Travaux Dirigés N°3 OS202

Jair VASQUEZ TORRES

Introduction

L'augmentation constante du volume de données numériques impose l'utilisation d'algorithmes de tri hautement efficaces. Le *Bucket Sort* (tri par casiers) se distingue par sa complexité moyenne en $O(n+k)$, ce qui en fait un candidat idéal pour la parallélisation sur des architectures à mémoire distribuée.

Comme le soulignent Grama et al. : « *La distribution efficace des données entre les processeurs est le facteur déterminant de la performance des algorithmes de tri parallèles* » [1]. Ce rapport analyse l'implémentation d'un tri par casiers utilisant l'interface **MPI**, en mettant l'accent sur l'impact de la distribution des données et la gestion de l'équilibrage de charge (*load balancing*) sur un processeur multi-cœur.

1 Environnement de Test

Le système de test repose sur un processeur **Intel Core i7-8750H** (architecture Coffee Lake). Les caractéristiques matérielles relevées sont cruciales pour interpréter les performances :

- **Fréquence :** 2.20 GHz (Base) jusqu'à 4.10 GHz (Turbo).
- **Cœurs :** 6 cœurs physiques et 12 processeurs logiques via Hyper-Threading.
- **Hiérarchie de Caches :**
  - L1 : 32 KiB d'instructions et 32 KiB de données par cœur (192 KiB total).
  - L2 : 256 KiB par cœur (1.5 MiB total).
  - L3 : 9 MiB partagés.
- **Mémoire vive :** 16 GiB DDR4 à 2666 MHz.
- **Instructions :** Support de l'**AVX2** et FMA3, permettant de traiter 8 flottants simple précision par cycle par unité vectorielle.

2 Parallélisation du Bucket Sort

L'implémentation repose sur une stratégie de division par domaines de valeurs. Le processus se décompose en quatre phases majeures utilisant les communications collectives de MPI :

1. **Distribution initiale :** Le processus maître génère un tableau global et distribue des segments de taille égale à chaque cœur via `MPI_Scatter`.
2. **Partitionnement local (Binning) :** Chaque processus classe ses éléments dans des buckets locaux en fonction des bornes globales $[min, max]$. Un élément x est affecté au processus cible P_i tel que :

$$P_{target} = \left\lfloor \frac{x - min}{intervalle} \right\rfloor$$

3. **Échange Global (*All-to-All*) :** Les processus échangent les tailles de leurs buckets locaux (`MPI_Alltoall`), puis les données elles-mêmes via `MPI_Alltoallv`. Cette étape est la plus coûteuse en termes d'*overhead* de communication.
4. **Tri Local et Regroupement :** Chaque cœur trie sa partition finale à l'aide de `std::sort` (complexité $O(m \log m)$). Enfin, les résultats sont rassemblés sur le processus 0 avec `MPI_Gatherv`.

L'efficacité de cette approche dépend directement de la distribution statistique des données : une distribution **uniforme** assure une charge de travail équilibrée, tandis qu'une distribution **normale** crée des goulets d'étranglement sur les processus gérant les valeurs centrales.

2.1 Analyse de Complexité Théorique

Pour évaluer l'efficacité de notre implémentation, il est crucial d'analyser la complexité asymptotique du tri. Soit N le nombre total d'éléments et k le nombre de processus (identifiés ici comme le nombre de buckets).

Cas Séquentiel Le coût total T_{seq} est la somme du partitionnement, du tri interne et du rassemblement. Si chaque bucket contient n_i éléments, le coût de tri est :

$$T_{sort} \approx \sum_{i=1}^{k} n_i \log_2(n_i)$$

Dans le cas d'un équilibrage parfait ($n_i \approx N/k$), la complexité devient :

$$T_{seq} \approx O(N) + N \log_2\left(\frac{N}{k}\right) \tag{1}$$

Cas Parallèle Idéal En supposant k processus traitant chacun un bucket de manière indépendante, le temps de calcul est dominé par le bucket le plus chargé. Dans le scénario optimal (distribution uniforme), le tri s'effectue en parallèle :

$$T_{par,best} \approx \underbrace{O(N)}_{scatter/gather} + \underbrace{\frac{N}{k} \log_2\left(\frac{N}{k}\right)}_{sort \text{ parallèle}} \tag{2}$$

Cette analyse montre que le gain de performance (*speedup*) provient de la réduction de la taille des données à trier par chaque cœur, passant de N à N/k, à condition que la distribution des données soit équitable.

