Tutoriel DASK

Auteur :
MOROOKA Fabio Eid

Correspondant pédagogique INSA Rennes :
NEZAN Jean-François

Tuteurs du stage :
ORIEX François
GAC Nicolas

26 août 2019

Sommaire

Table des abréviations

Abréviation	Explication
FFT	Fast Fourier Transform
IFFT	Inverse Fast Fourier Transform
PBS	Portable Batch System
PSF	Point Spread Function
SKA	Sky Kilometer Array

Glossaire

Chunk : Morceaux d'une *dask array*

DASK : Bibliothèque python pour faire le parallelisme

I Introduction

Le but de ce tutoriel est de fournir une introduction à la bibliothèque DASK python, avec des exemples pratiques et de montrer certains des résultats de l'étude de la bibliothèque dask dans les programmes de déconvolution d'images du projet SKA (Square Kilometer Array) qui ont été réalisés pendant cette partie du stage.

I.1 Contexte

Python est de plus en plus utilisé dans le domaine des *data science* ainsi que dans la programmation en général. Dask est une bibliothèque du langage de programmation python. Il a été créé pour permettre l'utilisation des principales bibliothèques python (pandas, numpy, scikit-learn) dans les machines multi-core ainsi que dans les clusters distribués.

Avant l'apparition de dask, les analystes n'avaient que leurs propres ordinateurs personnels pour analyser leurs données, parce que l'analyse des données dans le cluster de calcul en utilsant plusieurs de machines n'était pas possible. Ils ont utilisé les bibliothèques Numpy, Pandas, Scikit-learn car elles sont plus simples, plus efficaces et intuitives. Mais lorsqu'ils avaient besoin de travailler avec de plus grandes quantités de données, ils devenaient frustrés, car il fallait repenser leurs calculs et travailler avec un autre outil, normalement programmé dans un autre langage de programmation (java, C ou C++).

Dask propose une amélioration du *workflow* des bibliothèques comme pandas, numpy et scilit-learn sans avoir à apporter de modifications majeures à l'écriture du code. Pourtant, les analystes qui savent comment travailler avec Pandas/ Numpy/ Scikit-Learn n'ont pas beaucoup d'efforts pour apprendre les équivalents DASK.

I.2 Instalation de l'environement DASK

La preparation de l'environement est plus facile si vous avez les droits administateur de la machine, mais c'est aussi possible de le faire sans cette permission.

Pour faciliter l'installation de dask et d'autres bibliothèques python, il est plus facile de créer un environnement local à partir de zéro, de cette façon, vous savez où se trouvent chaque bibliothèque.

I.2.1 Installation du Python

Avant de commencer le tutoriel sur dask, vous devez configurer l'environnement pour que dask puisse être utilisé sur votre machine.

L'installation de dask se fait après l'installation de python. Pour vérifier si python est installé sur votre ordinateur, vous devez ouvrir un terminal et écrire "python". Si vous voyez l'environnement

python, donc il est installé, sinon pour installer vous devez écrire la commande "sudo apt get install python2.6" (n'importe quel version du python2, parce que ensuite il faut faire une mise à jour). Donc, pour mettre à jour le python, utilisez la commande "apt install --only -upgrade python". Enfin, pour installer "pip" qui est un utilitaire d'installation de bibliothèques python, utilisez la commande "sudo apt install python-pip".

Note : Si vous n'avez pas python instalée, vouz devez avoir la permission d'administrateur pour le faire.

Pour vérifier que pip a été correctement installé, dans le dossier *.local/bin/*. Ensuite, vous devez voir si ce chemin est bien dans la variable PATH. Sinon ajoutez le (via le .bashrc).

Note pour la suite Si vous n'avez pas les droits d'administrateur, vous devez ajouter la commande *"--user"*, donc la commande *pip install pipnameDeLaLibrarie* devient *pip install --user pipnameDeLaLibrarie*, et dans ce cas les bibliothèques python seront installées dans le répertoire *$HOME/.local/bin* au lieu de *usr/bin*.

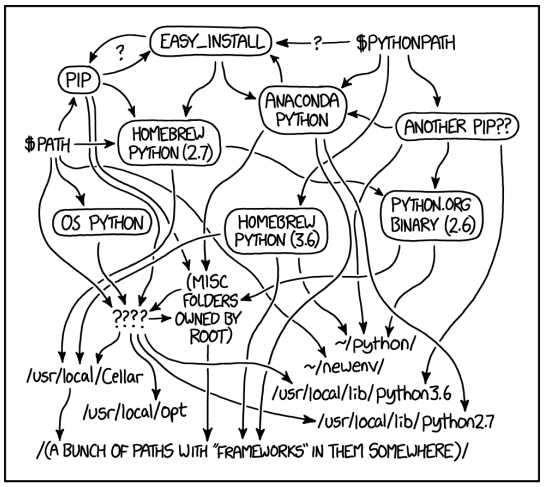
I.2.2 Installation de Anaconda/Miniconda

Anaconda ou Miniconda sont des distributions qui visent à simplifier la gestion des paquets. Les versions de paquetages sont gérées par le système de gestion de paquets conda.

L'installation de anaconda est simple, il faut télecharger l'executable en ligne dans le site anaconda (version 2.7) et ensuite lancer l'exedutable. les informations détaillé si trouvent dans la documentation de anaconda ou en ligne : `https://docs.anaconda.com/anaconda/install/linux/`.

I.2.3 Creation d'une environnement

Un environnement est un moyen de démarrer avec une nouvelle installation Python, qui ne regarde pas vos paquets déjà installés. De cette façon, il simule le fait d'avoir une nouvelle installation de Python.



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

La création d'une environnement sur linux est possible en utilisant pip pour créer une *virtual environment* (venv) ou utilser conda pour créer une environnement . La différence est que la création avec pip installe les paquets python dans n'importe quel environnement, par contre la création avec conda installe tous les paquets dans l'environnement conda. Si l'utilsation de l'environnement est que pour l'installation des paquets Python, il n'y a pas beaucoup de différence entre les deux.

Pour utiliser une environnement il y a que deux étapes à faire : Premierement il faut faire la création de l'environnement, soit à partir de zéro (un nouveau projet), soit à partir d'un fichier yaml (duplication d'un environnement) et ensuite activer l'environnement pour l'utilisation.

1. **Environnement pip :** Pour créer une environnement local python, il faut d'abord installer python et installer la bibliothèque de l'environnment local, pour cela, utilise la commande : "pip install virtualenv".

   Ensuite pour créer l'environnement local utilise la commande : "virtualenv nomEnvironnement", et pour l'activer utilse la commande : "source nomEnvironnement/bin/activate". Après l'activation, l'environnement local est prêt pour l'installation des bibliothèques comme montré avant. Enfin pour désactiver l'environnement local utiliser la commande : "deactivate".

2. **Environnement conda :** Pour créer une environnement local python avec conda, il faut d'abord installer conda (soit anaconda, soit miniconda) et ensuite utiliser la commande : "conda create--name nomEnvironnement python=2.7" pour la création de l'environnement.

