Parallélisme régulé

Objectifs

- gérer le parallélisme à gros grain
- paralléliser un algorithme par décomposition en sous-tâches
- connaître les services d'exécution de la plateforme Java

Exercices

Chaque exercice comporte un calcul séquentiel qu'il faut paralléliser.

Les exercices utilisent des tableaux d'entiers stockés sur disque. L'archive fournie comporte une classe `LargeIntArray` permettant de générer, charger en mémoire, sauvegarder, comparer ou trier de tels tableaux. La méthode `main` permet en outre d'appeler les méthodes de la classe `LargeIntArray` depuis la console.

Cette application sera utilisée pour générer les jeux de données utiles aux tests. En effet, pour que le gain apporté par les versions parallèles soit sensible, il est nécessaire que les volumes de données traités soient significatifs (tableaux de 1 à 100 millions d'entrées), ce qui aurait alourdi l'archive.

- Génération d'un gros fichier avec des valeurs entre -1000 et 1000 (`foo` est le nom du fichier créé) :
  `java LargeIntArray -g foo 1000000 1000`
- Tri du tableau (pour les exercices de comptage) :
  `java LargeIntArray -s foo`
- *Penser à supprimer les fichiers créés une fois le TP passé.*

Calcul du maximum d'un tableau (répertoire `max`).

Le but est de trouver l'élément maximal du tableau. La méthode `LargeIntArray.max(int[] array, int start, int end)` calcule le max en parcourant séquentiellement le tableau entre les indices `[start..end[`.

1) `MaxTabSequential.java` contient une version purement séquentielle.
2) Compléter `MaxTabThread.java` en s'inspirant de `exemples/SommeThreadPlus.java`.
3) Compléter `MaxTabPool.java` en s'inspirant de `exemples/SommePool.java`.
4) Compléter `MaxTabForkJoin.java` en s'inspirant de `exemples/SchemaForkJoin.java`.
5) Comparer les performances des solutions avec un gros tableau `foo` :
   `java MaxTabSequential foo 20` – 20 = nombre d'expériences
   `java MaxTabThread foo 20 4` – 4 = nombre de threads
   `java MaxTabPool foo 20 4 8` – 4 = taille du pool, 8 = nombre de jobs

`java MaxTabForJoin foo 20 1000` – 1000 = seuil d'arrêt de la décomposition

6) Faire varier le nombre de threads / taille du pool / seuil d'arrêt.
7) Analyser les résultats : quelle est la version la plus facile à écrire ? Quelle la version la plus efficace ?

Comptage des éléments dans un intervalle

Attention : le tableau doit être trié. Le but est de compter le nombre de valeurs dans le tableau entre 0 et 9 (constantes VMIN/VMAX arbitrairement fixées). La méthode `LargeIntArray.count(int[] array, int start, int end, int vmin, int vmax)` compte le nombre d'éléments entre vmin et vmax (inclus) en parcourant séquentiellement le tableau entre les indices `[start..end[`. Observer que vu que le tableau est trié, il est inutile de parcourir un tronçon si le premier élément du tronçon est supérieur à VMAX ou si le dernier est inférieur à VMIN : il y a nécessairement 0 valeur entre VMIN et VMAX. Ceci induit que la décomposition est irrégulière avec certaines tâches quasi-instantanées et d'autres longues.

1) `CountSequential.java` contient une version purement séquentielle.
2) Compléter `CountThread.java`.
3) Compléter `CountPool.java`.
4) Compléter `CountForkJoin.java`.
5) Comparer les performances des solutions avec un gros tableau `foo` :
   `java CountSequential foo 20` – 20 = nombre d'expériences
   `java CountThread foo 20 4` – 4 = nombre de threads
   `java CountPool foo 20 4 8` – 4 = taille du pool, 8 = nombre de jobs
   `java CountForJoin foo 20 1000` – 1000 = seuil d'arrêt de la décomposition
6) Faire varier le nombre de threads / taille du pool / seuil d'arrêt.
7) Analyser les résultats : quelle est la version la plus facile à écrire ? Quelle la version la plus efficace ?

Comptage des mots dans les fichiers d'une arborescence

Pour aller plus loin (non demandé pour ce TP), l'application de comptage de mots dans une arborescence (répertoire `wordcount`) réalise la commande `find repertoire -type f -exec grep mot {} \; | wc`. Elle permet d'illustrer la parallélisation d'un problème irrégulier.

- Paralléliser l'algorithme récursif proposé en utilisant le schéma fork/join
- Comparer cette version avec la version séquentielle en termes de facilité de conception et de performances. Pour le test, on pourra prendre un répertoire contenant des fichiers sources et rechercher un mot clé du langage.

