Tutorial Dask Python 3

Introducción a Dask

Dask es una librería flexible para computación paralela en Python. Está compuesta por dos partes:

- Planificación dinámica de tareas optimizada para el cómputo. Esto es similar a Airflow, Luigi, Celery o Make, pero está optimizado para cargas de trabajo computacionales interactivas.
- Las colecciones de "Big Data" como matrices paralelas, data-frames y listas que extienden las interfaces comunes como NumPy, Pandas o iteradores de Python a entornos de memoria distribuidos. Estas colecciones paralelas se ejecutan sobre el planificador(scheduler) de tareas dinámicas. [1]

La arquitectura del entorno distribuido se muestra en la siguiente imagen:

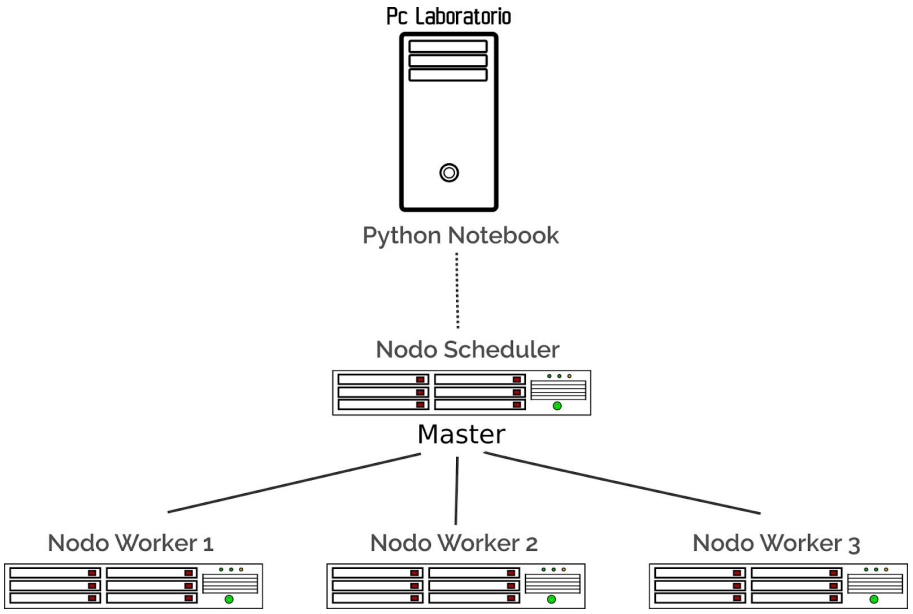

Figura 1: Arquitectura Dask

La instalación de Dask provee de las aplicaciones dask-scheduler y dask-worker, que se ejecutarán en los componentes de la arquitectura que se ven en la Figura 1.
Mediante la librería dask.distributed, puede accederse de forma local o remota al planificador.

Dask representa los cálculos paralelos con grafos de tareas (ver Figura 2). Estos grafos acíclicos dirigidos pueden tener una estructura arbitraria, lo que permite tanto a los desarrolladores como a los usuarios la libertad de construir algoritmos sofisticados y de manejar situaciones complicadas que no son fáciles de manejar por el paradigma de map/filter/groupby, común en la mayoría de los frameworks de ingeniería de datos.

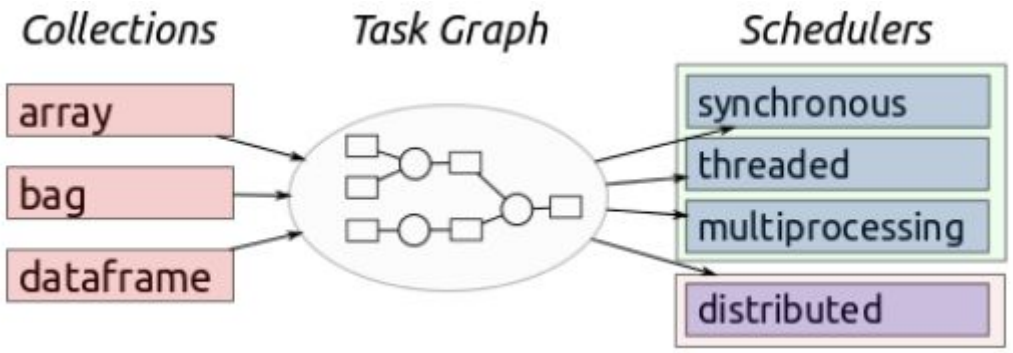

Figura 2. Representación de tareas en Dask

Requerimientos e instalación

Para obtener una instalación limpia, se propone instalar Dask en un entorno virtual, tanto en el master del cluster como en la máquina donde se ejecutarán los programas.

