Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université
Socialement engagée

Année 2023-24

Programmation avancée : TP 7 bonus
Code UE : SBICU05

Gestion de tâches dans une ferme de machines

Consignes à suivre

Consignes pour démarrer le TP

Comme pour le TP 2, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du TP 2. Le lien vers le projet à forker est le suivant : https://etulab.univ-amu.fr/alaboure/cluster-template

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le code cliquer deux fois sur `cluster` -> `application` -> `run`. Cela exécute la méthode `main` de la classe `cluster.App`.

Pour exécuter les tests, il faut passer par l'onglet gradle à droite et cliquer deux fois sur `cluster` -> `Tasks` -> `verification` -> `test`. Pour le moment, cela ne fonctionnera pas car les tests sont en commentaire.

Gestion de tâches dans une ferme de machines

Vous allez travailler dans ce TP à la gestion d'une ferme de nœuds (*nodes*) représentant des ordinateurs qui mettent à disposition de la mémoire (*memory*) exprimée en octets (o) et d'un nombre d'opérations exprimé en FLOP qui peut être consommées par des tâches (*job*). L'objectif est de proposer un ordonnanceur (*scheduler*) qui place un ensemble de tâches sur les nœuds de manière à maximiser l'utilisation de la mémoire et du nombre d'opération.

La tâche de calcul : classe `Job`

La classe `Job` va nous permettre de représenter une tâche de calcul qui a une quantité de mémoire à consommer sur un nœud ainsi qu'un nombre d'opérations à exécuter. La classe `Job` contiendra les attributs, méthodes et constructeurs suivants :

- un attribut `int memory` : la quantité de mémoire requise par la tâche exprimée en o.
- un attribut `int flop` : le nombre d'opérations en virgule flottante (en anglais : FLoating-point OPerations ou FLOP) requis par la tâche exprimé en FLOP.
- un attribut `int id` : l'identifiant de la tâche. La première tâche créée devra avoir un `id` égal à `0`, la deuxième tâche créée devra avoir un `id` égal à 1, la troisième tâche créée devra avoir un `id` égal à 2 et ainsi de suite.
- un attribut `static int jobCount` : le nombre total de tâches qui ont été créées (mis à jour à chaque appel du constructeur).
- un constructeur `public Job(int memory, int flop)` : qui initialise les attributs `memory`, `flop` et `id` de la tâche. Il devra lever une exception de type `IllegalArgumentException` si on essaye de créer un `Job` avec une mémoire ou un nombre d'opérations inférieur ou égal à 0.

— une méthode `int getMemory()` qui renvoie la mémoire de la tâche.
— une méthode `int getFlop()` qui renvoie le nombre d'opérations de la tâche.
— une méthode `static int getJobCount()` qui renvoie le nombre de tâches créées depuis le dernier appel à `resetJobCount()`.
— une méthode `static void resetJobCount()` : remet à 0 l'attribut `jobCount`. Un `Job` créé après un appel à `resetJobCount` devra avoir un `id` égal à `0`, la deuxième tâche créée devra avoir un `id` égal à 1, la troisième tâche créée devra avoir un `id` égal à 2 et ainsi de suite.
— une méthode `int getId()` : qui renvoie l'`id` du processus.
— une méthode `String toString()` : renvoie une chaîne de caractère représentant la tâche. La chaîne contiendra la chaîne `"Job"`, suivi de l'identifiant de la tâche, suivi de la quantité de mémoire de la tâche entre parenthèses, le tout séparé par des espaces. Par exemple, pour une tâche avec 100 octets de mémoire, 1000 FLOP et un identifiant égal à 0, un appel à `toString` devra renvoyer la chaîne de caractères `Job 0 (1000FLOP, 100o)`.

Modifier et compléter la classe `Job` située dans `src` -> `main` -> `java` -> `cluster` de sorte à respecter les consignes ci-dessus.

Tester la classe `Job` : classe `JobTest`

Vous trouverez dans le répertoire `test/java/cluster` une classe de test `JobTest`. Vous devez les décommenter les tests pour les lancer. Cette classe ne contient qu'un test pour la méthode `toString`. Vous devez donc rajouter des tests pour les autres méthodes. Pour compiler et lancer des tests sur votre programme, il faut passer par l'onglet *gradle* à droite et cliquer deux fois sur `cluster` -> `Tasks` -> `verification` -> `test`.

Vous allez devoir rajouter des méthodes de tests afin de tester les comportements suivants. Chaque test devra être dans une méthode de test différente.

