


# Paréllisation maximale automatique. Programmée en **python**



```
nomPrenom = "TRAN VALENTIN,  
SPUZNAR BARTHELEMI"
```

# Les modules

- Digraph
- threading
- time
- statistics

```
1 from graphviz import Digraph
2 import threading
3 import time
4 import statistics
```



# Les classes


## Class Task

Classe qui permet de créer une tâche

- name : le nom de la tâche, unique dans un système de tâche donné ;
- reads : le domaine de lecture de la tâche ;
- writes : le domaine d'écriture de la tâche ;
- run : la fonction qui déterminera le comportement de la tâche (ici None)

## Class TaskSystem

Classe qui contient la liste des tâches dans tasks et les dépendances (les parents) des tâches dans dictionary



# Le constructeur de **TaskSystem**

Permet de créer des objets à partir de notre classe **TaskSystem** et définir les attributs à rentrer lors de la création des objets.

Une vérification des attributs entrés sont ensuite effectué pour empêcher les doublons de tâche et vérifier l'existence d'une tâche.

# Les fonctions de TaskSystem

## recdep

Fonction utilisée : append, recdep

Entrée : name, res

Sortie: Aucune

Fonction récursive qui contient l'algo pour la fonction getDependencies.

Pour chaque dépendance dans le dictionnaire, si la dépendance n'est pas dans la liste des dépendances "res", on l'ajoute dans la liste "res".

```
def recdep(self, name, res):  
    for dep in self.dictionary[name]:  
        if not dep in res:  
            res.append(dep)  
            self.recdep(dep, res)
```

## getDependencies

Fonction utilisée : recdep

Entrée : name

Sortie : res

Fonction qui renvoie la liste ("res") des tâches qui doivent être exécutées avec une tâche donnée.

On crée la liste vide "res" qu'on va remplir avec les dépendances grâce à la fonction recdep.

```
def getDependencies(self, name):  
    res = []  
    self.recdep(name, res)  
    return res
```

## depth

Fonction utilisée : append, max

Entrée : name

Sortie : res

Fonction qui calcule la profondeur de la tâche dans l'arbre de dépendances.

On crée une liste "T" qui va contenir le nom des parents d'une tâche.

Pour chaque dépendance dans le dictionnaire de la tâche pour laquelle on a pas vu le parent avant on ajoute le parent dans la liste "T" et on calcul ça profondeur.

On finit par renvoyer la profondeur "res".

```
def depth(self, name):  
    res = 0  
    T = []  
    depth = 0  
    for dep in self.dictionary[name]:  
        if not dep in T:  
            T.append(dep)  
            res = max(res, 1 + self.depth(dep))  
    return res
```

## runSeq

Fonction utilisée : depth, append, run

Entrée : Aucune

Sortie : Aucune

Fonction qui va réaliser les tâches de façon séquentielle mais sans réaliser de manière parallèle celles qui peuvent l'être.

On crée un dictionnaire "depths".

Calcul de la profondeur de chaque tâche et ajout dans le dictionnaire.

Exécutes les tâches qui sont à la même profondeur dans l'ordre.

```
def runSeq(self):  
    depths = {}  
    for task in self.tasks:  
        depth = self.depth(task.name)  
        if (not depth in depths):  
            depths[depth] = []  
        depths[depth].append(task)  
    for depth in depths:  
        for task in depths[depth]:  
            task.run()
```

## run

Fonction utilisée : depth, append, start, join

Entrée : Aucune

Sortie : Aucune

Fonction qui permet d'exécuter les tâches en même temps à une même profondeur.

Création d'un dictionnaire depths contenant la profondeur de chaque tâche.

Calcul de la profondeur de chaque tâche et assignation de depth en tant qu'index.

Attribut chaque tâche d'une même dans une variable p est l'ajoute à la liste ps et exécutes ensuite parallèlement toutes ces tâches.

```
def run(self):
    depths = {}
    for task in self.tasks:
        depth = self.depth(task.name)
        if (not depth in depths):
            depths[depth] = []
        depths[depth].append(task)
    for depth in depths:
        ps = []
        for task in depths[depth]:
            p = threading.Thread(target=task.run)
            ps.append(p)
            p.start()
        for p in ps:
            p.join()
```



## getTree

Fonction utilisée : node, edge

Entrée : Aucune

Sortie : graph

Fonction qui définit les propriétés du graphe.

