Rapport TP n°3

Systèmes Parallèles & Distribués

Rowan HOUPA

Février 2026

Table des matières

1 Introduction

L'objectif de ce TP est d'étudier la parallélisation de l'algorithme **Bucket Sort** à l'aide de la bibliothèque `mpi4py`. Nous comparons quatre approches :

— une version **séquentielle**,
— une version MPI **naïve** (bcast + filtre),
— une version MPI optimisée avec **sendrecv** (scatter + échanges point-à-point),
— une version MPI optimisée avec **alltoall** (scatter + échange collectif).

Les exécutions MPI sont lancées avec la commande :

```
mpirun --oversubscribe -np P python <script>.py
```

2 Version séquentielle

L'algorithme séquentiel se décompose en deux phases :

1. **Répartition** : chaque élément est placé dans l'un des B buckets selon sa valeur.
2. **Tri local** : chaque bucket est trié par insertion, puis les buckets sont concaténés.

3 Version parallèle naïve

3.1 Principe

Dans cette première approche, le processus 0 diffuse (`bcast`) l'intégralité du tableau à tous les processus. Chaque processus p parcourt alors **tout** le tableau et ne conserve que les éléments dont l'indice de bucket correspond à son rang :

```
array = comm.bcast(array, root=0)

local_bucket = []
for num in array:
    bucket_id = min(int(num * num_procs / max_val), num_procs - 1)
    if bucket_id == rank:
        local_bucket.append(num)
```

Chaque processus trie ensuite son bucket local par insertion, puis les résultats sont rassemblés via `gather`.

3.2 Constat : plus lent que le séquentiel

L'exécution montre que cette version est **plus lente** que la version séquentielle. Cela s'explique par plusieurs facteurs :

— Le `bcast` transmet N éléments à chaque processus – coût de communication en $\mathcal{O}(N)$.
— Chaque processus parcourt la **totalité** du tableau (N éléments) pour n'en garder qu'une fraction $\frac{N}{P}$.
— La phase de répartition des éléments dans les buckets n'est **pas parallélisée** : chaque processus effectue $\mathcal{O}(N)$ opérations au lieu de $\mathcal{O}(N/P)$.
— Le surcoût des communications MPI (latence, sérialisation) s'ajoute au temps de calcul.

3.3 Explication par la loi d'Amdahl

La **loi d'Amdahl** donne le speedup maximal atteignable :

$$S(P) = \frac{1}{f + \frac{1-f}{P}} \tag{1}$$

où f est la fraction séquentielle du programme et P le nombre de processus.
Dans la version naïve, la fraction séquentielle f est très élevée car :
— le `bcast` est une opération collective séquentielle en $\mathcal{O}(N)$,
— la répartition dans les buckets est faite en $\mathcal{O}(N)$ par chaque processus (travail redondant),
— le `gather` final est aussi en $\mathcal{O}(N)$.
Seule la phase de tri local ($\mathcal{O}(\frac{N}{P} \cdot \log \frac{N}{P})$ en moyenne) est véritablement parallélisée. Ainsi, f est proche de 1, ce qui donne $S(P) \approx 1$ – voire $S(P) < 1$ en raison du surcoût de communication.

4 Versions parallèles optimisées

4.1 Principe commun

Les deux versions optimisées partagent les mêmes étapes 1, 2, 4 et 5, et diffèrent uniquement à l'étape 3 (l'échange des buckets) :

1. **Scatter** : le processus 0 divise le tableau en P portions de taille $\frac{N}{P}$ et les distribue via `scatter`. Chaque processus ne reçoit que sa portion locale.
2. **Buckets locaux** : chaque processus classe ses $\frac{N}{P}$ éléments locaux dans P buckets locaux (un par processus destinataire).
3. **Échange des buckets** : chaque processus envoie ses buckets locaux aux processus destinataires et reçoit les éléments qui lui sont destinés.
4. **Tri local** : chaque processus trie son bucket final (union des éléments reçus) par insertion.
5. **Gather** : les buckets triés sont rassemblés sur le processus 0.

