Projet programmation concurrente
Polytech'Nice Sophia - SI4 G1

David BORRY
Thomas GILLOT

November 14, 2016

Nous déclarons sur l'honneur que ce rapport et l'application qu'il décrit sont le fruit de notre propre travail, basé sur notre expérience scolaire et personnelle sur ces dernières années et que nous n'avons ni contrefait, ni falsifié, ni copié partiellement sur l'oeuvre d'autres binômes ou sur internet. Nous sommes conscients que le plagiat est considéré comme une faute grave pouvant être sévèrement sanctionnée.

1 Introduction

Ce rapport a pour sujet la conception et le développement d'une simulation de déplacement d'une foule d'individus dans un environnement comprenant plusieurs obstacles, basée sur les notions vues en cours et en TD de programmation concurrente.
L'application a été écrite en C++ et utilise la bibliothèque POSIX du langage C. Pour la représentation graphique, c'est la bibliothèque SFML qui est utilisée. Les tests unitaires sont quant à eux réalisés avec Google Test.
Il n'est pas nécessaire d'avoir ces bibliothèques installées au préalable et l'application peut être compilée et éxécutée sans la partie graphique.
Le rapport a pour but d'expliquer le fonctionnement de l'application et les décisions prises pour la conception des principaux algorithmes, analyser et comparer le fonctionnement des threads POSIX par rapport à Java ainsi que les performances en fonction des différentes configurations possibles

Contents

2 Conception

La simulation a un objectif clair: faire converger une quantité plus ou moins grande de personnes vers une unique destination et faire en sorte qu'ils finissent tous par l'atteindre. Toutefois, elle a trois manières différentes de remplir cet objectif. La première consiste à déplacer successivement chaque personne d'une unité jusqu'à ce qu'elles aient toutes atteint leur destination. La seconde permet de gérer de la même manière ces personnes, mais en fonction de leur position dans l'une des quatre régions de tailles égales constituant le monde. Une thread est associée à chaque région. Pour la troisième technique, on crée une thread pour chaque personne à déplacer.

2.1 Fonctionnement général

2.1.1 Base du programme

Trois scénarios bien différents peuvent donc s'appliquer, chacun ayant ses particularités en performances et en algorithmique. Il est néanmoins important qu'elles partagent la plupart des variables et des algorithmes du programme afin qu'ils soient plus simple à maintenir et à faire évoluer. Quelque soit le scénario, on a donc toujours :

- L'ensemble des murs et des personnes à déplacer sur le terrain
- Une destination fixée à une extrémité du terrain.
- Une carte du terrain constamment mise à jour où sont représentés les obstacles qu'une personne peut rencontrer (murs ou autres personnes).
- Les algorithmes permettant de créer et de déplacer d'une unité une personne en direction d'une destination donnée.

2.1.2 Déplacement d'une personne

Le bon fonctionnement du programme repose en grande partie sur l'algorithme de déplacement. Le programme ne se terminant que lorsqu'il n'y a plus personne à déplacer, il faut un algorithme rapide et solide prenant en compte la gestion des obstacles. Un algorithme de pathfinding pour éviter les obstacles comme *Dijkstra* ou $A*$ serait inneficace car peu optimal avec un grand nombre de personnes à gérer et/ou une grande carte. En revanche, un algorithme de type *steering behaviour* principalement basé sur les notions de vecteurs et de distances est bien plus approprié. L'entité à déplacer ne pourra pas éviter les obstacles de manière intelligente s'il y en a, c'est pourquoi il est possible qu'une destination fixée soit impossible à atteindre pour certaines entités (si la destination est derrière un mur et que la seule entrée pour y accéder se situe à l'autre bout du terrain, par exemple). Le choix de la destination est donc également essentiel pour que la simulation s'éxécute correctement.

L'algorithme choisi pour la simulation fonctionne donc de cette manière :

Pour un **terrain**, une **entité** et une **destination** donnés, si l'entité n'a pas atteint la destination :

- soit une **queue** q.
- Pour chaque case **c** de **bordure**(entite)
  - ajouter **chemin**(c,destination) à q
- tant que q n'est pas vide:
  - retirer **c** le plus petit chemin
  - **deplacer**(entite,c)
  - Si l'entité a bougé, arrêter.
  - Sinon, continuer.

