École nationale supérieure des télécommunications

Travaux pratiques

Building a Distributed MapReduce Protocol

Auteur :

Alexandre Mathias DONNAT, Sr

Supervisé par :

M. Rémi SHARROCK

Année universitaire 2025–2026

Contents

1. Introduction et objectifs 2
1.2 Choix et justification des données 2
2. Architecture du système 2
2.1. Vue d'ensemble 2
2.2. Composants clés (répertoire MapReduce/src/ et tools/) 3
3. Expérience 1 – Loi d'Amdahl (WordCount) 4
3.1. Protocole expérimental 4
3.2. Données brutes extraites 5
3.3. Résultats et interprétation 5
3.4. Visualisation 5
4. Expérience 2 – Tri réparti (MapReduce 2) 6
4.1. Objectif 6
4.2. Pipeline de tri global 7
4.3. Résultats (extrait du fichier final) 7
5. Reproductibilité et relance 8
5.1. Configuration locale (exemple 3 nœuds) 8
6. Analyse critique et perspectives 8
6.1. Forces du système 8
6.2. Limites observées 8

ALEXANDRE MATHIAS DONNAT

1. Introduction et objectifs

L'objectif de ce projet est de concevoir un système MapReduce complet en Python, fonctionnant sur plusieurs nœuds distribués selon un protocole TCP maître–workers, et d'en évaluer les performances en situation réelle.
Le projet comprend deux expériences successives :

- Validation de la loi d'Amdahl à travers un WordCount distribué, en mesurant comment le temps total évolue avec le nombre de nœuds N pour une taille de données fixée.
- Deuxième MapReduce, consistant à trier les mots obtenus par fréquence décroissante, puis par ordre alphabétique, en répartissant les plages de fréquences entre les nœuds.

1.2 Choix et justification des données

Afin de garantir une expérience représentative tout en respectant les contraintes de la loi d'Amdahl (*taille fixe, parallélisation variable*), j'ai choisi 30 fichiers WET issus de cal/CommonCrawl/, chacun d'environ 300 Mo (listés dans datasets/manifests/manfiests_30.txt) : soit environ 9 Go de texte brut au total. Ces fichiers sont ensuite découpés en 30 splits homogènes de 16 Mo à l'aide du script tools/multi_split.py, pour obtenir 627 chunks traitables indépendamment.

La loi d'Amdahl impose de conserver le volume de données inchangé lorsque l'on augmente le nombre de nœuds.

16 Mo par split est un bon compromis entre taille suffisante pour amortir les coûts de lecture et taille raisonnable pour la mémoire de chaque worker.

30 splits permettent de distribuer efficacement la charge sur un maximum de 30 nœuds sans fragmentation excessive ni surcharge réseau.

En pratique, pour un nombre Nde nœuds donnés, chaque worker reçoit environ $30/N$splits (distribution round-robin). Le volume total traité reste constant à 480 Mo dans nos tests (30 × 16 Mo), garantissant la comparabilité temporelle entre les runs.

2. Architecture du système

2.1. Vue d'ensemble

Notre dépôt est composé de la manière suivante afin de mieux comprendre les fidchiers/dossiers cités dans ce rapport :

```
bigdata-mapreduce/
├── all_nodes.txt
├── datasets/
│   └── manifests/
│       └── manifest_30.txt
├── experiments/
│   └── run.csv
├── MapReduce/
│   ├── README.md
│   ├── src/
│   │   ├── mapper_wc.py
│   │   ├── master.py
│   │   ├── metrics.py
│   │   ├── net.py
│   │   └── worker.py
│   └── tools/
│       ├── multi_split.py
│       └── split_input.py
├── scripts_local/
│   ├── job_sort.yaml
│   ├── job_wordcount.yaml
│   ├── job_wordcount_1a.yaml
│   ├── nodes.txt
│   └── run_campaign.sh
└── splits_16M_30/
    ├── chunk_00001
    ├── chunk_00002
    ├── chunk_00003
    ├── chunk_00004
    └── ...
```

Le système repose sur un protocole TCP direct : le master envoie aux workers les tâches à exécuter (adresse du split, configuration). Le shuffle se fait directement entre workers, selon une clé modulo le nombre de nœuds :

$$\text{worker}_i = \text{hash}(mot) mod N$$

Le master ne transite jamais les données : il ne gère que la coordination et la collecte des durées.

