PROG
TP#5

2020-10-07

Instructions de rendu

Ce TP est à réaliser en monôme ou en binôme. Vous devez envoyer votre travail sous la forme d'une archive à pierre.karpman@univ-grenoble-alpes.fr au plus tard le 23/10/2019 à 18 :00. Celle-ci doit contenir :

— Votre programme répondant aux questions ci-dessous, ***à l'exception de la question 4.3***. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
— Les instructions de compilation (idéalement avec un Makefile)
— Un court compte-rendu en format PDF qui détaille vos réponses

Quicksort

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

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;
```

Q.1 : Écrivez une fonction `int sorted(uint64_t *T, int64_t Tlen)` qui teste si un tableau `T` de `Tlen` `uint64_t` est trié.

Q.2 :

1. Écrivez une fonction `void qsortu(uint64_t *T, int64_t Tlen)` qui trie en place un tableau `T` de `Tlen` `uint64_t` en utilisant quicksort.
2. Testez votre fonction avec `sorted`, sur des tableaux remplis aléatoirement.
3. Evaluez expérimentalement la complexité de votre implémentation en mesurant le temps d'exécution sur des tableaux de taille variable (par ex. 1 000 000, 10 000 0000, 20 0000 000, 40 000 000, 80 000 000, 160 000 000 éléments).

Q.3 :

1. Que pouvez-vous prévoir sur le temps d'exécution de votre tri sur un tableau dont les éléments sont initialement strictement décroissant ? Vérifiez le expérimentalement.
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 de `T[d..f]`, puis de procéder comme auparavant.
3. Testez l'effet de votre changement sur le cas problématique précédent.

Q.4 :

1. Modifiez 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.
2. Comparez les performances de vos deux implémentations ainsi que celle de la bibliothèque standard pour des tableaux d'`uint64_t`.
3. Testez les performances de votre fonction sur votre programme du TP4.

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 et d'utiliser un tri *stable* pour trier ses entiers suivant d'abord les b bits de poids faible, puis les b bits suivants, etc. Le pseudocode d'un tel tri est le suivant :

```
/*****************************************
input: nlen, width
input: T, un tableau de nombres dont les
valeurs peuvent être représentées avec
nlen*width bits
output: Ts, qui contient les éléments de T
triés
*****************************************/
for (int i = 0; i < nlen; i++)
{
        denomsort(T, Ts, width, i);
        swap(T, Ts);
}
return T;
```

Ici, `denomsort` est un tri par dénombrement qui trie les entrées de `T` en considérant seulement leurs bits de position `i` $\times$ `width` $\dots$ (`i` $+ 1) \times$ `width` $- 1$.

Q.5 :

1. Écrivez une fonction

```
uint32_t *denomsort_r(const uint32_t *T, uint32_t *Ts,
↪  int64_t Tlen, uint64_t width, uint64_t pos, uint32_t
↪  *Aux);
```

   qui implémente un tel tri par dénombrement pour des entiers stockés sur 32 bits.
2. Testez-la sur un petit exemple
3. Testez les performances de votre implémentation pour trier un tableau de 60 000 000 nombres de 8 bits, et comparez avec votre implémentation de quicksort `qsortu` adaptée à des nombres de 32 bits.
4. Même question pour des nombres de 24 bits.

Q.6 :

1. Écrivez une fonction

```
uint32_t *radixsortu(const uint32_t *T, int64_t Tlen,
↪  uint64_t width, uint64_t nlen);
```

   qui implémente le tri par base pour des entiers stockés sur 32 bits.
2. 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.
3. Même question pour 240 000 000 nombres de 32 bits, pour les bases vous semblant appropriées.

Q.7 : Comparez expérimentalement la croissance des temps d'exécution de vos fonctions `qsortu` adaptée à des nombres de 32 bits, et `radixsortu` pour trier des tableaux de nombres de 24 bits de taille comprise entre 1000 et 500 000 000.