Système MapReduce “fait maison” et expérimentation de la loi d’Amdahl

Benjamin Lepourtois
Mastère Spécialisé Data – Télécom Paris, promotion 2026

Matière : Systèmes répartis - DATA701
Enseignant : Rémi SHARROCK

Résumé

Ce rapport présente la conception et l’implémentation d’un système MapReduce “fait maison”, développé dans le cadre du cours *Systèmes répartis* à Télécom Paris. L’objectif est double : (i) réaliser un système distribué de type MapReduce s’appuyant sur des sockets TCP pour coordonner un master et plusieurs workers, et (ii) mettre en place une expérimentation de la loi d’Amdahl en faisant varier le nombre de nœuds et la taille des données, à partir d’un corpus de type CommonCrawl. Le système respecte les contraintes du cahier des charges (protocole, absence de transit des données applicatives par le master, déploiement sur les machines de l’école, scénarios d’évaluation) et implémente deux jobs MapReduce : un *wordcount* enrichi de détection de langue, puis un tri réparti des mots par fréquence décroissante et ordre alphabétique. Les mesures obtenues sont agrégées dans un format exploitable pour l’analyse et la visualisation de la loi d’Amdahl.

Code source : https://github.com/BNJ02/mapreduce_homemade

1 Introduction

Le projet vise à concevoir un système de type MapReduce à partir de briques de base (sockets TCP, multiprocessus, scripts de déploiement), sans recourir à un framework existant. Ce système doit être déployable sur les machines de Télécom Paris, en respectant les contraintes d’infrastructure (NFS partagé, limite de connexions SSH, protocole TCP) et en permettant de réaliser une étude expérimentale de la loi d’Amdahl.

Le travail se structure autour de trois axes principaux :

— la définition du protocole entre un master et un ensemble de workers
— l’implémentation de deux jobs MapReduce complets (wordcount puis tri réparti)
— la mise en place de scripts d’orchestration et de benchmarks permettant l’analyse de la loi d’Amdahl

2 Architecture et implémentation

2.1 Organisation du dépôt

L'organisation réelle du dépôt s'articule autour des fichiers et répertoires suivants :

- `master.py` : implémentation du master, orchestration des jobs, gestion des splits et coordination des workers.
- `worker.py` : implémentation des workers, phases Map / Shuffle / Reduce, communication TCP.
- `config/nodes.txt` : liste des nœuds utilisés pour les expériences.
- `scripts/` :
  - `run_cluster.py`, `update_nodes.py` : remplissage/validation de `nodes.txt`, lancement des workers
  - `benchmark_cluster.py` : exécution automatisée des expériences pour divers nombres de workers et tailles de données
  - `results_summary.py` : agrégation des résultats dans un format exploitable.

2.2 Rôle du master

Le master (`master.py`) remplit plusieurs fonctions :

- lecture de `config/nodes.txt` et construction de la liste des nœuds
- découpe des données d'entrée en splits (fichiers ou intervalles de fichiers dans le NFS)
- distribution des splits selon une politique *roundrobin* à partir de deux nœuds
- ouverture d'un serveur TCP écoutant les connexions des workers
- envoi des paramètres de job (*job type*, liste de splits, ports de shuffle, etc.) aux workers
- suivi de l'avancement du job (accusés de réception, erreurs éventuelles) et collecte de métadonnées (durées de phases)

Le master ne transporte jamais les données applicatives (contenu des splits), ce qui respecte la contrainte du cahier des charges : les workers lisent directement les fichiers sur le NFS.

2.3 Rôle des workers

Chaque worker (`worker.py`) :

- se connecte au master via TCP et reçoit sa configuration (liste de splits à traiter, rôle dans la topologie)
- exécute la phase Map en exploitant le multicœurs (par exemple via `ProcessPoolExecutor`) pour paralléliser la lecture et le traitement des splits
- effectue le shuffle pair-à-pair : pour chaque paire $(\text{clé}, \text{valeur})$, le worker calcule un hash de la clé qui détermine le worker destinataire et envoie les données directement via TCP
- exécute la phase Reduce en agrégant les données reçues et en produisant un résultat local (fichier JSON ou équivalent)
- enregistre des métriques (temps de Map, Shuffle, Reduce) et les communique au master ou les écrit dans `results.jsonl`

La logique de communication (*handshake*, échanges de coordonnées de shuffle, formats de messages) est centralisée afin d'être réutilisable pour les deux jobs (wordcount et tri).

