Programmation Avancée - Complément sur la parallélisation et l'utilisation des Threads en Java

T.DUFAUD

UVSQ - IUT Vélizy - Informatique

BUT 3 R5-05 - 11 / 2024

Plan

1. Contexte de ce cours
2. Remarques sur l'API concurrent
   - *Executor* et *Future*
3. Introduction à l'évaluation de performance des programmes parallèles

Contexte de ce cours

Application du cours pour le Calcul de π par une méthode de Monte Carlo

- Analyse de l'algorithme
- Modèle de programmation parallèle par tâche
  - la tâche principale est la boucle : "génération des points et comptage"
  - On décompose la tâche en sous tâches :
    - **MODÈLE 1** une itération = une tâche
    - **OU MODÈLE 2** un groupe d'itération = une tâche
- Paradigme
  - Parallélisation de boucle : pour MODÈLE 1
  - Maître-Esclave : pour MODÈLE 2

Source en ligne

- Code 1 : Assignment 102 sur `https://gist.github.com/krthkj/9c1868c1f69142c2952683ea91ca2a37`
- Code 2 : Pi.java sur `http://web.cs.iastate.edu/~smkautz/cs430s14/examples/thread_pool_examples/pi/Pi.java`

Source en ligne

- Ces codes utilisent l'API Concurrent de Java
- **Quels sont les outils utilisés ici ?**

Remarques sur l'API concurrent

Le service **Executor**

- C’est un **support pour les Threads** en Java à un plus haut niveau que la classe Thread.
- Il permet de **découpler la soumission des tâches de la mécanique des Threads** (Execution, Ordonnancement)
- On peut ici gérer des pool de Threads. Chaque Thread du pool peut être réutilisé dans un Executor.
- **L’interface Executor définit la méthode *execute***
- *execute* prend comme argument des *Runnable* ou des *Callable* (Voir notes sur les Future)

Le service **Executor**

- **NE PLUS FAIRE**

```
new Thread( new RunnableTask() ).start();
```

- **FAIRE**

```
Executor executor = myExecutor;
executor.execute(new RunnableTask1())
executor.execute(new RunnableTask2())
```

Dans la documentation

```
class ThreadPerTaskExecutor implements Executor {
        public void execute( Runnable r) {
                new Thread(r).start;
        }
}
```

Méthode *execute*

execute la commande a un instant t dans le futur.

Les *Future*

Future

Une future est le résultat d'un calcul asynchrone. Le principe est de pouvoir :

- *isDone*
  - vérifier si une opération est complétée / terminée
- *get* : Attendre un résultat et le récupérer
  - attendre que l'opération soit terminée
  - récupérer le résultat de l'opération

[

fragile]FutureTask implémente l'interface Future

- peut être utilisée pour wrapper un *Callable*
- un *Callable* hérite d'un Runnable et le spécialise en donnant un type de retour et des exceptions.
- une *FutureTask* peut être soumise à un *Executor*.

Constructeur pour les Future

```
FutureTask( Callable<V> callable );
FutureTask( Runnable runnable, V result );
```

Opération Atomique

- C'est une opération qui ne peut être exécuté par plusieurs Thread en même temps
- C'est donc une section critique

Comment les gérer ?

- Par bloc synchronisé, *synchronized*
- Ou en déclarant des variables comme étant de type *Atomic <Type>*
  - toute les opérations sur la variable sont des sections critiques
  - exemple : un compteur peut être déclacré comme un Atomic Int (Cf. Exemple 71 blog Paumard, Cf. Code Assignment102))

Références

Références pour l'API Concurrent

- Blog Paumard : Java Avancé, section 8 : `http://blog.paumard.org/cours/java-api/chap05-concurrent-util.html`
- Cours Etienne Duris 2006 : `http://igm.univ-mlv.fr/~duris/TTT/threadsNew.pdf`

Introduction à l'évaluation de performance des programmes parallèles

Accélération (*speedup*)

L'accélération S_P est le gain de vitesse d'exécution en fonction du nombre de processus **P**. On l'exprime comme le rapport du temps d'éxécution sur un processus T_1, sur le temps d'exécution sur **P** processus, T_P.

$$S_p = \frac{T_1}{T_p}$$

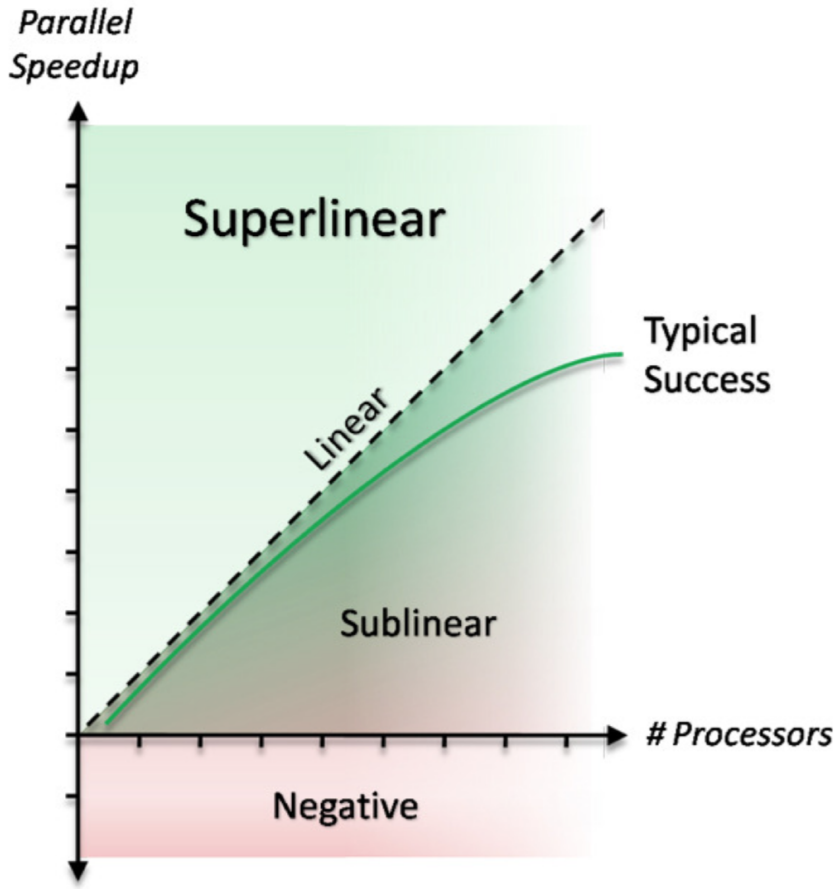

Figure: Speedup (extrait du cours F. Butelle et C. Coti, U. P13, `https://lipn.univ-paris13.fr/~coti/cours/coursmpi1.pdf`

Caractériser le gain de performance

Scalabilité forte

On fixe la taille du problème et on augmente le nombre de processus.
"Est-ce que je vais plus vite quand j'ajoute des processus ? "

Scalabilité faible

On fixe la taille du problème par processus et on augmente le nombre de processus. C'est à dire, la taille du problème augmente avec le nombre de processus. "Est-ce que mon code me permet de traiter des problèmes plus gros grace à l'augmentation du nombre de processus ? "

Exemple : code Monte Carlo pour le calcul de π