   Ensuite pour l'activer, utiliser la commande : "source activate nomEnvironnement", enfin pour installer les bibliothèques python après entrer dans l'environnement local (normalement le prompt va te monter), utilser la commande : "(nomEnvironnement) $ conda install nomDeLaBibliothèque". Enfin pour désactiver l'environnement local utiliser la commande : "conda deactivate".

I.2.4 DASK

Pour installer la bibliothèque dask, vous pouvez l'installer avec conda, pip ou à partir du code source.

— **Conda :** Dans Anaconda, dask est installé par défaut. Mais il est possible de le mettre à jour avec la commande : *"conda install dask"*. Cette commande vous permet d'installer toutes les autres bibliothèques dont le dask est dépendant (comme numpy et pandas). Si vous avez besoin d'installer le minimum pour pouvoir utiliser dask, vous devez utiliser la commande : *"conda install dask-core"*.

— **Pip :** Pour l'installation avec pip, il y a deux possibilités : soit installer l'ensemble de la bibliothèque de tâches et ses dépendances avec la commande *" pip install "dask[complete]" "*, soit installer une (ou plusieurs) collections de tâches. Dans ce cas, vous devez faire attention, car les collections dask.array, dask.dataframe, dask.delayed et dask.distributed ne fonctionneront pas si les bibliothèques NumPy, Pandas, Toolz et Tornado ne sont pas installées. Pour installer les bibliothèques principales de dask, utilisez la commande *"pip install dask"*, et pour installer les dépendances d'une collection unique de dask, utilisez la commande *"pip install "dask[**nomOfCollection**]" "*, soit nameDeLaCollection array, bag, dataframe, delayed ou distributed.

— **Code source :** Le code source de la bibliothèque de dask se trouve sur le GitHub. Donc pour installer à partir du code source, vous devez utiliser les commandes : *"git clone https ://-github.com/dask/dask.git"*, puis vérifiez si le fichier a été téléchargé avec la commande *"cd dask"* et enfin si le fichier a été téléchargé, installez-le avec cette commande : *"python setup.py install"*.

I.2.5 Bibliothèques complementaires

Ensuite, il y a d'autres bibliothèques qui doivent être installées pour l'ordonnancement des calculs, parce qu'elles supportent la bibliothèque principale des tâches.

— dask-image
— graphviz*
— scikit-image
— matplotlib
— bokeh**
— pyfftw

*La bibliothèque graphviz vous permet de générer des graphes de calcul de tâches, donc si vous n'avez pas besoin de voir et d'analyser le graphes, cette bibliothèque n'est pas nécessaire. Mais si vous avez besoin de l'installer, utilisez la commande : *"sudo apt install python-pydot python-pydot-ng graphviz"*. Si vous utilisez conda, vous devez utiliser la commande *"python-graphviz"*. Graphviz peut générer des graphiques avec un maximum de 100 nœuds.

**La bibliothèque Bokeh est important pour visualizer les ordonnanceurs en ligne ainsi que les images qui seront généré.

I.2.6 Editeur de texte

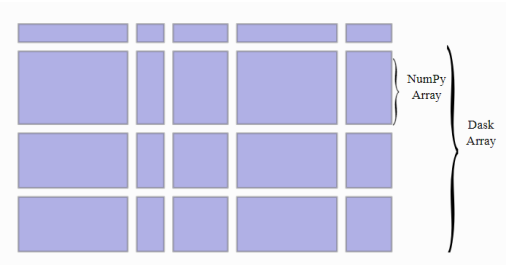
La dernière partie à vérifier est l'éditeur de texte pour écrire vos programmes, vous pouvez utiliser votre éditeur de texte préféré (gedit, vim, notepad++, notepad, atome, texte sublime, etc).


II Dask

Dask dispose de différentes interfaces qui permettent d'effectuer une programmation parallèle (aussi bien sur une seule machine que dans un cluster). Ce tutoriel vous montrera plus en détail les interfaces utilisées dans l'étude pendant le stage.

Les trois principales interfaces de dask sont : dask Array, dask Bag et dask DataFrame. Un code dask utilise une (ou plusieurs) interfaces et c'est le programmeur qui fait le choix de l'interface qui sera utilisé dans le programme. Le choix dépend des structures des données qui seront utilsées dans le code. Par exemple : une code qui utilise plus des donnnées comme des matrices et vecteurs (par exemple une image), est plus adapté à utiliser l'interface dask array. Par contre, si une identifiant a plusieurs des différentes informations (par exemple une base des données d'une entreprise), il est préférable d'utiliser l'interface dask DataFrame. De plus il y a certains codes où la définition d'une interface dask n'est pas possible, donc il y a aussi l'interface dask Delayed qui permmet la paralellization d'une code sans nune structure des donnnées défini.

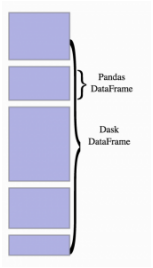
— **Arrays :** Dask array implémente un sous-ensemble de l'interface NumPy ndarray. Les tableaux de masques coupent un grand tableau numpy en plusieurs petits tableaux numpy (les *chunks*). Il permet d'utiliser tous les cœurs d'un processeur. Pour gérer ces blocs de tâches, dask array utilise des graphiques de tâches. Ces tableaux numpy peuvent être sauvegardés dans le disque de la machine ou dans d'autres machines.



Chunks : Les dask arrays sont composé par plusieurs des NumPy arrays, la manière comment les arrays sont organisé peut influencier dans la perfomance du programme. Les différentes arrangements des NumPy arrays peuvent augmenter ou diminuir la vitesse d'une programme. Donc penser sur le *chunking* et le controller est fondamental pour optimizer une algorithme.

— **Bags :** Dask bag fait la parallélisation d'une grande collection d'objets python, il est plus utilisé quand il y a une grande quantité de données JSON ou de fichiers logs. Cette collection gère les listes et les itérateurs python. Les Dask bags ont deux avantages : les données sont divisées de façon à ce que plusieurs noyaux ou machines puissent être exécutés en parallèle. De plus, les données sont exécutées d'une manière *lazy*, ce qui permet l'exécution de données qui théoriquement ne se font pas dans la mémoire, même dans les systèmes avec une machine. Pour faire le calcul du dask bag utilise la bibliothèque *dask.multiprocessing*, donc il n'utilise pas le *Global Interpreter Lock (GIL)* et utilise donc plusieurs cœurs dans les objets python.m

— **Dataframes :** Dask DataFrame est une union de Pandas DataFrames divisée par l'index. Les Pandas DataFrames peuvent être stockées sur disque pour une utilisation sur une seule machine ou sur plusieurs machines dans un cluster. Une opération sur une dask DataFrame effectue des opérations sur tous les Pandas DataFrales qui composent la dask DataFrame.



— **Machine Learning :** La collection dask-ml permmet créér des codes de *machine learning* python de plus en plus scalables. Cette collection parallelize des trois diffèrentes types de codes *machine learning* : les codes qu'on été créé avec la bibliothèque scikit-learn, les codes qu'ont des Numpy arrays (il faut simplesment changer les Numpy arrays pour dask arrays) et les codes qu'ont été créé avec d'autres bibliothèques de *machine learning*, par exemple *TensorFlow* ou *XGBoost*.

