles ne sont pas nécessaires. Ce mode de fonctionnement est appelé "paresseux" ; il permet d'effectuer des calculs complexes et volumineux de manière symbolique avant de les transférer au programmateur pour exécution.

Si nous voulons créer un tableau numérique de tous les calculs, nous le faisons comme ceci :

Entrée [9]:

```
import numpy as np
shape = (1000, 4000)
ones_np = np.ones(shape)
ones_np
```

Out[9]:

```
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

Ce tableau contient exactement 32 Mo de données :

Entrée [10]:

```
print('%.1f Mo' % (ones_np.nbytes / 1e6))
```

```
32.0 Mo
```

Maintenant, créons le même tableau en utilisant l'interface de tableau de dask.

Entrée [11]:

```
import dask.array as da
ones = da.ones(shape)
ones
```

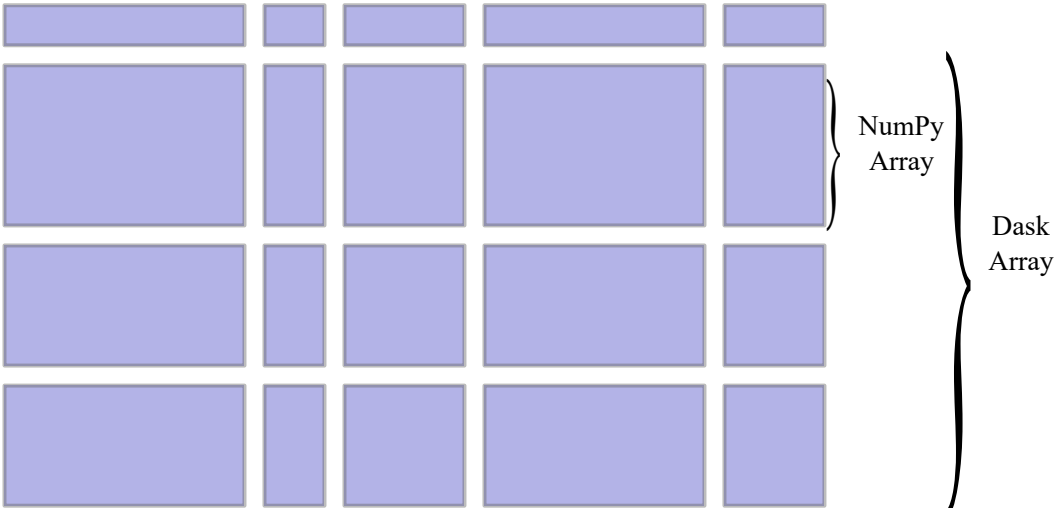
Out[11]:

	Array	Chunk
Bytes	32.00 MB	32.00 MB
Shape	(1000, 4000)	(1000, 4000)
Count	1 Tasks	1 Chunks
Type	float64	numpy.ndarray

4000 1000

Cela fonctionne, mais nous n'avons pas dit à Dask comment diviser le tableau, il n'est donc pas optimisé pour le calcul distribué.

Une différence essentielle avec dask est que nous devons spécifier l'argument `chunks`. `chunks` décrit comment le tableau est réparti sur plusieurs sous-tableaux.



Il y a plusieurs manières de spécifier le morcèlement des tableaux (https://docs.dask.org/en/latest/array-creation.html#chunks). Dans cette présentation, nous utiliserons une forme de bloc.

Entrée [12]:

```
chunk_shape = (1000, 1000)
ones = da.ones(shape, chunks=chunk_shape)
ones
```

Out[12]:

	Array	Chunk
Bytes	32.00 MB	8.00 MB
Shape	(1000, 4000)	(1000, 1000)
Count	4 Tasks	4 Chunks
Type	float64	numpy.ndarray

4000 1000

Remarquez que nous ne voyons qu'une représentation symbolique de l'ensemble, y compris sa forme, son type et sa taille. Aucune donnée n'a encore été générée. Lorsque nous appelons `.compute()` sur un tableau Dask, le calcul est déclenché et le tableau Dask devient un tableau numpy.

Entrée [13]:

```
ones.compute()
```

Out[13]:

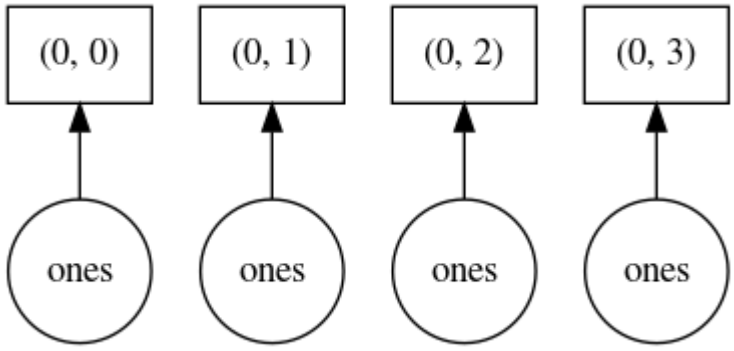
```
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

Afin de comprendre ce qui s'est passé lorsque nous avons appelé `.compute()`, nous pouvons visualiser le *graphe* Dask, c'est-à-dire les opérations symboliques qui composent le tableau.

Entrée [14]:

```
ones.visualize()
```

Out[14]:



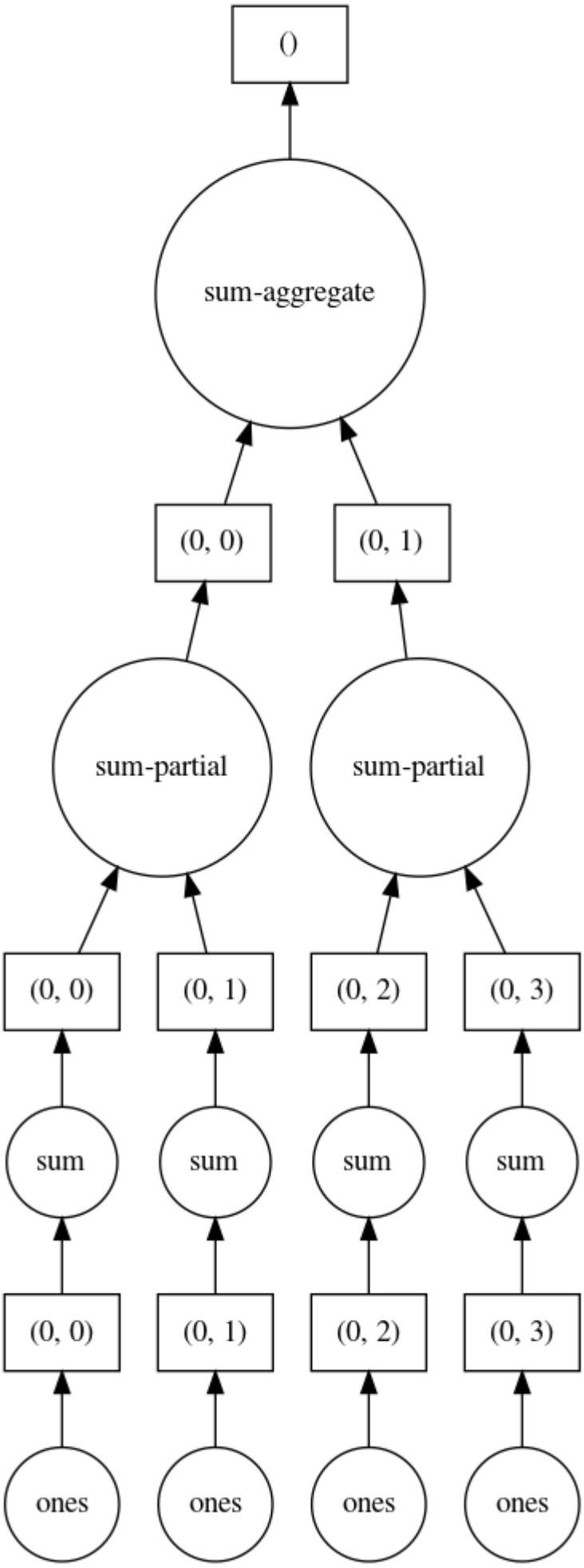
Notre tableau comporte 4 morceaux. Pour le générer, dask appelle 4 fois `np.ones` et les concatène ensuite en un tableau.

