# Projet Programmation Concurrente - 2éme Rendu

Ilyas El Bani

Augusto Sales de Queiroz

## I. Introduction :

Le projet consiste à modéliser le déplacement d'une foule dans un terrain de taille 512 x 128 représenté par une matrice. Ce terrain comportera en outre, des obstacles – aussi générés aléatoirement - qui ne peuvent pas être franchis.

Des personnes seront générées aléatoirement sur le terrain, une personne peut se déplacer dans les directions verticales, horizontales, ainsi que diagonales. Il est à noter que chaque déplacement doit forcément rapprocher la personne de la sortie.

Pour permettre la réutilisation des terrains générées avec les obstacles et personnes, on les sauvegardera dans des fichiers grâce à l'outil pickle.

Pour la premier rendu, on n'avait pas de contraintesliées au parallélisme et aux accès concurrents, puisqu'on a qu'un seul thread, du coup, toutes les instructions se font l'une après l'autre et donc sans qu'il y'ait de problèmes.

## II. Algorithmes mis en œuvre :

### a. Create Map :

```
CreateMap

Entrée:
    TerrainSize: Taille en x et en y du terrain
    Exit: Les positions que font la sortie
    N_Persons: Nombre de personnes présentent sur le terrain
Sortie:
    Map: Une matrice de la taille du terrain que codifie toutes les informations

Map <- matrice[terrainSize] of 0s

for each exit position do:
    Map[exit position] = -2

#Calculer le nombre de cases où on peut mettre des obstacles.
# 512 == Nombre de personnes maximal que peuvent être sur le terrain
# 3  == Nombre de sorties

TotalEmptyPlaces <- TerrainSize.Rows * terrainSize.Cols - 512 - 3


#GENERATION DES OBSTACLES
Tant que TotalEmptyPlaces > 0 :
    for each row in TerrainSize.Rows
        for each col in TerrainSize.Cols
            on génére un nombre aleatoire entre 0 et 1 nommé RAND
            si RAND dépasse un seuil ( 0.90 ) :
```

```
                on génere un sizeRow aleatoire entre 0 et 6
                on génere un sizeCol aleatoire entre 0 et 6
                 Si TotalEmptyPlaces – NombreCelluleDansObstacle <0 :
                          break
                on crée un obstacle commencant de (row,col) jusqu'à (row+sizeRow,
col+sizeCol)
                 TotalEmptyPlaces<- TotalEmptyPlaces – nombreCelluledansObstacle

#Liste des cases vides
on génére une liste EmptyCases, contenant toutes les cases qui sont vides

#GENERATION DES PERSONNES
Nombre_personnes <- 2**N_Persons
tant que Nombre_personnes > 0
    on génére un index "idx" aléatoire entre 1 et la taille de la liste EmptyCases
    Map[EmptyCases[idx]] = 1
    Nombre_personnes <- Nombre_personnes -1


return Map
```

## b. Move a Person:

```
Move Person

Entrée:
    Person
    Map: L'état du terrain

Sortie:
    Moved: Un boolean disant si la personne a reussi a bouger

if Person.x <= 0 and Person.y <= 0:
    # Si la personne est déjà sorti, il-n'y-a aucune chose a faire
    return False

positionsToMove = [(-1, -1), (-1, 0), (0, -1)]
for each position in positionsToMove:
    if Map[Person.x + position.x, Person.y + position.y] == 0:
        Map[Person.x + position.x, Person.y + position.y] = Person.id
        Person.x += position.x
        Person.y += position.y
        # A reussi a bouger mais pas sortir
        return True
    else if Map[Person.x + position.x, Person.y + position.y] <= -2:
        Map[Person.x + position.x, Person.y + position.y] = 0
        Person.x = -1
        Person.y = -1
        # A reussi a sortir
        return True

# N'a pas reussi a bouger
return False
```

## c. Scenario Un Seul Thread :

```
Entrée:
    Single Thread

Entrée:
        Map: L'état du terrain
```

```
Tantque (not Map.ListPersonnes.empty()):
        Pour Personne dans Map.ListPersonnes:
                Personne.EssayerDeBougerVers(Personne.SortieLaPlusProche()) // on
modifie aussi l'état de notre Map
                Si Map.estSortie(Personne.position):
                        Map.ListPersonnes.remove(Personne)
```

Pour cette partie, puisqu'il n'y a qu'un seul Thread, notre programme s'exécutera complétement en séquentiel. Du coup pas besoin de gérer les problèmes liés à la concurrence puisqu'ils n'existent pas.

