```
import pandas as pd
import numpy as np
import re

f = open("/kaggle/input/big-data-l2/large-txt.txt", "r",
encoding="latin")
text = f.read()

len(text)
```

```
223281915
```

Mono processeur

On essaye en premier de savoir le temps pour un programme classique de comptage de mot en mono processeur

```
import time

def calculer_occurence(text):
  start_time = time.time()

  # Nettoyage
  t = re.sub(r'[^\w\s]', '', text).lower()
  words = t.split()

  # Comptage
  word_counts = {}
  for word in words:
      if word in word_counts:
          word_counts[word] += 1
      else:
          word_counts[word] = 1

  tps = time.time() - start_time
  print("\n--- %s seconds ---" % tps)
  return tps

from tqdm import tqdm
temps_moyens = []

for i in tqdm(range(10)):
  temps = calculer_occurence(text)
  temps_moyens.append(temps)
```

```
 10%|▉         | 1/10 [00:20<03:03, 20.38s/it]


--- 19.620394945144653 seconds ---
```

```
 20%|██        | 2/10 [00:40<02:40, 20.07s/it]

--- 19.071309328079224 seconds ---
 30%|███       | 3/10 [01:00<02:20, 20.08s/it]

--- 19.331718921661377 seconds ---
 40%|████      | 4/10 [01:20<02:00, 20.03s/it]

--- 19.195183515548706 seconds ---
 50%|█████     | 5/10 [01:40<01:39, 19.96s/it]

--- 19.087004899978638 seconds ---
 60%|██████    | 6/10 [01:59<01:19, 19.90s/it]

--- 19.016866445541382 seconds ---
 70%|███████   | 7/10 [02:19<00:59, 19.83s/it]

--- 18.91770601272583 seconds ---
 80%|████████  | 8/10 [02:39<00:39, 19.80s/it]

--- 18.99219512939453 seconds ---
 90%|█████████ | 9/10 [02:59<00:19, 19.81s/it]

--- 19.04788565635681 seconds ---
100%|██████████| 10/10 [03:18<00:00, 19.90s/it]

--- 19.075124263763428 seconds ---


print(f'Temps moyen (10 itérations) en seconde :
{np.mean(temps_moyens):.2f}')

Temps moyen (10 itérations) en seconde : 19.14
```

Multi-processeurs

```
import multiprocessing
# Nombre de coeur disponible
multiprocessing.cpu_count()
```

```
4
```

Le code implémente un processus de comptage de mots à l'aide du paradigme de MapReduce. Ce paradigme est utilisé pour traiter des données volumineux de manière distribuée.

La fonction Map a pour objectif de traiter un segment de texte (chunk) pour en extraire les mots et compter leur fréquence d'apparition. En premier on utilise re et lower() pour supprimer tous les caractères non alphanumériques et on met le texte en minuscules pour tout normaliser. Ensuite on découpe le texte pour isoler chaque mot individuellement. On parcourt chaque mot de la liste et on enregistre la fréquence de chaque mot. Si le mot est déjà présent, sa valeur est incrémentée de 1, sinon on créer une nouvelle entrée avec comme valeur initiale 1.

La fonction Reduce combine les résultats intermédiaires générés par plusieurs appels à la fonction Map. En premier on créer un dictionnaire vide, final_counts, pour stocker le résultat final. Ensuite on a une boucle qui parcourt une liste de dictionnaires, chaque dictionnaire représente les résultats d'un appel précédent à la fonction map. Une seconde boucle parcourt les paires clé-valeur de chaque dictionnaire, elle additionne les fréquences de chaque mot. La fonction retourne un dictionnaire combinant les fréquences totales de chaque mot.

```
# Fonction de MAP
def map_function(text_chunk):
    # Nettoyage
    t = re.sub(r'[^\w\s]', '', text_chunk).lower()
    words = t.split()

    # Création des paires clé-valeur
    word_counts = {}
    for word in words:
        word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

# Fonction de REDUCE
def reduce_function(mapped_values):
    final_counts = {}
    for counts in mapped_values:
        for word, count in counts.items():
            final_counts[word] = final_counts.get(word, 0) + count
    return final_counts

# MAIN
def main(text, num_processes=multiprocessing.cpu_count()):
    start_time = time.time()

    # Séparation selon le nombre de coeur disponible
```

```
    chunk_size = len(text) // num_processes
    # Création des chunk
    chunks = [text[i:i + chunk_size] for i in range(0, len(text),
chunk_size)]

    # Pour chaque coeur, faire le map
    with multiprocessing.Pool(processes=num_processes) as pool:
        mapped_results = pool.map(map_function, chunks)

    # Appel de REDUCE sur les map
    final_counts = reduce_function(mapped_results)

    tps = time.time() - start_time
    print("\n--- %s seconds ---" % tps)
    return tps, final_counts
```

On a donc 2 étapes :

- Map Chaque portion de texte est traitée individuellement avec la fonction map_function, qui produit des dictionnaires de fréquences pour les mots trouvés dans cette portion.
- Reduce Les résultats des étapes Map sont agrégés à l'aide de la fonction reduce_function pour obtenir un décompte global des mots.