Plutôt que de charger immédiatement un tableau Dask (qui met toutes les données dans la RAM), il est plus courant de réduire les données d'une manière ou d'une autre. Par exemple :

Entrée [15]:

```
sum_of_ones = ones.sum()
sum_of_ones.visualize()
```

Out[15]:


()
sum-aggregate
(0, 0)
(0, 1)
sum-partial
sum-partial
(0, 0)
(0, 1)
(0, 2)
(0, 3)
sum
sum
sum
sum
(0, 0)
(0, 1)
(0, 2)
(0, 3)
ones
ones
ones
ones

Nous voyons ici la stratégie de Dask pour trouver la somme. Cet exemple simple illustre la beauté de dask : il conçoit automatiquement un algorithme approprié pour les opérations personnalisées avec de grosses données.

Si nous rendons notre opération plus complexe, le graphe devient plus complexe.

Entrée [16]:

```
import numpy as np
```

Entrée [17]:

```
np.arange(10)[::-1]
```

Out[17]:

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Entrée [18]:

```
ones[::-2, ::-2].compute()
```

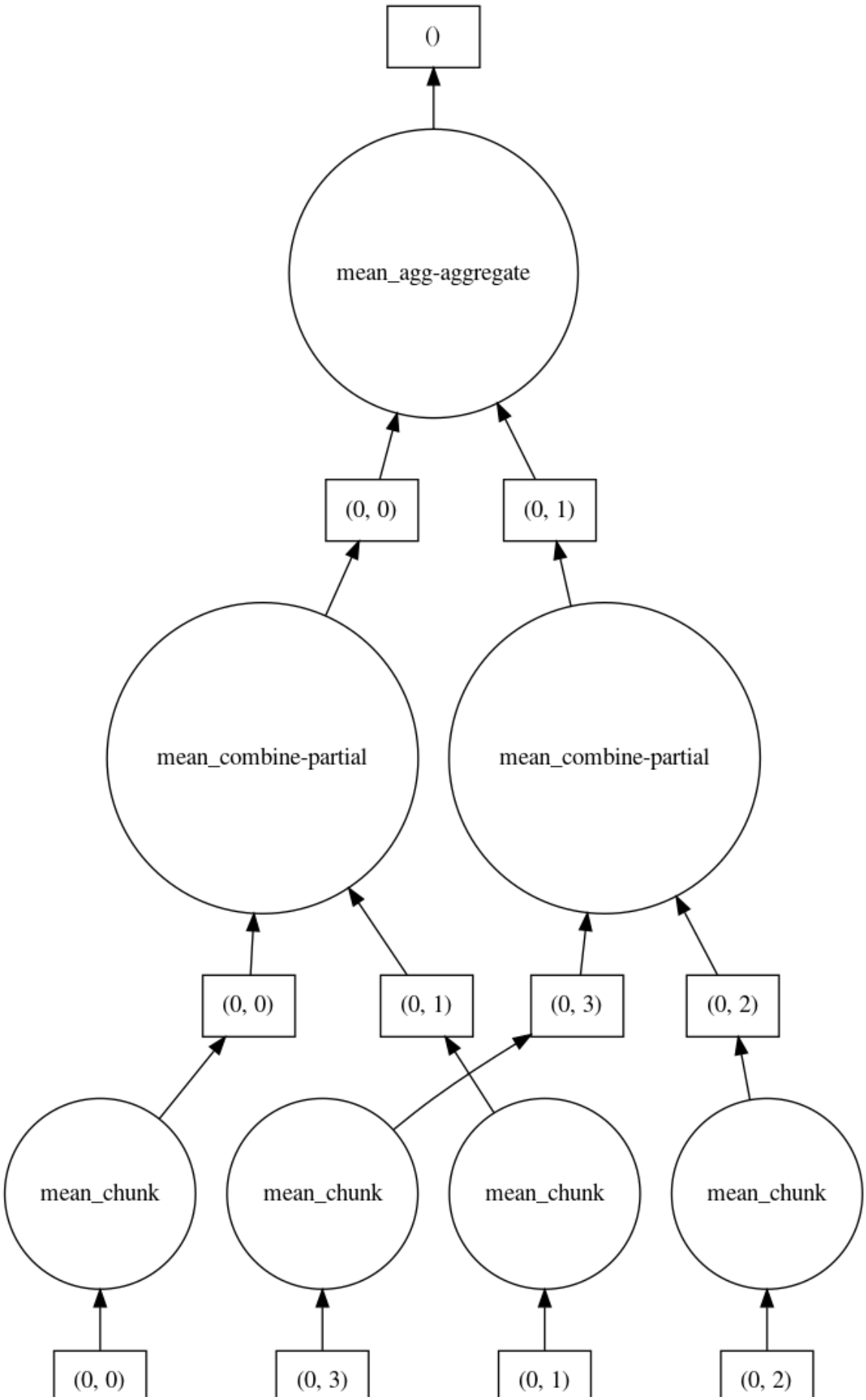
Out[18]:

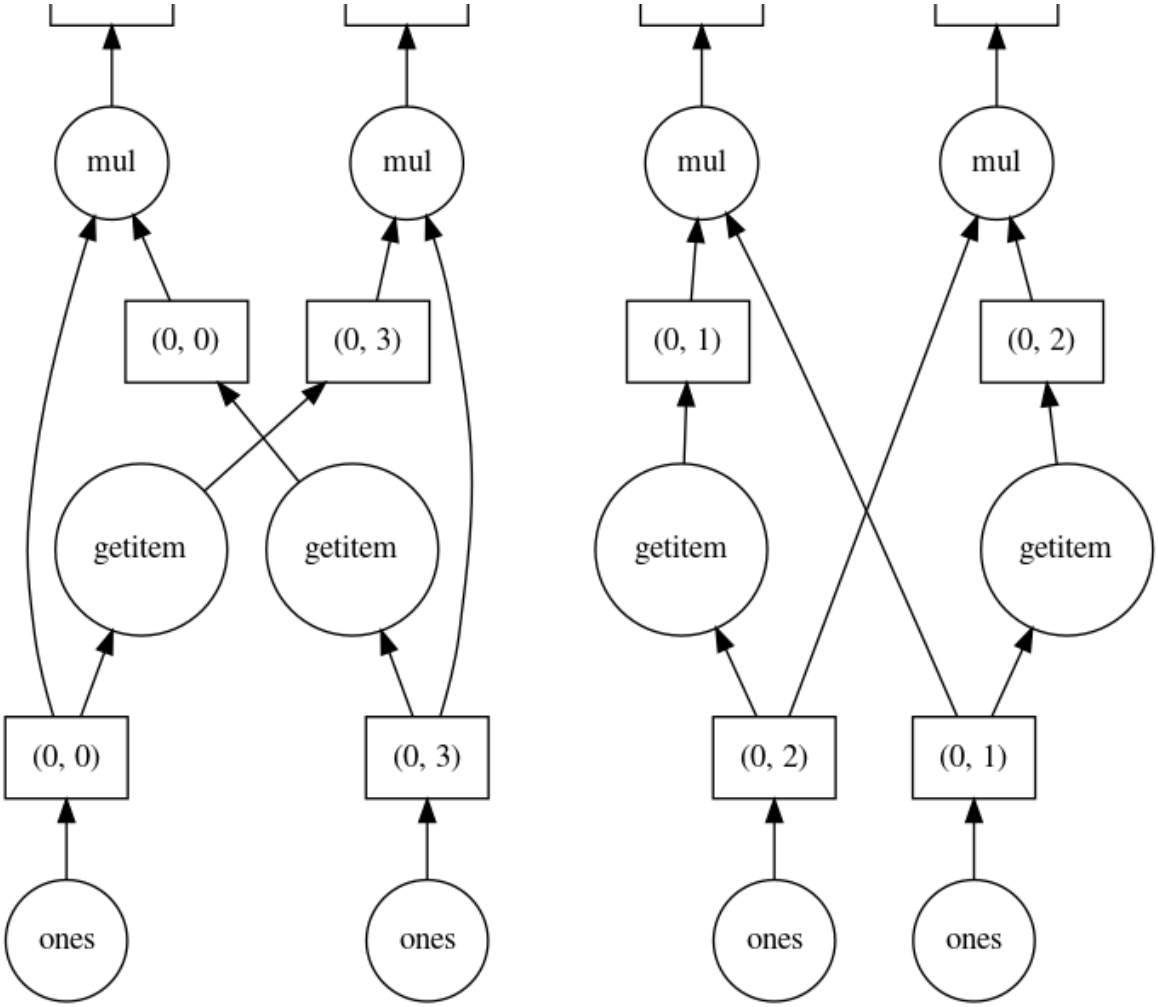
```
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

Entrée [19]:

```
fancy_calculation = (ones * ones[::-1, ::-1]).mean()
fancy_calculation.visualize()
```

Out[19]:





Un calcul plus grand

Les exemples ci-dessus étaient de petits exemples ; les données (32 Mo) sont loin d'être assez volumineuses pour justifier l'utilisation de dask.

Changeons d'échelle !

Entrée [20]:

```
bigshape = (200000, 4000)
big_ones = da.ones(bigshape, chunks=chunk_shape)
big_ones
```

Out[20]:

	Array	Chunk
Bytes	6.40 GB	8.00 MB
Shape	(200000, 4000)	(1000, 1000)
Count	800 Tasks	800 Chunks
Type	float64	numpy.ndarray

4000 200000

Cet ensemble de données est de 6,4 Go, au lieu de 32 Mo ! C'est probablement proche ou supérieur à la quantité de mémoire vive disponible que vous avez dans votre ordinateur. Néanmoins, Dask n'a aucun problème pour travailler dessus.

N'essayez pas d'appeler la méthode `.visualize()` sur ce tableau !

Lorsqu'il fait un gros calcul, dask a aussi quelques outils pour nous aider à comprendre ce qui se passe sous le capot. Regardons à nouveau le tableau de bord pendant que nous faisons un gros calcul.

Entrée [21]:

```
big_calc = (big_ones * big_ones[::-1, ::-1]).mean()

result = big_calc.compute()
result
```

Out[21]:

```
1.0
```

Réduction

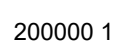
Toutes les méthodes habituelles numpy fonctionnent sur des tableaux de dask. Vous pouvez également appliquer la fonction numpy directement à un dask array, et il restera paresseux.

Entrée [22]:

```
big_ones_reduce = (np.cos(big_ones)**2).mean(axis=1)
big_ones_reduce
```

Out[22]:

	Array	Chunk
Bytes	1.60 MB	8.00 kB
Shape	(200000,)	(1000,)
Count	3400 Tasks	200 Chunks
Type	float64	numpy.ndarray



Le tracé déclenche également le calcul, car nous avons besoin des valeurs réelles.

Entrée [23]:

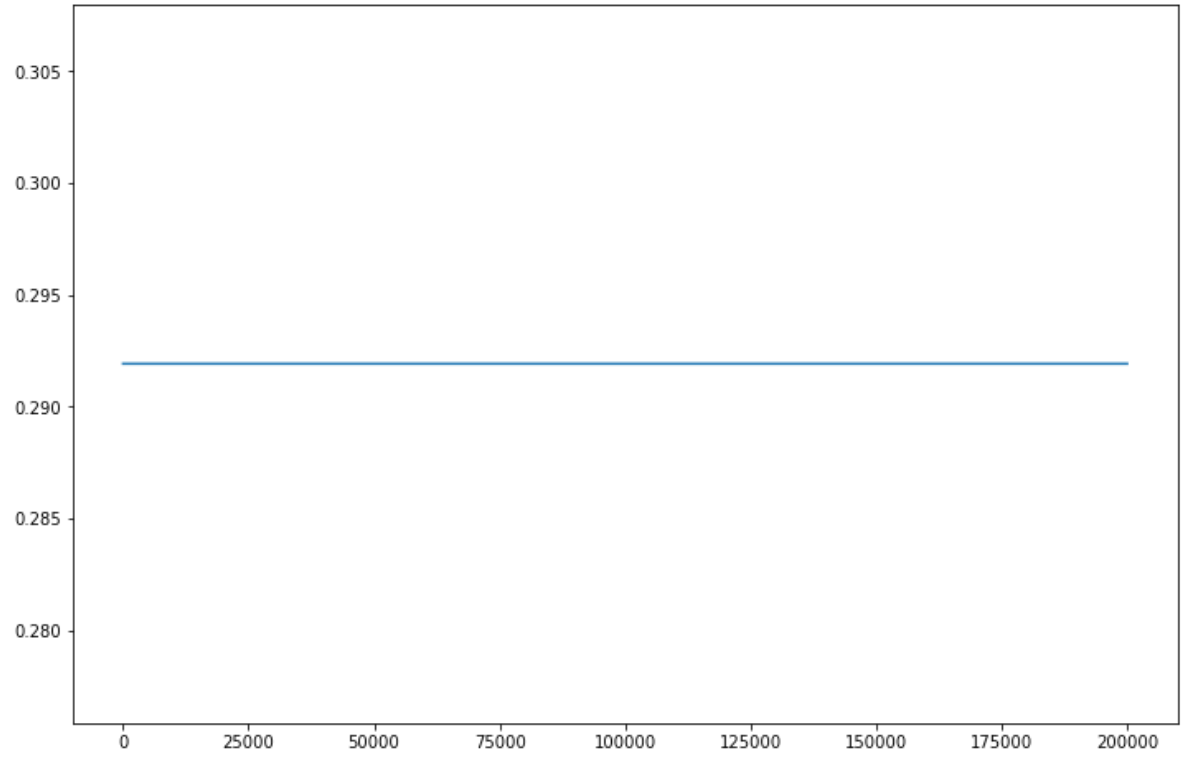
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,8)
```