A continuación, se muestran los pasos para instalar un entorno virtual con la herramienta **virtualenv:**
Instalar virtualenv:

$ apt install virtualenv --user

Crear entorno virtual con python3

$ virtualenv -p python3 nombreEntorno

Ingresar al entorno virtual

$ source nombreEntorno/bin/activate

Instalar Dask en el entorno virtual

$ pip install "dask[complete]"

Crear la red del cluster

Como se mencionó anteriormente, para crear la red del cluster, se deben ejecutar las aplicaciones dask-scheduler y dask-worker, en el planificador y los nodos respectivamente.
Si el cluster posee un sistema de archivos remoto como storage, el master y los nodos pueden usar la instalación del entorno virtual en el storage, para ejecutar las aplicaciones necesarias para crear la red del cluster.

A continuación se muestran los pasos para crear la red del cluster:

En el master, con el entorno virtual activado ejecutar:
$ dask-scheduler
Scheduler at: tcp:/*address-scheduler*:8786

En los nodos, con el entorno virtual activado ejecutar:
$ dask-worker address-scheduler:8786
Start worker at: tcp://address-worker:12345
Registered to: tcp://address-scheduler:8786

Instalación con contenedores Docker

Dask también provee imágenes docker, con contenedores que poseen las herramientas para Dask preinstaladas. Las imágenes están disponibles en los siguientes enlaces:

- Imagen Docker con dask-distributed. https://hub.docker.com/r/daskdev/dask
- Imagen Docker con JupyterLab: https://hub.docker.com/r/daskdev/dask-notebook

Para crear los contenedores, seguir los pasos que se encuentran en los repositorios de Docker.

Crear la red del cluster con contenedores Docker

Dar de alta el docker del máster con el comando:
$ docker run -it -d --network host daskdev/dask dask-scheduler

Dar de alta los nodos con el comando: (nodos 10 11 y 12)
$ docker run -d --network host daskdev/dask dask-worker address-scheduler:8786

Correr Jupyter notebook en la máquina local con el comando:
$ docker run -e GRANT_SUDO="yes" --user root -it --network host daskdev/dask-notebook

Desde el browser ir a **localhost:8888** e ingresar el token para loguearse.

Comandos útiles para el manejo de contenedores Docker

- Listar docker's corriendo
  docker ps
  -q (Muestra los detalles con id)
  -a (Muestra los nombres)
- Detener todos los docker
  docker stop $(docker ps -a -q)
- Eliminar los docker una vez detenidos para que no sigan guardados.
  docker rm $(docker ps -a -q)

Ejemplos de uso

Para comenzar con los primeros ejemplos de uso de la librería, se creó el repositorio disponible en https://github.com/MarioQuiroga/learning-dask-python, con distintas formas de usar la librería.
Para ejecutar los ejemplos, es necesario instalar las dependencias que se encuentran en el archivo requeriments.txt, para ello es necesario:

- Activar el entorno virtual en su máquina:
  $ cd nameEnv
  $ source bin/activate
- Clonar el repositorio con ejemplos:
  $ git clone https://github.com/MarioQuiroga/learning-dask-python.git
- Instalar las dependencias con pip:
  $ cd learning-dask-python
  $ pip install -r requeriments.txt

Los ejemplos funcionarán cuando el cluster esté activo y el planificador sea alcanzable por la red, desde su máquina local. Por lo tanto, debe verificar en cada uno de los ejemplos del repositorio, que la dirección IP al planificador del cluster sea la correcta.

Delayed

A continuación, se muestran ejemplos de cómo paralelizar una sentencia for con delayed:

```
def process(data):
    ...
    return ...
```

Normal Sequential Processing:

```
results = [process(x) for x in inputs]
```

Build Dask Computation:

```
from dask import compute, delayed
values = [delayed(process)(x) for x in inputs]
```

Multiple Threads:

```
import dask.threaded
results = compute(*values, scheduler='threads')
```

Multiple Processes:

```
import dask.multiprocessing
results = compute(*values, scheduler='processes')
```

Distributed Cluster:

```
from dask.distributed import Client
client = Client("cluster-address:8786")
results = compute(*values, scheduler='distributed')
```

Figura 3. Ejemplos de sentencia for con delayed

Futures

Dask soporta tareas en tiempo real que extiende la interfaz de Python concurrent.futures. Esta interfaz es buena para la programación de tareas arbitrarias como dask.delayed , pero es inmediata en lugar de perezosa, lo que proporciona un poco más de flexibilidad en situaciones en las que los cálculos pueden evolucionar con el tiempo.