3 Jobs MapReduce implémentés

3.1 Job 1 : Wordcount avec détection de langue

Le premier job est un *wordcount* enrichi, dont l'objectif est de :

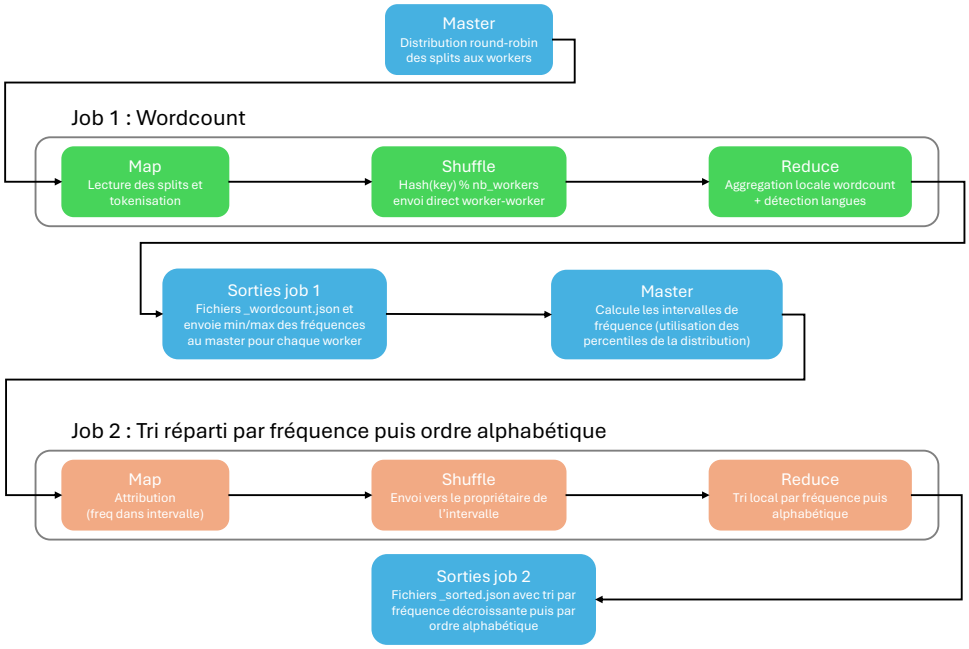

FIGURE 1 – Pipeline MapReduce : wordcount puis tri réparti des mots par fréquence.

— compter le nombre d'occurrences de chaque mot dans le corpus CommonCrawl
— estimer la langue dominante des documents ou des segments de texte (*langdetect*)
— produire des statistiques de fréquences par langue

Phase Map. Chaque worker lit les splits qui lui sont assignés, applique une tokenisation simple (nettoyage, mise en minuscules, filtrage) et émet des paires $(\text{mot}, 1)$. En parallèle, une détection de langue est effectuée (par exemple pour chaque document ou bloc de texte) et agrégée dans des compteurs de type $(\text{langue}, 1)$.

Phase Shuffle. Pour chaque mot, le worker calcule un hash de la clé et envoie les paires au worker propriétaire du bucket correspondant. Ce *shuffle* pairàpair se fait via TCP, en respectant le fait que le master ne voit jamais les données applicatives.

Phase Reduce. Chaque worker agrège les paires reçues pour produire un *wordcount* local (fichier `_wordcount.json` contenant $(\text{mot}, \text{fréquence})$), ainsi que des statistiques de langue. Les résultats sont stockés sur le NFS pour être réutilisés par le job 2 et pour l'analyse.

3.2 Job 2 : Tri réparti par fréquence puis ordre alphabétique

Le second job prend comme entrée les résultats du wordcount et produit un tri global des mots :

— ordre principal : fréquence décroissante
— ordre secondaire : ordre alphabétique pour les mots de même fréquence

Définition des intervalles de fréquence. À partir des fichiers de wordcount, le master calcule des statistiques globales (min/max de fréquences, échantillons, quantiles) et définit des intervalles de fréquences associés à chaque worker. Par exemple :

— worker 1 : mots de fréquence $[1, 10]$
— worker 2 : mots de fréquence $[11, 100]$

— etc.

