OS202 - Programming Parallel Computers

TP3

9 février 2026

NOUKOUA TATMFO Maëva Sandy

2025–2026

Table des matières

1 Introduction

Le tri est une opération fondamentale en informatique, dont l'efficacité devient critique lorsque la taille des données augmente. Dans le cadre des systèmes parallèles et distribués, il est nécessaire d'exploiter le parallélisme afin de réduire le temps d'exécution.

L'objectif de ce travail est d'implémenter et de paralléliser l'algorithme *Bucket Sort* à l'aide de MPI. L'algorithme est réalisé en Python avec la bibliothèque `mpi4py`, en respectant les contraintes de l'énoncé.

2 Principe du Bucket Sort

Le *Bucket Sort* repose sur l'idée de répartir les données dans plusieurs intervalles appelés *buckets*. Chaque bucket contient les valeurs appartenant à un intervalle donné. Une fois les données réparties, chaque bucket est trié indépendamment, puis l'ensemble des buckets est concaténé pour obtenir le tableau final trié.

Cet algorithme est particulièrement adapté au parallélisme car :

— la répartition des données peut être effectuée en parallèle ;

— chaque bucket peut être trié indépendamment des autres.

3 Stratégie de parallélisation

La parallélisation repose sur le modèle MPI et suit les étapes suivantes :

1. Le processus de rang 0 génère un tableau de nombres arbitraires.
2. Le tableau est découpé et distribué aux autres processus.
3. Chaque processus participe au tri en construisant et en traitant ses buckets.
4. Les données triées sont rassemblées sur le processus 0 afin de reconstruire le tableau global trié.

Chaque processus travaille donc sur une portion distincte des données, ce qui permet un parallélisme de type *data-parallel*.

4 Implémentation

L'implémentation a été réalisée en Python en utilisant la bibliothèque `mpi4py`. Les bornes des buckets sont calculées à partir des valeurs minimales et maximales globales, obtenues via une opération de réduction MPI.

4.1 Génération et distribution des données

Le processus de rang 0 génère un tableau de nombres réels aléatoires. Ce tableau est ensuite découpé et distribué aux différents processus à l'aide de l'opération `scatter`.

4.2 Répartition en buckets

Chaque processus place ses données locales dans les buckets correspondant aux intervalles définis. Les données sont ensuite échangées entre les processus afin que chaque processus récupère les valeurs appartenant à son intervalle.

4.3 Tri local et reconstruction

Une fois les échanges terminés, chaque processus trie localement les données reçues. Le processus 0 rassemble ensuite les résultats pour reconstruire le tableau global trié.

5 Analyse d'une exécution du programme

L'exécution suivante illustre le fonctionnement complet de l'algorithme de *Bucket Sort* parallèle avec quatre processus MPI :

```
mpiexec -np 4 python bucketSort.py

Tableau initial : [227.33602247 316.75833971 797.36545733 676.25467075
                   391.1095506  332.81392787 598.30875359 186.7341856 ]
[rank 0] N=8, p=4, temps (max) = 0.000878s, tri OK ? True
Tableau trié final : [186.7341856  227.33602247 316.75833971 332.81392787
                      391.1095506  598.30875359 676.25467075 797.36545733]
```

Dans cet exemple, le programme est lancé avec `p=4` processus MPI. Conformément à l'énoncé, le processus de rang 0 génère un tableau de nombres réels arbitraires, ici de taille `N=8`. Ce tableau correspond aux données initiales à trier.

Les valeurs sont ensuite distribuées entre les différents processus, qui participent tous au tri en parallèle selon le principe du *Bucket Sort*. Chaque processus traite les valeurs appartenant à son intervalle, puis effectue un tri local.

Une fois le tri distribué terminé, le tableau global est reconstruit sur le processus de rang 0. Le tableau final affiché est correctement ordonné par ordre croissant, ce qui valide le bon fonctionnement de l'algorithme.

La ligne indiquant `tri OK ? True` confirme que le tableau final est bien trié. Le temps d'exécution affiché correspond au temps maximal mesuré parmi l'ensemble des processus, ce qui est une mesure classique en calcul parallèle.

Cet exemple, volontairement réalisé avec une petite taille de données, permet de visualiser clairement les différentes étapes de l'algorithme : génération des données, distribution, tri parallèle et reconstruction du résultat final.

6 Validation et performances

La validité du tri est vérifiée en contrôlant que le tableau final est bien ordonné. Les tests ont été réalisés avec plusieurs processus MPI.

Pour des tailles de données plus importantes, l'affichage est désactivé et seul le temps d'exécution est mesuré. L'algorithme exploite efficacement le parallélisme en répartissant le travail de tri entre les différents processus.

7 Conclusion

Ce travail a permis de mettre en œuvre une version parallèle du *Bucket Sort* à l'aide de MPI. La solution respecte les contraintes de l'énoncé et illustre clairement l'intérêt du parallélisme pour le traitement de grandes quantités de données.

Le découpage des données en buckets et le tri local indépendant constituent une approche naturelle et efficace pour une exécution distribuée. Ce TP met ainsi en évidence les principes fondamentaux de la programmation parallèle et distribuée.