## d. Scenario Un Thread Par Personne :

```
One Thread Per Person
Entrée :
        Map: L'état du terrain

Thread_list = []
Pour Personne dans Map.ListPersonnes:
        on crée un thread T qui va lancer la méthode Loop pour cette personne
        Thread_list.add(T)
        T.start()
Pour Thread dans Thread_list:
        Thread.join()


Définition de la méthode loop, propre à chaque personne:
        Tant que Map.estSortie(Personne.position):
                Map.lock[Personne.position.x][Personne.position.y].acquire()
                Map.lock[Personne.position.x-1][Personne.position.y].acquire()
                Map.lock[Personne.position.x][Personne.position.y-1].acquire()
                Map.lock[Personne.position.x-1][Personne.position.y-1].acquire()

                Personne.EssayerDeBougerVers(Personne.SortieLaPlusProche()) // on
modifie aussi l'état de notre Map

                Map.lock[Personne.position.x-1][Personne.position.y-1].Release()
                Map.lock[Personne.position.x][Personne.position.y-1].Release()
                Map.lock[Personne.position.x-1][Personne.position.y].Release()
                Map.lock[Personne.position.x][Personne.position.y].Release()
```

Dans cette partie, comme nous avons utilisé Threading en python, et comme le GIL n'utilise qu'un seul cœur. Tous nos threads auront access à la même zone mémoire. Du coup on n'a pas à envoyer des messages entre les différents Thread.

Par contre, dans cette partie là on doit bien faire attention au problème liés à la concurrence, surtout en ce qui concerne les Race Conditions et les Dead Lock.

C'est pourquoi avant d'essayer de bouger une personne, on verrouille 4 cases, la position de la personne elle-même, et les positions susceptibles qu'elle prenne dans le futur. Ceci permet d'eviter les Race Conditions.

Il est à noté que le fait de toujours acquérir les lock dans le même ordre, et de les release dans l'ordre inverse, nous a permis de ne pas avoir de problèmes de deadlock.

## e. Scenario Un Thread Par Quadrant :

```
One Thread Per Quadrant
Entrée :
 Map: L'état du terrain
 quadrant1 : Map[0:256][0:64]
 quadrant2 : Map[0:256][64:128]
 quadrant3 : Map[256:512][0:64]
 quadrant4 : Map[256:512][64:128]
 quadrantExits : contiendra les sorties correspondantes pour chaque quadrant, toujour le coté haut Gauche

ProcessList = []
Queue_Quadrant = [SharedQueue(),SharedQueue(),SharedQueue(),SharedQueue()]
pour i dans range(1,5):
 on crée un process P qui va lancer la méthode runQuadrant en copiant dans sa mémoire,
 le Quadrant correspondant et les sorties correspondantes, chaque process aura aussi access
 aux queues crées précedemment.
 P.start()
pour i dans range(1,5):
 P.join()

Définition de la méthode RunQuadrant, propre à chaque Quadrant:
 Flag = [False,False,False,False] // sert pour informer un quadrant quand il peut arreter de chercher des personnes à bouger
 ok = True // tant que c'est true, on essaye de cherche des personne à bouger

 TantQue ok ou len(quadrant.people):
  On Essaye de tirer un message dans la Queue correspondantes
  Si on reussie :
   Si Message est bloquant et vient de la part du Quadrant4:
    flag[3] = true
   Si Message  est bloquant et vient de la part du Quadrant3:
    flag[2] = true
   Si Message  est bloquant et vient de la part du Quadrant2:
    flag[1] = true
   Sinon //le message contient une personne
    quadrant.Map[Message.position.x][Message.position.y] = PERSON_IDENTIFEIR
    quadrant.people.add(Message)
  Pour personne dans quadrant.ListPersonnes:
   Map.lock[Personne.position.x][Personne.position.y].acquire()
   Map.lock[Personne.position.x-1][Personne.position.y].acquire()
   Map.lock[Personne.position.x][Personne.position.y-1].acquire()
   Map.lock[Personne.position.x-1][Personne.position.y-1].acquire()

   Personne.EssayerDeBougerVers(Personne.SortieLaPlusProche()) // on modifie aussi l'état de notre Map

   Map.lock[Personne.position.x-1][Personne.position.y-1].Release()
   Map.lock[Personne.position.x][Personne.position.y-1].Release()
   Map.lock[Personne.position.x-1][Personne.position.y].Release()
   Map.lock[Personne.position.x][Personne.position.y].Release()

   Si Personne est sortie:
    Si elle était dans le quadrant4 , on l'envoie vers le quadrant2 ou quadrant3
    Si elle était dans le quadrant3 , on l'envoie vers le quadrant1
    Si elle était dans le quadrant2 , on l'envoie vers le quadrant1
  Si len(quadrant.ListPersonnes) == 0 :
```

