BIG DATA ET SCIENCE DE DONNÉES

SCHEMA DE PARALLELISATION

Master 1 Intelligence Artificielle

Université de M'sila, Département d'Informatique

Dr Mehenni Tahar

2020-2021

Introduction

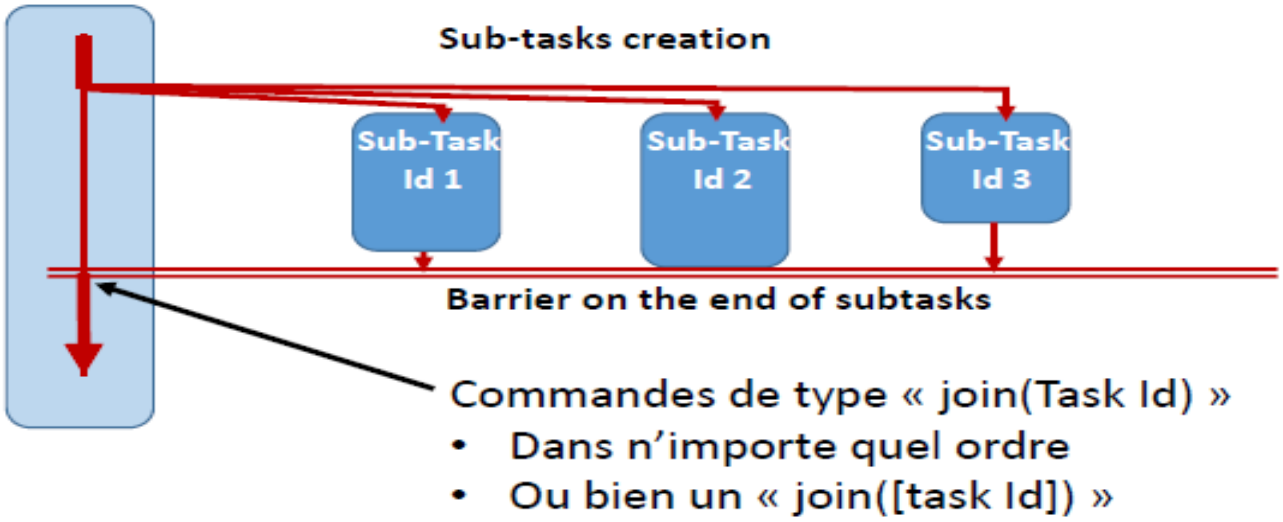
- Les PC contiennent plusieurs processeurs qui sont tous multi-cores,
- Agrégation de plusieurs de ces PC au sein de clusters pour atteindre des capacités de calcul et de stockage importantes et extensibles (nécessaire pour passer à l'échelle).
- Mais exploiter ces architectures demande de développer des applications parallèles et distribuées, et donc de concevoir des algorithmiques plus complexes.
- Ce chapitre introduit des schémas d'applications multi-tâches, visant à exploiter en parallèle plusieurs coeurs d'une même machine ou plusieurs machines.
- Il se focalise sur deux schémas de parallélisation classiques et de haut niveau : le schéma SPMD apparu dans le HPC, et le schéma Map-Reduce associé au Big Data.

Outils de synchronisation

- Les barrières de synchronisation sont typiques de la programmation parallèle visant à exécuter concurremment plusieurs tâches dans le but d'accélérer les traitements.

Barrières sur fin de tâches

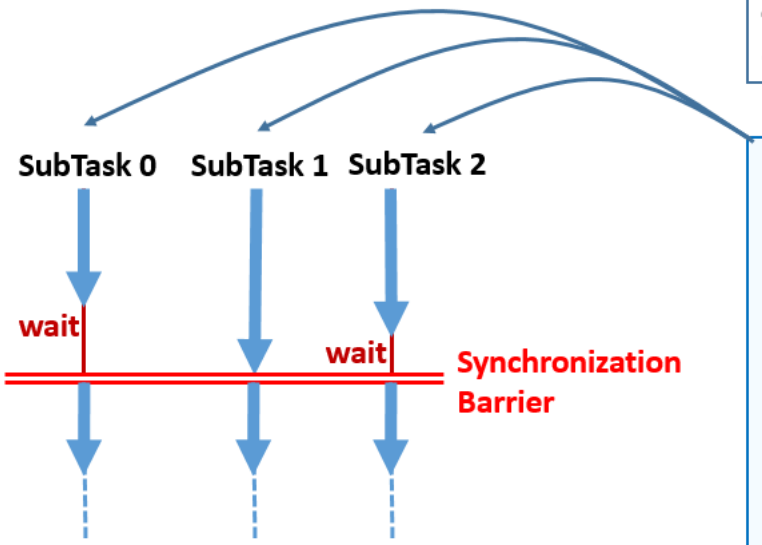
- Chaque tâche arrivant sur une barrière se bloque
- La dernière du groupe à se bloquer débloque toutes les tâches
- Concept de « groupe de tâches » associé à une barrière



