PROG
TP#7

2022-W43–46

Instructions de rendu

Ce TP fait l'objet d'un rendu *en binôme*. Vous devez envoyer votre travail sous la forme d'une archive à philip.scales@univ-grenoble-alpes.fr, au plus tard le 2022-11-18 à 18 :00. Celle-ci doit contenir :

— Votre programme répondant aux questions ci-dessous. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
— Un Makefile et les instructions permettant de l'utiliser.
— Un compte-rendu détaillé en format PDF qui décrit vos choix de conception et vos réponses aux questions.

Première partie : quicksort

Le but de cet exercice est d'implémenter l'algorithme de tri *quicksort*, d'abord sur un tableau d'entiers puis pour des structures de donnée arbitraires dotées d'une fonction de comparaison.

On commence par rappeler l'algorithme de partition en ***pseudocode***, l'algorithme complet pouvant s'obtenir par une série d'appels récursifs à ce dernier.

```
/****************************************
input: T
input: d, f, x = T[f]
output: n, la nouvelle position de x
dans le tableau modifié en place t.q.
T[d..n-1] <= x; T[n+1..f] > x
****************************************/
x = T[f];
n = f;
for (i = f - 1; i >= d; i--)
{
        if (T[i] > x)
        {
                T[n] = T[i];
                T[i] = T[n-1];
                n = n - 1;
        }
}
T[n] = x;
```

Remarque : Lors de l'implémentation de cet algorithme, faites particulièrement attention aux bornes sur `T` et vos variables de boucles ; une utilisation préventive d'ASan et de Valgrind permet de détecter de nombreux bugs classiques.

Q.1 : Écrivez une fonction `int sorted(size_t Tlen, const uint32_t T[Tlen])` qui teste si un tableau `T` de `Tlen` `uint32_t` est trié.

Q.2 :

1. Écrivez une fonction `void qsortu(size_t Tlen, uint32_t T[Tlen])` qui trie en place un tableau `T` de `Tlen` `uint32_t` en utilisant quicksort.
2. Testez votre fonction. Ces tests devront au minimum utiliser `sorted`, sur des tableaux remplis aléatoirement. Pourquoi l'utilisation de `sorted` ne permet elle pas de détecter tous les bugs possibles dans `qsortu` ?
3. Évaluez expérimentalement le coût de votre implémentation en mesurant le temps d'exécution sur des tableaux de taille variable (par ex. de quelques millions à quelques centaines de millions d'éléments remplis aléatoirement. Faites attention à tirer vos éléments aléatoires (par ex. uniformément) dans un ensemble suffisamment grand par rapport à la taille du tableau (pourquoi ?), et à exclure le temps de génération des tableaux de vos mesures. Vous pourrez par exemple utiliser la fonction `gettimeofday` pour une mesure raisonnablement précise du temps d'exécution.

Q.3 :

1. Que pouvez-vous prévoir sur le temps d'exécution de votre tri sur un tableau dont les éléments sont initialement strictement décroissants ? Vérifiez le expérimentalement sur de petits tableaux de tailles variables.
2. Modifiez votre fonction de partition afin de résoudre ce problème en moyenne. Une solution suffisante est d'échanger `T[f]` avec un élément au hasard (uniforme) de `T[d..f]`, puis de procéder comme auparavant.
3. Testez l'effet de votre changement sur le cas problématique précédent.

Q.4 :

1. Écrivez une nouvelle fonction `qsortg` basée sur votre fonction `qsortu` pour qu'elle offre le même prototype et comportement que la fonction `qsort` de la bibliothèque standard. Vous pourrez par exemple utiliser `memcpy` pour effectuer les affectations nécessaires.

   Remarque : La logique de `qsortg` est identique à `qsortu` (et de même pour leurs fonctions de partition respectives) ; il y a donc peu de changements à faire à `qsortu` pour obtenir `qsortg`.
2. Comparez les performances de vos deux implémentations (`qsortu` et `qsortg`, toutes deux avec un choix aléatoire du pivot) ainsi que celle de la bibliothèque standard pour des tableaux d'`uint32_t`. D'où peut selon vous provenir une éventuelle différence ?

Seconde partie : radixsort

On souhaite maintenant écrire une fonction `radixsortu` qui implémente le tri par base pour des entiers de 32 bits, en utilisant un tri par dénombrement comme sous-routine. L'idée du tri par base est d'écrire les entrées en base 2^b, avec b un petit entier, et d'utiliser un tri *stable*[1] pour trier ces entiers d'abord suivant le chiffre (en base 2^b) de poids faible (c-à-d suivant les b bits de poids faible de l'entier), puis le chiffre suivant, etc. Le ***pseudocode*** d'un tel tri est le suivant :

```
/*****************************************
input: nlen, b
input: T, un tableau de nombres entiers non
signés dont les valeurs peuvent être
représentées avec nlen*b bits (autrement dit,
les éléments de T ont nlen chiffres en
base 2^b)
output: Un nouveau tableau contenant les
éléments de T triés
*****************************************/
T1 = T
T2 = T
for (int i = 0; i < nlen; i++)
{
        denomsort(T1, T2, b, i);
        swap(T1, T2);
}
return T1;
```

1. On rappelle qu'un tri est *stable* si l'ordre relatif des éléments égaux reste inchangé dans le tableau trié par rapport au tableau non trié.

Ici, `denomsort` est un tri par dénombrement qui trie les entrées de `T1` en considérant seulement le $i^{\text{ème}}$ chiffre en base 2^b (ou dit autrement, les bits de position $i \times b \dots (i+1) \times b - 1$) des éléments, et écrit la sortie dans `T2`.

Q.5 :

1. Écrivez une fonction `denomsort` qui implémente un tri par dénombrement pour des entiers stockés sur 32 bits. Faites particulièrement attention à son prototype, et réfléchissez bien (étant donné son usage au sein de `radixsortu`) à ses arguments et à votre politique d'allocation/désallocation mémoire. Vous pouvez ici ajouter autant d'arguments que vous souhaitez par rapport à la version appelée dans le pseudocode de `radixsortu`. Expliquez vos choix de conception pour cette fonction.
2. Testez votre fonction pour de petites valeurs de b (par exemple inférieures à 24).
3. Testez les performances de votre implémentation pour trier un tableau de 60 000 000 nombres de 24 bits avec $b = 24$, et comparez avec votre implémentation de quicksort `qsortu`.

Q.6 :

1. Écrivez une fonction `radixsort` qui implémente le tri par base pour des entiers stockés sur 32 bits, et permettant d'aisément changer la base utilisée.
2. Testez votre fonction pour de petites valeurs de b (par exemple inférieures à 24).
3. Testez les performances de votre implémentation pour trier un tableau de 240 000 000 nombres de 24 bits pour les bases 2^{24}, 2^{12}, 2^8, 2^6, 2^4, 2^3, 2^2.
4. Même question pour 240 000 000 nombres de 32 bits, pour des bases vous semblant appropriées.

Q.7 : Comparez expérimentalement la croissance des temps d'exécution de vos fonctions `qsortu` et `radixsortu` pour trier des tableaux de nombres de 24 bits de taille comprise entre 1000 et 500 000 000, avec une valeur de b vous semblant appropriée.