Entrée [24]:

```
plt.plot(big_ones_reduce)
```

Out[24]:

```
[<matplotlib.lines.Line2D at 0x2b65435cae50>]
```



dask.delayed

`dask.delayed` est un moyen simple et puissant de paralléliser le code existant. Il permet aux utilisateurs de retarder les appels de fonctions dans un graphe de tâches avec des dépendances. `dask.delayed` ne fournit pas d'algorithmes parallèles sophistiqués comme `dask.dataframe` , mais il donne à l'utilisateur un contrôle complet sur ce qu'il veut construire.

Les systèmes comme `dask.dataframe` sont construits avec des objets `dask.delayed` . Si vous avez un problème qui est parallélisable, mais qui n'est pas aussi simple qu'un grand tableau ou une grande trame de données, alors `dask.delayed` peut être le bon choix.

Créer des fonctions simples

Ces fonctions effectuent des opérations simples comme l'addition de deux nombres, mais elles dorment pendant un temps aléatoire pour simuler un travail réel.

Entrée [25]:

```
import time


def inc(x):
    time.sleep(0.1)
    return x + 1

def dec(x):
    time.sleep(0.1)
    return x - 1

def add(x, y):
    time.sleep(0.2)
    return x + y
```

Réalisons un petit algorithme à l'aide des fonctions ci-dessus.

Entrée [26]:

```
%%time
x = inc(1)
y = dec(2)
z = add(x, y)
z
```

```
CPU times: user 13.8 ms, sys: 3.23 ms, total: 17 ms
Wall time: 401 ms
```

Out[26]:

```
3
```

Les appels se sont succédés les uns après les autres, dans l'ordre. Notez cependant que les deux premières lignes `inc(1)` et `dec(2)` ne dépendent pas l'une de l'autre, nous *aurions pu* les appeler en parallèle si nous avions été intelligents.

Annoter les fonctions avec `dask.delayed` pour les rendre paresseuses

Nous pouvons appeler `dask.delayed` pour rendr nos fonctions paresseuses. Plutôt que de calculer leurs résultats immédiatement, ils enregistrent ce que nous voulons calculer dans un graphe que nous exécuterons plus tard en parallèle.

Entrée [27]:

```
import dask
inc = dask.delayed(inc)
dec = dask.delayed(dec)
add = dask.delayed(add)
```

L'appel de ces fonctions paresseuses est désormais presque gratuit. Nous construisons juste un graphe

Entrée [28]:

```
%%time
x = inc(1)
y = dec(2)
z = add(x, y)
z
```

```
CPU times: user 241 µs, sys: 37 µs, total: 278 µs
Wall time: 286 µs
```

Out[28]:

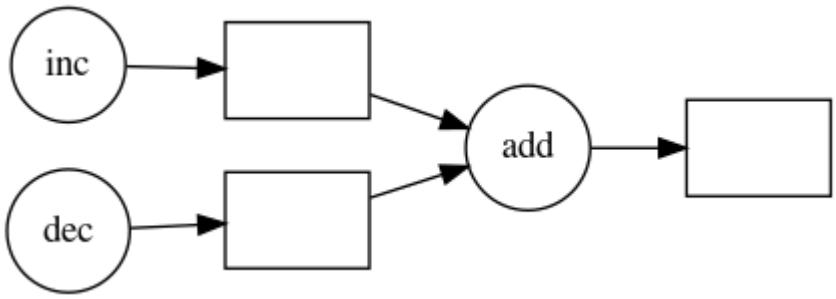
```
Delayed('add-d83e6bf0-6573-4ed3-9db8-a17faadafb19')
```

Graphe de calcul

Entrée [29]:

```
z.visualize(rankdir='LR')
```

Out[29]:



Calcul en parallèle

Appelez `.compute()` quand vous voulez que votre résultat soit un objet Python normal

Si vous avez démarré `Client()` ci-dessus, vous pouvez regarder la progression du calcul dans le tableau de bord.