Définition du sens du graphe en TB et format png.

Création d'un noeud à chaque tâche et liaison entre le parent de chaque tâche.

```
def getTree(self):  
    graph = Digraph(name='tree', graph_attr={'rankdir': 'TB'}, format='png')  
    for task in self.tasks:  
        graph.node(task.name, task.name, shape='circle')  
    for task in self.dictionary:  
        for edge in self.dictionary[task]:  
            graph.edge(edge, task)  
    return graph
```

## draw

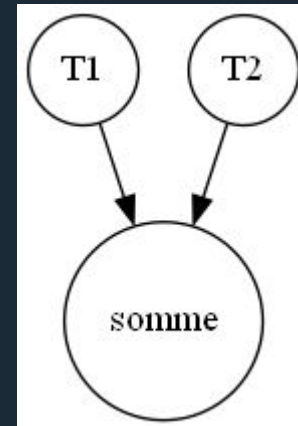
Fonction utilisée : getTree, render

Entrée : Aucune

Sortie : Aucune

Fonction qui permet de récupérer et dessiner le graphe

```
def draw(self):  
    graph = self.getTree()  
    graph.render('tree', view=True)
```



## parCost

Fonction utilisée : range, time, run, append, runSeq, mean, print

Entrée : Aucune

Sortie : Aucune

Fonction qui calcule la moyenne du temps d'exécution du système en para et seq.

Créer 2 listes de temps pour para et seq.

Répétition de 10 fois : sauvegarde de l'heure actuelle, lancement des tâches en parallèles, soustraction de l'heure actuelle au temps sauvegardé et ajout du résultat. De même pour le séquentiel.

Affichage des moyennes de temps d'exécution.

```
def parCost(self):  
    par = []  
    seq = []  
    for count in range(10):  
        startTime = time.time()  
        self.run()  
        par.append(time.time() - startTime)  
        startTime = time.time()  
        self.runSeq()  
        seq.append(time.time() - startTime)  
    print("En moyenne, sur 10 essais, le systme s'execute durant ", statistics.mean(seq),  
          " en séquentiel, tandis qu'il s'execute durant ", statistics.mean(par), " en parallele.")
```

# TEST ET RESULTAT

```
X = None
Y = None
Z = None

def runT1():
    global X
    X = 60
    time.sleep(0.5)

def runT2():
    global Y
    Y = 40
    time.sleep(0.5)

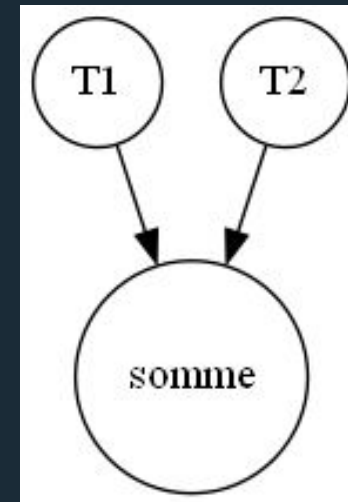
def runTsomme():
    global X, Y, Z
    Z = X + Y
    print(Z)

t1 = Task()
t1.name = "T1"
t1.writes = ["X"]
t1.run = runT1
t2 = Task()
t2.name = "T2"
t2.writes = ["Y"]
t2.run = runT2
tSomme = Task()
tSomme.name = "somme"
tSomme.reads = ["X", "Y"]
tSomme.writes = ["Z"]
tSomme.run = runTsomme

s1 = TaskSystem([t1, t2, tSomme], {"T1": [], "T2": [], "somme": ["T1", "T2"]})

s1.run()
s1.draw()
s1.parCost()
```

En moyenne, sur 10 essais, le système s'exécute durant 1.0009247303009032 en séquentiel, tandis qu'il s'exécute durant 0.5008606672286987 en parallèle.





FIN