Les mots très rares peuvent être répartis par hashing pour éviter le déséquilibre de charge.

Bucketisation (Map 2) et Shuffle 2. Les workers lisent les fichiers de wordcount et affectent chaque mot à l'intervalle de fréquence approprié. Les paires (mot, fréquence) sont envoyées au worker propriétaire de l'intervalle, via un second *shuffle* pairàpair.

Reduce 2 et sortie globale. Chaque worker trie localement les mots de son intervalle par fréquence décroissante, puis par ordre alphabétique. La concaténation des sorties ordonnées produit un tri global des mots du corpus. Ces sorties peuvent ensuite être utilisées pour générer des tableaux ou figures (par exemple top 50 des mots par langue).

4 Déploiement et scripts d'orchestration

L'automatisation des expériences repose sur plusieurs scripts Python dans `scripts/` :

— `update_nodes.py` : récupération et mise à jour de la liste des machines dans `config/nodes.txt`, à partir de l'API fournie par l'école.
— `run_cluster.py` : lancement des workers sur les machines listées, en respectant la limite de 5 connexions SSH par minute depuis une même machine.
— `benchmark_cluster.py` : boucle d'expériences pour différentes combinaisons (nombre de workers, nombre de splits), lancées à distance depuis sa machine locale connectée au réseau de Télécom Paris.
— `results_summary.py` : agrégation des résultats dans `benchmarks/results.jsonl`.

Les résultats bruts (`results.jsonl`) contiennent typiquement, pour chaque run :

— le nombre de workers utilisés
— le nombre de splits
— les temps de Map, Shuffle, Reduce au total
— d'autres métriques comme le timestamp, le nombre de mots traités, le nombre de mots uniques, le top 20 des mots les plus fréquents, les langues détectées, etc.

Ces données sont ensuite exploitées dans des notebooks (non détaillés ici) pour visualiser les courbes de speedup, l'effet de la taille de données et la répartition des langues.

5 Expérimentation de la loi d'Amdahl

5.1 Méthodologie

La loi d'Amdahl exprime le speedup théorique $S(N)$ d'un programme en fonction du nombre de processeurs N et de la fraction parallélisable p :

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}.$$

Dans le cadre de ce projet, la définition du *speedup* est adaptée au protocole :

— la référence n'est pas nécessairement un run strictement séquentiel, mais peut être définie comme le cas où le nombre de workers est égal au nombre de splits
— seules les phases Map + Shuffle + Reduce sont prises en compte dans les temps mesurés

Pour chaque taille de données, plusieurs campagnes sont réalisées en faisant varier le nombre de workers (par exemple $N \in \{1, 2, 4, 8, \dots\}$). Le speedup observé est alors :

$$S_{\text{obs}}(N) = \frac{T_{\text{ref}}}{T(N)},$$

où T_{ref} est le temps de référence et $T(N)$ le temps mesuré avec N workers.

5.2 Résultats et estimation de p

À partir des mesures stockées dans `benchmarks/results.jsonl`, une régression est effectuée pour estimer la fraction parallélisable p (et donc la fraction sérielle $1 - p$). En pratique :

— on ajuste la courbe prévue par la loi d'Amdahl sur les points $(N, S_{\text{obs}}(N))$;
— on compare les prédictions et les observations pour différents volumes de données.

La Figure 2 synthétise ce fit empirique : pour chaque configuration (workers, splits), on calcule le speedup observé et on le compare à la surface prédite par la loi d'Amdahl.

