



UNIVERSITE CHEIKH ANTA DIOP DE DAKAR



ECOLE SUPERIEURE
POLYTECHNIQUE

Département génie informatique

Rapport : Programme séquentiel Word Count et analyse du temps d'exécution

Groupe 1 :

Anna Ndoye

Mame Diarra Mbacké

Khadim Mbaye

Thierno Abdoulaye Sall

Youssou Gnaga Diatta

Serigne Mame Sarr

Professeur :

Mouhamadou Lamine Ba

Lien vers le github du tp :

[https://github.com/AnnaNdoye/TP1--Introduction-au-Big-
Data.git](https://github.com/AnnaNdoye/TP1--Introduction-au-Big-Data.git)

Introduction

Ce rapport présente l'implémentation d'un programme séquentiel de comptage de mots (Word Count) en Python, ainsi qu'une analyse expérimentale de son temps d'exécution en fonction de la taille de l'entrée. L'objectif est de mesurer les performances pour des tailles d'entrée variant de 1 MB à 50 MB, avec 5 répétitions par taille, et d'analyser la complexité empirique.

1. Algorithme et complexité

Description de l'algorithme

L'algorithme implémenté est un compteur de mots séquentiel simple :

1. Ouvrir le fichier texte en mode lecture.
2. Lire le fichier ligne par ligne.
3. Pour chaque ligne, diviser en mots en utilisant split() (séparation par espaces).
4. Utiliser un dictionnaire (collections.Counter) pour compter la fréquence de chaque mot.
5. Retourner le dictionnaire des fréquences.

Le code est le suivant :

```
import collections
```

```
def word_count(filename):  
    word_freq = collections.Counter()  
    with open(filename, 'r') as f:  
        for line in f:  
            words = line.strip().split()  
            word_freq.update(words)  
    return word_freq
```

Complexité théorique

- **Temps** : $O(n)$, où n est le nombre total de mots dans le fichier. Chaque mot est traité une fois pour le comptage.
- **Espace** : $O(m)$, où m est le nombre de mots uniques. Le dictionnaire stocke une entrée par mot unique.

L'algorithme est séquentiel et ne tire pas parti du parallélisme.

Méthodologie expérimentale

Génération des données

Les fichiers de test ont été générés à l'aide du script gen_corpus.py, qui crée des textes aléatoires composés de mots de 5 lettres en minuscules. Les tailles sont : 1 MB, 5 MB, 10 MB, 20 MB, 50 MB.

Mesures de performance

Le script run_bench.py exécute le comptage de mots 5 fois par taille d'entrée et mesure le temps d'exécution à l'aide de `time.time()`. Les résultats sont sauvegardés dans `results.csv` avec les colonnes : Taille_MB, Temps1, Temps2, Temps3, Temps4, Temps5, Moyenne.

Résultats

Les temps moyens d'exécution sont :

- 1 MB : 0.0635 s
- 5 MB : 0.4691 s
- 10 MB : 0.8518 s
- 20 MB : 2.2190 s
- 50 MB : 6.0870 s

Le graphique ci-dessous montre la relation entre la taille d'entrée et le temps moyen :

2. Analyse des résultats

Complexité empirique

Le graphique montre une relation approximativement linéaire entre la taille d'entrée et le temps d'exécution. En effet, le temps augmente proportionnellement à la taille du fichier, ce qui confirme la complexité $O(n)$ théorique. Pour une taille doublée, le temps est multiplié par environ 2-3, avec une légère variation due aux fluctuations système.

3. La complexité est-elle linéaire ?

Oui, la complexité est linéaire. Les points sur le graphique s'alignent sur une droite, indiquant que le temps d'exécution évolue linéairement avec la taille de l'entrée.

4. Sources d'erreurs expérimentales

Plusieurs sources d'erreurs peuvent affecter les mesures :

- **Fluctuations système** : Autres processus en cours (antivirus, mises à jour) peuvent interférer.
- **Cache disque** : Les lectures répétées peuvent bénéficier du cache, rendant les mesures non représentatives.

- **Précision du chronomètre** : `time.time()` a une résolution limitée (environ 1 ms), ce qui peut introduire du bruit pour les petits fichiers.
- **Génération de données** : Les textes aléatoires ne reflètent pas toujours des distributions réelles de mots, mais cela n'affecte pas la complexité.
- **Variabilité matérielle** : Température du CPU, fragmentation disque, etc.

Pour minimiser ces erreurs, les mesures ont été répétées 5 fois et moyennées.

5. Traitement de 1 TB

Pour traiter 1 TB de données avec cet algorithme séquentiel, le temps estimé serait d'environ 120 000 secondes (environ 33 heures), basé sur l'extrapolation linéaire (50 MB ~ 6 s, donc 1 TB = 1 000 000 MB ~ 120 000 s). Cependant, cela n'est pas pratique.

Solutions :

- **Parallélisation** : Diviser le fichier en chunks et traiter en parallèle sur plusieurs coeurs ou machines
- **Distribution** : Utiliser un cluster de machines pour traiter des parties du fichier simultanément.
- **Optimisations** : Utiliser des structures de données plus efficaces ou des bibliothèques comme Dask pour le calcul distribué.

6. Option : Version parallèle

Une version parallèle pourrait utiliser multiprocessing en Python pour diviser le fichier en blocs et compter les mots en parallèle. Par exemple :

```
import multiprocessing as mp
```

```
def parallel_word_count(filename, num_processes=4):
```

 Diviser le fichier en chunks

 Traiter chaque chunk en parallèle

 Fusionner les compteurs

```
    pass
```

Cela pourrait réduire le temps pour de gros fichiers, mais introduit une surcharge pour la synchronisation.