Outils de synchronisation

Barrières génériques

- Ex : Calculs itératifs & Synchronisation à chaque itération




```
1 // Main task code
2 Data_t InTab1[N];
3 Data_t InOutTab2[N];
4 Data_t OutTab3[N];
5 .... // - Create and launch SubTasks
```

```
1 // SubTask code
2 int Me = .... // Ranking each task
3 ....
4 // - read InTab1, write into InOutTab2
5 f(InTab1, InOutTab2,N/3*Me, N/3*(Me+1))
6 // - Wait all tasks have finished to write into InOutTab2
7 Barrier(TaskGroupId)
8 // - read InOutTab2, write into OutTab3
9 g(InOutTab2,OutTab3,N/3*Me, N/3*(Me+1))
10 ....
```

Barrière : pour être sur que le tableau InOutTab2[ ] a été entièrement généré avant d'être exploité

Outils de synchronisation

Mise en oeuvre de barrières génériques

- Assez simple et rapide en mémoire partagée (dans un PC): sémaphores, PAS d'attente active…
- Plus complexe et plus couteux en mémoire distribué (clusters): échanges de messages entre PC
- Traverser une barrière de synchronisation à toujours un coût (même si les tâches terminent en même temps)

- En mettre le moins possible !

Schéma de parallélisation SPMD

- SPMD signifie *Single Program Multiple Data*, et correspond à un paradigme de programmation parallèle adapté aux architectures à base de CPU.
- 1 seul programme répété dans toutes les tâches
- Toutes les tâches s'exécutent en parallèle mais sur des données différentes.
- **Exemple:** dans une structure *if (cond) then ... else ...*, une tâche peut passer dans le *then* et une autre dans le *else*, selon les valeurs de leurs variables testées dans la condition du *if*.
- Contrairement au paradigme SIMD (*Single Instruction Multiple Data*), les tâches ne sont pas resynchronisées à chaque tic d'horloge, mais seulement à des points de synchronisation explicites.