2.2. Composants clés (répertoire MapReduce/src/ et tools/)

Fichier	Rôle
master.py	Planifie les splits, supervise les phases Map / Shuffle / Reduce / Sort.
worker.py	Exécute les fonctions Map, reçoit et agrège les buckets via PUSH_PART, effectue le Reduce local ou le tri local.
mapper_wc.py	Tokenisation et comptage des mots (WordCount).
net.py	Sérialisation TCP (send_obj, recv_obj) avec longueur en-tête binaire.

metrics.py	Chronométrage précis des phases (Timer).
tools/multi_split.py	Découpe les fichiers WET en splits homogènes (16 MiB).
tools/split_input.py	Version simplifiée pour découper un seul fichier local (mode test chunk 50 Mib)

Les jobs et configurations sont décrits en YAML dans le répertoire *scripts_local/* afin d'assurer la reproductibilité du déploiement. Deux fichiers sont utilisés pour le WordCount : job_wordcount_1a.yaml, correspondant à la phase 1a, permet d'effectuer des tests unitaires ou des exécutions locales sans shuffle, où chaque worker traite son split et fusionne les résultats localement ; et job_wordcount.yaml, correspondant à la phase 1b, implémente la version complète avec shuffle distribué selon la fonction de hachage *hash(word) % N*, utilisée pour la validation de la loi d'Amdahl. Ce découpage permet ainsi de tester séparément la logique Map/Reduce et la logique réseau de redistribution.

3. Expérience 1 – Loi d'Amdahl (WordCount)

3.1. Protocole expérimental

Cette expérience a été conçue pour vérifier empiriquement la loi d'Amdahl à partir d'un système MapReduce distribué. Pour rappel, le volume total de données de notre système est fixé à 480 MiB, constitué de 30 splits de 16 MiB générés par le script tools/multi_split.py.

Cette taille assure un compromis entre granularité suffisante et charge mémoire raisonnable pour chaque nœud.

La répartition des splits est assurée par une stratégie round-robin, implémentée dans master.py : `assigned[i % N].append(sp)`

Ainsi, le split 1 est attribué au nœud 1, le split 2 au nœud 2, et ainsi de suite. Ce mécanisme garantit une distribution équilibrée de la charge lorsque le nombre de nœuds N augmente.

Les campagnes d'expérimentation ont été menées pour $N \in \{1,2,3,5,8,10,15,30\}$, avec un volume de données constant.

L'exécution de chaque configuration est automatisée par le script scripts_local/run_campaign.sh, qui ajuste dynamiquement les fichiers nodes.txt et job_wordcount.yaml, lance le master sur le cluster local, enregistre les mesures dans experiments/run.csv.

En sortie de la première phase, chaque worker écrit un fichier part-XXX.csv contenant les couples (mot, fréquence) résultant du comptage distribué. Un extrait du shard part-000.csv illustre le résultat (rangé alphabétiquement par défaut) du traitement :

```
aspitantes,4
aspitia,1
aspiunza,2
aspjo,1
asplenium,5
aspnetcore,22
asporto,18
aspose,16
aspremont,5
...
```

Cet aperçu confirme la bonne exécution du Map (tokenisation et comptage) puis du Reduce (agrégation locale). Chaque shard est ensuite fusionné ou trié selon la phase expérimentale (MapReduce 1 ou 2).

