Travaux dirigés n°3

Nicolas RINCON VIJA

nicolas.rincon@ensta.fr

École Nationale Supérieure de Techniques Avancées - ENSTA
Systèmes parallèles et distribués - OS02
Palaiseau, France

Année académique 2024-2025

Table des matières

Remarque préalable

Ce travail a été réalisé sur la base des résultats obtenus à partir des exécutions disponibles sur le dépôt GitHub suivant : https://github.com/Nrinconv/OS02_ENSTA.

1 Introduction

Le Bucket Sort est un algorithme de tri basé sur la distribution des éléments dans plusieurs buckets. Chaque élément est placé dans un bucket selon sa valeur, puis chaque bucket est trié individuellement. Enfin, les paniers sont concaténés pour obtenir la liste triée finale.

L'idée derrière l'utilisation de cet algorithme en parallèle est de distribuer la charge de travail entre plusieurs processus, ce qui peut accélérer le tri de grandes quantités de données.

Dans ce projet, une version parallèle du Bucket Sort a été implémentée en utilisant MPI (Message Passing Interface). L'objectif est d'analyser l'impact de la distribution des données et de la communication entre processus sur la performance du tri.

2 Algorithme du Bucket Sort en parallèle

2.1 Principe du tri par paniers

L'algorithme du Bucket Sort repose sur une approche en plusieurs étapes :

— Définition des bornes des paniers pour répartir les données.

— Distribution des nombres vers le bon panier en fonction de ces bornes.

— Chaque processus trie son propre panier.

— Les paniers triés sont rassemblés par le processus 0 pour former la liste finale triée.

L'objectif est d'utiliser plusieurs processus pour exécuter ces étapes en parallèle, afin d'améliorer les performances du tri.

2.2 Définition des bornes des paniers

Une étape essentielle de l'algorithme est la définition des bornes des paniers, car une mauvaise répartition peut entraîner un déséquilibre de charge entre les processus.

Une approche naïve consiste à diviser l'intervalle des valeurs en sous-intervalles de largeur égale :

```
1 bucket_intervals = np.linspace(0, 1, size + 1)
```

Cependant, cette méthode suppose que les valeurs sont réparties uniformément, ce qui peut entraîner des paniers vides ou déséquilibrés. Une alternative plus efficace est de calculer dynamiquement les bornes en fonction de la distribution des données :

```
1 bucket_intervals = np.percentile(data, np.linspace(0, 100, size + 1))
```

Cela permet d'obtenir des paniers contenant un nombre d'éléments plus équilibré, optimisant ainsi le temps de calcul de chaque processus.

2.3 Distribution des nombres aux processus

Une fois les bornes définies, chaque nombre est assigné au bon panier selon l'intervalle dans lequel il se situe. Cette étape est réalisée uniquement par le **processus 0**, qui distribue ensuite les paniers aux autres processus. Le processus 0 crée une liste contenant les éléments de chaque panier :

```
local_data = [[] for _ in range(size)]
for num in data:
    for i in range(size):
        if bucket_intervals[i] <= num < bucket_intervals[i + 1]:
            local_data[i].append(num)
            break
```

Puis, il envoie chaque panier au processus correspondant :

```
for i in range(1, size):
    comm.send(local_data[i], dest=i, tag=0)
```

Le processus 0 conserve son propre panier et l'exécute localement.
De leur côté, les autres processus reçoivent leurs données :

```
local_bucket = comm.recv(source=0, tag=0)
```

Ainsi, chaque processus ne travaille que sur un sous-ensemble des données globales.

2.4 Tri des paniers par chaque processus

Une fois les données reçues, chaque processus exécute un tri sur son propre panier. Une *implémentation simple du Bucket Sort* a été utilisée pour trier ces éléments localement.

```
def bucket_sort(arr):
    if len(arr) == 0:
        return arr  # Rien    trier si le panier est vide

    min_val, max_val = min(arr), max(arr)  # Bornes du panier
    bucket_count = len(arr)  # Nombre de sous-paniers
    buckets = [[] for _ in range(bucket_count)]

    for num in arr:
        index = int(bucket_count * (num - min_val) / (max_val - min_val + 1e-6))
        buckets[index].append(num)

    sorted_array = []
    for bucket in buckets:
        sorted_array.extend(sorted(bucket))  # Tri final des sous-paniers

    return sorted_array
```

Ce tri est réalisé indépendamment par chaque processus, en parallèle avec les autres.

2.5 Rassemblement des données triées

Une fois que chaque processus a trié son panier, il envoie son résultat au **processus 0** via 'MPI.gather()' :

```
sorted_data = comm.gather(sorted_local_bucket, root=0)
```

Cette commande récupère les listes triées de chaque processus et les rassemble dans un tableau *trié par blocs*, où chaque élément représente un panier trié.