Schéma de parallélisation SPMD

Exemple de SPMD réalisé par multithreading

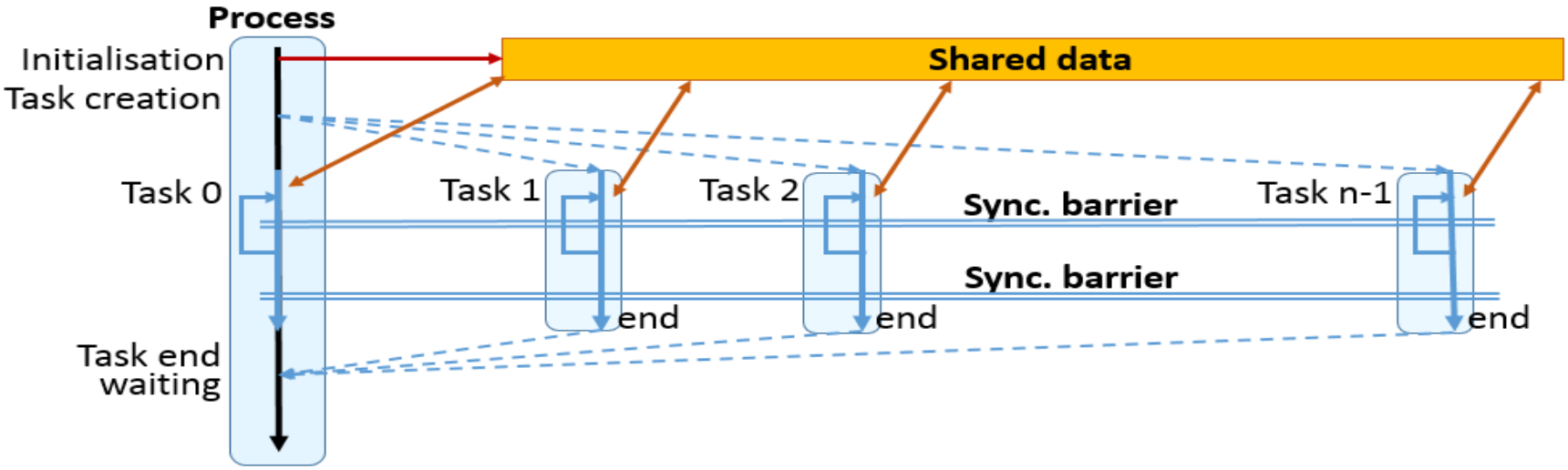
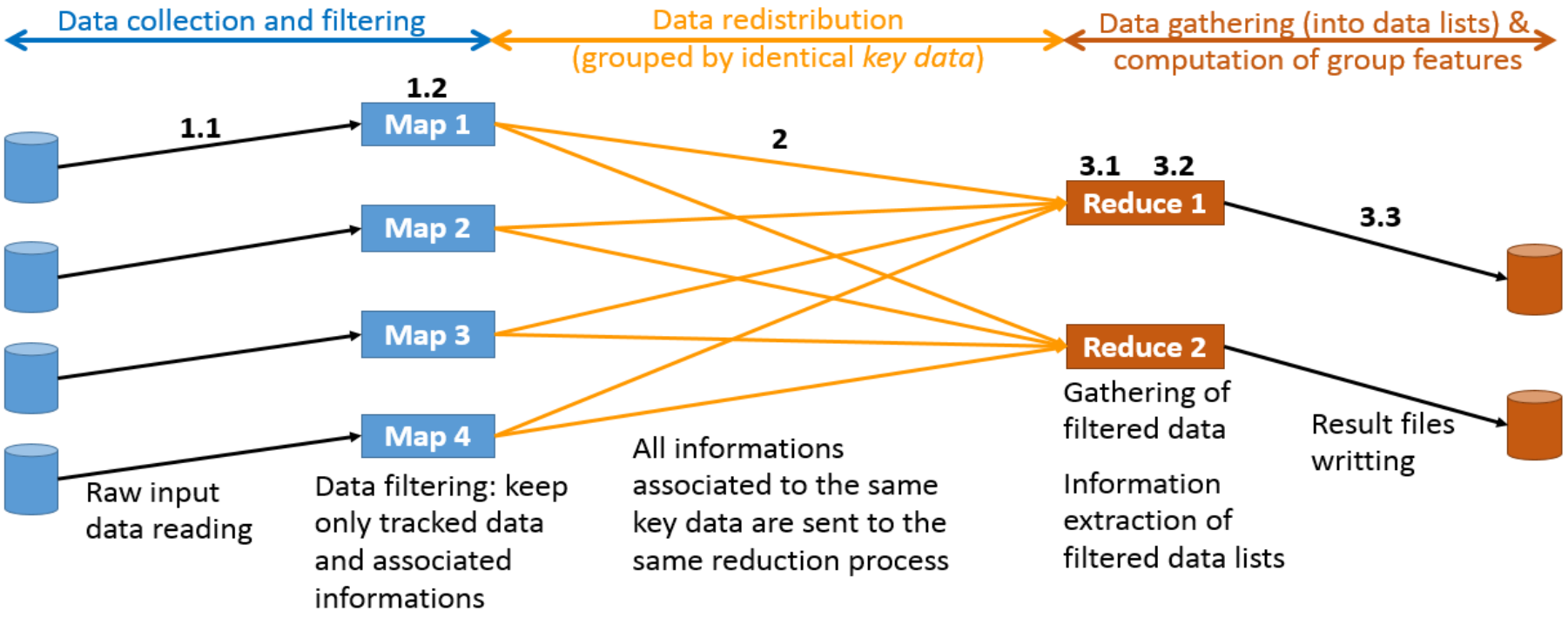