— **Delayed :** Parfois, le problème ne peut pas être résolu par l'une des collections DASK (dask array, dask bag ou dask DataFrame). Dans ces cas, il est possible de paralléliser le code avec l'interface dask.delayed. Il vous permet de créer des graphiques d'un code python.

— **Futures :** Cette collection de dask permmet une calcul immedit au lieu d'une calcul *lazy*, ça donne une flexibilité au code python.

Dans la suite nous interessons surtout au dask array, parce que dans l'étude de cas nous avons le problème de déconvolution du projet SKA, donc nous travailerons avec les images (dans ce cas fichiers avec des données correspondant à images du ciel) et nous avons le but de trouver une image la plus pareil possible de la réel à partir de la dirty et de la psf.

III Collections

III.1 Dask Graphes

Dask est une bibliothèque qui crée une graphe interne avant effectivement éxécuter le code. Ces graphes peuvent être utilisé dans une code sans d'autres collections dask. Mais il est rare cette type d'utilisation, parce que dans ces cas il y a la collection Dask Delayed.

Task Scheduling : La découpage du code dans plusieurs tâches de taille moyenne est appelé *task scheduling*. Dans les graphes, les tâches sont représéntées pour nœuds avec des fléches entre les nœuds si une tâche dépend des données produites par une autre. Ensuite une *task scheduler* est appelé pour exécuter cette graphe de façon à respecter la dépéndence des données et faire le parallelisme dans les parties possibles, parce que les divers tâches indépéndentes peuvent être exécuté au même temps.

III.1.1 Génération des graphes

Dask propose une manière simple de générer ces graphes de calcul, en utilisant les structures simples de python (dictionaires, tuples and callables). Donc dask propose une manière simple, facille d'utiliser et de comprendre pour une vaste communuté.

A Dask graphe est une structure python (*dictionary*) qu'utilise les *keys* pour faire les calculs. A *key* est une nom arbitraire (une valeur arbitraire) qui n'est pas une tâche. La tâche est une *tuple* où son première élément est une fonction et les éléments suivants sont des arguments pour cette fonction.

Enfin le *entry point* est une fonction qui permet faire le calcul d'une graphe. Dans dask, le *entry point* est la fonction *get* de la bibliothèque *dask.multiprocessing*. Cette fonction fait toutes les calculs necessaires pour obtenir le résultat.

Example : Pour bien visualizer la base des dask graphes, trouve ci-dessous deux codes python, le premier est une code simple qui fait l'incrementation d'une valeur et ensuite faire la somme de deux variables. Le deuxième fait la même opération, mais il fait la construction d'une graphe de calcul.

Code simple

```
>>> def inc(i):
...     return i + 1
>>>
>>> def add(a,b):
...     return a + b
>>>
>>> x = 1
>>> y = inc(x)
>>> z = add(y, 10)
>>> print(z)
```

Code DASK - Création et éxecution d'une graphe

```
>>> from dask.multiprocessing import get
>>>
>>> def inc(i):
...     return i + 1
>>>
>>> def add(a,b):
...     return a + b
>>>
>>> #Definition du graphe
>>> dsk = {'x': 1,
...     'y': (inc, 'x'),
...     'z': (add, 'y', 10)}
>>>
>>> get(dsk, 'z')
```

Il y a d'autres bibliothèques qui font la parallelization d'une code comme : Joblib, Multiprocessing, Ipython Parallel, Concurrent.futures et Luigi. Tous permetent qui le programmeur fait le choix des relations entre les tâches. Mais les différentiels de dask schedulers sont :

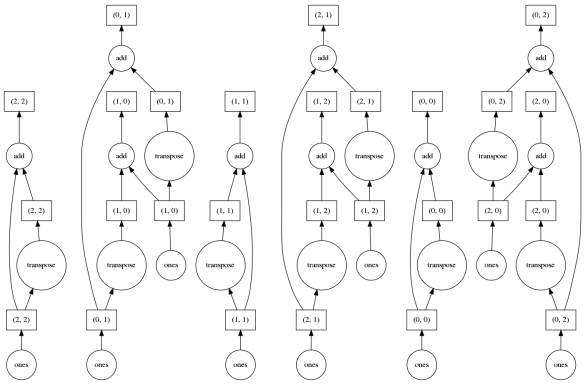
1. Utiliser une structure python *dictionary* pour décrire toute le graph.
2. Pouvoir utiliser une grand variété des différentes ordonnanceurs, allant d'une seule machine, d'un seul noyau à *threaded*, multiprocesseurs, distribués.
3. Les dask *schedulers* qui sont executé dans une seule machine font le calcul du graphe d'une manière qui minimise l'empreinte mémoire.

III.1.2 Visualization des graphes

Avant exécuter une code, il est recommendé de visualizer le graphe, parce que en regardant les connexions entre les tâches et les données est plus facile d'apprendre les possibles blocages où le parallelisme pêut-être pas possible où les lieux où il y a les tâches dépendents, qui peut générer une grande quantité de communication. Pour cela, la méthode "*.visualize()*" montre le graphe des tâches*. Par défaut les graphes sont rendu de haut en bas. Si vous préférez le visualiser de gauche à droite, passez "*rankdir="LR""* comme argument de la méthode "*.visualize()*".

Code DASK - Visualization des graphes

```
>>> import dask.array as da
>>>
>>> x = da.ones((15, 15), chunks=(5, 5))
>>> y = x + x.T
>>> y.visualize()
```



* Pour montrer les graphes il faut avoir la bibliothèque python GraphViz installé ainsi que la bibliothèque du système GraphViz (installation montré sur le site : `https://www.graphviz.org/`). De plus GraphViz prendre plus des temps pour générer les graphes qu'ont plus de 100 nœuds. Donc pour les grandes calculs il est recommendé de simplifier le calcul avant générer le graphe.

III.1.3 Calcul des graphes

Dask est une bicliothèque qui fait une calcul *lazy* de chaque fonction, donc pour obtenir la valeur d'une fonction il faut utiliser la methode *.compute()*. Cette methode fait le calcul des fonctions en faisant le paralelisme où possible (parce qu'il fait le calcul en se basant sur le graphe). Donc c'est possible de voir la relation direct entre les dask graphes et le temps de computation d'une code.

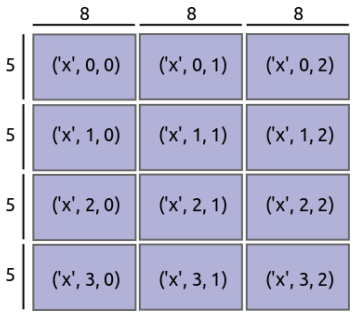
Code DASK - Computation des graphes

```
>>> import dask.array as da
>>>
>>> x = da.ones((15, 15), chunks=(5, 5))
>>> y = x + x.T
>>> y.compute()
```

Une de ses arguments est le nombre de *threads* qu'il va utiliser pour faire le calcul. (par exemple : *".compute(num_workers=4)"*.

III.2 Dask Array

Dask array est une collection dask qui est une n-dimentional array *parallel* et *larger-than-memory* qui utilise les *blocked algorithms*. *Parallel* parce que cette collection utilise toutes les cœurs du processeur pour le calcul, *larger-than-memory* parce que cette collection sépare les array en *chunks* afin de minimizer l'empreinte mémoire du calcul en distribuant efficcacement les donnés.