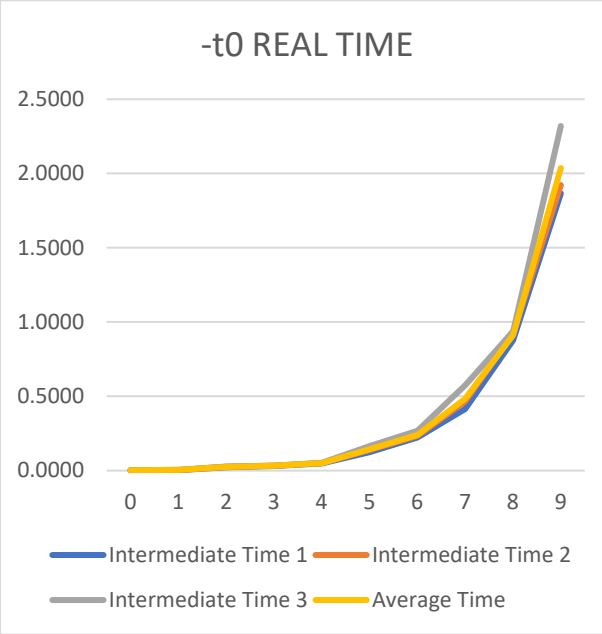
```
    si on est dans quadrant4, on envoie un message bloquant dans toutes les queues
    si on est dans quadrant3, et qu'on a deja recu un message bloquant de 4, on envoie un
message bloquant dans toutes les queues
    si on est dans quadrant2, et qu'on a deja recu un message bloquant de 4, on envoie un
message bloquant dans toutes les queues
    si on est dans quadrant1, et qu'on a deja recu un message bloquant de 4, 3 et 2, on envoie
un message bloquant dans toutes les queues
```

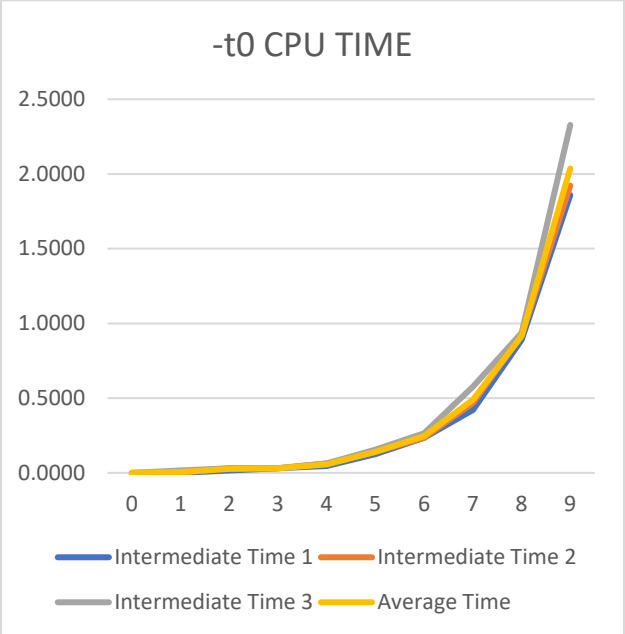
Dans cette partie, comme nous avons utilisé multiprocessing en python, pour éviter que le GIL n'utilise qu'un seul cœur. Tous nos threads/Processus auront accès à des zone de mémoire différentes donc. Du coup on a dû cherche un moyen pour envoyer des messages entre les différents processus, pour faire ça, on a utilisé Multiprocessing.Queue qui permet d'envoyer des messages et en recevoir à tout moment

De même, dans cette partie-là, On n'a pas dû faire face à des Race-conditions, puisque chaque Processus travaille sur une mémoire propre à lui-même, et comme il n y'a qu'un seul thread par processus, il n y'a donc pas de problème.