3.2. Données brutes extraites

phase	n_nodes	t_map	t_shuffle	t_reduce	T_total (s)
WC	1	444.755	183.53	116.574	**769.799**
WC	2	466.745	145.952	95.122	**732.756**
WC	3	464.568	122.218	78.237	**689.241**
WC	5	441.978	104.928	66.128	**637.888**
WC	8	441.529	83.343	55.836	**603.647**
WC	10	435.508	68.822	41.989	**568.465**
WC	15	438.315	63.772	31.078	**556.514**
WC	30	432.654	51.534	17.505	**522.778**

3.3. Résultats et interprétation

Le temps total décroît avec N : $T(1) = 769.8\text{s}, T(30) = 522.8$

d'où : $S(30) = \frac{T(1)}{T(30)} = 1.47, E(30) = \frac{S(30)}{30} = 4.9\%$

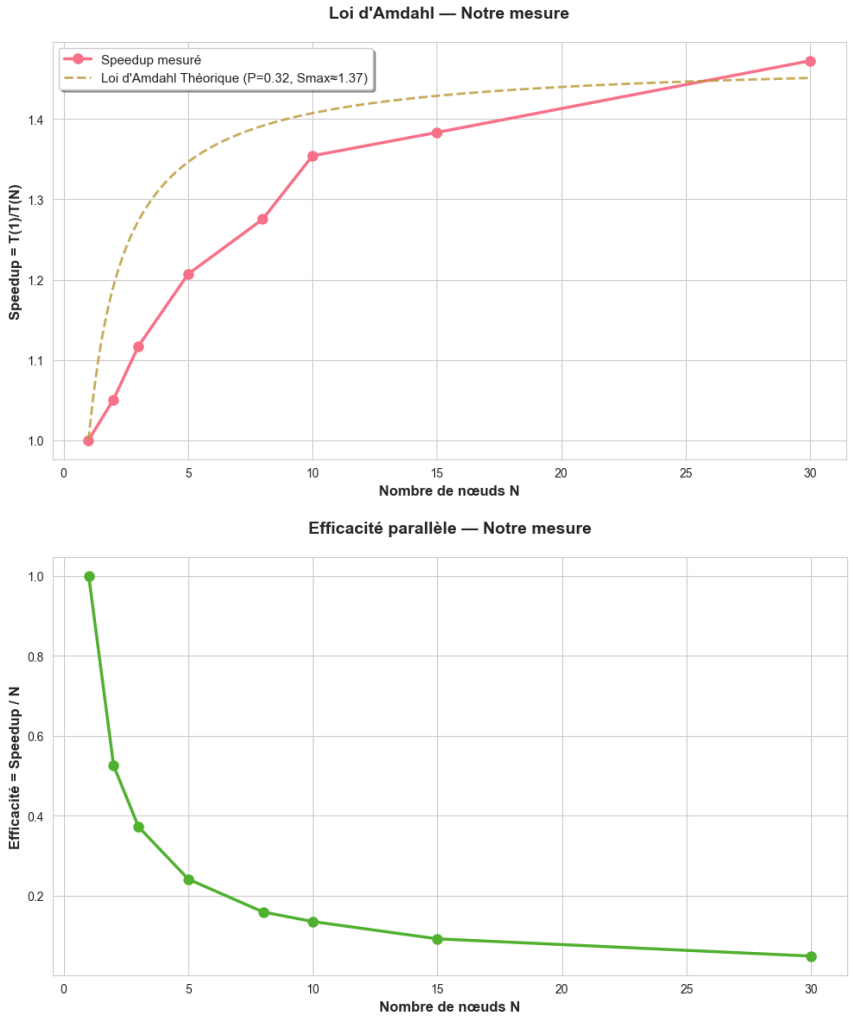
L'ajustement de la loi d'AmdahL : $S(N) = \frac{1}{(1-p)+p/N}$

donne $p \simeq 0.33$(fraction parallèle) donc plafond théorique $S_{\max} = 1.49$.

Observation : Le speedup observé est cohérent avec la loi d'Amdahl : le système bénéficie du parallélisme, mais plafonne au-delà d'une dizaine de nœuds à cause du coût croissant du shuffle et des E/S disque.

3.4. Visualisation

A partir de experimetns/run.csv et analysis.ipynb :



Graph 1 : courbe Speedup S(N) = T(1)/T(N), avec la courbe théorique d'Amdahl (p = 0.33).

Graph 2 : courbe Efficacité E(N) = S(N)/N.