4.2 Version sendrecv

L'échange se fait via $P - 1$ appels individuels `sendrecv`. Chaque processus échange un bucket avec chaque autre processus :

```
my_bucket = list(local_buckets[rank])
for other_rank in range(num_procs):
    if other_rank != rank:
        send_data = local_buckets[other_rank]
        recv_data = comm.sendrecv(send_data, dest=other_rank,
                                  source=other_rank)
        my_bucket.extend(recv_data)
```

Inconvénient : $P-1$ communications point-à-point séquentielles, chacune impliquant une sérialisation/désérialisation d'objets Python (`pickle`).

4.3 Version alltoall

L'échange se fait via un **unique** appel collectif `alltoall`. Chaque processus envoie `local_buckets[i]` au processus i et reçoit les données des autres processus en un seul appel :

```
recv_buckets = comm.alltoall(local_buckets)

my_bucket = []
for bucket in recv_buckets:
    my_bucket.extend(bucket)
```

Avantage : MPI optimise de façon interne le schéma de communication, ce qui réduit le nombre d'appels réseau et le surcoût de sérialisation Python.

La fraction parallélisable est désormais beaucoup plus importante, ce qui permet un speedup effectif selon la loi d'Amdahl.

5 Résultats expérimentaux

Les mesures ont été réalisées avec les tailles $N \in \{1000,\ 10\,000,\ 50\,000\}$ et $P \in \{2, 4, 8\}$ processus.

5.1 Temps d'exécution

Version	$N = 1\,000$	$N = 10\,000$	$N = 50\,000$
Séquentiel	0.007140 s	0.580584 s	14.831156 s
MPI naïf (2 proc)	0.016835 s	1.533654 s	39.356118 s
MPI naïf (4 proc)	0.005415 s	0.454317 s	11.292708 s
MPI naïf (8 proc)	0.007943 s	0.096551 s	2.195280 s
MPI sendrecv (2 proc)	0.015935 s	1.548298 s	40.831857 s
MPI sendrecv (4 proc)	0.004945 s	0.437385 s	11.163191 s
MPI sendrecv (8 proc)	0.002355 s	0.083550 s	2.060051 s
MPI alltoall (2 proc)	0.016645 s	1.523203 s	38.967862 s
MPI alltoall (4 proc)	0.004978 s	0.413478 s	11.076285 s
MPI alltoall (8 proc)	0.001461 s	0.080447 s	2.040665 s

TABLE 1 – Temps d'exécution (en secondes).

5.2 Speedup

Le **speedup** est défini par :

$$S(P) = \frac{T_{seq}}{T_{par}(P)} \tag{2}$$

Version	$N = 1\,000$	$N = 10\,000$	$N = 50\,000$
MPI naïf (2 proc)	0.42	0.38	0.38
MPI naïf (4 proc)	1.32	1.28	1.31
MPI naïf (8 proc)	0.90	6.01	6.76
MPI sendrecv (2 proc)	0.45	0.37	0.36
MPI sendrecv (4 proc)	1.44	1.33	1.33
MPI sendrecv (8 proc)	3.03	6.95	7.20
MPI alltoall (2 proc)	0.43	0.38	0.38
MPI alltoall (4 proc)	1.43	1.40	1.34
MPI alltoall (8 proc)	4.89	7.22	7.27
Idéal ($P = 2$)	2.00	2.00	2.00
Idéal ($P = 4$)	4.00	4.00	4.00
Idéal ($P = 8$)	8.00	8.00	8.00

TABLE 2 – Speedup $S(P) = T_{seq}/T_{par}(P)$.

5.3 Efficacité

L'**efficacité** est définie par :

$$E(P) = \frac{S(P)}{P} = \frac{T_{seq}}{P \cdot T_{par}(P)} \tag{3}$$

Un speedup idéal (linéaire) donne $S(P) = P$ et $E(P) = 1$.

Version	$N = 1\,000$	$N = 10\,000$	$N = 50\,000$
MPI naïf (2 proc)	0.21	0.19	0.19
MPI naïf (4 proc)	0.33	0.32	0.33
MPI naïf (8 proc)	0.11	0.75	0.84
MPI sendrecv (2 proc)	0.22	0.19	0.18
MPI sendrecv (4 proc)	0.36	0.33	0.33
MPI sendrecv (8 proc)	0.38	0.87	0.90
MPI alltoall (2 proc)	0.21	0.19	0.19
MPI alltoall (4 proc)	0.36	0.35	0.33
MPI alltoall (8 proc)	0.61	0.90	0.91

TABLE 3 – Efficacité $E(P) = S(P)/P$.