Un *blocked algorithms* effectue un calcul d'un grand tableau en effectuant des calculs de plusieurs petits blocs de ce tableau.

Prenons par exemple la somme d'un milliard de chiffres. Nous pourrions plutôt diviser le tableau en mille *chunks*, chacun de taille un million, prendre la somme de chaque *chunk*, et ensuite prendre la somme des sommes intermédiaires.

Dask, ainsi que Numpy, utilise ce type de calcul, de sorte qu'il permet des calculs qui ont une plus grande quantité de données que la taille de la mémoire. Elle utilise la même interface que la bibliothèque Numpy. Par exemple la fonction dask array *dask_ array.sum()* correspond à la fonction Numpy *numpy_ array.sum()*. Mais les deux fonctions ne fonctionnent pas de la même manière. Dask, contrairement à Numpy, avant l'execution fait une construction du calcul.

Cette différence est possible parce que les dask arrays sont divisés en *chunks* et chaque *chunk* doit avoir des calculs effectués sur ce *chunk* explicitement. Si la réponse souhaitée provient d'une petite tranche de l'ensemble des données, l'exécution du calcul sur toutes les données serait un gaspillage du temps de calcul du CPU et de la mémoire.

Ainsi, les dask arrays sont des objets qui sont évalués d'une manière *lazy*, donc toutes les opérations qui sont faites dans les dask arrays (par exemple *dask_ array.sum()*) constituent un graphique des tâches bloquées à exécuter. Pour déclencher le calcul réel, vous devez utiliser la fonction *.compute()* de l'objet dask array.

Example : Pour voir la différence de performance entre un code dask et un code numpy vous pouvez voir avec le code suivant :

Code Numpy - Simple teste

```
>>> import numpy as np
>>>
>>> %%time
...     x = np.random.normal(10, 0.1, size=(20000, 20000))
...     y = x.mean(axis=0)[::100]
...     y
```

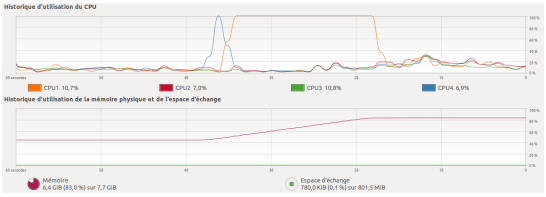
Code DASK - Simple teste

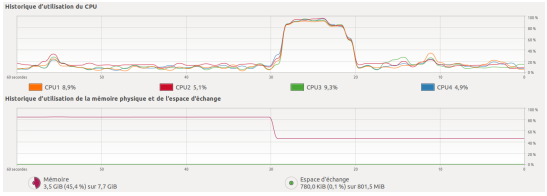
```
>>> import dask.array as da
>>>
>>> %%time
```

```
...     x = da.random.normal(10, 0.1, size=(20000, 20000), chunks=(1000, 1000))
...     y = x.mean(axis=0)[::100]
...     y.compute()
```

Dans cette exemple il y a une différence dans le temps de calcul de chaque code, le code Numpy fait le calcul plus lentement que le code dask, par contre cette programme utilse plus de temps de CPU. Alors, c'est possible d'observer que dask termine d'éxecuter le code plus vite, mais utilise plus des temps de CPU, parce que dask est capable de parallelizer le calcul à cause de la taille du *chunk*.

Dans les images ci-dessous on observe le comportement des CPU dans l'ordinateur ainsi que la memoire utilisé.





C'est possbile de voir que dajs utilise bien les 4 CPU de l'ordinateur, cependant Numpy utilise qu'une CPU.

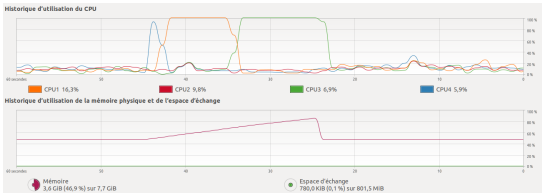
Maintenant si on fait deux changements dans le code dask, premièrement nous pouvons changer la taille du *chunk* par une *chunk*=(20000,20000), ensuite on changera la valeur du *chunk*=(25,25).

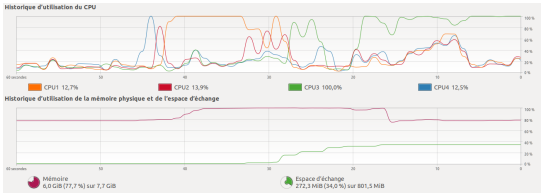
Code DASK - Augmenter la taille du *chunk*

```
>>> import dask.array as da
>>>
>>> %%time
...     x = da.random.normal(10, 0.1, size=(20000, 20000), chunks=(20000, 20000))
...     y = x.mean(axis=0)[::100]
...     y.compute()
```

Code DASK - Diminuer la taille du *chunk*

```
>>> import dask.array as da
>>>
>>> %%time
...     x = da.random.normal(10, 0.1, size=(20000, 20000), chunks=(25, 25))
...     y = x.mean(axis=0)[::100]
...     y.compute()
```





À partir des exemples on voit bien que si on augmente la taille du *chunk* par la même taille de l'array c'est comme si nous avons qu'une array, donc le calcul est proche d'une calcul en utilisant Numpy. Si on diminue la valeur du *chunk* par une taille trop petite on a une augmentation du temps de calcul, parce que comme il y a plus des morceaux il prends plus de temps pour faire le calcul de chaque morceaux (de plus il utilise plus de la memoire, on voit bien qu'il fait le *swap* dans le deuxième cas).

À partir de cette exemple, on voit bien que la choix d'une bonne valeur du *chunk* est fondamental pour l'optimization du programme, donc pour bien choisir la bon valeur du *chunk* il faut penser en 4 choses :

1. Le *chunk* doit être petit sufisant pour peût être mis dans la memoire.
2. La taille du *chunk* normalement est entre 10MB et 1GB, il peût changer selon la taille de la mémoire RAM et la duration du calcul
3. Le *chunk* doit être d'accord avec son calcul. Par exemple, si on fait une calcule de somme de deux arrays, le plus efficace c'est le cas où les arrays ont le même *chunk*.
4. Une *chunk* doit être assez gros pour que les calculs sur ce *chunk* prennent beaucoup plus de temps que la surcharge de 1ms par tâche que l'ordonnancement de Dask. Une tâche doit durer plus de 100 ms.

Enfin dask array n'est pas parfait, il y a des limitations, par exemple ce n'est pas possible de faire des *sort* avec cette collections, car il est très difficile d'implementer un algorithme d'ordonnacement en parallèle. De plus, dask array ne fait pas toutes les fonctions de *np.linalg*.

III.3 Dask DataFrame

Les Pandas DataFrames peuvent être stockée sur le disque pour faire le calcul *larger-than-memory* dans une seule machine ou dans plusieurs différentes machines dans une cluster de calcul.