## III. Mesure du temps d'exécution pour t0:

| | Real Time | | | | CPU TIME | | | |
|---|---|---|---|---|---|---|---|---|
| N_People | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time |
| 0 | 0.0020 | 0.0020 | 0.0030 | 0.0023 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 1 | 0.0040 | 0.0040 | 0.0050 | 0.0043 | 0.0000 | 0.0000 | 0.0156 | 0.0052 |
| 2 | 0.0229 | 0.0259 | 0.0259 | 0.0249 | 0.0156 | 0.0313 | 0.0313 | 0.0260 |
| 3 | 0.0309 | 0.0339 | 0.0339 | 0.0329 | 0.0313 | 0.0313 | 0.0313 | 0.0313 |
| 4 | 0.0479 | 0.0498 | 0.0519 | 0.0499 | 0.0469 | 0.0625 | 0.0625 | 0.0573 |
| 5 | 0.1247 | 0.1386 | 0.1656 | 0.1430 | 0.1250 | 0.1406 | 0.1563 | 0.1406 |
| 6 | 0.2224 | 0.2333 | 0.2673 | 0.2410 | 0.2344 | 0.2344 | 0.2656 | 0.2448 |
| 7 | 0.4149 | 0.4657 | 0.5734 | 0.4847 | 0.4219 | 0.4688 | 0.5781 | 0.4896 |
| 8 | 0.8777 | 0.9225 | 0.9395 | 0.9133 | 0.8906 | 0.9219 | 0.9375 | 0.9167 |
| 9 | 1.8666 | 1.9224 | 2.3198 | 2.0363 | 1.8594 | 1.9219 | 2.3281 | 2.0365 |

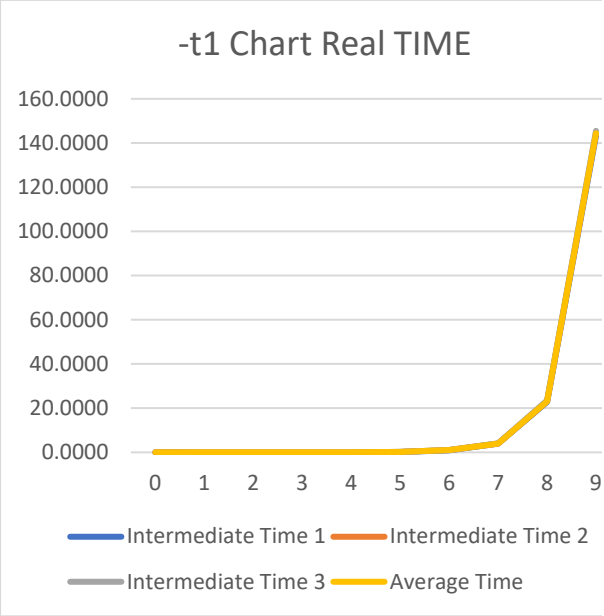


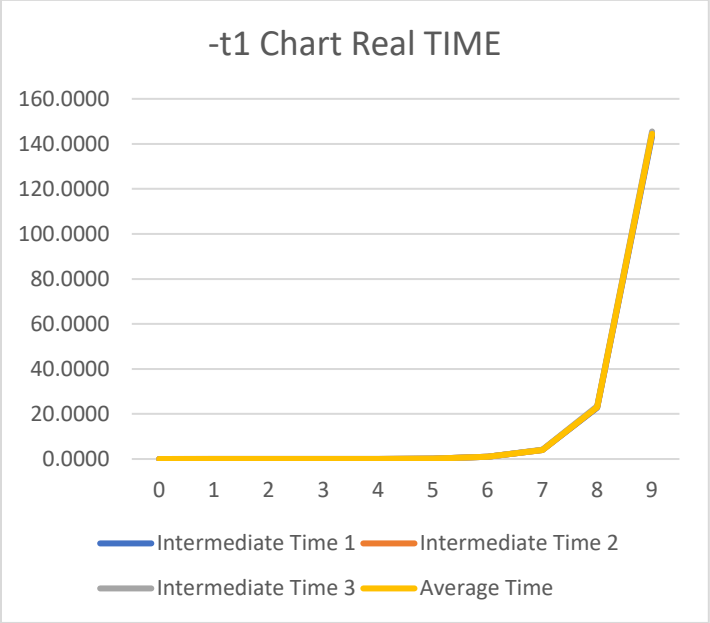



On remarque que le temps d'exécution a une allure exponentielle. Le temps Cpu est est aussi égale au temps réelle  puisque notre programme s'éxecute en séquentielle et il n y'a pas de thread qui attendent dans le background.