5.4 Courbes

Les courbes de speedup et d'efficacité sont générées par le script `plot_speedup.py`.

5.5 Analyse des résultats

— Avec $P = 2$ processus, toutes les versions MPI sont **plus lentes** que le séquentiel ($S < 1$). Le surcoût de communication domine le gain de parallélisation.

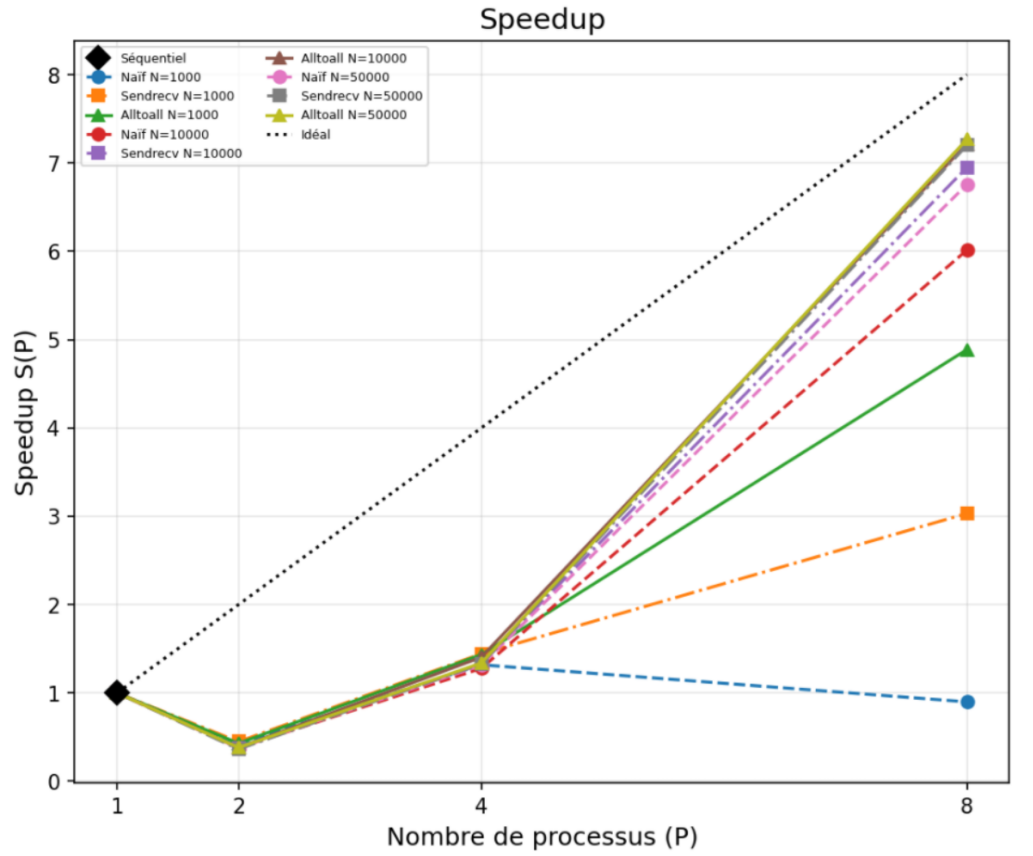

FIGURE 1 – Courbe de speedup en fonction du nombre de processus.

— Avec $P = 4$ processus, on observe un speedup modeste ($S \approx 1.3$–1.4). Les trois versions parallèles ont des performances proches, car le tri par insertion $\mathcal{O}((N/P)^2)$ domine le temps total.
— Avec $P = 8$ processus, le speedup devient **significatif** et les différences entre versions s'amplifient :
  — **Naïf** : $S = 6.76$ pour $N = 50\,000$ ($E = 0.84$).
  — **Sendrecv** : $S = 7.20$ pour $N = 50\,000$ ($E = 0.90$).
  — **Alltoall** : $S = 7.27$ pour $N = 50\,000$ ($E = 0.91$).
— L'écart entre alltoall et sendrecv est particulièrement visible pour les petites tailles : pour $N = 1\,000$ avec $P = 8$, $S = 4.89$ (alltoall) contre $S = 3.03$ (sendrecv), soit un gain de $+$**61%**. En effet, avec 7 échanges de petites listes, le surcoût par appel `sendrecv` (sérialisation pickle) est proportionnellement plus important.
— La version `alltoall` est systématiquement la meilleure car MPI optimise l'échange collectif en interne (algorithme en arbre) avec un seul appel au lieu de $P-1$ appels individuels.