Una vez instanciado el cliente del módulo distributed, como se mostró en los ejemplos anteriores, es posible utilizar las siguientes funciones para enviar tareas al planificador:

- Enviar una aplicación de función al planificador, devuelve un objeto future
  `Client.submit`(func, * args, ** kwargs)
- Mapear una función en una secuencia de argumentos, devuelve una lista de future
  `Client.map`(func, * iterables, ** kwargs)
- Esperar hasta que se complete el cálculo, reúna el resultado en el proceso local.
  `Future.result`([timeout])
- Obtener el resultado de una lista de futures
  `Client.gather`(futures [, errors, maxsize,…])

Ejemplos de uso con colecciones

Dask contiene largas colecciones paralelas para arrays n-dimensional (similar a NumPy), DataFrames (similar a Pandas), y listas (similar a PyToolz o PySpark).

Dask Array

El módulo dask.array coordina muchas matrices Numpy, organizadas en trozos (chunks) dentro de una cuadrícula. Soportan un gran subconjunto de la API de Numpy.

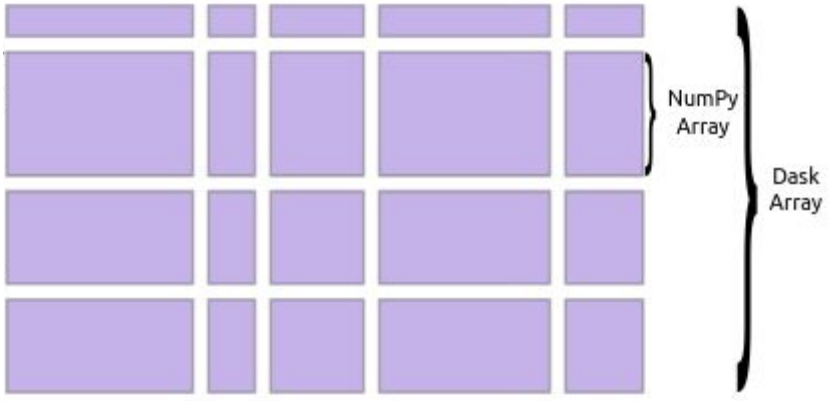

Figura 4. Estructura de datos Dask Array

Por ejemplo, la siguiente porción de código[2] crea una matriz de 10000x10000 de números aleatorios, representados como muchas matrices numpy de tamaño 1000x1000 (o más pequeñas si la matriz no se puede dividir en partes iguales). En este caso hay 100 (10x10) matrices numpy de tamaño 1000x1000.

```
import dask.array as da
x = da.random.random((10000, 10000), chunks=(1000, 1000))
```

Luego, se usa la sintaxis usual de NumPy:

```
y = x + x.T
z = y[::2, 5000:].mean(axis=1)
```

Dask Data Frame

Similar a dask.array, dask.dataframe coordina muchos dataframe de Pandas, divididos a lo largo de un índice. Soportan un gran subconjunto de la API de Pandas.

Por ejemplo[3], creamos un simple dataframe provisto por la librería dask.dataframe:

```
import dask
import dask.dataframe as dd
df = dask.datasets.timeseries()
```

Luego, se utilizan las operaciones estándar de Pandas Dataframe:

```
df.head(3)
df2 = df[df.y > 0]
df3 = df2.groupby('name').x.std()
computed_df = df3.compute()
```

Dask Bags

Dask Bags implementa operaciones como **map, filter, groupby** y agregaciones de colecciones de objetos de Python. Lo hace en paralelo y en pequeña memoria usando iteradores de Python. Es similar a una versión paralela de itertools o una versión Pythonic del PySpark RDD.
Dask Bags a menudo se usa para realizar un preprocesamiento simple en archivos de registro, registros JSON u otros objetos de Python definidos por el usuario.