## IV. Mesure du temps d'éxecution pour t1 :

| | Real Time | | | | CPU TIME | | | |
|---|---|---|---|---|---|---|---|---|
| N_People | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time |
| 0 | 0.0030 | 0.0030 | 0.0040 | 0.0033 | 0.0000 | 0.0000 | 0.0156 | 0.0052 |
| 1 | 0.0130 | 0.0160 | 0.0160 | 0.0150 | 0.0156 | 0.0156 | 0.0156 | 0.0156 |
| 2 | 0.0159 | 0.0160 | 0.0189 | 0.0169 | 0.0156 | 0.0156 | 0.0156 | 0.0156 |
| 3 | 0.0379 | 0.0379 | 0.0399 | 0.0386 | 0.0313 | 0.0313 | 0.0469 | 0.0365 |
| 4 | 0.0708 | 0.0728 | 0.0771 | 0.0736 | 0.0625 | 0.0781 | 0.0781 | 0.0729 |
| 5 | 0.1433 | 0.1436 | 0.1506 | 0.1458 | 0.1406 | 0.1406 | 0.1563 | 0.1458 |
| 6 | 0.9779 | 1.0066 | 1.0383 | 1.0076 | 0.9844 | 1.0469 | 1.0469 | 1.0260 |
| 7 | 3.9612 | 4.0001 | 4.1143 | 4.0252 | 3.9844 | 4.0469 | 4.1094 | 4.0469 |
| 8 | 22.7849 | 23.0692 | 23.5182 | 23.1241 | 22.8125 | 23.0156 | 23.4375 | 23.0885 |
| 9 | 143.3680 | 144.4984 | 145.5160 | 144.4608 | 143.2031 | 144.1563 | 145.2344 | 144.1979 |



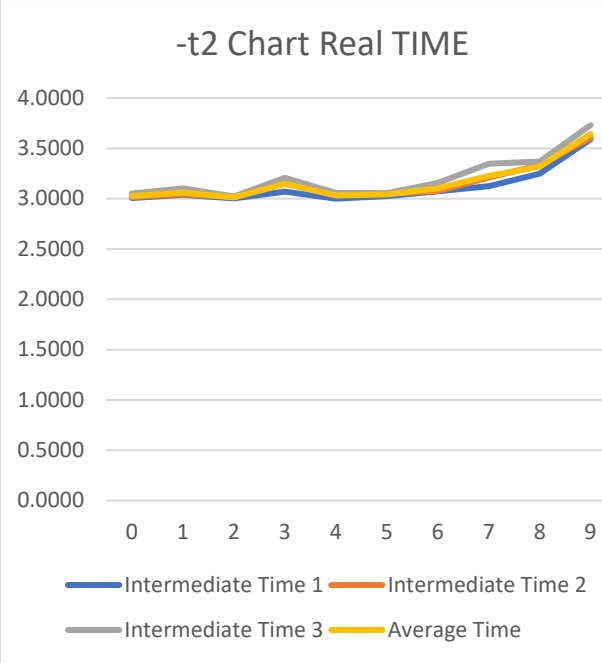



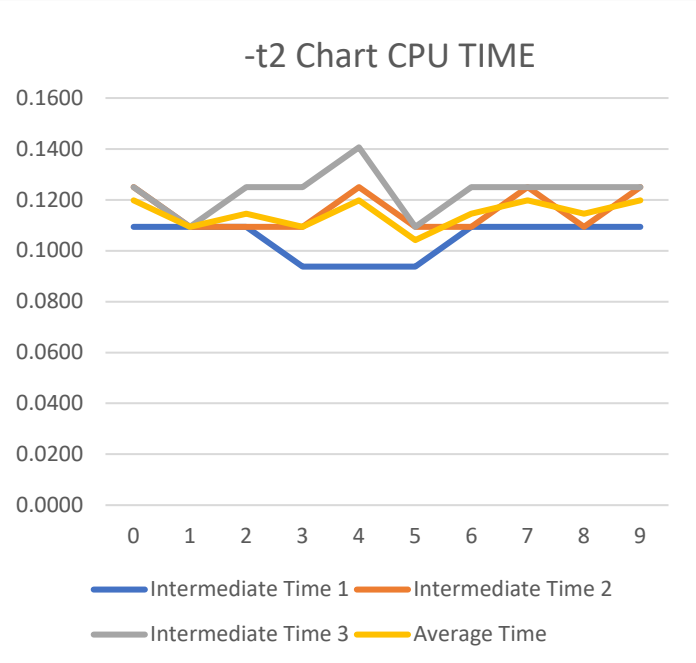

On observe que le temps pris explose exponentiellement … Avoir plusieurs thread n'est pas toujours une bonne solution, on doit gérer l'overhead causés par la gestion des locks/sémaphore, ce qui va ralentir notre programme. Avoir plusieurs thread n'est pas toujours la bonne solution.

