Comparaison entre deux implémentations de génération et traitement de grilles de valeurs

Emmanuel OSERET
Université de Versailles Saint-Quentin-en-Yvelines, France

20 février 2025

Table des matières

1 Introduction

1.1 Contexte et Objectifs

Ce rapport présente une comparaison entre deux versions d'un programme dont le but est de générer pseudo-aléatoirement des paires de valeurs `(v1,v2)` dans une grille 2D et d'en extraire les valeurs maximales. L'analyse porte sur divers aspects tels que la performance, la gestion mémoire, la lisibilité et la robustesse. La première version (code initial) utilise un fichier texte et des allocations multiples, tandis que la version optimisée exploite un fichier binaire et des allocations contiguës.

1.2 Méthodologie

La comparaison s'effectue en analysant les structures de données utilisées, les techniques d'I/O, les algorithmes de recherche des maximums et la gestion de la mémoire. Chaque section du rapport détaille ces points pour chacune des implémentations.

2 Présentation du Code Initial

2.1 Structures de Données

Dans le code initial, les données sont stockées à l'aide des structures suivantes :
— `value_t` : Représente une paire de valeurs `v1` et `v2`.
— `value_grid_t` : Contient un tableau de pointeurs vers des instances de `value_t`, chaque valeur étant allouée séparément.
— `pos_val_t` et `pos_val_grid_t` : Associent chaque paire à ses coordonnées dans la grille.

2.2 Gestion de la Mémoire

Le code effectue de nombreuses allocations dynamiques via `malloc` et `realloc` pour chaque valeur chargée. Un suivi global de la mémoire allouée est assuré par la variable `sum_bytes`.

2.3 Lecture et Écriture des Données

La génération et la sauvegarde des données se font dans un fichier texte. Les fonctions `fprintf` et `sscanf` sont utilisées pour l'écriture et la lecture, respectivement, ce qui implique une opération de parsing coûteuse pour de grands volumes de données.

2.4 Recherche des Maximums

Pour trouver le maximum, le code utilise la fonction `qsort` afin de trier le tableau de structures par valeur, puis récupère le dernier élément du tableau trié. Cette approche a une complexité en $\mathcal{O}(n \log n)$.

3 Présentation du Code Optimisé

3.1 Structures de Données

La version optimisée modifie les structures de manière à :
— Utiliser des tableaux contigus pour stocker directement les instances de `value_t` et `pos_val_t`.
— Réduire le nombre d'allocations dynamiques et améliorer la localité mémoire.

3.2 Gestion de la Mémoire

L'allocation contiguë dans la version optimisée permet de réduire la surcharge due aux appels répétés à `malloc` et minimise le risque de fragmentation. Le suivi de la mémoire est maintenu par `sum_bytes`.

3.3 Lecture et Écriture des Données

La version optimisée opte pour l'utilisation d'un fichier binaire au lieu d'un fichier texte. Les fonctions `fwrite` et `fread` permettent une lecture et une écriture plus rapide en évitant le parsing et en diminuant la taille des fichiers générés.

3.4 Recherche des Maximums

Au lieu de trier le tableau complet, la version optimisée parcourt les données via une simple boucle pour trouver le maximum, ce qui réduit la complexité à $\mathcal{O}(n)$.

4 Analyse Comparative

4.1 Performance

- **I/O :** Le passage du fichier texte au fichier binaire améliore considérablement la vitesse de lecture et d'écriture.
- **Algorithme de Recherche :** L'utilisation d'une boucle linéaire pour trouver le maximum dans le code optimisé est plus efficace que le tri complet du code initial.

4.2 Gestion Mémoire

- **Code Initial :** Multiples allocations individuelles entraînent une surcharge et un risque de fragmentation.
- **Code Optimisé :** L'allocation contiguë permet une meilleure utilisation de la mémoire et améliore la localité, réduisant ainsi les coûts d'accès.

4.3 Lisibilité et Maintenabilité

La version optimisée, par sa simplification des structures et de l'algorithme de recherche, offre un code plus lisible et plus facile à maintenir.

4.4 Robustesse et Sécurité

Les deux versions intègrent des vérifications d'erreurs (ouverture des fichiers, allocations réussies). Toutefois, la version optimisée présente une robustesse accrue grâce à la simplification de la gestion des ressources.

5 Conclusion

La comparaison démontre que la version optimisée offre des améliorations significatives en termes de performance et de gestion mémoire. L'utilisation d'un fichier binaire et d'allocations contiguës, ainsi que le remplacement du tri complet par une boucle linéaire pour la recherche des maximums, permettent de réduire le temps d'exécution et d'optimiser l'utilisation de la mémoire. Ces optimisations rendent le code plus adapté aux applications manipulant de grands volumes de données.

6 Annexes et Références

6.1 Extraits de Code

Les extraits suivants illustrent les différences majeures entre les deux implémentations :

Génération et Sauvegarde des Données

- **Code Initial :** Utilisation de `fprintf` pour écrire dans un fichier texte.
- **Code Optimisé :** Utilisation de `fwrite` pour écrire dans un fichier binaire.

Recherche des Maximums

— **Code Initial :** Tri complet avec `qsort` (complexité en $\mathcal{O}(n \log n)$).
— **Code Optimisé :** Parcours linéaire (complexité en $\mathcal{O}(n)$).

6.2 Références

— Documentation C standard (pour `fread`, `fwrite`, `malloc`, etc.).
— Articles et guides sur l'optimisation de la gestion mémoire et des performances en C.