2.2 Gestion des threads

Si les scénarios partagent donc la plupart des ressources du programmes, ils les utilisent tous d'une manière différente. Pour l'option **-t0** où la simulation est gérée par la thread principale, le fonctionnement est assez simple : tant que toutes les personnes n'ont pas atteint la destination, répéter pour chacune d'entre elles l'algorithme de déplacement décrit précédemment. On doit donc mettre à jour le monde et l'ensemble des personnes un certain nombre de fois, ce qui peut faire penser au design d'une boucle de jeu. C'est le but: de cette manière, on peut facilement intégrer la représentation graphique à la simulation en redessinant le terrain à chaque mise à jour.
Pour l'option **-t1**, le fonctionnement est assez semblable, mais on divise le terrain en 4 parties à gérer pour chaque thread créée. Les threads ont toujours une boucle devant déplacer des entités, mais seulement celles se trouvant dans la région associée.
L'option **-t2** utilise toujours une boucle, mais elle ne gère qu'une entité par thread. La boucle continue tant que l'entité n'a pas atteint sa destination.

2.2.1 Les threads POSIX

L'année dernière en cours d'**IHM** et de **Réseaux**, nous avons déjà eu un aperçu de la gestion des threads en **Java**. L'utilisation des threads java est plutôt orientée objet car nous les avons principalement utilisées en créant des classes héritées de **Thread**, où l'on redéfinissait la méthode **run**. Cela facilite grandement le passage de paramètres car on peut directement les mettre dans le constructeur personnalisé d'une classe fille de Thread. Les threads java sont d'abord créés avec le constructeur et on les éxécute avec la méthode **start**. On attend la fin d'une tâche avec la méthode **join**. Les méthodes **suspend** et **resume** permettent de mettre en pause et de relancer une thread. En java,

on ne peut pas détruire directement une thread. La fin de l'éxécution est en général suffisante, mais on peut s'aider de la méthode **interrupt** pour arrêter la thread en avance.

Les thread POSIX ont le même objectif, mais fonctionnent différemment.
Tout d'abord contrairement à Java, la création et l'éxécution d'une thread ne sont pas séparées et sont directement gérées par la fonction **pthread_create**, qui prend en paramètre l'adresse d'un type **p_thread**, une fonction statique et un paramètre qui sera passé à cette fonction. On ne crée donc pas de classe héritée ici et le passage de paramètres est plus délicat car la fonction statique ne peut accepter qu'un seul argument. On doit donc créer par-dessus une structure contenant les différents arguments que l'on veut mettre et on la passe en paramètre à cette fonction. On doit également utiliser la fonction **pthread_join** pour attendre la fin d'éxécution d'une thread. Sans cela, le programme risquerait de s'arrêter avant. Il n'y a pas non plus de véritable fonction de mise en pause et de reprise comme en Java, on doit pour cela utiliser des **Mutex** et des **Signaux**.

2.2.2 Intégrer les threads à la simulation

Pour les deux configurations multithread, on doit faire en sorte que chaque thread puisse s'éxécuter entièrement, sans que le programme ne se termine avant. C'est rendu possible grâce aux fonctions **pthread_create** et **pthread_join** de la bibliothèque POSIX. Les algorithmes de gestion des threads sont donc très semblables pour les simulations t1 et t2. Ils fonctionnent de la manière suivante :

t1 :

- creer une liste **l**
- Pour chaque region **r** du terrain :
  - Créer une thread **t**
  - gerer_regionr,t
  - ajouter r a l
- Pour chaque thread t de l
  - attendre(t)

t2 :

- creer une liste **l**
- Pour chaque entité e :
  - Créer une thread **t**
  - gerer_personnee,t
  - ajouter r a l

- Pour chaque thread t de l
  - attendre(t)

2.2.3 Synchronisation

Plusieurs choix de synchronisations peuvent être implémentés. Le plus simple serait d'associer un **mutex**, sémaphore avec un compteur de 1, à la carte entière, mais cette solution serait inefficace et reviendrait à répéter le scénario itératif de **t0** (qui n'a pas besoin de synchronisation inter-thread puisque le scénario de t0 ne contient qu'une seule thread).