Pour finir, la fonction main combine l'étape MapReduce avec la parallélisation avec la librairie multiprocessing. Cela permet d'accélérer le traitement lorsque le texte est volumineux. La taille de chaque segment est calculée en divisant la longueur totale du texte par le nombre de coeur alloué. Ensuite, le texte est découpé en chunks. Chaque chunk est une portion du texte que chaque processus traitera indépendamment. On met ensuite en oeuvre la parallélisation avec un pool. Le pool crée plusieurs processus en parallèle pour traiter les chunks. Chaque processus applique la fonction map_function sur un chunk, produisant un dictionnaire de fréquences de mots pour ce chunk. Une fois que tous les processus ont terminé leur travail, les résultats sont combinés à l'aide de la fonction Reduce.

```
temps_moyens_multi = []
coeurs = [1,2,3,4]
for c in coeurs:
    temps_moyens_coeurs = []
    print('-'*10)
    print(f'Nb Coeur : {c}')
    for i in tqdm(range(10)):
        temps, _ = main(text,c)
        temps_moyens_coeurs.append(temps)
    temps_moyens_multi.append(temps_moyens_coeurs)
    print(f'Temps moyen (10 itérations) en seconde :
{np.mean(temps_moyens_coeurs):.2f}')
```

```
----------
Nb Coeur : 1
 10%|          | 1/10 [00:21<03:14, 21.65s/it]
```

```
--- 21.633089780807495 seconds ---
 20%|██        | 2/10 [00:42<02:48, 21.09s/it]

--- 20.662687301635742 seconds ---
 30%|███       | 3/10 [01:03<02:27, 21.05s/it]

--- 20.958998203277588 seconds ---
 40%|████      | 4/10 [01:24<02:06, 21.09s/it]

--- 21.119631052017212 seconds ---
 50%|█████     | 5/10 [01:45<01:45, 21.11s/it]

--- 21.107937812805176 seconds ---
 60%|██████    | 6/10 [02:06<01:24, 21.10s/it]

--- 21.063379049301147 seconds ---
 70%|███████   | 7/10 [02:27<01:03, 21.08s/it]

--- 20.98696804046631 seconds ---
 80%|████████  | 8/10 [02:48<00:41, 20.94s/it]

--- 20.62382745742798 seconds ---
 90%|█████████ | 9/10 [03:09<00:20, 21.00s/it]

--- 21.079743146896362 seconds ---
100%|██████████| 10/10 [03:30<00:00, 21.03s/it]

--- 20.699782371520996 seconds ---
Temps moyen (10 itérations) en seconde : 20.99
----------
Nb Coeur : 2
 10%|█         | 1/10 [00:12<01:53, 12.59s/it]

--- 12.505043029785156 seconds ---
```

```
 20%|██        | 2/10 [00:25<01:42, 12.75s/it]

--- 12.780176401138306 seconds ---
 30%|███       | 3/10 [00:37<01:28, 12.65s/it]

--- 12.445419549942017 seconds ---
 40%|████      | 4/10 [00:50<01:16, 12.69s/it]

--- 12.673441648483276 seconds ---
 50%|█████     | 5/10 [01:03<01:04, 12.90s/it]

--- 13.175751209259033 seconds ---
 60%|██████    | 6/10 [01:16<00:51, 12.84s/it]

--- 12.658288955688477 seconds ---
 70%|███████   | 7/10 [01:29<00:38, 12.80s/it]

--- 12.609986066818237 seconds ---
 80%|████████  | 8/10 [01:42<00:25, 12.74s/it]

--- 12.544441938400269 seconds ---
 90%|█████████ | 9/10 [01:54<00:12, 12.65s/it]

--- 12.35572624206543 seconds ---
100%|██████████| 10/10 [02:07<00:00, 12.79s/it]

--- 13.330689430236816 seconds ---
Temps moyen (10 itérations) en seconde : 12.71
----------
Nb Coeur : 3
 10%|█         | 1/10 [00:12<01:48, 12.07s/it]

--- 11.974326133728027 seconds ---
 20%|██        | 2/10 [00:23<01:34, 11.84s/it]
```