— une tâche créée juste après un appel à `resetJobCount()` a un `id` égal à 0.
— une tâche créée avec une certaine mémoire a bien cette mémoire selon l'accesseur (*getter*) correspondant.
— une tâche créée avec un certain nombre d'opérations a bien ce nombre d'opérations selon l'accesseur (*getter*) correspondant.
— un appel à `getJobCount()` donne bien le nombre de tâches créées depuis le dernier appel à `resetJobCount()`.

Rajouter les tests demandés en utilisant JUnit ou assertJ.

Les nœuds de la ferme de calcul : classe `Node`

Une ferme de calcul se compose d'un ensemble de nœuds (*nodes*) auxquels on va affecter des tâches utilisant de la mémoire et des opérations. Chaque nœud a une capacité de mémoire (quantité de mémoire maximale que peut fournir le nœud) et une quantité de mémoire disponible (égal à la capacité en mémoire du nœud moins la mémoire utilisée par les tâches affectées au nœud). La quantité de mémoire disponible prend donc la valeur 0 quand il n'y a plus de mémoire disponible et la valeur de la capacité quand aucune tâche n'est affectée au nœud.

De même, chaque nœud a une capacité en termes de nombre d'opérations de calcul possible et un nombre d'opérations de calcul disponible. La classe `Node` contiendra les attributs, méthodes et constructeurs suivants :

- un attribut `String name` : le nom du nœud.
- un attribut `int memoryCapacity` : la capacité de mémoire du nœud.
- un attribut `int availableMemory` : la quantité de mémoire disponible dans le nœud.
- un attribut `int flopCapacity` : la capacité en nombre d'opérations du nœud.
- un attribut `int availableFlop` : la quantité d'opérations disponible dans le nœud.
- un attribut `List<Job> assignedJobs` : la liste des tâches affectées au nœud.
- un constructeur `Node(String name, int memoryCapacity, int flopCapacity)` qui crée un nœud avec le nom et les capacités en mémoire et opérations spécifiés. Le nœud créé n'a aucune tâche affectée.
- une méthode `boolean canAccept(Job job)` renvoie `true` si le nœud a assez de mémoire et d'opérations disponibles pour accepter le `job` (mémoire et nombre d'opérations du `Job` tous les deux inférieurs ou égaux à la mémoire et le nombre d'opérations disponibles du `Node`).
- une méthode `boolean canHandle(Job job)` renvoie `true` si le nœud a une capacité de mémoire et d'opérations suffisants pour gérer le `job` (mémoire et nombre d'opérations du `Job` tous les deux inférieurs ou égaux aux capacités de mémoire et de nombre d'opérations du `Node`).
- une méthode `void accept(Job job)` : affecte le `job` au nœud et met à jour la quantité de mémoire et le nombre d'opérations disponible en conséquence.
- une méthode `int usedMemory()` : renvoie la mémoire consommée par les tâches affectées au Nœud.
- une méthode `int usedFlop()` : renvoie le nombre d'opérations consommé par les tâches affectées au Nœud.
- une méthode `String toString()` : renvoie une chaîne de caractères représentant le nœud. La chaîne contiendra la chaîne `"Node"`, suivi du nom du nœud, suivi de la quantité de mémoire utilisée sur la capacité du nœud entre parenthèses, le tout séparé par des espaces. Par exemple, pour un nœud ayant pour nom `Calcul` avec 4000o de mémoire utilisée, une capacité mémoire de 10000o, 2000 opérations utilisées, une capacité en opérations de 3000 et un appel à `toString` devra renvoyer la chaîne de caractères `Node Calcul (2000/3000FLOP, 4000/10000o)`.
- une méthode `void printJobs()` : affiche le nœud et toutes les tâches qui lui ont été affectées, chacun sur une ligne différente. L'exemple suivant vous indique un exemple de ce qui est demandé. Ce code est déjà présent dans la classe `App` du paquetage `cluster`. Il vous faut décommenter le code afin de l'exécuter.

```java
package cluster;

public class App {
    public static void main(String[] args) throws Exception{
    Job job1 = new Job(1000, 1000);
    Job job2 = new Job(3000, 1000);
    Node node = new Node("Calcul", 10000, 3000);
        node.accept(job1);
        node.accept(job2);
        node.printJobs();
    }
}
```

L'exécution du code ci-dessus devra donner l'affichage suivant :

```
Node Calcul (2000/3000FLOP, 4000/10000o)
Job 0 (1000FLOP, 1000o)
Job 1 (1000FLOP, 3000o)
```

Créer la classe `Node` avec les attributs et méthodes demandées.