- Synchronisation de t1

  Malheureusement, pour le moment nous n'avons pas trouvé de solution plus optimale pour t1.
  En effet, la synchronisation pour t1 se fait au niveau de chaque zone et de la carte entière. Chaque zone contient une file, contenant les entités qui le contiennent. La carte entière contient une variable contenant le nombre de personnes restant sur la carte. Il existe donc un mutex sur la carte entière, car une seule thread doit pouvoir accéder à cette variable (en écriture et en lecture). Dès qu'une entité a quitté la carte, la thread correspondante accèdera à cette variable pour la décrémenter.

  Il existe aussi un mutex sur chaque zone. En effet, si une entité d'une zone quitte cette dernière, la zone courante va chercher la nouvelle zone correspondante en fonction des nouvelles coordonnées de l'entité. Dès que la nouvelle zone est trouvée, la thread va chercher à ajouter l'entité sur la file de la nouvelle zone. C'est donc à ce niveau que le mutex associé au zone entre en jeu : une seule thread doit avoir accès en écriture à la file d'une zone, si une thread y a déjà accès on bloque le processus courant.

  Comme nous disions précédemment, cette solution n'est vraiment pas la plus optimale (il y a même certains cas où des interblocages interviennent). En effet, comme nous bloquons l'accès à la variable de la carte entière à une seule thread, une seule boucle à la fois peut mettre à jour son compteur local du nombre de personne encore présente. Il y a donc un coût temporel important avec cette solution.

  Nous nous efforcerons d'améliorer au mieux cette solution (afin d'éviter, en premier, les cas d'interblocages et/ou de famines) et nous la présenterons lors de la prochaine version de ce projet et donc de ce rapport.

- Synchronisation de t2

  Pour t2, une solution plus optimale serait d'associer un mutex à chaque case, et en fonction du scénario, seulement certaines cases devront être synchronisées. La synchronisation se fait au niveau du déplacement d'une entité. Quelle que soit la direction, une entité se déplaçant va essayer d'occuper 4 cases (celles vers lesquelles on se dirige) et d'en libérer 4 (celles laissées après déplacement). La synchronisation

doit donc se faire au niveau des 4 cases qu'on essaie d'occuper. Avant de vérifier si elles sont libres ou non, on attend que le mutex de chacune des cases soit libre pour éviter une collision avec une entité occupant déjà la case. Ainsi pour chaque entité déplacée, ce sont 4 mutex qui sont utilisés.

3 Choix de développement

En dehors de la gestion des threads et des options entrées par l'utilisateur, le programme est principalement développé en C++ et tire parti de la programmation orientée objet, du polymorphisme et de la surcharge d'opérateurs.
La carte du monde est ainsi représentée par un tableau à deux dimensions contenant des cellules pouvant être **solides** (occupées par un obstacle) ou non.
Les vecteurs utilisés pour déplacer nos entités ont leur propre structure, **Vector2i** et peuvent être additionnés, soustraits, comparés par leur longueur et multipliés par un scalaire. Les régions du terrain à gérer pour le scénario **t1** ont également une structure **Rectangle** comprennant les points Nord-Ouest et Sud-Est de la région.
La classe **World** permet de gérer les déplacements d'une partie ou de la totalité des entités du terrain. Les entités à déplacer sont stockées dans une liste de pointeurs, et sont modélisées dans la plus grande classe de l'application.