Normalement dask DataFrame est utilisé dans le cas ou l'utilisation de Pandas est essentiel. Notamment, quand Pandas n'est peut pas être utilisé car la taille des données est grand ou quand la vitesse d'exécution est insufisant. Donc dask DataFrame est essentiel quand :

1. Il y a une besoin de manipuler une grande quantité de *datasets*, même quand les *datesets* ne tiennet pas en mémoire.
2. Il faut accélerer l'éxecution du code en utilsant plusieur cœurs.
3. Il a la necessité de faire le calcul distribué des opérations de grandes Pandas *datasets* (comme GroupBy, Join ...)

Comme dask array, dask DataFrame n'est pas toujours la meilleure collection pour une code, quelque fois utiliser Pandas est plus efficace, par exemple quand les données peuvent être sauvegardé dans la memoire RAM, ou quand les données ne peuvent pas s'insérir dans une tableau (il est meillleur d'utiliser dask array ou dask bag). S'il y a des fonctions qui ne sont pas implementé avec les dask DataFrame il est recommendé d'utilser dask Delayed. Enfin s'il y a une besoin d'avoir une base de données appropriée avec tout ce que les bases de données offrent, vous pourriez préférer quelque chose comme Postgres.

Example : Ainsi que la relation entre numpy et dask array, la relation entre les fonctions Pandas et les fonctions dask DataFrame sont les mêmes. Donc une utilisateur Pandas n'a pas des grandes difficultés pour utiliser dask DataFrame.

Code DASK - Simple teste

```
>>> import dask.dataframe as dd
>>>
>>> df = dd.read_csv('staderennais.csv')
>>>
>>> df.head(len(df)) #Le header du tableau avec les types
>>> df.compute() #Tous les données du tableau
>>>
>>> df.mean().compute() #Moyennes des columnes ou il y a les numeros
>>>
>>> df.Age.mean().compute() #Moyenne des ages
>>>
>>> df.loc[df['Numero']==1].compute() #Select une line ou le numero=1
>>>
>>> df.iloc[:,1].compute() #Toutes les valeus de la première colonne
>>>
>>> df.DebutContrat.compute() #Toutes les valeurs de la colonne "debutContrat"
>>>
>>> df.Age.max().compute() #La valeur maximalle des Ages
>>>
>>> df.Age.value_counts().compute() #Compter l'aparition pour age
>>>
>>> df.loc[df['ClubOrigine']=='Stade_de_Rennes'].compute()
>>>
>>> df.loc[df['Pays_(origine)']!='France'].compute()
```

III.4 Dask Delayed

Il y a des problèmes qui sont parallelizables, mais qui ne peuvent pas être programmé en utilisant dask array ou dask DataFrame. Dans ce cas il faut utiliser l'interface dask delayed. Cette interface permmet de paralellizer une code python d'une façon simple, parce qu'elle crée des objets *delayed* qui sont des objets qui seront calculé de manière *lazy*. Donc au lieu d'être calculé au moment, ils seront metre dans le graph et seront calculé après.

Example 1 Dans cette premier exemple sera montre dans le cas où il y a une somme de deux différents variables qui ensuite seront sommées qui la paralelization est possible, dans le premier moment il seront executé au moment qu'ils sont appellé, mais après sera montré le calcul en parallel.

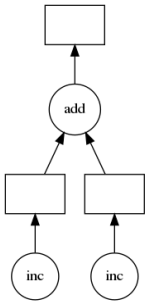
Code sequentiel

```
>>> from time import sleep
>>>
>>> def inc(x):
...     sleep(1)
...     return x+1
>>>
>>> def add(x,y):
...     sleep(1)
...     return x+y
>>>
>>> %%time
...     x = inc(1)
...     y = inc(2)
...     z = add(x,y)
```

Code DASK - Parallèlization

```
>>> from time import sleep
>>> from dask import delayed
>>>
>>> def inc(x):
...     sleep(1)
...     return x+1
>>>
>>> def add(x,y):
...     sleep(1)
...     return x+y
>>>
>>> %%time
...     x = delayed(inc)(1)
...     y = delayed(inc)(2)
...     z = delayed(add)(x,y)
```

Dans le premier exemple, le programme execute en 3s, donc on voit bien qu'il a été executé sequencielement, mais dans le deuxième cas il a executé en 2s, donc il a parallelize les deux fonction "inc". On peut visualizer le graph de cette programme et on voit bien la parallèlization.



Example 2 Cette collection fait aussi plusieurs parallèlization, donc si dans le code il y a plusieurs des fonctions qui peuvent être executées au même temps une fois qu'il n'y a pas des données dépendentes, cette collection peut être utilisé.

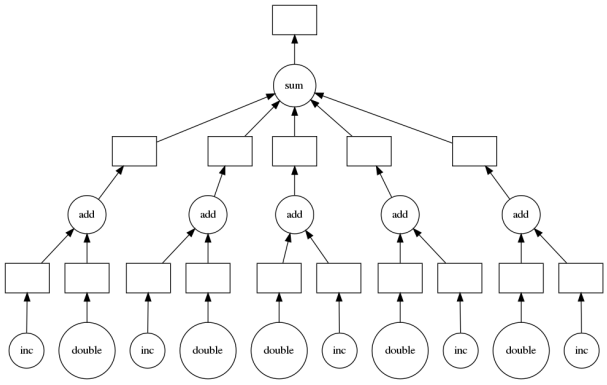
Code sequentiel

```
>>> def inc(x):
...     return x+1
>>>
>>> def double(x):
...     return x+2
>>>
>>> def add(x,y):
...     return x+y
>>>
>>> data = [1, 2, 3, 4, 5]
>>> output = []
>>> for x in data:
...     a = inc(x)
...     b = inc(x)
```

```
...         c = add(a,b)
...         output.append(c)
>>>
>>> total = sum(output)
```

Code DASK - Parallèlization

```
>>> from dask import delayed
>>>
>>> def inc(x):
...         return x+1
>>>
>>> def double(x):
...         return x+2
>>>
>>> def add(x,y):
...         return x+y
>>>
>>> data = [1, 2, 3, 4, 5]
>>> output = []
>>> for x in data:
...         a = delayed(inc)(x)
...         b = delayed(inc)(x)
...         c = delayed(add)(a,b)
...         output.append(c)
>>>
>>> total = delayed(sum)(output)
>>> total.visualize()
```



Dans le graphe du code DASK c'est possible d'apercevoir que les fonctions seront executées en parallèle, une fois que le fonctions sont mis une à côte d'autre. Enfin, cette collection est aussi une *decorator*, donc le code d'avant peut être écrit comme ci-dessous.

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

```
>>> def double(x):
...     return x+2
>>>
>>> @dask.delayed
>>> def add(x,y):
...     return x+y
>>>
>>> data = [1, 2, 3, 4, 5]
>>> output = []
>>> for x in data:
...     a = inc(x)
...     b = inc(x)
...     c = add(a,b)
...     output.append(c)
>>>
>>> total = dask.delayed(sum)(output)
>>> total.compute()
```

III.5 Dask Bag

Dask bag est une structure qui permet l'implementation des fonctions *map*, *filter*, *fold* et *groupby* sur les groupes des objets python genériques. Il fait la paralellization avec peu d'empreinte mémoire en utilsant itérateurs python. Il y a deux grandes bénéfices : l'exécution en parallel parce que il divise le data, qui permet l'exécution dans plusieurs cœrs ou plusieurs machines (cluster), et aussi fait le calcul de manière *lazy*, qui permet l'exécution des données *larger-then-memory*.