Schéma de parallélisation Map-Reduce

- **Principes de fonctionnement de Map-Reduce**

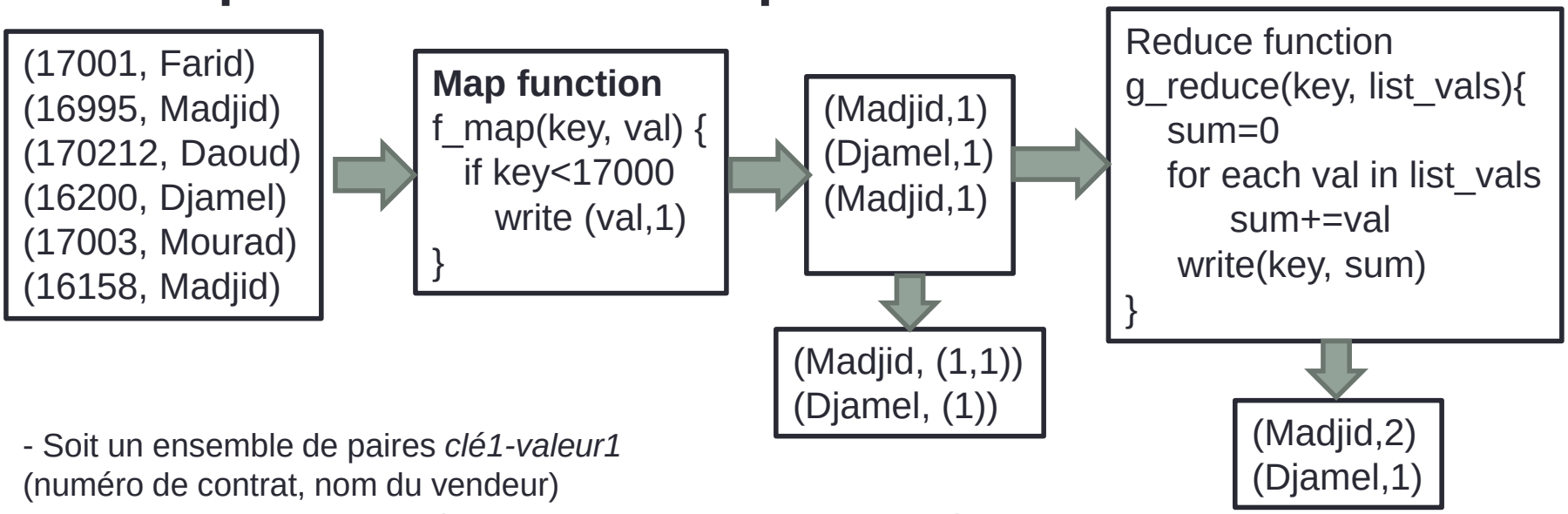


Map ⟶ Shuffle & Sort ⟶ Reduce

À base de paires (clé – valeur(s))

Schéma de parallélisation Map-Reduce

- **Exemple de traitement Map-Reduce**



- Soit un ensemble de paires *clé1-valeur1* (numéro de contrat, nom du vendeur)

- Cet ensemble est lu et traité par la phase Map qui applique indépendamment sur chaque paire une fonction f_map qui teste la clé de la paire et ne retient que les clés inférieure à 17000, pour lesquelles elle génère une nouvelle paire *clé2-valeur2* (nom du vendeur, 1), avec 1 = un contrat vendu. La clé de la deuxième paire est donc la valeur de la paire initiale, et sa valeur est imposée à 1.