Après nos expériences on a aussi déduit, qu'avoir plusieurs threads sur un seul cœur pour un programme aussi simple n'est pas vraiment utile. Ça n'ajoute rien. C'est plutôt utile si on veut faire quelque chose d'autre que le calcul durant l'éxecution de notre programme. L'utilisation des threads sur un seul cœur est plutôt utile si on gére des serveurs ou si on programme une interface graphique.

## V. Mesure du temps d'exécution pour t2 :

| | Real Time | | | | CPU TIME | | | |
|---|---|---|---|---|---|---|---|---|
| N_People | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time | Intermediate Time 1 | Intermediate Time 2 | Intermediate Time 3 | Average Time |
| 0 | 3.0063 | 3.0158 | 3.0512 | 3.0244 | 0.1094 | 0.1250 | 0.1250 | 0.1198 |
| 1 | 3.0372 | 3.0439 | 3.1022 | 3.0611 | 0.1094 | 0.1094 | 0.1094 | 0.1094 |
| 2 | 3.0035 | 3.0144 | 3.0222 | 3.0133 | 0.1094 | 0.1094 | 0.1250 | 0.1146 |
| 3 | 3.0691 | 3.1599 | 3.2083 | 3.1458 | 0.0938 | 0.1094 | 0.1250 | 0.1094 |
| 4 | 3.0002 | 3.0416 | 3.0565 | 3.0328 | 0.0938 | 0.1250 | 0.1406 | 0.1198 |
| 5 | 3.0251 | 3.0489 | 3.0552 | 3.0431 | 0.0938 | 0.1094 | 0.1094 | 0.1042 |
| 6 | 3.0725 | 3.0765 | 3.1568 | 3.1019 | 0.1094 | 0.1094 | 0.1250 | 0.1146 |
| 7 | 3.1241 | 3.2099 | 3.3465 | 3.2268 | 0.1094 | 0.1250 | 0.1250 | 0.1198 |
| 8 | 3.2493 | 3.3342 | 3.3678 | 3.3171 | 0.1094 | 0.1094 | 0.1250 | 0.1146 |
| 9 | 3.5915 | 3.6045 | 3.7307 | 3.6422 | 0.1094 | 0.1250 | 0.1250 | 0.1198 |




Maintenant qu'on a essayé le multithreading dans l'étape précédente, on a décidé de voir ce que ça donne d'exécuter notre programme sur plusieurs cœurs, du coup on a utilisé la bibliothèque multiprocessing.

Première chose bizarre qu'on observe, c'est que notre programme prend toujours au moins 3 secondes. Ceci est dû au temps nécessaire pour la création des 4 processus, ainsi que la copie des mémoires propres à chaque processus, comme on copié la totalité de nos map 4 fois avec quelques modification, ça a du prendre un peu de temps.

Par contre, si on ignore les 3 secondes prises pour l'initialisation, on observe que cette méthode est la plus efficace des trois, pour -p 9, elle ne prend que 0.90 seconde comparé aux 2 secondes prises par -t0.

Ceci peut être expliqué par le fait, qu'on peut faire bouger de 1 à 4 personnes à la fois en même temps. Le fait de ne pas avoir des locks à gérer aussi, rend notre programme plus rapide aussi.

En ce qui concerne le CPU TIME, il tourne autour de 0.11secondes ; ceci est dû au fait qu'on calcule que le CPU TIME du process main, on ne prend pas en compte le CPU TIME de chaque processus. C'est donc normal qu'il soit aussi faible, vû que le process main, est en attente la plupart du temps.

# VI. Conclusion :

A travers ce projet, on a pû experimenter avec threading, et multiprocessing. On a découvert leur importance, ainsi que leurs défauts. Avoir plusieurs processus/threads pour rendre un programme rapide, n'est pas toujours la bonne solution. Pour optimiser le temps d'éxecution, il faudra bien gérer la balance entre les parties concurrentes/parallèles et les parties séquentielles.

On trouve aussi que le fait de rendre un programme parallèle/concurrent, conduit à une complexité plus élevée pour nos programmes. Du coup, il faut aussi savoir quand est-ce que c'est bénéfique. Utiliser les deux notions vûs pour rendre un programme quelques secondes plus rapides n'est pas vraiment quelque chose de bon si cela complique notre programme.