Scripts Python principaux :

- gen_corpus.py : Génère les fichiers corpus de tailles variables (1 à 50 MB). Il utilise des fonctions pour créer des mots aléatoires et écrire des lignes dans les fichiers jusqu'à atteindre la taille cible.
- word_count.py : Contient la fonction word_count(filename) qui lit un fichier ligne par ligne, divise chaque ligne en mots, et compte les fréquences à l'aide de collections.Counter. C'est l'implémentation de base du comptage de mots séquentiel.
- run_bench.py : Script de benchmarking qui exécute word_count sur chaque taille de fichier (5 répétitions par taille), mesure les temps d'exécution avec time.time(), calcule les moyennes, et sauvegarde les résultats dans results.csv.
- plot_results.py : Lit les résultats de results.csv, extrait les tailles et moyennes, puis génère un graphique linéaire avec matplotlib pour visualiser la relation taille vs temps d'exécution. Le graphique est sauvégarde dans plot.png.

Fichiers de sortie et rapport :

- results.csv : Fichier CSV contenant les données brutes des benchmarks (colonnes : Size_MB, Time1 à Time5, Average).
- plot.png : Image du graphique généré par plot_results.py.

Flux d'exécution :

1. gen_corpus.py crée les fichiers de test.
2. run_bench.py benchmark word_count sur ces fichiers et produit results.csv.
3. plot_results.py analyse results.csv et génère plot.png.
4. report.md documente tout le processus et les conclusions.

Test

Exécution dans l'ordre :

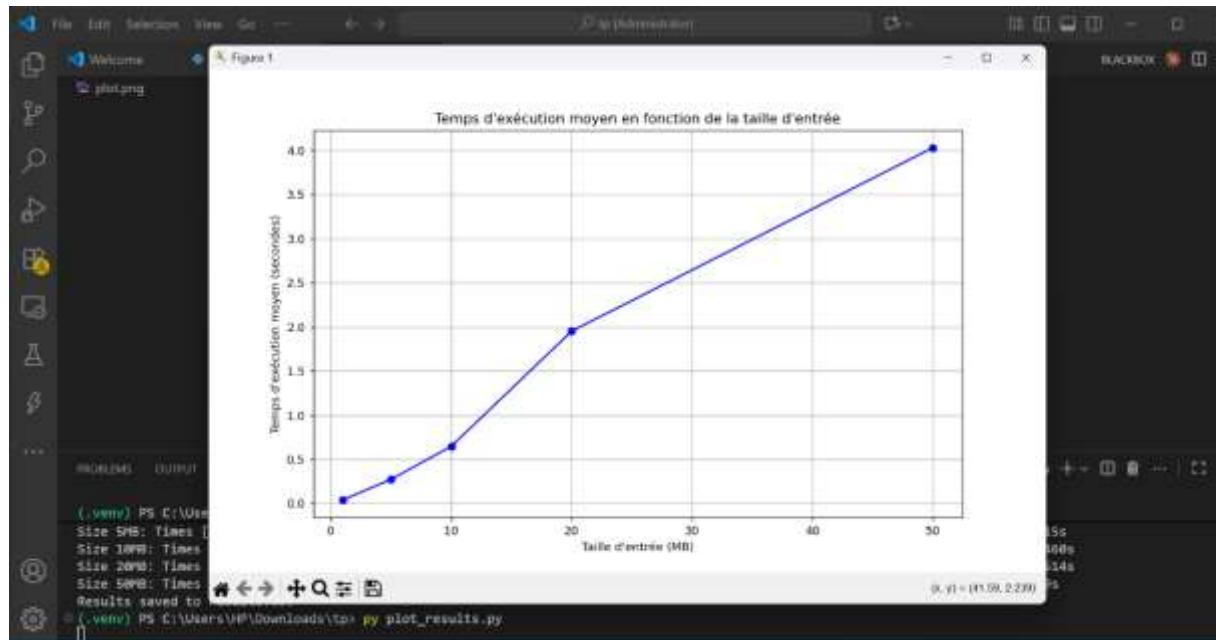
- python gen_corpus.py (Crée les fichiers de test)

```
● (.venv) PS C:\Users\HP\Downloads\tp> py gen_corpus.py
Generated corpus_1MB.txt
Generated corpus_5MB.txt
Generated corpus_10MB.txt
Generated corpus_20MB.txt
Generated corpus_50MB.txt
○ (.venv) PS C:\Users\HP\Downloads\tp> █
```

- **python run_bench.py (Fait les mesures et crée results.csv)**

```
* (.venv) PS C:\Users\HP\Downloads\tp> py run_bench.py
Size 1MB: Times [0.05742955287824287, 0.03439458263977053, 0.03442430496215#2, 0.03722816745239258, 0.03572273254394531], Avg 0.0398s
Size 5MB: Times [0.2986266613086592, 0.268099825286895, 0.2697688470916748, 0.253068447119371, 0.268838272857866], Avg 0.2715s
Size 10MB: Times [0.670685830841864, 0.6266660690807617, 0.6485276222220084, 0.6317752342224121, 0.651015041258667], Avg 0.6460s
Size 20MB: Times [1.5362937458488936, 2.9281204776763916, 2.43452705447827, 1.4694838523864746, 1.4165120124816895], Avg 1.9514s
Size 50MB: Times [3.988532866345215, 3.9526214599609375, 4.260281085968818, 4.041295528411865, 3.906986474998447], Avg 4.0299s
Results saved to results.csv
(.venv) PS C:\Users\HP\Downloads\tp>
```

- **python plot_results.py(Génère plot.png)**



Conclusion

L'implémentation séquentielle du Word Count fonctionne correctement et présente une complexité linéaire confirmée empiriquement. Pour des volumes plus importants, une approche parallèle ou distribuée est nécessaire. Les scripts fournis permettent de reproduire les expériences.