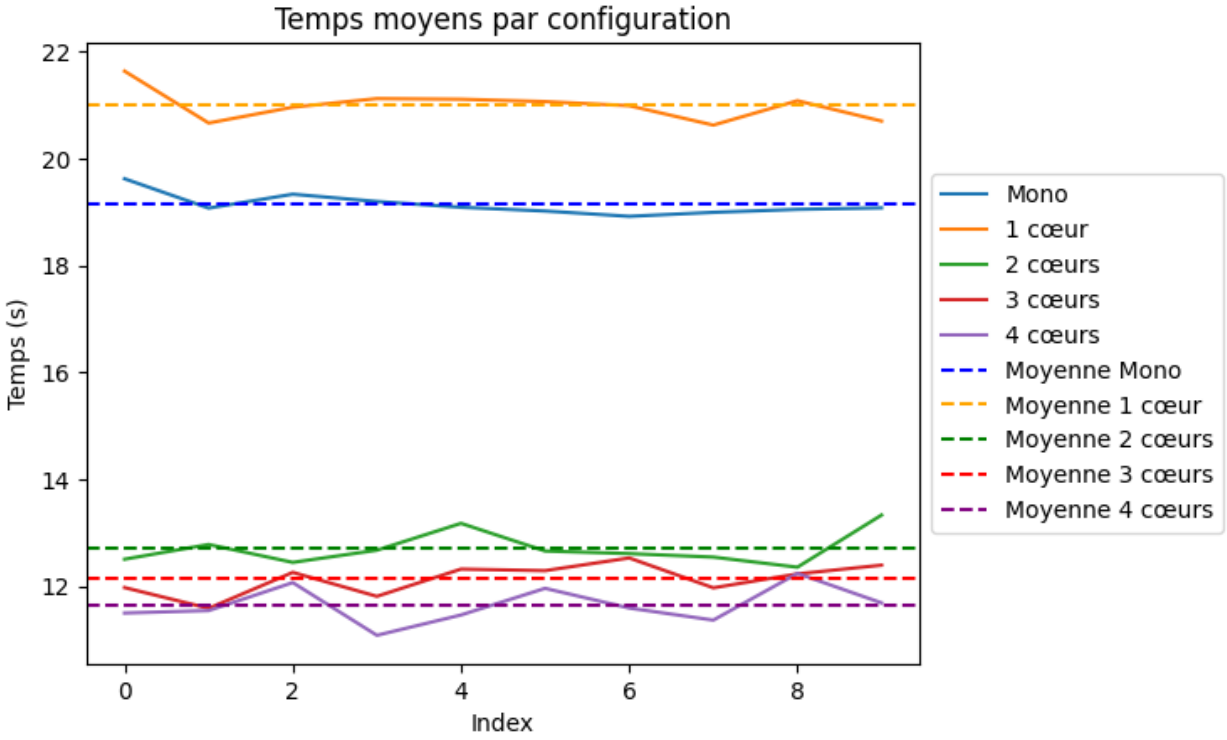
```
--- 11.586336612701416 seconds ---
 30%|███       | 3/10 [00:36<01:24, 12.08s/it]

--- 12.258901119232178 seconds ---
 40%|████      | 4/10 [00:48<01:12, 12.01s/it]

--- 11.810555934906006 seconds ---
 50%|█████     | 5/10 [01:00<01:00, 12.16s/it]

--- 12.319989442825317 seconds ---
 60%|██████    | 6/10 [01:12<00:48, 12.24s/it]

--- 12.291039943695068 seconds ---
 70%|███████   | 7/10 [01:25<00:37, 12.37s/it]

--- 12.52835202217102 seconds ---
 80%|████████  | 8/10 [01:37<00:24, 12.27s/it]

--- 11.969957828521729 seconds ---
 90%|█████████ | 9/10 [01:49<00:12, 12.29s/it]

--- 12.229958295822144 seconds ---
100%|██████████| 10/10 [02:02<00:00, 12.24s/it]

--- 12.393692970275879 seconds ---
Temps moyen (10 itérations) en seconde : 12.14
----------
Nb Coeur : 4
 10%|          | 1/10 [00:11<01:44, 11.60s/it]

--- 11.498280763626099 seconds ---
 20%|          | 2/10 [00:23<01:33, 11.63s/it]

--- 11.544209718704224 seconds ---
```

```
 30%|███       | 3/10 [00:35<01:23, 11.88s/it]

--- 12.0678071975708 seconds ---
 40%|████      | 4/10 [00:46<01:09, 11.61s/it]

--- 11.08136534690857 seconds ---
 50%|█████     | 5/10 [00:58<00:57, 11.59s/it]

--- 11.462043046951294 seconds ---
 60%|██████    | 6/10 [01:10<00:47, 11.75s/it]

--- 11.960410356521606 seconds ---
 70%|███████   | 7/10 [01:21<00:35, 11.74s/it]

--- 11.59125566482544 seconds ---
 80%|████████  | 8/10 [01:33<00:23, 11.66s/it]

--- 11.365809202194214 seconds ---
 90%|█████████ | 9/10 [01:45<00:11, 11.87s/it]

--- 12.239530086517334 seconds ---
100%|██████████| 10/10 [01:57<00:00, 11.76s/it]

--- 11.688145399093628 seconds ---
Temps moyen (10 itérations) en seconde : 11.65
```