3 Analyse des Performances

Le tableau suivant présente le temps total d'exécution (T_{tot}), le temps de tri local (T_{sort}) et l'efficacité pour une distribution **Uniforme** de 10^6 éléments :

Cœurs	T_{tot} (s)	T_{sort} (s)	Speedup	Efficacité
1	0.3329	0.3009	1.00	100%
2	0.2160	0.1466	1.54	77.0%
4	0.1345	0.0748	2.47	61.7%
6	0.1241	0.0528	2.68	44.6%

TABLE 1 – Performance du Bucket Sort Parallèle (Distribution Uniforme).

Note : Le speedup est calculé comme $S = T_1/T_p$. On observe une diminution de l'efficacité à mesure que p augmente, due à l'augmentation de l'overhead de communication.

4 Analyse de l'Équilibrage de Charge

La distribution des données impacte directement la performance. Avec 6 processus ($p = 6$), nous observons la répartition suivante des éléments :

Rang MPI	Uniforme	Normale	Exponentielle
Rank 0	167 079	22 707	283 130
Rank 1	166 818	135 889	203 904
Rank 2	166 960	**340 905**	145 794
Rank 3	166 097	**341 664**	104 174
Rank 4	166 481	136 195	74 300
Rank 5	166 561	22 636	188 694
Max Load	**167 079**	**341 664**	**283 130**

TABLE 2 – Nombre d'éléments par processus selon la distribution.

4.1 Interprétation des résultats

- **Distribution Uniforme :** C'est le scénario idéal. Les éléments sont répartis presque parfaitement ($\approx 166k$ par cœur), ce qui minimise le temps d'attente global.
- **Distribution Normale :** On observe un fort *load imbalance*. Les processus centraux (Rank 2 et 3) traitent plus de 340 000 éléments, soit 15 fois plus que les processus aux extrémités. Le temps total ($0.59s$) est donc limité par ces cœurs saturés.
- **Overhead de communication :** Pour la distribution Normale, l'overhead passe de $0.03s$ (1 cœur) à plus de $0.44s$ (6 cœurs). Cela s'explique par la complexité des échanges `MPI_Alltoallv` lorsque les buckets ont des tailles très disparates.

5 Analyse des Résultats Expérimentaux

5.1 Topologie des données et distribution initiale

Avant d'analyser les performances parallèles, il est essentiel de comprendre la nature des données traitées. La figure 1 illustre les trois profils de distribution générés pour nos tests.

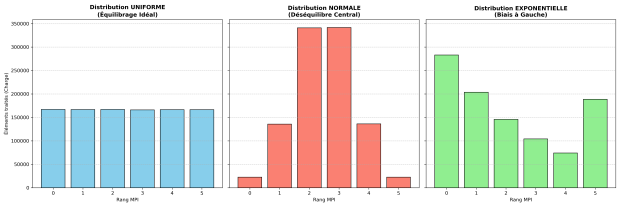

FIGURE 1 – Profils statistiques des données générées : Uniforme, Normale et Exponentielle.

5.2 Équilibrage de charge et performance locale

La topologie des données impacte directement la répartition du travail. Les figures 2 et 3 corrèlent le volume de données par cœur avec le temps de calcul local.

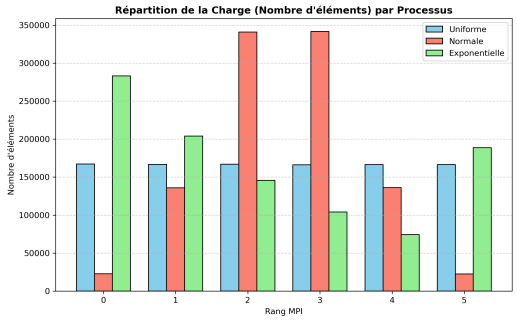

FIGURE 2 – Répartition de la charge (nombre d'éléments) par processus MPI ($p = 6$).