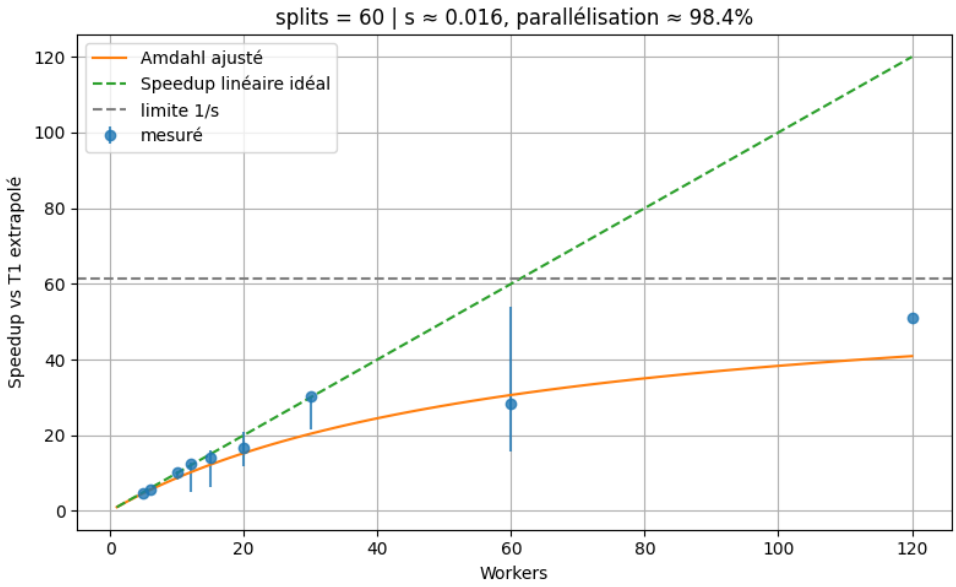

FIGURE 2 – Speedup empirique en fonction du nombre de workers et des splits, et surface prédite par la loi d'Amdahl.

Les visualisations (et en particulier le notebook `amdahl_splits60.ipynb`) montrent que la fraction parallélisable estimée est très élevée. Cela s'explique par plusieurs facteurs :

— la part de calcul pur (tokenisation, agrégation de clés, tri local) domine les coûts sériels et la coordination par le master reste relativement légère
— le système exploite non seulement le parallélisme entre machines, mais aussi le parallélisme intra-nœud : chaque worker lance plusieurs processus de Map/Reduce en parallèle sur les cœurs locaux (jusqu'à 12 par machine). En combinant par exemple 60 workers avec 12 cœurs chacun, on atteint théoriquement jusqu'à $60 \times 12 = 720$ processus de traitement en parallèle
— les entrées/sorties disque sont en grande partie masquées par le pipelining entre Map, Shuffle et Reduce, ce qui réduit encore la fraction sérielle effective

On observe néanmoins que, au-delà d'un certain nombre de workers, les coûts sériels (coordination, latence réseau, synchronisations, contention sur le NFS) finissent par dominer. À volume de données constant, la loi d'Amdahl rend bien compte de cette saturation : le fit de p reste élevé, mais la courbe empirique tend vers une asymptote fixée par la fraction sérielle résiduelle.

6 Analyse des langues

La phase Map du job wordcount intègre une détection de la langue pour les segments de texte traités. Les résultats agrégés permettent de compter le nombre de tokens par langue.

On constate que les occurrences sont dominées très largement par l'anglais, puis par quelques autres langues majeures (russe, allemand, français, espagnol). Les autres langues se retrouvent

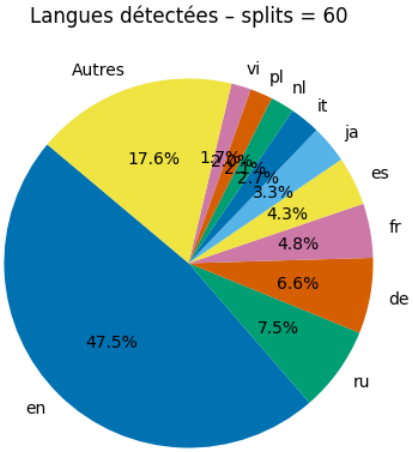

FIGURE 3 – Répartition des occurrences par langue dans le corpus CommonCrawl traité.

dans une longue traîne de fréquences plus faibles. Cette distribution est cohérente avec la composition attendue d'un corpus web mondial, et valide le fait que les statistiques agrégées par le système capturent bien la structure multilingue des données.

