Programmation Concurrente

TD - Partager le travail pour aller plus vite - le pattern fork-join

Nous allons mettre en oeuvre la pattern fork-join de Java sur l'algorithme de tri Quick Sort qui est assez aisé à paralléliser et qui permet d'utiliser le pattern Fork-Join de Java. Pour ceux qui ne se rappelle pas de cet algorithme, voici une courte vidéo de présentation (https://www.youtube.com/watch?v=3DV8GO9g7B4).

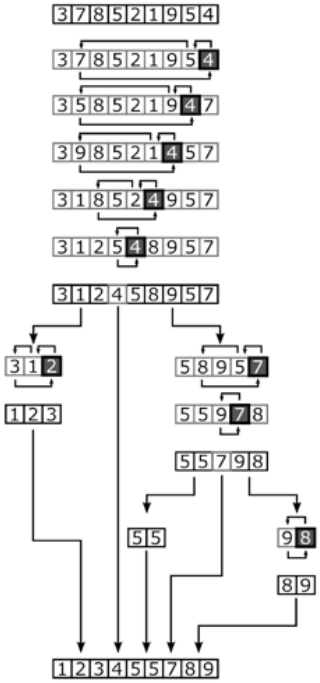
La 'recette de cuisine' pour cet algorithme est assez simple. Nous avons besoin : d'un tableau qui a une taille, un indice minimal (ind_min) et un indice maximal (ind_max) et dont la valeur des éléments est comprise entre borne_min et borne_max. La stratégie employée suit une approche de type "diviser pour régner". Elle consiste à placer un élément du tableau, appelé "pivot", à sa place définitive en permutant tous les éléments du tableau de telle sorte que :

- tous ceux qui sont inférieurs au pivot soient à sa gauche,
- tous ceux qui sont supérieurs au pivot soient à sa droite.

Cette opération s'appelle le "Partitionnement". Dès lors, il ne reste plus qu'à trier les deux sous-tableaux de chaque côté du pivot. Ce sont là deux tâches indépendantes qui peuvent être traité en parallèle (cf. cette petite vidéo (https://www.youtube.com/watch?v=ARfR7ForyN4)). En effet, pour chacun des deux tableaux résiduels, on choisit un nouveau pivot et on répète l'opération de partitionnement récursivement jusqu'à ce que chaque élément soit correctement placé. C'est le rôle dévolue à la méthode "`TriRapide`".

En pratique, pour partitionner un sous-tableau : on place le pivot choisi à la fin du sous-tableau, en l'échangeant avec le dernier élément du sous-tableau et on place tous les éléments inférieurs au pivot en début du sous-tableau, en conservant l'indice de celui le moins à gauche. Puis on place le pivot à la fin des éléments déplacés. Ces opérations sont réalisées par la méthode "`Partitionner`".

Le choix du pivot étant arbitraire, il peut par exemple correspondre systématiquement au dernier élément du tableau. La figure ci-dessous illustre le fonctionnement de l'algorithme.



L'objectif de ce TD est de concevoir et implémenter une version parallèle de cet algorithme en s'appuyant sur 4 threads rassemblées dans un "`Executor`", puis à me surer le gain en terme de temps de calcul. Pour cela, le programme devra mesurer et comparer le temps de calcul de la version séquentielle avec celui de la version parallèle obtenue sur le *même tableau* avant d'afficher le rapport entre ces mesures. Celles-ci seront effectuées sur une taille de tableau de "entier long" d'au-moins 100.000 éléments ayant une valeur entre 0 et 2^64. En guise de test, vous devrez comparer que tous les tableaux sont triés de la même manière.

- Etape 1 : quel est le pire cas i.e. celui ou QuickSort prendra le plus de temps ? Créer de manière aléatoire le tableau demandé ou choisissez le pire cas.
- Etape 2 : mettre en oeuvre l'algorithme itératif. Vérifier que le tableau est bien trié.
- Etape 3 : mettre en oeuvre la version 1 de l'algorithme parallèle en utilisant uniquement les Executor et des 'Runnable' ou 'Callable' Vérifier que le tableau est bien trié.
- Etape 4 : mettre en oeuvre la version 2 de l'algorithme parallèle en utilisant le pattern fork/join et les pools de taches (ForkJoinPool). Vérifier que le tableau est bien trié.
- Etape 5 : comparer et analyser les résultats obtenus. Pour cela vous mesurerez le temps d'exécution des 3 algorithmes pour des tableaux de tailles 1000 à 1.000.000 en multipliant à chaque fois la taille par 10 (ou la taille max acceptée par votre machine virtuelle) et pour la solution de l'étape 4 vous ferez varier le nombre de thread de 2^0 à 2^10. Attention à bien gérer la mémoire et à ne pas trop gaspiller cette précieuse ressource.
  Bien évidemment, vous exécuterez plusieurs fois chacun des algorithmes pour mesurer le temps d'exécution. Vous présenterez les résultats dans un graphique et vous proposez une approche permettant de déterminer le meilleur algorithme en fonction de la taille du tableau.
- Etape 6 (facultative) : reprenez l'algorithme de l'étape 4 mais pour un tableau inférieur à une taille donnée que vous préciserez, au lieu de paralléliser le tri, vous utiliserez une approche séquentielle. Est-ce que le temps d'exécution a été amélioré.

Vous trouverez ci-dessous des éléments permettant de construire rapidement les étapes 1 et 2.

- Pour initialiser un tableau entre les bornes min et max

```
Random rand=new Random();
long taille = XXX;
long max = XXX;
long min = XXX;

long tableau[]=new long[taille];

for (int i=0; i<taille; i++)
      tableau[i] = rand.nextInt(2*max - min + 1) + min;
```

- La fonction Partition

```
int partition (int [] t, int début, int fin) {
    long valeurPivot = t[début]; // on prend le pivot au début mais on pourrait le prendre n'importe ou entre les deux bornes.
    long d = début+1;
    long f = fin;
    while (d < f) {
        while(d < f && t[f] >= valeurPivot) f--;
        while(d < f && t[d] <= valeurPivot) d++;
        int temp = t[d];
        t[d]= t[f];
        t[f] = temp;
    }
    if (t[d] > valeurPivot) d--;
    t[début] = t[d];
    t[d] = valeurPivot;
    return d;
}
```

- La procédure TriRapide

```
 void TriRapide(int [] tableau, long début, long fin) {
    if (début < fin) {
        long indicePivot = Partition(tableau, début, fin);
        TriRapide(tableau, début, indicePivot-1);
        TriRapide(tableau, indicePivot+1, fin);
    }
}
```