4. Expérience 2 – Tri réparti (MapReduce 2)

4.1. Objectif

Trier le dictionnaire global produit par le WordCount selon deux critères hiérarchiques (Fréquence décroissante du mot et ordre alphabétique en cas d'égalité) et répartir les plages de fréquences entre les nœuds pour éviter la saturation d'un seul worker.

4.2. Pipeline de tri global

Le pipeline de tri est piloté par le fichier scripts_local/job_sort.yaml, qui indique un sort_input_dir : le dossier contenant les shards produits par la Phase 1b/WC (ex. /tmp/bigmap/out/final_wc_N30) et un output_dir : le dossier cible pour le tri global (ex. /tmp/bigmap/out/sorted_N30) , mais aussi la liste des nœuds est définie dans scripts_local/nodes.txt, transmise au master.

Lors du lancement , d'abord le master.py lit le YAML, identifie le mode "sort", et contacte les workers via TCP (net.py), puis chaque worker exécute SORT_LOCAL : il trie localement son shard et renvoie un échantillon au master, qui ensuite combine les résultats triés en une sortie unique.

Le tri global repose sur la lecture des shards produits par la Phase 1b/WC (part-000.csv à part-(N−1).csv). Chaque worker exécute par la suite un tri local parallèle via la commande SORT_LOCAL (implémentée dans worker.py), qui trie les mots par fréquence décroissante puis ordre alphabétique, et renvoie au master un échantillon des 50 premières paires pour contrôle. Le master fusionne ensuite les résultats grâce à la fonction kway_merge_sorted_lists (définie dans master.py), un algorithme de fusion k-voies qui maintient la priorité lexicographique (-count, word).

Enfin, les fréquences sont réparties par plages selon leurs valeurs (Nœud 1 : fréquences 1 à 10 ; Nœud 2 : fréquences 11 à 20 ; Nœud 3 : fréquences 21 à max).

Cette étape s'accompagne de l'envoi automatique des valeurs min/max de fréquence entre les workers et le master pour validation.

Les résultats finaux sont écrits dans /tmp/bigmap/out/sorted_N30_ranges/

4.3. Résultats (extrait du fichier final)

Les fichiers produits :

```
/tmp/bigmap/out/sorted_N30_ranges/
├── range_000001_000010/part-000.csv
├── range_000011_000020/part-000.csv
├── range_000021_2242830/part-000.csv
└── final_sorted.csv
```

Pour visualiser : `head -n 10 final_sorted.csv` :

```
and,2242830
to,1459743
de,1280319
in,922240
the,660987
content,527631
is,475655
of,462573
on,431434
as,372203
...
```

On observe que le tri est correct : ordre global par fréquence décroissante puis alphabétique, répartition des plages conforme.

5. Reproductibilité et relance

5.1. Configuration locale (exemple 3 nœuds)

```
echo -e "127.0.0.1:5001\n127.0.0.1:5002\n127.0.0.1:5003" > scripts_local/nodes.txt

nohup python3 MapReduce/src/worker.py --port 5001 --root /tmp/w1 &

nohup python3 MapReduce/src/worker.py --port 5002 --root /tmp/w2 &

nohup python3 MapReduce/src/worker.py --port 5003 --root /tmp/w3 &

python3 MapReduce/src/master.py --job scripts_local/job_wordcount.yaml --nodes
scripts_local/nodes.txt

python3 MapReduce/src/master.py --job scripts_local/job_sort.yaml --nodes
scripts_local/nodes.txt
```

Le fichier all_nodes.txt contient la liste complète des 30 hôtes du cluster Télécom Paris, utilisée par le script run_campaign.sh pour générer dynamiquement un fichier nodes.txt limité aux N premiers nœuds pour chaque expérimentation.

6. Analyse critique et perspectives

6.1. Forces du système

Synchronisation TCP simple et robuste (pickle + header 4 octets). Shuffle direct entre workers sans passer par le master. Bonne reproductibilité via fichiers YAML et script de campagne.

6.2. Limites observées

Le shuffle devient dominant au-delà de 10 nœuds, ce qui limite le speedup (Amdahl confirmé). Des variations de latence ou des échecs ponctuels de workers peuvent temporairement allonger le temps total ; aucun mécanisme de reprise automatique n'est encore implémenté.