7 Les limites

Plusieurs limites sont à noter :

— la robustesse face aux pannes de nœuds reste limitée (pas de réallocation automatique des splits)
— la politique de répartition par intervalles de fréquences peut encore être raffinée pour mieux équilibrer la charge
— la collecte de métriques pourrait être enrichie (latences réseau détaillées, taille des buffers de shuffle, etc.) pour ensuite gagner du temps en prenant des décisions d'optimisation

8 Conclusion

Ce projet a permis de mettre en œuvre un système de type MapReduce sur une infrastructure réelle, en respectant un cahier des charges proche de scénarios industriels tout en restant suffisamment simple pour être entièrement implémenté en Python. Le système :

— implémente un protocole TCP master–workers avec shuffle pairàpair
— réalise deux jobs MapReduce, dont un tri réparti par fréquence et ordre alphabétique
— exploite le multicœurs et le NFS partagé des machines de Télécom Paris
— fournit des benchmarks et des analyses conformes à la loi d'Amdahl, ainsi qu'une exploration de la répartition des langues dans le corpus

Les extensions possibles incluent la tolérance aux pannes, des algorithmes de partitionnement plus sophistiqués pour le tri réparti, et l'intégration d'autres types de jobs analytiques sur le corpus CommonCrawl.

9 Pour aller plus loin

Au-delà des expériences principales, la loi d'Amdahl a été étudiée sur des configurations plus fines, en faisant varier le nombre de splits indépendamment du nombre de workers. Dans

ces expériences supplémentaires, le nombre de splits n'est plus nécessairement un multiple du nombre de workers, ce qui rompt la symétrie idéale où chaque worker reçoit exactement le même nombre de splits.

La Figure 4 présente un exemple de ces résultats étendus, et la Figure 5 propose une vue tridimensionnelle.

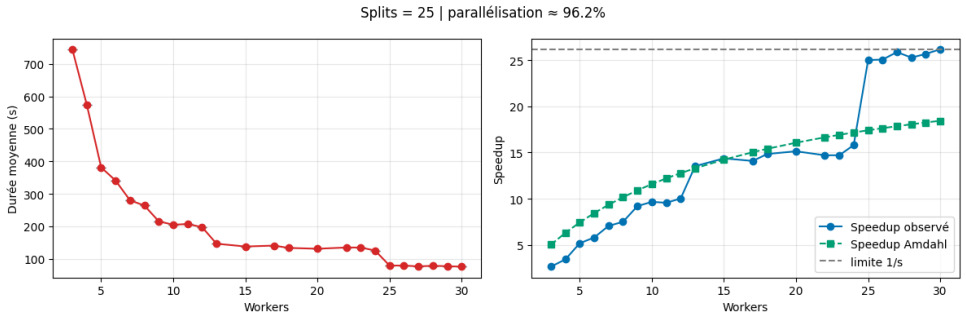

FIGURE 4 – Speedup observé pour différents nombres de splits, y compris lorsque le nombre de splits n'est pas un multiple du nombre de workers.

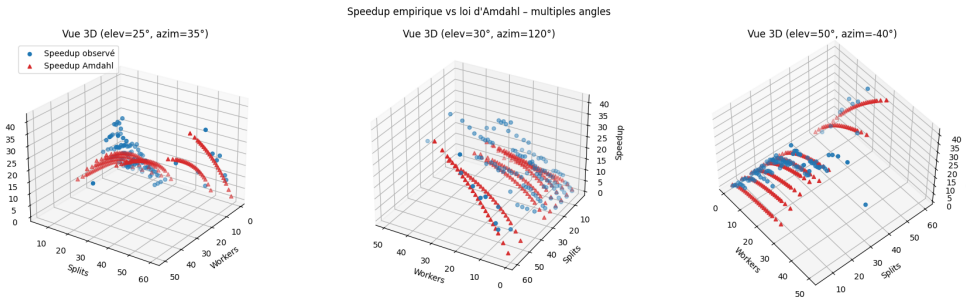

FIGURE 5 – Visualisation 3D de la loi d'Amdahl : nombre de workers, nombre de splits (quantité de données) et speedup.