Par contre, dask bags a quelques limitations :

1. Dépendre d'une ordonnanceur multiprocesseur
2. C'est impossible de changer les Bags, donc c'est impossible de changer la valeur individual d'une element.
3. Les opérations sont normalement plus lent que les calculs en utilisant dask DataFrame ou dask array.
4. La fonction *groupby* est lent.

Code simple

```
>>> def iseven(n):
...     return n % 2 == 0
>>>
>>> def squared(x):
...     return x ** 2
>>>
>>> map(squared, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>>
>>> filter(iseven, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Code DASK - Parallèlization

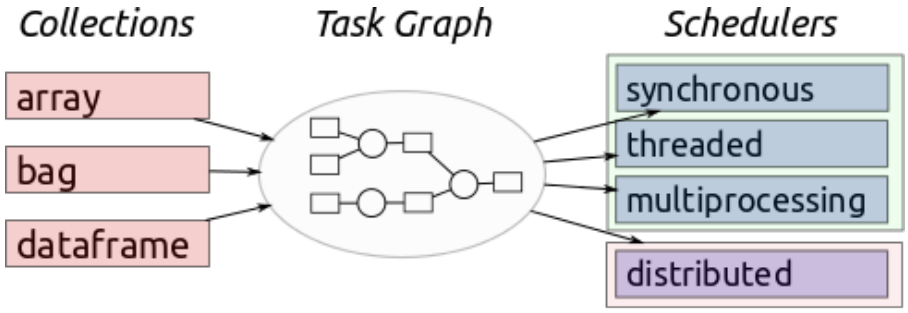
```
>>> import dask.bag as db
>>>
>>> def iseven(n):
```

```
...     return n % 2 == 0
>>>
>>> def squared(x):
...     return x ** 2
>>>
>>> b = db.from_sequence([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> b.map(squared).compute()
>>> b.filter(iseven).compute()
```

IV Analyse de performance

IV.1 Dask *schedulers*

Les collections dask (dask array, dask DataFrame, dask Bag, dask Delayed et dask Futures) génèrent une dask graphe où chaque nœds dans le graphe est une fonction python et les connections entre les nœds sont les objets python qui sont une sortie d'une autre fonction python et seront une entrée d'autre tâche. Le but d'une *task scheduler* est éxécuter en parallèle cette graphe dans une *hardware* parallèle. Il y a plusieurs des *task schedulers* chaqu'un fait le calcul d'une graphe,et tous obtiennet le mêm résultat, mais chaqu'un a une caractéristique.



Il y a deux groupes de *task schdulers* dans dask :

1. ***Single machine scheduler :*** Il est une *scheduler* simple et pas chèr pour utiliser, par contre il peut être utilisé seulement dans une seule machine. Il est le *scheduler* par défaut, car il était le première.
2. ***Distributed scheduler :*** Il est plus sophistiqué, avec plus des caractéristiques, mais il a besoin de plus d'effort pour initialiser. De plus il peut être executé localment ou dans une *cluster*.

IV.2 Dask *diagnostics*

Premièrment pour accélerer le code, il faut comprendre mieux toutes les coûts du code. Pour cela, normalenment les programmeurs utilisent les modules : CProfile, %%prun Ipython magic, VMProf ou snakeviz. Par contre ce n'est pas toutes ses modules qui font bien dans les codes multi-threaded ou code multi-processes, et très peu font le *profiling* dans plusieurs machines (cluster). De plus il y a des nouveau coûts comme le transfer des données, la serialization et d'autres coûts qui auparavant n'étaient pas identifier. Donc dask *schedulers* propose des outils qui permet une analyse des codes dask python pour mieux comprendre la performance du code.

IV.2.1 Dask graphes

Les graphes qui dask génére sont le premier outil qui permet analyser la perfomance du code. À partir de la visualization des graphes (avec la fonction "*.visualize()*"), c'est possible de mieux comprendre les relations des fonctions et des objets python qui sont utilisé dans le calcul, ainsi que voir l'ordre et la façon du calcul.

IV.2.2 Diagnostiques local

1. ***ProgressBar :***Le *ProgressBar* est une outil qui montre une barre de progression dans le terminal au cours du calcul. Cette outil permet une retour du calcul pendant l'éxecution du graphe.

   Exemple 1 : Code simple avec une comparaison avec le %%prun Ipython magic.

```
>>> from dask.diagnostics import ProgressBar
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> with ProgressBar():
...     out = res.compute()
>>>
>>> %%time #%%prun Ipython magic pour comparer avec le ProgressBar
...     res.compute()
```

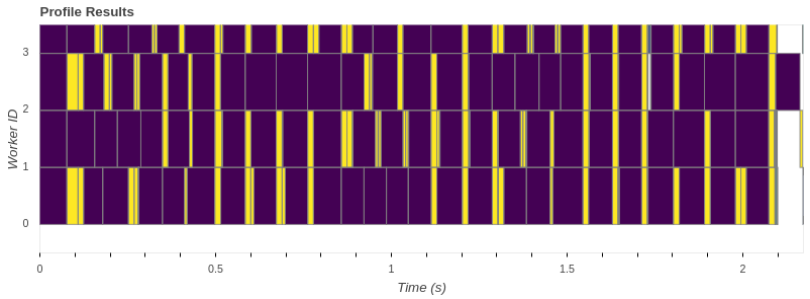
   Exemple 2 : C'est possible de enregistrer l'outil globalement dans le code et l'utiliser.

```
>>> from dask.diagnostics import ProgressBar
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> pbar = ProgressBar()
>>> pbar.register()
>>> out = res.compute()
>>>
>>> pbar.unregister() #Pour désenregistrer l'outil
>>> pbar.last_duration #Variable qui sauvegarde la valeur du temps de
    calcul
```

2. ***Profiler :*** La classe *Profiler* fait une profilage d'execution du calcul à une niveau d'une tâche. Donc il montre dans une image les *workers* et quel calcul est fait dans chaqu'un. De plus il sauvegarde les informations (la clé, la tâche, le temps de démarrage en secondes depuis l'epoch, le temps d'arrivée en secondes depuis l'epoch et le identifiant du *worker*) de chaque tâche.

   Exemple : Code simple pour montrer le résultat de l'outil *Profiler*.

```
>>> from dask.diagnostics import Profiler
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> with Profiler() as prof:
...     out = res.compute()
>>> prof.visualize()
>>> prof.results #Montre toutes les informations sauvegardés
```

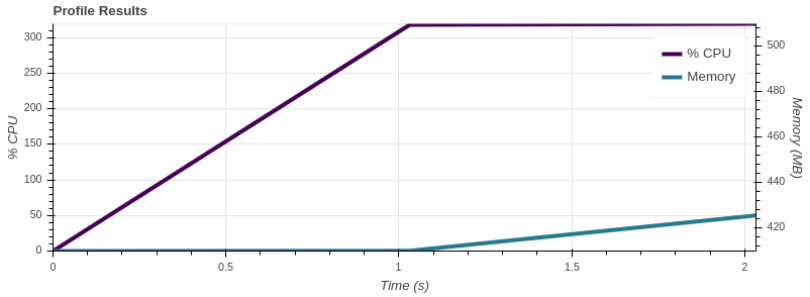


3. ***ResourceProfiler :*** Le *ResourceProfiler* permet faire le profilage de l'execution en visant les ressources utilisées. Donc il sauvegarde les informations (temps en secondes depuis l'epoch, l'utilsation de la memoire en MB et la percentage d'utilisation de la CPU) pour chaque pas de temps.

   Par défaut le pas de temps est de 1s, mais c'est possible de changer avec le paramètre "dt".

   Exemple : Code simple pour montrer le résultat de la classe *ResourceProfiler*.