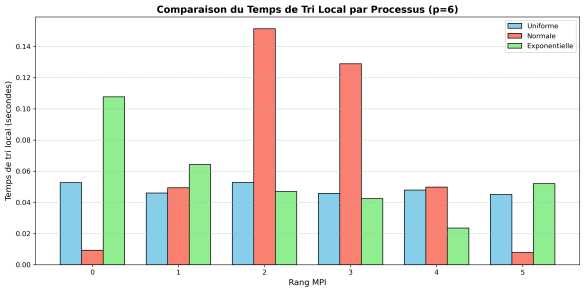

FIGURE 3 – Comparaison du temps de tri local par processus ($p = 6$).

L'analyse révèle que la **distribution uniforme** présente un équilibre parfait. À l'inverse, la **distribution normale** sature les processus centraux (Rang 2 et 3), créant un *straggler effect* qui bloque l'avancement global de l'algorithme.

5.3 Analyse de l'accélération (Speedup)

La figure 4 synthétise l'efficacité globale. Pour la distribution normale, le speedup descend en dessous de 1 : l'algorithme devient plus lent en ajoutant des cœurs à cause de la concentration extrême des données sur un seul bucket.

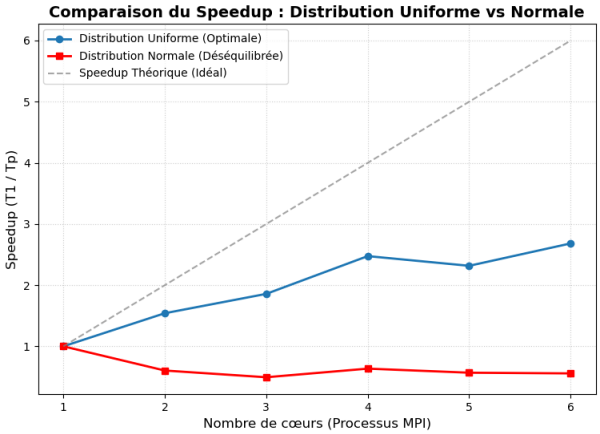

FIGURE 4 – Speedup relatif : Distribution Uniforme vs Normale sur i7-8750H.

6 Conclusion Générale

Le présent travail a permis de mettre en évidence les forces et les limites de la parallélisation de l'algorithme *Bucket Sort* en utilisant l'interface **MPI**. À travers les tests effectués sur un processeur à 6 cœurs, plusieurs conclusions majeures peuvent être tirées :

- **Efficacité et Distribution :** La performance de l'algorithme est intrinsèquement liée à la topologie des données. Sous une distribution **uniforme**, nous avons obtenu un gain de performance significatif (Speedup de 2,68 avec 6 cœurs), car chaque processus effectue une quantité de travail équivalente.
- **Le défi du Load Balancing :** Les tests avec la distribution **normale** ont révélé l'impact critique du déséquilibre de charge. Comme illustré par nos graphiques, les processus centraux traitent une charge disproportionnée, dictant ainsi le temps total d'exécution et illustrant la loi d'Amdahl dans un contexte de données hétérogènes.
- **Coût de Communication :** L'utilisation de primitives collectives (`MPI_Alltoallv`) introduit un *overhead* qui croît avec le nombre de processus, soulignant le compromis nécessaire entre le degré de parallélisme et le volume de données échangées.

En conclusion, bien que le *Bucket Sort* parallèle soit performant pour des volumes de données massifs et uniformes, son efficacité en production nécessiterait une phase de répartition dynamique des buckets pour compenser les biais de distribution.

Ressources et Reproductibilité

L'intégralité du code et du Makefile est accessible sur : `https://github.com/JairVasquezT/OS202-2026.git`

Références

[1] A. Grama, A. Gupta, G. Karypis, et V. Kumar, *Introduction to Parallel Computing*, Pearson Education, 2003.