Visualization

```
import matplotlib.pyplot as plt

plt.plot(temps_moyens, label="Mono")
plt.plot(temps_moyens_multi[0], label="1 cœur")
plt.plot(temps_moyens_multi[1], label="2 cœurs")
plt.plot(temps_moyens_multi[2], label="3 cœurs")
plt.plot(temps_moyens_multi[3], label="4 cœurs")
```

```
plt.axhline(y=np.mean(temps_moyens), color='blue', linestyle='--',
label="Moyenne Mono")
plt.axhline(y=np.mean(temps_moyens_multi[0]), color='orange',
linestyle='--', label="Moyenne 1 cœur")
plt.axhline(y=np.mean(temps_moyens_multi[1]), color='green',
linestyle='--', label="Moyenne 2 cœurs")
plt.axhline(y=np.mean(temps_moyens_multi[2]), color='red',
linestyle='--', label="Moyenne 3 cœurs")
plt.axhline(y=np.mean(temps_moyens_multi[3]), color='purple',
linestyle='--', label="Moyenne 4 cœurs")

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Temps moyens par configuration")
plt.xlabel("Index")
plt.ylabel("Temps (s)")

plt.show()
```



Mono en multi

Cette fois-ci, on teste le code du mono processeur (sans le Map Reduce) en multi processeur