2.6 Fusion des résultats dans le processus 0

Le processus 0 doit ensuite fusionner les paniers triés en une seule liste finale triée. Comme chaque panier contenait déjà des valeurs dans un ordre globalement correct, une simple concaténation garantit que la séquence est triée :

```
final_sorted_data = np.concatenate(sorted_data)
```

3 Exemple de fonctionnement

Prenons un exemple où nous avons trois processus et une liste d'entiers :

$$\{3, 0, 4, 1, 2, 5, 2, 6\}$$

Le processus 0 détermine les bornes des paniers en fonction de la distribution des valeurs. Supposons que les percentiles produisent les intervalles suivants :

$$[0\text{-}1],\ [2\text{-}3],\ [4\text{-}6]$$

Les nombres sont distribués aux processus en fonction de ces bornes :

— Processus 0 : $\{0, 1\}$

— Processus 1 : $\{2, 2, 3\}$

— Processus 2 : $\{4, 5, 6\}$

Chaque processus trie ses nombres localement :

— Processus 0 : $\{0, 1\}$

— Processus 1 : $\{2, 2, 3\}$

— Processus 2 : $\{4, 5, 6\}$

Ensuite, les résultats sont rassemblés :

$$\{0, 1, 2, 2, 3, 4, 5, 6\}$$

L'utilisation des percentiles permet une répartition équilibrée des données entre les processus, assurant un tri plus efficace.

4 Résultats et analyse des performances

Pour évaluer la performance du tri par paniers en parallèle, nous avons mesuré le temps d'exécution total en fonction du nombre de processus pour deux tailles de données :

— $N = 100$: Petit jeu de données pour observer l'impact du parallélisme sur un faible volume de données.

— $N = 1000000$: Grande taille de données pour mieux exploiter le parallélisme.

Le code a été exécuté avec différentes valeurs de p (nombre de processus) :

```
mpirun -np p python bucket_sort_parallel.py
```

4.1 Temps d'exécution et accélération

Les tableaux ci-dessous présentent les temps d'exécution obtenus pour $N = 100$ et $N = 1000000$.

Nombre de processus (p)	Temps d'exécution (s)	Speedup
1	0.001752	1.00
2	0.001971	0.89
4	0.012166	0.14
8	0.021198	0.08
16	0.065796	0.03

TABLE 1 – Temps d'exécution et accélération pour $N = 100$

Le speedup est calculée comme suit :

$$S(p) = \frac{T(1)}{T(p)} \tag{1}$$

où $T(1)$ est le temps d'exécution en séquentiel et $T(p)$ est le temps d'exécution avec p processus.

Nombre de processus (p)	Temps d'exécution (s)	Accélération (Speedup)
1	5.431597	1.00
2	5.296733	1.03
4	5.170709	1.05
8	13.110076	0.41
16	34.686297	0.16

TABLE 2 – Temps d'exécution et accélération pour $N = 1000000$

4.2 Analyse des performances et impact de l'architecture

Les résultats montrent que l'augmentation du nombre de processus ne réduit pas nécessairement le temps d'exécution, en particulier pour un nombre élevé de processus. Pour $N = 100$, le faible volume de données fait quele coût de communication dépasse les bénéfices du parallélisme.

Pour $N = 1000000$, le parallélisme semble initialement prometteur (de $p = 1$ à $p = 4$), mais les performances se détériorent rapidement pour $p = 8$ et $p = 16$. Cela peut être expliqué par plusieurs facteurs :

— **Overhead de communication** : Lorsque le nombre de processus augmente, la communication via MPI devient plus coûteuse que l'exécution locale.

— **Nombre de cœurs physiques limité** : L'architecture testée possède 4 cœurs physiques et 8 threads via l'hyper-threading. L'ajout de processus au-delà de ces limites ne profite pas réellement aux performances.

— **Problème de charge** : Pour $p = 8$ et $p = 16$, chaque processus a très peu de données à traiter, rendant le temps d'attente et la synchronisation plus coûteux que le calcul lui-même.

5 Conclusion

Nous avons observé que :

— Pour des petites tailles de données ($N = 100$), le parallélisme est contre-productif en raison du coût de communication.

— Pour $N = 1000000$, le parallélisme est bénéfique jusqu'à $p = 4$, mais au-delà, l'overhead de communication devient dominant.

— L'architecture processeur impacte fortement la scalabilité : ici, un processeur avec 4 cœurs physiques et 8 threads limite l'efficacité du parallélisme.

L'implémentation du Bucket Sort en parallèle avec MPI permet d'améliorer les performances du tri en distribuant les données entre plusieurs processus. Cependant, l'efficacité dépend fortement de la répartition des paniers. L'utilisation des percentiles pour définir les intervalles permet d'obtenir une charge de travail plus équilibrée.

Nous avons observé une accélération significative en augmentant le nombre de processus, mais une saturation apparaît lorsque le coût de communication devient trop élevé. Une analyse plus détaillée avec des tailles de données plus grandes et différents algorithmes de tri local pourrait être un axe d'amélioration futur.