Outils

- `System.nanoTime()` et `System.currentTimeMillis()` fournissent une durée écoulée (en ns et ms) depuis une date d'origine non spécifiée. La différence entre les valeurs retournées par deux appels successifs permet d'évaluer le temps écoulé entre ces deux appels.

- `Runtime.getRuntime().availableProcessors()` fournit le nombre de processeurs disponibles pour la JVM.

Tester les performances d'applications concurrentes en Java

- sources de perturbation : cache, compilateur à la volée, ramasse miettes et optimiseur, charge de l'environnement (matériel, réseau) -> répéter les mesures
- tester sur des volumes de données significatifs
- connaître le nombre de processeurs réels disponibles
- éviter les optimisations sauvages
  - avoir des tâches suffisamment complexes
  - avoir un jeu de données varié (non constant en valeur et dans le temps)
- arrêter la décomposition en sous-tâches à un seuil raisonnable

Rappel : Activités en Java

La présentation qui suit peut être complétée par la lecture de la partie correspondante du cours sur les activités (chapitre 6) et par la documentation Java en ligne.

Création manuelle des activités

La classe `Thread` peut être instanciée avec un `Runnable` pour créer une nouvelle activité. Le cycle de vie de l'activité est à la charge du programmeur. Si l'activité doit fournir un résultat quand elle termine, le programmeur doit prévoir un mécanisme de transmission de ce résultat via des variables partagées (ce qui induit la nécessité de synchronisation, a minima l'exclusion mutuelle pour protéger les accès concurrents).

Un exemple

```
import java.util.*;
class Sum implements Runnable {
    private long from;
    private long to;

    Sum(long from, long to) { this.from = from; this.to = to;}

    @Override
    public void run() {
        long s = 0;
        for (long i = from; i <= to; i++) s = s + i;
        System.out.println("Calcul terminé. Sigma(" + from + "," + to + ") = " + s);
    }
}

public class SommeThread {
    public static void main(String[] args) throws Exception {
        List<Sum> jobs = Arrays.asList(new Sum(0, 1_000_000_000L),
                                       new Sum(0, 4_000_000_000L),
                                       new Sum(900_000, 1_000_000_000L),
                                       new Sum(1, 100),
                                       new Sum(0, 3_000_000_000L));
        Set<Thread> threads = new HashSet<>();
        for (Sum j : jobs) {
            Thread t = new Thread(j);
            threads.add(t);
            t.start();
        }
        for (Thread t : threads) {
            t.join();
        }
    }
}
```

Commentaires

- L'application crée autant d'activités qu'il y a de calculs à faire. Ces activités sont démarrées et l'application attend ensuite leur terminaison.
- L'archive contient une variante `SommeThreadPlus` où chaque activité fournit un résultat qui est récupéré dans le programme principal.

Création automatique des activités

La plateforme Java fournit la classe `Executor` destinée à séparer la gestion des activités des aspects purement applicatifs. Le principe est qu'un objet de la classe `Executor` (« exécuteur ») fournit un *service* de gestion et d'ordonnancement d'activités, auquel on soumet des *tâches* à traiter. Une application est donc vue comme un ensemble de tâches qui sont fournies à l'exécuteur. L'exécuteur gère l'exécution des tâches qui lui sont soumises de manière indépendante et transparente pour l'application. L'objectif de ce service est de permettre :

- de simplifier la tâche du programmeur puisqu'il n'a plus à gérer le démarrage des activités ni leur ordonnancement ;
- d'adapter le nombre d'activités exécutées à la charge et au nombre de processeurs physiques disponibles.

Le paquetage `java.util.concurrent` définit trois interfaces pour les exécuteurs :

- `Executor` fournit une méthode `execute`, permettant de soumettre une tâche `Runnable`.
- `ExecutorService` étend `Executor` avec une méthode `submit` permettant de soumettre une tâche `Callable` et renvoyant un objet `Future`, lequel permet de récupérer la valeur de retour de la tâche `Callable` soumise. Un `ExecutorService` permet en outre de soumettre des ensembles de tâches `Callable` et de gérer la terminaison de l'exécuteur.
- `ScheduledExecutorService` étend `ExecutorService` avec des méthodes permettant de spécifier l'ordonnancement des tâches soumises.

Différentes implantations d'exécuteurs sont fournies. Le principe commun est de distribuer les tâches soumises à un ensemble d'ouvriers. Chaque ouvrier est un thread cyclique qui traite une par une les tâches qui lui sont attribuées.

Les exécuteurs fournis par le paquetage `java.util.concurrent` sont de deux sortes : les pools d'activités et les pools pour fork/join.