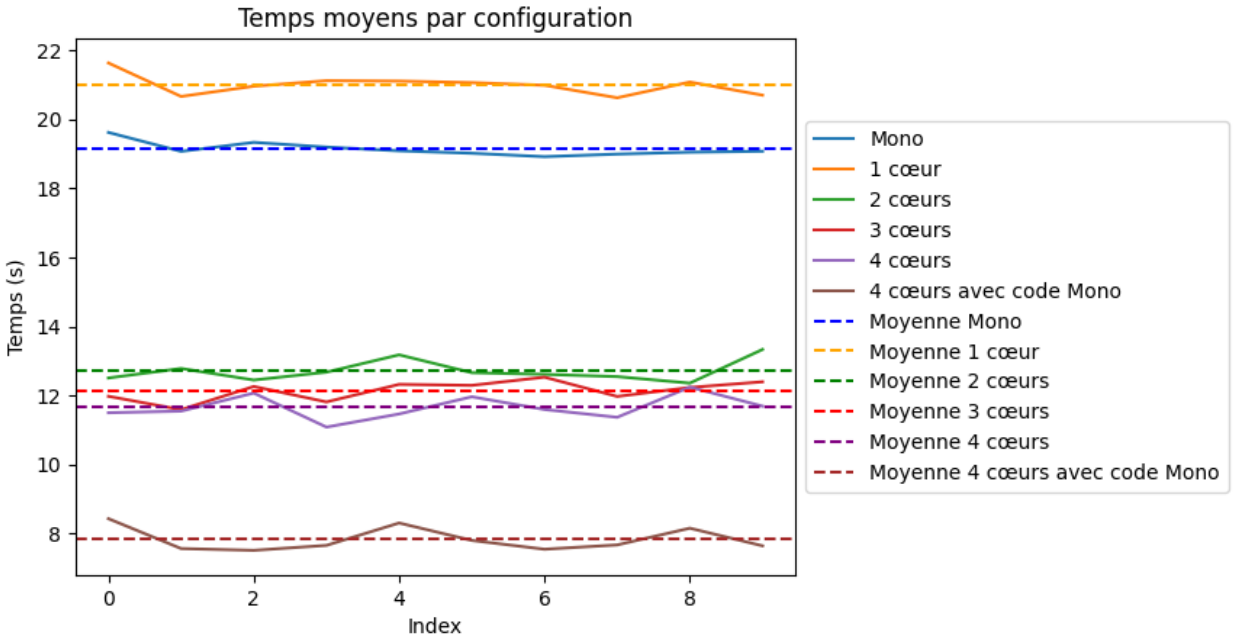
```python
def calculer_occurence_2(text):
  # Split
  words = text.split()

  # Comptage
  word_counts = {}
  for word in words:
      if word in word_counts:
          word_counts[word] += 1
      else:
          word_counts[word] = 1

  return word_counts

def main_2(text, num_processes=multiprocessing.cpu_count()):
    start_time = time.time()

    # Séparation selon le nombre de coeur disponible
    chunk_size = len(text) // num_processes
    # Création des chunk
    chunks = [text[i:i + chunk_size] for i in range(0, len(text),
chunk_size)]

    # Pour chaque coeur, faire le map
    with multiprocessing.Pool(processes=num_processes) as pool:
        results = pool.map(calculer_occurence_2, chunks)

    # Appel de REDUCE sur les map
    final_counts = results

    tps = time.time() - start_time
    print("\n--- %s seconds ---" % tps)
    return tps, final_counts

temps_moyens_coeurs_2 = []
print('-'*10)
print(f'Nb Coeur : 4')
for i in tqdm(range(10)):
    temps, _ = main_2(text,c)
    temps_moyens_coeurs_2.append(temps)

print(f'Temps moyen (10 itérations) en seconde :
{np.mean(temps_moyens_coeurs_2):.2f}')
```

```
----------
Nb Coeur : 4

 10%|          | 1/10 [00:08<01:16,  8.51s/it]


--- 8.42768120765686 seconds ---
```

```
 20%|██        | 2/10 [00:16<01:04,  8.02s/it]

--- 7.561983346939087 seconds ---
 30%|███       | 3/10 [00:23<00:54,  7.83s/it]

--- 7.510143280029297 seconds ---
 40%|████      | 4/10 [00:31<00:46,  7.81s/it]

--- 7.6540021896362305 seconds ---
 50%|█████     | 5/10 [00:39<00:40,  8.03s/it]

--- 8.302795886993408 seconds ---
 60%|██████    | 6/10 [00:47<00:31,  7.99s/it]

--- 7.79752254486084 seconds ---
 70%|███████   | 7/10 [00:55<00:23,  7.88s/it]

--- 7.542543172836304 seconds ---
 80%|████████  | 8/10 [01:03<00:15,  7.85s/it]

--- 7.667322874069214 seconds ---
 90%|█████████ | 9/10 [01:11<00:07,  7.98s/it]

--- 8.151808500289917 seconds ---
100%|██████████| 10/10 [01:19<00:00,  7.93s/it]

--- 7.641369342803955 seconds ---
Temps moyen (10 itérations) en seconde : 7.83
```

```python
import matplotlib.pyplot as plt

plt.plot(temps_moyens, label="Mono")
plt.plot(temps_moyens_multi[0], label="1 cœur")
plt.plot(temps_moyens_multi[1], label="2 cœurs")
plt.plot(temps_moyens_multi[2], label="3 cœurs")
```

```
plt.plot(temps_moyens_multi[3], label="4 cœurs")
plt.plot(temps_moyens_coeurs_2, label="4 cœurs avec code Mono")


plt.axhline(y=np.mean(temps_moyens), color='blue', linestyle='--',
label="Moyenne Mono")
plt.axhline(y=np.mean(temps_moyens_multi[0]), color='orange',
linestyle='--', label="Moyenne 1 cœur")
plt.axhline(y=np.mean(temps_moyens_multi[1]), color='green',
linestyle='--', label="Moyenne 2 cœurs")
plt.axhline(y=np.mean(temps_moyens_multi[2]), color='red',
linestyle='--', label="Moyenne 3 cœurs")
plt.axhline(y=np.mean(temps_moyens_multi[3]), color='purple',
linestyle='--', label="Moyenne 4 cœurs")
plt.axhline(y=np.mean(temps_moyens_coeurs_2), color='brown',
linestyle='--', label="Moyenne 4 cœurs avec code Mono")


plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Temps moyens par configuration")
plt.xlabel("Index")
plt.ylabel("Temps (s)")

plt.show()
```



Le plus rapide est le code du mono processeur avec 4 coeurs. Cela s'explique par la simplicité de la tâche, les fonctions map-reduce font plus d'appel et donc cela prend plus de temps. Avec encore plus de données ou avec une tâche plus complexe, map-reduce fonctionnera mieux. Je n'ai pas pu tester avec plus de données car cela prenait trop de place en mémoire et la boucle ne pouvait pas finir de tous calculer.