3.1 Modéliser une entité

Il faut d'abord pouvoir définir ce qu'est une entité dans un monde représenté par un ensemble de cellules. Une **entité** est une forme rectangulaire composée de cellules, pouvant être solide ou non. Elle est définie par une position, une longueur, une largeur et un booléen définissant sa solidité. Elle peut également se déplacer dans toutes les directions et elle peut être détruite avant de réapparaître à sa position d'origine.
Une entité peut donc être une personne, un mur ou un trou. Les seules données qui changent à chaque type sont les dimensions et la solidité, c'est pourquoi il est inutile de créer une nouvelle classe fille pour chaque type. Il est préférable de stocker ces données dans un fichier (ici **DataTable**) afin de les utiliser simplement et rapidement dans le constructeur. Le reste des attributs et fonctions, en particulier celle de déplacement, ne changent pas. L'algorithme de déplacement cité précédemment est implémenté dans la fonction **update**.
Cette fonction n'est appelée que si l'entité a une destination a atteindre (**mTarget**). Grâce à la structure **Vector2i**, le calcul d'une direction à suivre est assez simple: on soustrait au vecteur destination un vecteur position donné. Pour l'entité à déplacer, cette position est basée sur les bordures, à savoir les cellules extérieures du rectangle. Ainsi, pour un humain occupant 4x4 cellules, 12 directions sont calculées. Ces directions sont ensuite stockées dans une structure de données **Priority Queue** où elles sont rangées dans l'ordre croissant. De cette manière, chaque retrait de la tête de file renverra la direction à la plus petite distance.

Quant au déplacement d'une case en lui-même, son implémentation est également basique: on vérifie qu'il n'y a pas d'obstacle et s'il y en a un, on peut annuler la vélocité verticale et/ou horizontale de l'entité en fonction de sa position.
On remarquera que la classe World dispose également d'une fonction **update**, dans laquelle on applique la fonction du même nom pour chacune des entités à déplacer. Comme il a été dit précédemment cela ressemble fortement à une boucle de jeu, facilitant ainsi l'intégration de la représentation graphique.

3.2 Simulation et polymorphisme

L'application en elle-même est lancée à travers la classe **Simulation**. On génère le monde et les entités dans le constructeur, et on lance la boucle dans la fonction **run**. Cette fois-ci, l'usage du polymorphisme est avantageux car cette fonction doit avoir un comportement bien différent pour chaque scénario. On crée donc des classes filles **MTSim1** et **MTSim2** respectivement associées aux configurations **-t1** et **-t2**. La fonction run est alors redéfinie pour pouvoir gérer la création des threads et l'attente de leur terminaison. Comme les threads POSIX ne peuvent pas gérer des fonctions membres directement, on recrée dans ces classes des fonctions **statiques** pour gérer les déplacements dans le monde.

4 Tester l'application

Le fonctionnement de l'application repose essentiellement sur trois données: La position du point azimuth, qui doit être accessible à tout le monde, l'algorithme de déplacement et la gestion des threads. Cette dernière est uniquement nécessaire pour les configurations **t1** et **t2**.

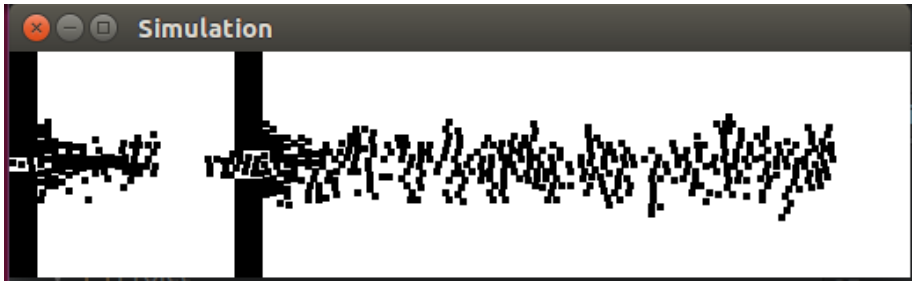
4.1 Correction

Le problème du point azimuth est facile à résoudre: il ne doit pas être entouré d'obstacles et doit être situé vers la sortie du terrain. Dans cette simulation, le monde est occupé par deux murs séparés d'une certaine distance avec un trou de largeur suffisante pour qu'on puisse le traverser. Le point azimuth est fixé dans le trou du premier mur, celui de la sortie. Il est donc placé vers le milieu du mur et chaque entité finira par l'atteindre quelque soit sa position, à condition que l'algorithme de déplacement fonctionne correctement.

