TP4 : Parallélisme et concurrence

Le but de ce TP est de se familiariser avec les concepts de parallélisme et de concurrence et certains mécanismes de Java qui y sont liés.

1 Parallélisme

Dans cette partie, nous programmerons deux algorithmes qui opèrent sur des tableaux (de nombres entiers). Une technique fréquente consiste à découper le tableau en deux parties disjointes et de faire exécuter le même traitement sur les deux parties en parallèle, par deux *exétrons*.

1.1 Maximum Pour commencer, recherchons le maximum d'un tableau en cherchant en parallèle le maximum de sa première moitié et le maximum de sa seconde moitié.

a) Programmez un objet `maxInRange(int[] a, int i, int j)` qui satisfait l'interface `Runnable` et qui, lorsqu'exécuté, recherche séquentiellement l'élément maximal dans le tableau `a` entre les indices `i` (inclus) et `j` (exclus). Cet objet devra également fournir une méthode permettant de récupérer le résultat, une fois le calcul terminé.

b) Programmez une fonction qui recherche en parallèle l'élément maximal de chaque moitié d'un tableau puis renvoie l'élément maximal du tableau.

1.2 Tri fusion Le tri fusion est un des algorithmes de tri efficaces qui se prête le mieux à une exécution en parallèle.

a) Programmez un algorithme de tri fusion en place et séquentiel. Il s'appuiera sur une fonction `merge(int[] a, int i, int j, int k, int l)` qui opère la fusion (en place) des deux segments triés `a[i..j]` et `a[k..l]`.

La fonction `System.arraycopy` se montrera certainement utile.

b) Programmez un objet (récursif) `sort(int[] a, int b, int e)` qui satisfait l'interface `Runnable` et qui, lorsqu'exécuté, trie en parallèle les deux moitiés du segment de tableau `a[b..e]` puis les fusionne.

En déduire une fonction `sort(int[] a)` qui trie en place le tableau `a`.

2 Concurrence

Imaginons deux fils d'exécution concurrents ; l'un trie un tableau pendant que l'autre recherche le maximum du même tableau. Que peut-il se passer ? Est-ce problématique ? auquel cas comment y remédier ?

Il s'agit maintenant de programmer une file de messages pour un serveur : celui-ci consacre un fil d'exécution à chaque client pour recevoir leurs requêtes mais traite toutes lesdites requêtes dans un même fil d'exécution.

Ainsi, lorsqu'une requête est reçue d'un client, un message est placée à la fin de la file. Pendant ce temps, le fil d'exécution dédié au traitement des requêtes récupère un par un les messages au début de la file.

Un tel serveur est déjà programmé et disponible à l'adresse suivante : http://people.irisa.fr/Vincent.Laporte/teaching/para/EchoServer.java.

Il vous faut donc implémenter une file de messages qui satisfera l'interface suivante.

```
public interface Queue<E> {
  boolean enqueue(E e);
  E dequeue();
}
```

La méthode `enqueue(e)` ajoute l'élément `e` à la fin de la file et renvoie `true` ou ne fait rien et renvoie `false`.

La méthode `dequeue()` retire le premier élément de la file et le renvoie. Si la file est vide, elle ne fait rien et renvoie `null`.

Une implémentation possible est la *two-lock queue* de Michael et Scott [1]. Le *package* `java.util.concurrent.locks` contient des définitions et implémentations de verrous.

Références

[1] M. M. Michael et M. L. Scott. « Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms ». In : *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, p. 267–275.