```
>>> from dask.diagnostics import ResourceProfiler
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> with ResourceProfiler() as rprof: #pour changer le temps: "with
    ResourceProfiler(dt=0.5) as rprof":
...     out = res.compute()
>>> rprof.visualize()
>>> rprof.results #Montre toutes les informations sauvegardés
```

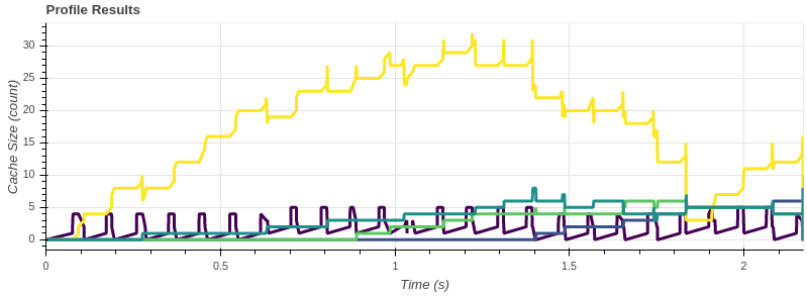


4. ***CacheProfiler :*** La classe *CacheProfiler* fait une profilage de l'éxecution à niveau de la cache. Donc elle sauvegarde les informations suivantes : la clé, la tâche, le *size metric*, le temps de démarrage de la cache en secondes depuis l'epoch et le temps d'arrivée de la cache en secondes depuis l'epoch.

   Le *size metric* est la sortie d'une fonction appelée au résultat de chaque tâche. Par défaut sa valeur est égal à 1. Mais c'est possible de changer pour mesurer par exemple la quantité des *bytes* dans la cache d'ordonnanceur (avec la fonction "nbytes" de la bibliothèque "cachey").

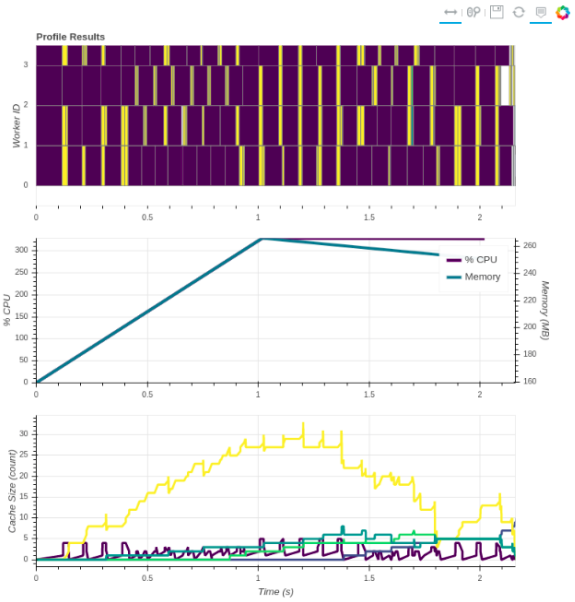
   Exemple : Code simple pour montrer le résultat de la classe *CacheProfiler*.

```
>>> from dask.diagnostics import CacheProfiler
>>> import dask.array as da
>>> #from cachey import nbytes
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> with CacheProfiler() as cprof: #pour changer le temps: "with
    CacheProfiler(metric=nbytes) as cprof":
...     out = res.compute()
>>> cprof.visualize()
>>> cprof.results #Montre toutes les informations sauvegardés
```



Note : C'est aussi possible d'utilser les 3 profilage au même temps dans le même code, comme l'exemple suivant.

```
>>> from dask.diagnostics import ProgressBar, Profiler, ResourceProfiler,
    CacheProfiler, visualize
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>> pbar = ProgressBar()
>>>
>>> with pbar, Profiler() as prof, ResourceProfiler() as rprof,
    CacheProfiler() as cprof:
...     out = res.compute()
>>> visualize([prof, rprof, cprof]) #L'ordre ici c'est l'ordre de l'image
```



5. ***Custom Callbacks :*** Les *callbacks* permet analyser l'execution d'orordonnanceur
   Exemple : Code simple pour montrer le résultat de la fonction *callbacks*.

```
>>> from dask.callbacks import Callback
>>> import dask.array as da
>>>
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
>>> class PrintKeys(Callback):
...     def _pretask(self, key, dask, state):
...         """Print the key of every task as it's started"""
...         print("Computing: {0}!".format(repr(key)))
>>>
>>> with PrintKeys():
...     res.compute()
```

IV.2.3 Diagnostiques distribué

Pour le diagnostiques des calculs distribuées sont fait avec l'aide d'une *dashboard* en ligne. Elle containt plusieurs des informations qui permmet une analyse complet de l'execution do code. De plus il est possible d'utiliser une *ProgressBar* comme montré aupuravant.

Pour utiliser ProgressBar dans une calcul distribué il faut changer un peu le code d'aupuravant.

```
>>> from dask.diagnostics import ProgressBar
>>> import dask.array as da
>>> from dask.distributed import Client, progress
>>>
>>> client = Client()
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = (a.T + a).mean(axis=0)
>>>
```

```
>>> res = res.persist()
>>> progress(res)
```

Pour utiliser le *dashboard* il faut avoir Bokeh installé. Le *dashboard* est normalement dans le serveur : `http://localhost:8787/status`, mais si cette porte est déjà pris, elle sera dans une autre porte. S'il y a besoin de la vérifier, vous pouvez l'acceder à partir de la commande : "client.scheduler_info()['services']".

Pour initialiser une *dashboard*, il y a deux manières, une plus simple qui ne permet pas changer le nombre de *workers*, et d'autre manière qui permet utiliser plusieurs machines comme *workers*

Configuration plus simple

```
>>> from dask.distributed import Client
>>>
>>> client = Client()  # start distributed scheduler locally. Launch dashboard
>>> client.scheduler_info()['services'] #verifier la porte
```

Le configuration où c'est possible de changer le nombre de *workers* est fait dans le terminal. Premierement dans une terminal utilsez la commande : "dask-scheduler". Attendez une peu, la deuxième ligne *"distributed.scheduler - INFO - Scheduler at : tcp ://160.228.203.193 :8786"* montre l'adresse ou le *dashboard* est hébergé, dans ce cas dans l'adresse : "tcp ://160.228.203.193 :8786". Ensuite, il faut ouvrir une autre terminal et utiliser la commande *dask-worker ADRESSE* pour initialiser les *workers*.

Si vous avez besoin de plusieurs *workers* il faut utiliser les options "-npcs N", où N est la quantité de *workers* qui vous avez besoin.

Pour mieux comprendre toutes les fenetres du *scheduler*, vous pouvez regarder un video en ligne : `https://www.youtube.com/watch?time_continue=1&v=N_GqzcuGLCY`.

V Cluster de calcul

Il est possible d'utiliser DASK sur une cluster de calcul. Pour ces testes le cluster utilisé est le cluster fusion, une cluster de calcul du mésocentre Moulon (`http://mesocentre.centralesupelec.fr/`).

Le cluster de calcul fusion a été crée par deux institutions : CentraleSupélec et ENS Paris-Saclay. Le but de cette création était de creer une structure des calculs puissantes pour les deux établissements.

Il est compris par plusieurs resources matériel. Sur le site web du mesocentre il est possible de trouver les spécifications plus détailés de chaque partie matériel du cluster.

V.1 Conexion avec une fichier PBS

Pour utilser le cluster il faut utliser une fichier de communication avec le serveur, pour cette cluster la communication est la PBS.