On observe un effet marqué de « marches d'escalier » : dès que l'on atteint une configuration où le nombre de splits devient un multiple du nombre de workers, le speedup augmente brusquement d'un échelon. Ce phénomène est lié à la structure même du pipeline :

— le premier passage de Map sur les splits est la phase la plus coûteuse en temps, car chaque worker doit parcourir l'ensemble des splits qui lui sont attribués
— l'attribution des splits par le master se fait en roundrobin, et dans notre expérience les splits sont atomiques : un split ne peut pas être subdivisé ni partagé entre deux workers
— tant que la division des splits ne tombe pas pile sur un multiple du nombre de workers, certains workers se retrouvent avec un split supplémentaire à traiter, ce qui crée un déséquilibre de charge et limite le speedup global

Lorsque l'on atteint un multiple, la charge se répartit beaucoup plus uniformément, d'où le saut visible dans le speedup. Malgré ces effets de granularité, les taux de parallélisation estimés restent très élevés dans l'ensemble du spectre exploré, ce qui confirme la pertinence de l'architecture choisie pour exploiter massivement le parallélisme inter et intranœuds.

10 Reproductabilité

Pour garantir la reproductabilité des expériences, plusieurs éléments sont nécessaires : l'installation de `uv` sur le réseau de machines TP, la mise à jour régulière de la liste de nœuds disponibles, et l'utilisation des scripts `benchmark_cluster.py` et/ou `run_cluster.py`.

10.1 Installation de uv sur le NFS

Il suffit d'installer uv une fois sur une machine du réseau TP dont le $HOME est partagé via NFS. Par exemple :

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Par défaut, uv est alors disponible dans $HOME/.local/bin/uv. Grâce au NFS, ce chemin est visible depuis toutes les machines de la salle (même chemin /cal/exterieurs/blepourt-25/.local/bin/uv).

Pour disposer d'un Python 3.14 (et exploiter le multi-cœurs via la pile uv), on peut demander à uv de l'installer, créer un environnement virtuel partagé via NFS, puis installer les dépendances du projet :

```
uv python install 3.14
uv venv --python 3.14 ~/.venv
uv pip install langdetect
```

L'environnement /.venv ainsi créé sera visible depuis toutes les machines partageant le même NFS et aura les mêmes dépendances installées.

10.2 Mise à jour de config/nodes.txt

Avant de lancer des expériences depuis sa machine locale, il est recommandé de rafraîchir la liste des machines disponibles depuis sa machine locale (connectée au réseau TP ou via VPN). Pour cela, on utilise le script :

```
python3 scripts/update_nodes.py
```

Ce script interroge l'API https://tp.telecom-paris.fr/ajax.php, récupère les machines marquées comme disponibles, les ordonne par groupes prioritaires (par exemple tp-1d23, tp-1a226, etc.) et écrit le résultat dans config/nodes.txt. Ce fichier est ensuite utilisé par run_cluster.py et copié sur la machine distante par benchmark_cluster.py.

10.3 Lancer des benchmarks depuis sa machine locale

Depuis une machine personnelle (connectée au réseau TP ou via VPN) et à la racine du dépôt, il est possible d'orchestrer des runs complets en utilisant scripts/benchmark_cluster.py. Une commande typique est :

```
python3 scripts/benchmark_cluster.py \
  --remote-host blepourt-25@tp-1a226-00 \
  --remote-path /cal/exterieurs/blepourt-25/hadoop_homemade \
  --remote-uv-bin /cal/exterieurs/blepourt-25/.local/bin/uv \
  --splits 5 \
  --worker-counts 1 2 5 \
  --fetch-output
```

où :

- -remote-host désigne la machine TP qui servira d'hôte de référence (ici blepourt-25@tp-1a226-00)
- -remote-path est le chemin du projet sur le NFS de cette machine (par ex. /hadoop_homemade)
- -remote-uv-bin pointe vers l'exécutable uv installé sur le NFS (par défaut $HOME/.local/bin/uv)
- -splits fixe le nombre de splits à traiter (ici 5)
- -worker-counts liste les nombres de workers à tester successivement (ici 1, 2 puis 5 workers)
- -fetch-output : permet de récupérer en local les résultats produits

Autres options utiles de `benchmark_cluster.py` :

- `-paths` : liste des fichiers/dossiers à copier via `scp` (par défaut `master.py`, `worker.py`, `config`, `scripts/run_cluster.py`, `scripts/results_summary.py`)
- `-nodes-file` : chemin distant vers `nodes.txt` utilisé par `run_cluster.py` (par défaut `config/nodes.txt`)
- `-benchmark-file` : fichier local où append les résultats (`benchmarks/results.jsonl` par défaut)
- `-export-dir` : dossier local où rapatrier les `output/` si souhaité (`benchmarks/exports` par défaut)

Le script se charge de copier les fichiers nécessaires (`master.py`, `worker.py`, `config`, `scripts/run_cluster.p scripts/results_summary.py`) vers `remote-path`, de lancer `run_cluster.py` à distance pour chaque configuration, puis d'agréger les résultats dans `benchmarks/results.jsonl`. Optionnellement, il peut aussi rapatrier les répertoires `output/`.