Les trois scénarios ont en commun le fait qu'ils utilisent des boucles pour déplacer les individus : tant que la destination n'a pas été atteinte, continuer à essayer de déplacer. La simulation ne s'arrêtera donc qu'une fois que tout le monde aura quitté le terrain. Ces scénarios fonctionnent si le point azimuth est bien placé vers la sortie, comme le montre la figure 7. Pour vérifier l'efficacité de l'algorithme de déplacement, une série de tests unitaires doit être effectuée pour voir si le programme traite correctement le calcul des vecteurs, les directions et la gestion des obstacles. Le programme peut toutefois boucler à l'infini si on place volontairement le point à un endroit inatteignable. Aussi, si la simulation fonctionne correctement pour la thread principale, ce n'est évidemment

Figure 1: **Simulation lancée avec les options -t0 et -p9**

Figure 2: **La même simulation 40 itérations après**



pas encore le cas pour les cas **t1** et **t2**. Les entités finissent quand même par atteindre leur destination car elles utilisent toujours le même algorithme de déplacement, mais des problèmes de collisions sont présents. En effet, les ressources partagées par les threads crées dans les deux cas sont la carte du terrain et la liste des entités humaines, or elles sont constamment mises à jour. Sans modèle de synchronisation implémenté, les threads peuvent se baser sur des informations faussées ou obsolètes. Une entité peut alors se déplacer sur une case non solide, mais qui parallèlement est occupée par une autre entité déplacée par une thread. Les personnes se déplacent toujours, mais en traversant certains obstacles ou en bloquant là où il n'y en a pas.

4.2 Tests unitaires

Les tests sont placés dans un répertoire du même nom et le projet peut être compilé sans les inclures. Aussi, **il n'est pas nécessaire d'avoir installé Google Test** pour les compiler, la bibliothèque étant directement incluse dans le projet. Il faut toutefois avoir **CMake** installé pour pouvoir compiler le CMakeLists à la racine du projet. Il y aura alors une installation complète de la simulation avec les tests inclus.

Pour l'instant, ils servent principalement à tester l'algorithme de déplacement qui se base sur de nombreuses parties du programme, la première et plus importante étant les vecteurs. La structure **Vector2i** et tous les opérateurs qui y sont associés sont testés dans le fichier **vector_test.cpp**. On s'assure ainsi que le programme sait calculer

correctement une direction entre deux points et une distance minimale, ce qui constitue la base de l'algorithme de déplacement. Les déplacements des entités sont également testés dans le fichier **entity_test.cpp**. On doit vérifier qu'elle ne traverse pas d'obstacle et qu'elle adapte sa trajectoire si elle en rencontre un. On a pour cela plusieurs tests où l'on crée une entité sur une petite carte où on y a placé manuellement plusieurs obstacles et on analyse le comportement de l'entité. Combinés avec les vecteurs, ces tests prouvent l'efficacité de l'algorithme de déplacement. Pour que ces tests fonctionnent efficacement, les entités sont toujours placées au même endroit.

4.3 Performances

La simulation peut être éxécutée avec l'option **-m**, permettant d'effectuer des mesures moyennes sur le temps de réponse et de calcul du processeur. On lance ainsi la simulation 5 fois d'affilée, on effectue les mesures à chaque fois et on calcule la moyenne des trois moyennes intermédiaires à la fin. La réinitialisation de la simulation qui consiste à faire réapparaître les entités à leurs positions d'origines n'est pas incluse dans la mesure. Pour ces mesures, on met bien l'accent sur les différences entre le temps de réponse, c'est à dire le temps écoulé du point de vue d'un utilisateur, et le temps CPU consommé qui peut vite augmenter pour une application multithread. On utilise ainsi les fonctions POSIX **getrusage** et **gettimeofday** dans une classe **Measures** afin de calculer le **temps de réponse**, le **temps CPU utilisateur** et le **temps CPU système**. Les figures suivantes montrent pour les trois scénarios les différences en secondes entre les temps de réponses et CPU (utilisateur + système) pour 4, 16 et 256 personnes à déplacer. On remarquera que pour chacune de ces mesures, le temps CPU système a une valeur négligeable.

