Sorbonne Université

Résolution de problème : problème d'ordonnancement

Rapport de projet

Encadrant :
Evripidis Bampis

Étudiants :
Yannis Elrharbi-Fleury
Yuan Fangzheng

Table des Matières

1 Introduction

Ce projet porte sur l'étude de solutions au problème d'ordonnancement.

Étant donné N tâches et 1 machine, il s'agit de trouver un ordonnancement de ces tâches minimisant la somme de leur temps de complétude (minimiser le temps d'attente total).

La machine ne connaît pas forcément leur durée d'exécution réelle.

Dans ce rapport, nous étudions et mesurons la qualité de plusieurs solutions en fonction de l'erreur de prédiction.

Le code est écrit en Python et sera fourni en annexe.

2 Structure du programme

Nous avons naturellement opté pour une approche orientée objet du problème.

2.1 Les distibutions

La classe *Distribution* représente un ensemble de distributions de probabilité et leurs paramètres.

Lors de son instanciation, elle prend en argument des fonctions permettant de générer un tuple de valeurs.

La méthode *sample* renvoie un tuple contenant :

- une durée réelle
- une durée erreur de prédiction
- un instant d'arrivée

2.2 Les tâches

La classe *Task* représente une tâche, dont les attributs sont générés à partir d'un objet de type *Distribution*.

Elle possède notamment comme attributs :

- un ensemble de durée : durée réelle, durée prédite (générées à partir de *Distribution*)
- un état : *paused, running, finished, not available*
- un curseur *currentStep* permettant d'avancer dans l'exécution de la tâche
- un numéro d'identification

Une tâche possède trois méthodes :

- *hasFinished* : renvoie si la tâche est achevée ou non
- *forward* : exécute la tâche d'un pas de temps, renvoie une exécution de *hasFinished*
- *restart* : réinitialise la tâche à son état initial

2.3 Les machines

Notre idée était de créer une classe *Machine* représentant une machine capable de travailler sur un ensemble de tâches. Les différents algorithmes que nous présenterons dans la partie suivante en héritent.

Chaque machine possède notamment comme attributs :

- des dictionnaires de tâches à différents états : *allTasks, workingTasks, pausedTasks, finishedTasks*
- une vitesse d'exécution
- une horloge donnant le temps de la machine
- une clée d'affichage (une fonction lmabda) permettant de trier les tâches de la machine lors de son affichage

Ainsi, *Machine* possède plusieurs méthodes concernant ses tâches : ajouter ou supprimer des tâches; démarrer, mettre en pause ou terminer une tâche.

Elle possède aussi des méthodes permettant de les traîter :

- une méthode de travail *work* faisant travailler les tâches sur un pas de temps de la machine
- une méthode abstraite de traitement *run* traîtant les tâches avant chaque étape de travail, elle permet d'introduire l'algorithme de la machine
- une méthode de démarrage *boot* démarrant la machine et la faisant tourner jusqu'à ce que toutes ses tâches soient terminées

2.4 Les machines parallèles

La classe *Parallel* fonctionne de façon analogue à *Machine*, à ceci près qu'elle ne fournit pas le travail aux tâches elle même.

Le travail sur les tâches est effectué par deux machines (*Prediction* et *Round-Robin*) que possède *Parallel*, les exécutant avec une vitesse λ et $1 - \lambda$.

3 Implémentation des algorithmes

Grâce à notre refléxion claire sur les structures de données, nous avons pu facilement implémenter les aglorithmes demandés.

Dans cette partie, nous nous contenterons de présenter le fonctionnement de ceux-ci. Nous étudierons leurs performances dans la prochaine section.

Les algorithmes suivant héritent de *Machine* ou *Parallel* et ne font que surchager la méthode *run*.

3.1 Algorithme optimal : Shortest processing time

L'agorithme *SPT* est un algorithme optimal pour ce problème.

Il connaît la durée réelles des tâches et les exécute de la plus courte à la plus grande.

```
    def run(self, step):
        if len(self.workingTasks) == 0:
            nextTask = sorted(list(self.pausedTasks.values()), key=lambda
            ↪  x:x.realLength)[0]
            self.startTask(nextTask)
        return self.work(step)
```

Sa méthode *run* est claire : si aucune tâche n'est en cours d'exécution, alors la machine exécute la plus courte.

3.2 Exécution par prédiction

Pour l'algorithme *Prediction*, la machine n'a pas accès aux durées réelles des tâches. Elle ne connait que leur durée prédite, avec par conséquent une certaine erreur.

```
    def run(self, step):
        if len(self.workingTasks) == 0:
            nextTask = sorted(list(self.pausedTasks.values()), key=lambda
            ↪  x:x.predLength)[0]
            self.startTask(nextTask)
```

```
5           return self.work(step)
```

La méthode est quasiment identique à l'algorithme précédent, à ceci près que les tâches sont triées en fonction de leur durée prédite.

3.3 Round-Robin

L'algorithme *Round-Robin* partage son travail de façon égale entre les tâches.

Cet algorithme est utile dans le cas où les prédictions sont mauvaises, car il possède un rapport de compétitivité $\max_i \frac{A(I)}{OPT(I)}$ de 2.

Notre implémentation se déroule en deux phases : l'initialisation (toutes les tâches démarrent) et l'exécution (*run*).

```
1       def _initRun(self):
2           for task in self.allTasks.values():
3               self.startTask(task)
4
5       def run(self, step):
6           if self.currentTime == 0:
7               self._initRun()
8
9           self.speed = self.initSpeed / len(self.workingTasks)
10          return self.work(step)
```

Il suffit de changer la vitesse de la machine en fonction du nombre de tâches restantes à chaque pas de temps.

3.4 Exécution parallèle

Comme vu précédemment, *Parallel* exécute *Predicition* et *Round-Robin* aux vitesses λ et $1-\lambda$.

Ainsi, lors de l'initialisation de la machine, on définit ses sous-machines :

```
1           self.speed = speed
2           self.prediction = Prediction(speed * lmb, key)
3           self.roundRobin = RoundRobin(speed * (1-lmb), key)
```

Les deux machines possèdent les même tâches en référence, et travaillent ensemble à leur avancement.

La méthode *run* prend la forme suivante :

```
    def run(self, step):
        self.currentStep += step

        self.finishTasks()

        if not bool(self):
            self.prediction.run(step)
            self.roundRobin.run(step)

        return bool(self)

    def __bool__(self):
        return bool(self.prediction) or bool(self.roundRobin)
```

Avec les méthodes *bool* renvoyant si les machines ont fini leur exécution ou non.

3.5 Exécution parallèle dynamique

4 Résulats expérimentaux

4.1 Cadre classique

4.2 Introduction des dates d'arrivée