Exception `NotEnoughMemoryException`

Il y a pour le moment un problème si on appelle la méthode `accept(Job job)` de la classe `Node` alors que le nœud n'a pas assez de mémoire ou d'opérations disponibles. Pour éviter cela, vous allez rajouter du code pour déclencher une exception `NotEnoughResourceException` si le nœud ne dispose pas de la mémoire suffisante ou ne dispose pas du nombre d'opérations suffisant pour accepter la tâche fournie.

Le message d'erreur de l'exception dépendra de la situation (pas assez de capacité de mémoire ou d'opération ou pas assez de mémoire ou d'opérations disponibles soit 4 cas à considérer au total) et qui utilisera des appels à `toString()` sur la tâche et le nœud concerné.

Par exemple, l'affectation d'une tâche nécessitant ayant 1000FLOP et 1000o sur un nœud ayant 10o et 1000FLOP devra lever une exception ayant le message suivant :

```
Node Calcul (0/1000FLOP, 0/10o) has not enough total memory to handle Job 0
    (1000FLOP, 1000o).
```

L'affectation d'une tâche nécessitant 1FLOP et 1o sur un nœud ayant 10FLOP et 10o de capacité, mais aucune opération restante (10 opérations déjà utilisées par les tâches déjà affectées) devra lever une exception ayant le message suivant :

```
Node Calcul (10/10FLOP, 1/10o) has not enough remaining FLOP to handle Job 1
    (1FLOP, 1o).
```

Créer la classe `NotEnoughResourceException` et modifier le code de la méthode `void accept(Job job)` de la classe `Node` afin qu'elle lève une exception en cas de mémoire insuffisante.

Interface `JobGenerator` et classe `UniformJobGenerator`

On souhaite générer des tâches. Pour cela, vous allez créer une classe `UniformJobGenerator`. Cette classe contiendra le constructeur et la méthode suivante :

- Le constructeur `UniformJobGenerator(int memory, int flop)` qui crée un générateur de tâches, chaque tâche générée aura une mémoire égale à `memory` compris et un nombre d'opérations égal à `flop`.
- Une méthode `Job generateJob()` qui génère et renvoie une tâche.
- Des attributs que vous aurez à définir vous-même.

La classe `UniformJobGenerator` devra implémenter une interface `JobGenerator` que vous devrez définir.

Créer l'interface `JobGenerator` et la classe `UniformJobGenerator`.

Classe `RandomJobGenerator`

On souhaite générer aléatoirement des tâches. Pour cela, vous allez créer une classe `RandomJobGenerator` qui utilisera la classe `java.util.Random`. Cette classe contiendra le constructeur suivant et devra implémenter `JobGenerator` :

— Le constructeur `RandomJobGenerator(int minMemory, int maxMemory, int minFlop, int maxFlop, long seed)` qui crée un générateur de tâches, chaque tâche générée aura une mémoire entre `minMemory` et `maxMemory` compris, un nombre d'opérations entre `minFlop` et `maxFlop` compris. La génération aléatoire sera assuré par une instance de `Random` construite à l'aide de la graine `seed`.

— Des attributs et méthodes que vous aurez à définir vous-même.

```
1  package cluster;
2
3  public class Main {
4    public static void main(String[] args){
5      JobGenerator randomJobGenerator = new RandomJobGenerator(10, 100, 10, 11,
           0);
6      Node node = new Node("Calcul", 10000, 3000);
7      for(int i = 0; i<10; i++){
8          try {
9          node.accept(randomJobGenerator.generateJob());
10         }
11         catch (NotEnoughResourceException e){
12             e.printStackTrace();
13         }
14     }
15     node.printJobs();
16   }
17 }
```

L'exécution du code ci-dessus devra donner l'affichage suivant :

```
1  Node Calcul (105/3000FLOP, 478/10000o)
2  Job 0 (11FLOP, 15o)
3  Job 1 (11FLOP, 98o)
4  Job 2 (10FLOP, 49o)
5  Job 3 (10FLOP, 82o)
6  Job 4 (11FLOP, 65o)
7  Job 5 (10FLOP, 39o)
8  Job 6 (11FLOP, 31o)
9  Job 7 (11FLOP, 14o)
10 Job 8 (10FLOP, 41o)
11 Job 9 (10FLOP, 44o)
```

Créer la classe `RandomJobGenerator`.

Ordonnanceur : interface `Scheduler`

Afin d'affecter des tâches aux nœuds, on va utiliser des ordonnanceurs (*scheduler*). Pour cela on va définir une interface. Cette interface sera générique (type paramétré) avec un seul paramètre de type que l'on nommera `J`.