Figure 3: Comparaison des temps réels et processeurs pour 4 personnes

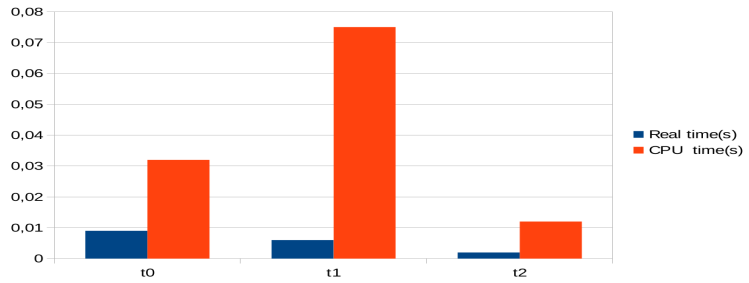


Figure 4: Comparaison des temps réels et processeurs pour 16 personnes

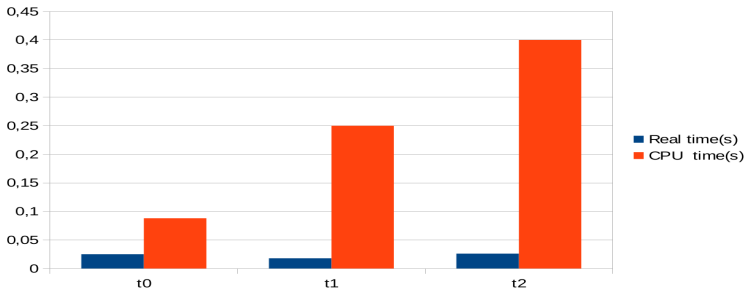
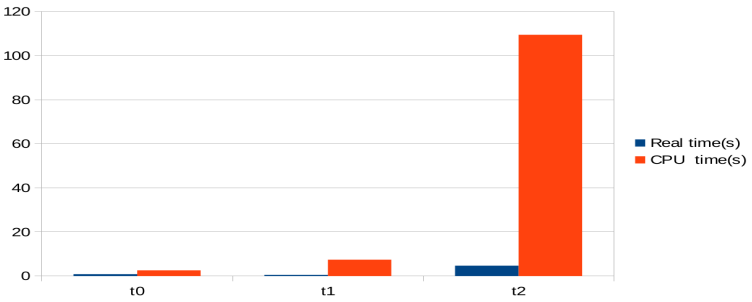


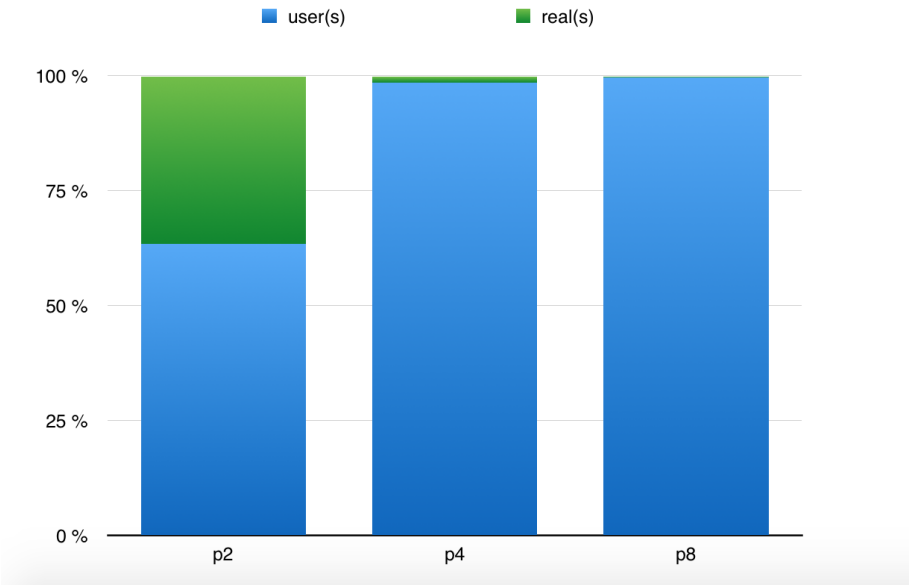
Figure 5: Comparaison des temps réels et processeurs pour 256 personnes