Creamos un conjunto aleatorio de datos de registro y los almacenamos en el disco como muchos archivos JSON[4].

```
import dask
import json
import os

os.makedirs('data', exist_ok=True)                # Create data/ directory
b = dask.datasets.make_people()                   # Make records of people
b.map(json.dumps).to_textfiles('data/*.json')     # Encode as JSON, write to disk
```

Leemos los datos JSON generados en el paso anterior:

```
import dask.bag as db
import json

b = db.read_text('data/*.json').map(json.loads)
print(b.take(2)) # Muestra dos elementos dentro del JSON
```

Podemos procesar estos datos filtrando solo ciertos registros de interés, asignando funciones sobre ellos para procesar nuestros datos y agregando esos resultados a un valor total.

```
print(b.filter(lambda record: record['age'] > 30).take(2))   # Select only people over 30
b.map(lambda record: record['occupation']).take(2)  # Select the occupation field
b.count().compute()  # Count total number of records
```

Es común hacer todos estos pasos en un pipeline, com por ejemplo:

```
result = (b.filter(lambda record: record['age'] > 30)
           .map(lambda record: record['occupation'])
           .frequencies(sort=True)
           .topk(10, key=1))
print(result.compute())
```

Persistir datos en la memoria

Si está en un cluster, puede conservar los datos en la memoria distribuida . Esto permite que los futuros cálculos sean mucho más rápidos. Luego de haber creado la variable, con algunos de los ejemplos de colección vistos, usar las siguientes líneas:

```
nombre_variable = nombre_variable.persist()
print(nombre_variable.compute())
```

Ejemplos de uso con Scikit-Learn

Dask-ML proporciona aprendizaje automático escalable en Python utilizando Dask junto con bibliotecas de aprendizaje automático populares como Scikit-Learn .

Scikit-Learn ya proporciona computación en paralelo en una sola máquina con Joblib . Dask extiende este paralelismo a muchas máquinas en un clúster. Esto funciona bien para tamaños de datos modestos, pero grandes cálculos, como random forest, optimización de hiper-parámetros y más[5].

```
from dask.distributed import Client
from sklearn.externals.joblib import parallel_backend
client = Client('address-cluster:8786')  # Connect to a Dask Cluster

with parallel_backend('dask'):
    # Your normal scikit-learn code here
```

Asociarse con otras bibliotecas distribuidas

Otras bibliotecas de aprendizaje automático como XGBoost y TensorFlow ya tienen soluciones distribuidas que funcionan bastante bien. Dask-ML no intenta reimplementar estos sistemas. En cambio, Dask-ML facilita el uso de los flujos de trabajo normales de Dask para preparar y configurar los datos, luego implementa XGBoost o Tensorflow junto con Dask y entrega los datos.

```
from dask_ml.xgboost import XGBRegressor
est = XGBRegressor(...)
est.fit(train, train_labels)
```

Consulte la documentación de Dask-ML + XGBoost o Dask-ML + TensorFlow para obtener más información.

Pruebas de Escalabilidad

Se hacen pruebas del framework modificando la cantidad de nodos y núcleos por nodo del cluster. Para el deployment se crean dos scripts **init_cluster.sh y kill_cluster.sh**, los cuales pueden usarse mediante funciones en python disponibles en el archivo scale_cluster.py.

Multiplicación de matrices

Se hacen pruebas con la función matmult para Dask Arrays que ofrece Dask, para la multiplicación de matrices. Las pruebas se encuentran en: . Donde puede verse que la función matmult no escala mientras se modifican la cantidad de nodos y la cantidad de núcleos por nodo en el cluster.

Estimación de pi por el método Montecarlo

Se implementó un ejemplo de estimación del número pi por el método Montecarlo distribuido, y se utilizan la funciones init_cluster y kill_cluster. Como el algoritmo es implementado con los decoradores delayed. Por lo tanto el planificador hace bien el trabajo y la función escala correctamente.

Todos los scripts pueden encontrarse en:
https://github.com/MarioQuiroga/learning-dask-python/tree/master/scaling_tests

Bibliografía

- [1] https://buildmedia.readthedocs.org/media/pdf/dask/latest/dask.pdf (Documentación oficial de la librería)
- [2] https://examples.dask.org/array.html
- [3] https://examples.dask.org/dataframe.html
- [4] https://examples.dask.org/bag.html
- [5] https://ml.dask.org/#parallelize-scikit-learn-directly
- Link oficial al repositorio git-hub: https://github.com/dask