Pools de threads

La classe `java.util.concurrent.Executors` fournit des méthodes permettant de créer des pools de threads implantant `ExecutorService` avec un nombre d'ouvriers fixe (`newFixedThreadPool`), variable (`newCachedThreadPool`) ou permettant une régulation par vol de tâches (`newWorkStealingPool`). Une variante implantant `ScheduledExecutorService` est proposée pour chacune de ces méthodes afin d'intervenir sur l'ordonnancement des tâches. Enfin, les classes `ThreadPoolExecutor` et `ScheduledThreadPoolExecutor` proposent encore davantage d'options sur la paramétrage et la supervision de l'ordonnancement.

Les pools de threads évitent la création de nouveaux threads pour chaque tâche à traiter puisque qu'un même ouvrier est réutilisé pour traiter une suite de tâches, ce qui présente plusieurs avantages :

- éviter la création de threads apporte un gain (significatif lorsque les tâches sont nombreuses) en termes de consommation de ressources mémoire et processeur ;
- le délai de prise en charge des requêtes est réduit du temps de la création du traitant de la requête ;
- le contrôle du nombre d'ouvriers va permettre de réguler l'exécution en fonction des ressources matérielles disponibles au lieu d'avoir une exécution directement dépendante du flux de tâches à traiter. Ainsi dans le cas d'un flux de tâches augmentant brutalement, les performances se dégraderont progressivement (car le délai de prise en charge augmentera proportionnellement au nombre de tâches en attente) mais il n'y aura pas d'écroulement dû à un épuisement des ressources nécessaires.
- Les pools de threads sont bien adaptés au traitement de problèmes réguliers, décomposables en sous-problèmes de « taille » équivalente, ce qui garantit une bonne répartition des tâches entre ouvriers.

Classes et méthodes utiles

- La classe `java.util.concurrent.Executors` permet de créer des pools de threads par appel de `newFixedThreadPool()` ou `newCachedThreadPool()`.
- La classe `ExecutorService` et sa superclasse `Executor` définissent l'interface d'un exécuteur, avec notamment les méthodes `submit()`, `execute()` et `shutdown()` (gestion de la terminaison de l'exécuteur).
- La classe `Future` fournit immédiatement une référence vers le résultat à venir d'une tâche `Callable`soumise à l'exécuteur par `submit()`. L'appel de la méthode `get()` permet d'obtenir le résultat effectif, en bloquant s'il n'est pas encore disponible.
- Les tâches ne renvoyant pas de résultat sont des `Runnable`, soumises à l'exécuteur par `execute()`.

Un exemple

```
import java.util.concurrent.Future;
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

class SigmaC implements Callable<Long> {
    private long from;
    private long to;

    SigmaC(long from, long to) { this.from = from; this.to = to;}

    @Override
    public Long call() { // le résultat doit être un objet
        long s = 0;
```

```
        for (long i = from; i <= to; i++) s = s + i;
        return s;
    }
}

class SigmaR implements Runnable {
    private long from;
    private long to;

    SigmaR(long from, long to) { this.from = from; this.to = to;}

    @Override
    public void run() {
        long s = 0;
        for (long i = from; i <= to; i++) s = s + i;
        System.out.println("Calcul terminé. Sigma(" + from + "," + to + ") = " + s);
    }
}

public class SommePool {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Future<Long> f1 = executor.submit(new SigmaC(0L, 1_000_000_000L));
        Future<Long> f2 = executor.submit(new SigmaC(0L, 4_000_000_000L));
        executor.execute(new SigmaR(900_000L, 1_000_000_000L));
        Future<Long> f3 = executor.submit(new SigmaC(1, 100));
        Future<Long> f4 = executor.submit(new SigmaC(0L, 3_000_000_000L));

        System.out.println("Résultat obtenu. f1 = " + f1.get());
        System.out.println("Résultat obtenu. f2 = " + f2.get());
        System.out.println("Résultat obtenu. f3 = " + f3.get());
        System.out.println("Résultat obtenu. f4 = " + f4.get());

        executor.shutdown();
    }
}
```