```
#!/bin/bash

#PBS -l walltime=00:20:00
#PBS -l select=1:ncpus=4:mem=8gb
#PBS -M fabioeid.morooka@l2s.centralesupelec.fr
#PBS -m be
#PBS -N dask_seq
#PBS -j oe
#PBS -P gpi

# Load necessary modules
module load anaconda2/2019.03

# Activate anaconda environment
source activate myenv

# Move to directory where the job was submitted
cd $PBS_O_WORKDIR

# Run python script
python simple.py
```

Ensuite pour utiliser cette fichier comme configuration du cluster il faut d'abord se conecter au Cluster fusion (avec *ssh*). Il y a 3 principal commandes basiques à utiliser sur le cluster : *qsub* qui est la commande pour sousmettre une programme dans une queue, *qstat* la commande pour voir la situation d'une programme dans la queue (avec des options "-H" NUMERO_TACHE) pour voir plus d'information et *qdel* pour suprimmer une programme dans la queue.

V.2 Conexion avec PBS-DASK

Une autre option pour utiliser DASK dans le cluster de calcul est avec la librarie *dask-jobqueue*. Cette librarie permet de faire la connexion à une cluster et ensuite l'utiliser pour faire le calcul du code.

Dans ce cas, il faut d'abord lancer une Jupyter Notebook dans le cluster pour que soit possible les calculs en utilisant les ressources du cluster. Le Jupyter notebook est une application web utilisée pour programmer plusieurs langages de programmation.

Pour lancer une jupyter dans le cluster il faut installer jupyter notebook dans l'environemment virtuel avec la commande : "conda install jupyterlab". Ensuite avant lancer le notebook jupyter il est plus sécure de mettre une mot de passe dans votre notebook, pour cela il faut creer une fichier de configuration ("jupyter lab --generate-config") en cas il n'y a pas. Pour vérifier si le ficher exist, il est dans le dossier : /.jupyter/jupyter_notebook_config.py

Ouvrir cette fichier et ensuite changer la ligne "c.NotebookApp.password = ”", il faut mettre une mot de passe 'hashed', pour cela il faut ouvrir une nouveau terminal local. Dans cette terminal il faut lancer python et ensuite utiliser la fonction passwd().

Mettre une mot de passe dans le Jupyter Notebook

```
>>> from notebook.auth import passwd
>>>
>>> passwd()
>>> Enter password: #Ici vous écrivez une mot de passe
>>> Verify password: #Répetez le mot de passe
>>> 'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed' #Sortie
```

Le mot de passe definit avant sera le mot de passe de votre notebook. Ensuite copiez la sortie 'sha1 :67c9e60bb8b6 :9ffede0825894254b2e042ea597d771089e11aed' (dans ce cas) et colez cette sortie dans la ligne "c.NotebookApp.password = " Après vous pouvez sauvegarder et sortir du fichier de configuration.

Ensuite pour lancer le notebook jupyter il faut utiliser la commande : "jupyter lab –no-browser"

Enfin vous pouvez faire une tunnel ssh pour eviter de lancer le navegateur dans la frontal : "ssh -L 2000 :localhost :8888 morooka@fusion.centralesupelec.fr" Ensuite dans votre machine lance une navegateur et utiliser l'adresse définit avant pour entrer dans votre notebook "localhost :2000"

Nb. : Quand vous lancez la dashboard il faut faire une autre tunnel ssh pour cette adresse aussi.

Maintenant toutes les codes écrites dans le jupyter notebook ouvert dans une navegateur utilisent les ressources du cluster de calcul. Pour bien definir la configuration du cluster il faut utiliser le code suivant.

Configuration du Jupyter notebook

```
>>> from dask_jobqueue import PBSCluster
>>>
>>> cluster = PBSCluster(cores = 24, memory = '32GB', job_extra=['-P decowska'])
>>> cluster.job_script() #Voir le 'fichier' PBS
>>> N = 10 #Nombre de workers
>>> cluster.scale(N) #Demande de workers
>>> ##Attend que toutes les workers sont prêt
>>> from dask.distributed import Client
>>>
>>> client = Client(cluster) #Conecter la dashboard au cluster
>>> client.scheduler_info()['services'] #Voir la port de la dashboard
>>>
>>> #Après toutes les calcul fait dans le notebook utilisent les ressources du
    cluster.
```

Références

[1] Rioja, Maria J., *Advanced Topic in Astrophysics, Lecture 3* [Diapositives]. Récupéré le 20 mai 2019, de `https://www.web.uwa.edu.au/__data/assets/pdf_file/0008/791126/rioja-icrar-lecture3_small.pdf`.

[2] Smirnov, O., *The radio interferometric data challenge : From MeerKAT towards the SKA* [Diapositives]. Récupéré le 20 mai 2019, de `http://sites.ieee.org/sips2018/files/2018/11/meerkat.pdf`.

[3] Giovannelli, J.-F., La méthode de Wiener-Hunt en déconvolution de signaux et d'images.

[4] Gary, D. E., *Physics 738, Radio Astronomy : Lecture #6*. Accédé le 21 mai 2019, en `https://web.njit.edu/~gary/728/Lecture6.html`.

[5] Dask Development Team, 2016, Dask : Library for dynamic task scheduling. Accédé le 22 mai 2019, en `https://dask.org`.

[6] Jupyter Notebook, Cours *Fundamentals of Radio Interferometry* du master NASSP. Accédé le 23 mai 2019, en : `https://github.com/ratt-ru/fundamentals_of_interferometry`.

[7] Jupyter Notebook, Dask Tutorial. Accédé le 24 mai 2019, en : `https://github.com/dask/dask-tutorial`.

[8] Site, Site personnel Matthew Rocklin. Accédé le 26 mai 2019, en : `https://matthewrocklin.com/`.

[9] Site, Site du projet SKA. Accédé le 20 mai 2019, en : `https://astronomers.skatelescope.org/`.

[10] Site, Livre blanc SKA. Récupéré le 29 Juillet 2019, de `https://arxiv.org/pdf/1712.06950.pdf`

[11] Github, DDFACET program. Récupéré le 26 Aôut 2019, de `https://github.com/saopicc/DDFacet`

[12] Site, Moulon Mesocentre, documentation of the cluster Fusion Accédé le 10 Spetembre 2019, en `http://mesocentre.centralesupelec.fr/`