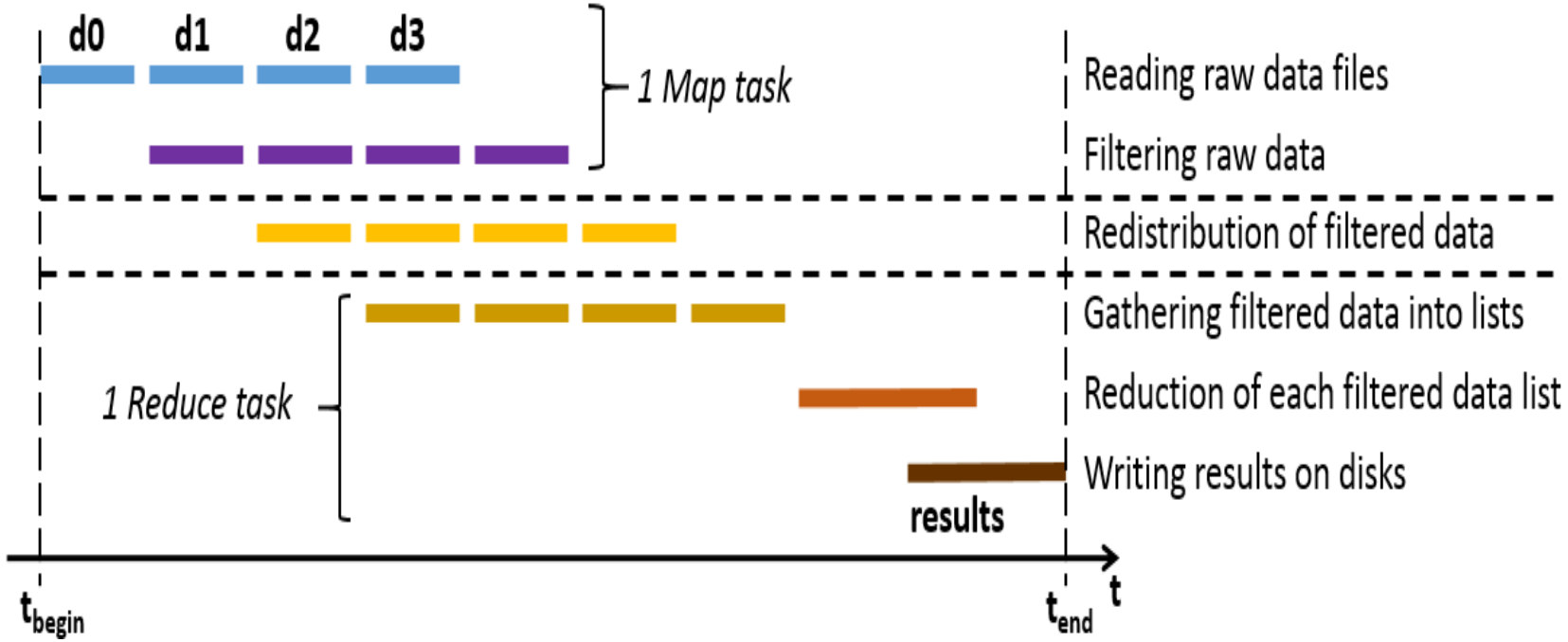
- La phase *shuffle & sort* regroupe tout d'abord toutes les paires de même clé, et crée une nouvelle paire *clé2 - liste de valeur2*. Deux paires de clé *Madjid* et de valeur 1 vont donc donner naissance à une paire (Madjid, (1,1)).

- Enfin, on applique une fonction g_reduce à chaque paire *clé2 - liste de valeur2*, qui calcule la somme des valeur2 de la liste, c'est à dire le nombre de contrats vendu par un même vendeur. La fonction g_reduce génère alors une nouvelle paire *clé3-valeur3*, avec toujours le nom du vendeur comme clé, et le nombre de contrats vendus par ce vendeur comme valeur.

Schéma de parallélisation Map-Reduce

Pipelining de Map-Reduce

- Pour augmenter le parallélisme, et réduire le temps d'exécution global
- Pour tenter de masquer les temps d'IO (masquer les lectures et écritures de fichiers temporaires)



Impact du gros volume de données

- **Amener les traitements aux données**
- plus rapides que l'inverse pour de gros volumes sur des sites distants
- **Organiser des traitements par blocs**
- traiter des problèmes qui ne tiennent pas en RAM
- **Paralléliser efficacement les traitements**
- tâches *Map* indépendantes, tâches *Reduce* indépendantes, communications optimisées et recouvertes avec les calculs
- **Pipeliner les traitements et les communications**
- masquer les coûts des communications, réduire les temps d'execution.

+ Rendre simples les développements applicatifs

- métier de Data Scientist ≠ Expert d'informatique distribuée

⟶ **Map-Reduce est souvent un bon compromis**