Commentaires

- L'application crée un pool avec un nombre fixe d'ouvriers (deux) puis lance cinq tâches : les deux premières et les deux dernières soumises (`Callable`, soumises par appel à `submit()`) rendent un résultat `Future`, récupéré de manière bloquante par l'appel à la méthode `get()`. La troisième (`Runnable`, soumise par appel à `execute()`) s'exécute de manière asynchrone.
- L'exécution voit la tâche `Runnable` terminer après la première soumise (`f1`), car bien que plus courte, elle ne peut démarrer tant que l'une des deux premières tâches lancées n'est pas terminée, la taille du pool étant de deux. L'appel `f2.get()` entraîne l'attente de la terminaison de `f2`, plus longue que `f1` et la tâche `Runnable` cumulées. L'appel de `f3.get()` retourne

immédiatement, car `f3`, courte, est déjà terminée. L'appel `f4.get()` entraîne l'attente de la terminaison de `f4`.
- `shutdown` permet de terminer proprement l'exécuteur qui dès lors n'accepte plus de nouvelles tâches. L'application Java termine avec la dernière tâche traitée.
- L'archive contient une variante (`SommePoolPlus`) qui illustre l'utilisation de `invokeAll()` pour soumettre une collection de `Callable`.

Pool Fork/Join (Schéma Map/Reduce)

La classe `ForkJoinPool` est un exécuteur dont l'ordonnancement est adapté à une parallélisation selon le schéma *fork/join* (voir cours). Le principe récursif est

- de traiter directement (séquentiellement) un problème si sa taille est suffisamment petite ;
- sinon de diviser le problème en sous-problèmes qui seront traités en parallèle (`fork`) et dont les résultats seront attendus et agrégés (`join`).

Ce schéma de programmation permet de créer dynamiquement un nombre de tâches adapté à la taille de chacun des problèmes rencontrés, chacune des tâches créées représentant une charge de travail équivalente. Ce schéma est donc bien adapté au traitement de problèmes irréguliers de grande taille. L'ordonnanceur de la classe `ForkJoinPool` comporte en outre une régulation (vol de tâches) qui permet l'adaptation de l'exécution aux capacités de calcul disponibles.

Classes et méthodes utiles

- `ForkJoinPool`: classe définissant l'exécuteur.
- `RecursiveTask<V>` : définit une tâche soumise à l'exécuteur, fournissant un résultat.
- `RecursiveAction` : définit une tâche soumise à l'exécuteur, ne fournissant pas de résultat.
- `ForkJoinTask<V>` : superclasse de `RecursiveTask` et `RecursiveAction`, définissant la plupart des méthodes utiles comme `fork()` et `join()`.

Un exemple

Dans cette application, les données à traiter sont représentées par un simple entier qui symbolise leur volume.

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class Problem extends RecursiveTask<Integer> {
    static private final int SEUIL = 10;
    private int size;
    private String indent;

    Problem(int size, String indent) {
        this.size = size;
        this.indent = indent;
    }
```

```
    protected Integer compute() {
        if (this.size > SEUIL) { // la tâche est trop grosse, on la décompose en deux
            System.out.println(indent + "Fork : " + size);
            Problem sp1 = new Problem(size / 2, indent + "  ");
            Problem sp2 = new Problem(size / 2, indent + "  ");
            sp1.fork();
            sp2.fork();
            // en commençant par sp2.join, la jvm va réutiliser ce thread pour exécuter sp
            int result = sp2.join() + sp1.join();
            return result;
        } else {
            System.out.println(indent + "Traitement direct : " + size);
            return 10 * size;
        }
    }
}

public class SchemaForkJoin {
    static final int TAILLE = 1024; // Puissance de 2 pour cet exemple naïf

    public static void main(String[] args) throws Exception {
        RecursiveTask<Integer> pb = new Problem(TAILLE, "");
        ForkJoinPool fjp = new ForkJoinPool();
        long startTime = System.nanoTime();
        int result = fjp.invoke(pb);
        long endTime = System.nanoTime();
        System.out.println("Résultat final = " + result);
        System.out.println("Durée exécution : " + (endTime - startTime) / 1000000L + " ms"
    }
}
```

Commentaires

- La méthode abstraite `compute()` définie dans `RecursiveTask` et `RecursiveAction` contient le code du calcul récursif proprement dit. C'est l'analogue de la méthode `run()` pour la classe `Runnnable` ou de la méthode `call()` pour la classe `Callable`.
- `SEUIL` est la taille de problème à partir duquel le travail n'est plus subdivisé. Sa valeur est un compromis dépendant de la nature du problème. Une règle empirique est que le nombre de sous-tâches créées devrait être compris entre 100 et 10 000. Il faut aussi savoir que le pool ne peut comporter plus de 32K ouvriers.
- Le `ForkJoinPool` doit être créé une fois et une seule pour toute la durée de l'application (ce n'est pas obligatoire mais c'est conseillé même pour les experts). On peut aussi utiliser le pool par défaut (`ForkJoinPool.commonPool()`).
- L'appel `fjp.invoke(pb);` permet de soumettre la tâche racine au pool.