On peut voir que l'évolution des mesures est la même, sauf pour le cas avec 4 personnes, ce qui est normal.
En effet, les configurations **t1** et **t2** créent ici 4 threads chacune. Le temps CPU du scénario t1 est plus élevé que celui de t2 car il ne doit pas seulement gérer les déplacements d'une entité. Il doit d'abord parcourir la liste des 4 entités présentes et gérer celle située dans la région associée au thread, ce qui augmente la complexité de l'algorithme et donc le temps de calcul. Le fait que le temps CPU soit plus long que le temps réel à chaque configuration est logique: la fonction **getrusage** permet de calculer la somme des temps utilisateurs de chacune des threads actives. Plus il y a de threads, plus le temps CPU sera long. Les mesures effectuées montrent bien que la répartition des tâches sur plusieurs threads peut être avantageuse en gain de performances. Le

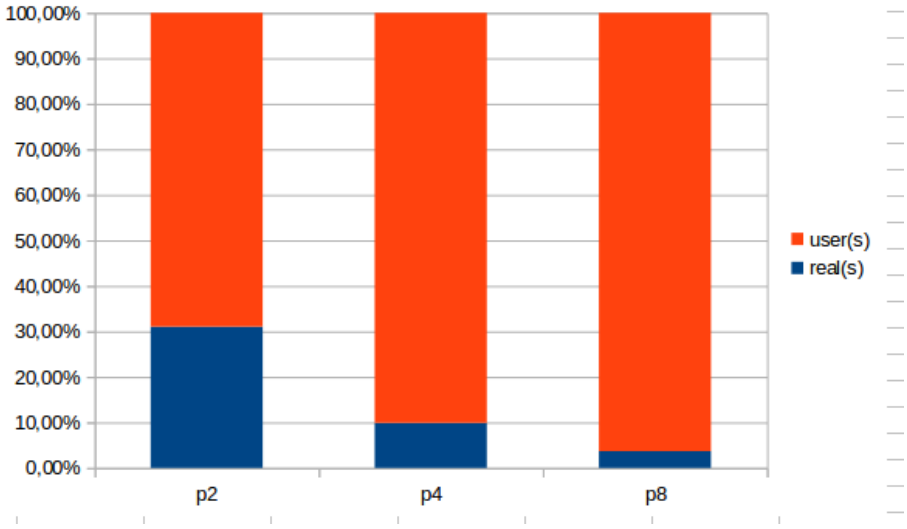
temps de réponse du programme est réduit pour les deux scénarios multithreads avec 4 personnes.
Pour 16 et 256 personnes, c'est la configuration **t1** qui est la plus optimale. Le temps de réponse est le plus petit et le temps CPU est largement supérieur, ce qui prouve que l'application bénéficie de l'usage des threads. La configuration **t2** en revanche montre ses limites à partir de 16 personnes. Le processus de création de threads est en effet coûteux et augmentera le temps de réponse si on les utilise en grande quantité. Ainsi pour 16 personnes le temps de réponse est quasiment le même que pour **t0** et pour 256 personnes, il est beaucoup plus long, ce qui rend ce scénario inefficace passé un certain nombre de personnes à déplacer.

Figure 6: **Execution de T1 en étape E2 pour 4, 16 et 256 personnes**



Nous pouvons voir, après exécution de T1 en étape E2, que l'inverse de étape E1 se produit. Le temps CPU est beaucoup moins long que le temps réel. Nous pouvons en conclure que plus le nombre de personne est important (i-e : à partir de 16 personnes) l'utilisation de T1 en étape E2 est beaucoup plus efficace que T2.

Figure 7: **Execution de T2 en étape E2 pour 4, 16 et 256 personnes**



5 Conclusion

La première étape a permis de mettre en place une simulation pleinement fonctionnelle avec la thread principale et de poser les bases pour développer les scénarios multithreads. Avec la seconde, nous avons mis en place des cas simples de synchronisations des scénarios multithreads t1 et t2. Les mesures effectuées ont bien montré l'intérêt d'un programme multithread pour ce genre de problème, qui donne un véritable gain de performances sous certaines conditions. La prochaine étape consistera donc à synchroniser efficacement les tâches et à intégrer la représentation graphique pour les scénarios multithreads.