Le type `J` représentera un `Job` et vous devez donc ajouter dans la définition de l'interface le code nécessaire pour forcer le fait que `J` sera une extension de `Job`. L'interface n'aura qu'une seule méthode `List<J> scheduleJobs (List<J> jobs, List<Node> nodes)` qui essaye d'affecter les tâches `jobs` (en utilisant la méthode `accept` sur le nœud) aux nœuds `nodes` et qui renvoie la liste des tâches qu'elle n'a pas affectées (faute de disponibilité des ressources en fonction des tâches déjà affectées). Ceci est le contrat de l'interface que devra respecter les classes qui l'implémenteront. L'interface en elle-même ne devra pas donner d'implémentations de ses méthodes.

Créer l'interface `Scheduler` avec la définition de méthode demandée.

Ordonnanceur aléatoire : classe `RandomScheduler`

Vous allez maintenant créer une classe `RandomScheduler` qui implémentera `Scheduler<Job>`.

Un `RandomScheduler` tente d'affecter les tâches au hasard sur les nœuds. Pour chacune des tâches à affecter, un nœud est pioché au hasard et on essaie d'y affecter la tâche. Si le nœud n'accepte pas la tâche, celle-ci est ajoutée à la liste des tâches non affectées. Pour tirer un nœud au hasard, vous pourrez utiliser une instance de `Random`.

Créer la classe `RandomScheduler` implémentant `Scheduler<Job>`.

Classe `PriorityJob`

Dans certains cas, des tâches sont plus importantes que d'autres. Afin de prendre cela en compte, vous allez créer une classe `PriorityJob` qui étendra `Job` et qui implémentera `Comparable<PriorityJob>`. Cette classe rajoutera l'attribut, le constructeur et la méthode suivants :

- un attribut `int priority` représentant la priorité de la tâche. Plus une tâche est prioritaire et plus sa priorité est élevée.
- un constructeur `PriorityJob(int memory, int priority)` qui crée un `PriorityJob` avec la mémoire et la priorité spécifiées.
- une méthode `int compareTo(PriorityJob job)` qui renvoie :
  - un entier négatif si `this` a une priorité supérieure à celle de l'argument `job`,
  - `0` si `this` et `job` ont la même priorité,
  - un entier positif si `this` a une priorité inférieure à celle de l'argument `job`,

Créer la classe `PriorityJob` étendant `Job`.

Ordonnanceur avec priorité : classe `PriorityScheduler`

Nous allez maintenant créer une classe `PriorityScheduler` qui implémentera `Scheduler<PriorityJob>`.

L'idée derrière un `PriorityScheduler` est de trier les taches par priorité. Pour trier, vous pouvez utiliser la méthode `static void sort(List<T> list)` de `Collections` qui trie une liste d'éléments pouvant se comparer à eux-même. Si vous avez implémenter correctement méthode `compareTo` de la classe `PriorityJob`, un appel à `sort` va trier la liste des tâches en mettant les tâches les plus prioritaire en premier.

Une fois les tâches triées, le `PriorityScheduler` va tenter d'affecter les tâches dans cet ordre. Pour cela, il va tenter de l'affecter à tous les nœuds. Dès qu'il trouve un nœud acceptant la tâche, le `PriorityScheduler` passe à la tâche suivante. Si jamais aucun nœud ne convient, la tâche est rajoutée à la liste des tâches non-affectées.

Créer la classe `PriorityScheduler` implémentant `Scheduler<PriorityJob>`.

Classe `Controller`

Un contrôleur (*controller*) permet d'organiser la répartition des différentes tâches sur les nœuds de la ferme. Pour cela il faut préalablement ajouter les nœuds gérés et soumettre les tâches à prendre en compte. Comme `Scheduler`, la classe `controller` définira un type paramétré avec un paramètre de type `J` qui devra être une extension de `Job`. La classe `Controller` devra avoir les méthodes suivantes :

- `void addNode(Node node)` : ajoute un nœud.
- `Controller(String name, Scheduler<J> scheduler)` : construit un contrôleur avec un nom et un ordonnanceur.
- `void submitJob(J job)` : ajoute une tâche non-affectée au contrôleur. Cette méthode devra lever une `IllegalArgumentException` si aucun nœud du contrôleur n'a assez de capacité de mémoire pour gérer la tâche.
- `void scheduleJobs()` : affecte (en utilisant le `Scheduler` donné au constructeur) les tâches non-encore affectées. Toutes les tâches qui ne sont pas affectées par le `Scheduler` sont conservées dans les taches non affectées du contrôleur.
- `void printNodesAndNonScheduledJobs()` : affiche les nœuds et les tâches non affectées.

Créer la classe `Controller` avec les méthodes demandées et les attributs qui vous semblent nécessaires.

Tests supplémentaires

Rajouter des classes et des méthodes de tests afin de vérifier la validité de votre code.