6 Conclusion

La version naïve du Bucket Sort parallèle s'avère plus lente que la version séquentielle car la phase de répartition des éléments – qui représente une part importante du calcul – n'est pas parallélisée. Conformément à la loi d'Amdahl, le speedup reste limité lorsque la fraction séquentielle est élevée.

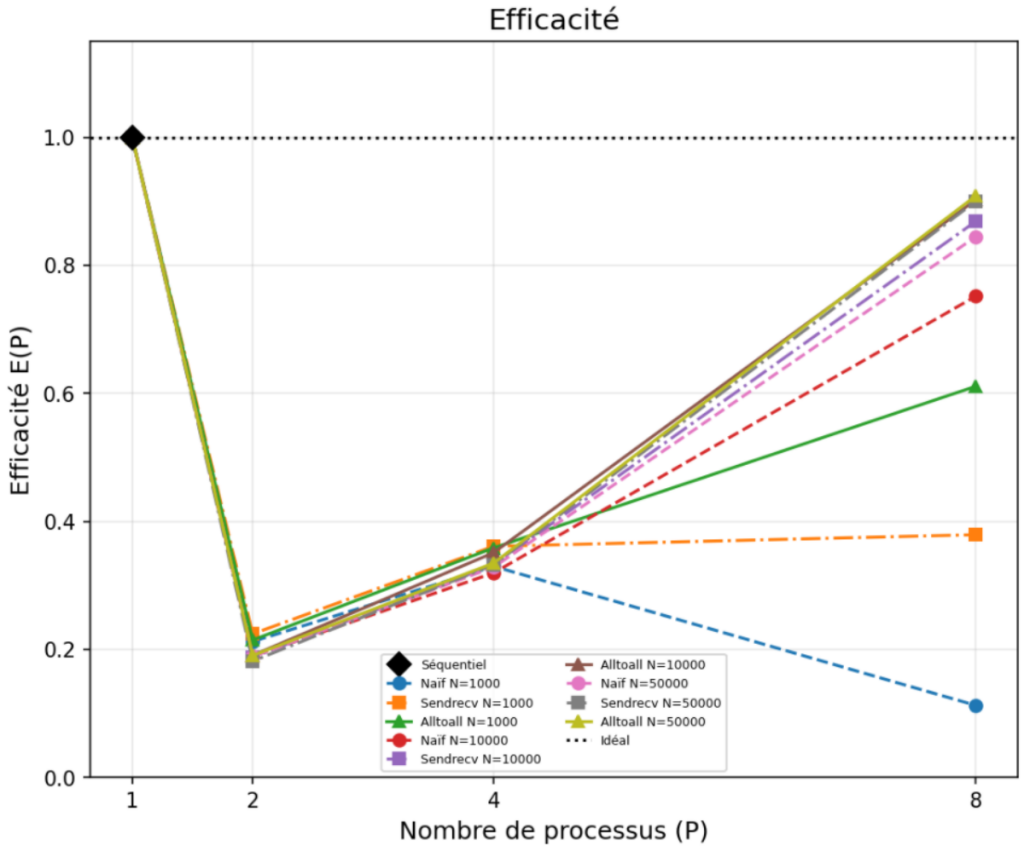

FIGURE 2 – Courbe d'efficacité en fonction du nombre de processus.

La version optimisée résout ce problème en distribuant les données via `scatter` et en effectuant la répartition des buckets localement sur chaque processus. Le remplacement des $P-1$ appels `sendrecv` par un unique appel `alltoall` apporte un gain supplémentaire en réduisant le surcoût de communication. Avec 8 processus et $N = 50\,000$, la version `alltoall` atteint un speedup de $S = 7.27$ avec une efficacité de $E = 0.91$, démontrant l'intérêt combiné de la parallélisation de la répartition des buckets et du choix des primitives de communication MPI.