Quand `-fetch-output` est utilisé, les résultats bruts (wordcount et tri) sont rapatriés en local sous `benchmarks/exports/`. Exemple :

```
benchmarks/
  exports/
    run_workers1_splits1/
      master_metrics.json
      tp-1d23-02_sorted.json
      tp-1d23-02_wordcount.json
  results.jsonl
```

Chaque exécution de benchmark ajoute une ligne à `benchmarks/results.jsonl`. Le sous-dossier `exports` peut rapidement occuper de l'espace disque si l'on rapatrie de nombreux runs.

En cas de crash de certains workers, il est possible de nettoyer les processus Python sur l'ensemble des machines listées dans `config/nodes.txt` grâce au script shell (à exécuter depuis sa machine locale) :

```
bash scripts/kill_python_all.sh
```

Le master, lancé sur une machine en particulier, doit être terminé manuellement (par exemple via `htop` ou `kill -9`).

10.4 Lancer directement le cluster depuis une machine TP

Il est également possible de lancer directement le cluster à partir d'une machine TP (par exemple `tp-1a226-00`), sans passer par `benchmark_cluster.py`. Une commande typique, exécutée depuis la racine du dépôt sur cette machine, est :

```
uv run python scripts/run_cluster.py \
  --nodes-file config/nodes.txt \
  --workers 1 \
  --splits 1 \
  --remote-path /cal/exterieurs/blepourt-25/hadoop_homemade \
  --uv-bin /cal/exterieurs/blepourt-25/.local/bin/uv
```

Les principaux arguments de `run_cluster.py` sont :

- `-nodes-file` : fichier listant les machines disponibles (par défaut `config/nodes.txt`)
- `-master-node` : machine sur laquelle lancer le master (par défaut la première du fichier)
- `-workers` : nombre de workers à lancer (si `-worker-nodes` n'est pas fourni)
- `-splits` : nombre de splits à traiter

— `-remote-path` : chemin du projet sur les machines distantes (partagé via NFS)
— `-data-dir`, `-file-prefix`, `-file-suffix`, `-split-padding` : décrivent la localisation et le format des fichiers CommonCrawl
— `-output-dir` : répertoire de sortie relatif à `remote-path` (par défaut `output`)
— `-uv-bin` : chemin vers l'exécutable `uv` utilisé pour lancer `master.py` et `worker.py`

Pour que ce script fonctionne, le NFS doit contenir au minimum la structure suivante (au même `remote-path` sur toutes les machines) :

```
config/nodes.txt
master.py
worker.py
scripts/run_cluster.py
scripts/results_summary.py
```

Les résultats sont écrits dans le répertoire `output` sous `remote-path`. Par exemple :

```
output/
  master_metrics.json
  tp-1d23-02_sorted.json
  tp-1d23-02_wordcount.json
  workers5_splits5/
    master_metrics.json
    tp-1d23-02_sorted.json
    tp-1d23-02_wordcount.json
    tp-1d23-03_sorted.json
    tp-1d23-03_wordcount.json
    tp-1d23-04_sorted.json
    tp-1d23-04_wordcount.json
    tp-1d23-05_sorted.json
    tp-1d23-05_wordcount.json
    tp-1d23-06_sorted.json
    tp-1d23-06_wordcount.json
```

Ces fichiers contiennent les résultats détaillés du wordcount et du tri réparti pour chaque worker, ainsi que les métriques globales du master (`master_metrics.json`) utilisées pour l'analyse de la loi d'Amdahl.

Auteur : Benjamin Lepourtois – Mastère Spécialisé Data, Télécom Paris (promotion 2026)
Matière : Systèmes répartis - DATA701
Enseignant : Rémi SHARROCK