Entrée [30]:

```
%%time
z.compute()
```

```
CPU times: user 20.5 ms, sys: 5.54 ms, total: 26 ms
Wall time: 421 ms
```

Out[30]:

```
3
```

Soumettre des appels de fonction au cluster

Nous pouvons soumettre des appels de fonction individuels avec la méthode `client.submit` ou de nombreux appels de fonction avec la méthode `client.map`

Entrée [31]:

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

x = client.submit(inc, 10)
x
```

Out[31]:

Future: inc status: pending, key: inc-8993bacc55179215f7eadc7dcff25597

Entrée [32]:

```
L = client.map(inc, range(1000))
L[:5]
```

Out[32]:

```
[<Future: pending, key: inc-c6ec8fa9c5d2d7ec7a8e480e4cc5e069>,
 <Future: pending, key: inc-2830b9fd354929158fff7cf060e7601d>,
 <Future: pending, key: inc-57652607f0b4084d590a36ea5c65b37c>,
 <Future: pending, key: inc-19b76c9c350736407c7d1171f119bf5a>,
 <Future: pending, key: inc-9c85eb8a7dc711a0e42d22440f3b5e40>]
```

Ces résultats vivent sur les workers distribués.

Nous pouvons soumettre des `futures` aux fonctions. La fonction ira sur le worker où les futures sont stockées et s'exécutera sur le résultat une fois qu'il sera disponible.

Entrée [33]:

```
y = client.submit(inc, x)
total = client.submit(sum, L)
```

Nous recueillons les résultats en utilisant soit la méthode `Future.result` pour un seul objet `Future`, soit la méthode `client.gather` pour plusieurs objets `Future` à la fois.

Entrée [34]:

```
x.result()
```

Out[34]:

```
11
```

Entrée [35]:

```
client.gather(L)[:5]
```

Out[35]:

```
[1, 2, 3, 4, 5]
```

Il est préférable de minimiser le nombre de rapatriements des résultats vers le processus local. Pour cela, les données sur le cluster seront, de préférence, exploitées à distance avec des fonctions telles que `submit`, `map`, `get` et `compute`.

Dask Schedulers

Les ordonnanceurs orchestrent les tâches dans les graphes de calcul afin qu'elles puissent être exécutées en parallèle. La manière dont elles sont exécutées en parallèle dépend de l'ordonnenceur que vous choisissez.

Il y a 3 ordonnanceurs locaux :

- **Single thread Local:** Pour le débogage, le profilage et diagnostiquer des problèmes
- **Multi-thread:** Utilisation du paquet `threading` (par défaut pour toutes les opérations de Dask sauf `Bags`)
- **Multi-process:** Utilisation du paquet `multiprocessing` (par défaut pour `Bags`)

et 1 ordonnanceur distribué:

- **Distributed:** Utilisation du module `dask.distributed` (qui utilise `tornado` pour les communications TCP). L'ordonnanceur distribué utilise un `Cluster` pour gérer la communication entre l'ordonnanceur et les "workers".

Dask Distributed (http://distributed.dask.org/ (http://distributed.dask.org/))

Dask peut être déployé sur une infrastructure distribuée, comme un système HPC ou un système de cloud computing.

- `LocalCluster` - Crée un `Cluster` " qui peut être exécuté localement. Chaque `Cluster` " comprend un `Scheduler` et des `Worker` s.
- `Client` - Se connecte à un `Cluster` " distribué et pilote le calcul.

Dask Jobqueue (http://jobqueue.dask.org/ (http://jobqueue.dask.org/))

- `PBSCluster`
- `SlurmCluster`
- `LSFCluster`
- etc.

Dask Kubernetes (http://kubernetes.dask.org/ (http://kubernetes.dask.org/))

- `KubeCluster`

Entrée [ ]: