Gestion de la Concurrence

César COLLÉ

Rémy KALOUSTIAN

POLYTECH NICE-SOPHIA

SI4

10/10/2016

1. Introduction

L'objectif de ce projet est de simuler le déplacement de personnes sur une grille et ce, à l'aide de threads qui géreront le déplacement de ces personnes. Outre l'objectif fonctionnel, nous avons un objectif pédagogique qui est de bien nous rendre compte de l'utilité et du fonctionnement des threads.

Nous avons choisi d'implémenter ce projet en C car le choix d'un langage orienté objet ne nous semblait pas nécessaire (pas de polymorphisme ni d'héritage).

2. Algorithme de déplacement

Précondition : toutes les entités de la carte sont situées dans une zone libre.

Post-condition : L'entité se déplacera obligatoirement vers la gauche, ou se mettra en attente.

Variables : P – Décrit une personne sur la carte

X – décrit l'abscisse de P ; Y son ordonné

Déplacement d'une personne

```
if(dans_limites_du_thread() && statut(P) == disponible)
    if( Y < limite_milieu1)
         if(case_bas_gauche_libre())
               bouger_bas_gauche() ;
         else if(case_gauche_libre())
               bouger_gauche() ;
         else if(case_bas _libre())
               bouger_bas() ;
    else if ( Y > limite_milieu2)
         if(case_haut_gauche_libre ())
               bouger_haut_gauche() ;
         else if(case_gauche_libre())
               bouger_gauche() ;
         else if(case_haut_libre())
               bouger_haut() ;
    else if (Y > limite_milieu1 && Y < limite_milieu2 && X > limite_gauche )
         if(case_gauche_libre())
               bouger_gauche() ;
```

3. Java vs Posix

Création :

- En C, on appelle pthread_create() en lui passant en paramètre le thread, la fonction à exécuter, et des informations complémentaires sur le comportement du thread. Une majeure partie de la création se fait via des paramètres et on a juste à déclarer une variable de type pthread_t.

- En Java, on doit tout d'abord créer une classe héritant de Thread, qui représentera notre thread. Il faudra donc Override la méthode run(). Contrairement au C, il n'y a pas de paramètres à passer lors de la création, qui se fait comme une instanciation d'objet classique. C'est une solution rapide dans certains cas.

  Le gros problème avec cette méthode c'est que le langage java n'autorisant pas l'héritage multiple, on perd donc la chance d'utiliser pleinement la puissance de Java et du polymorphisme.

  Il y a une deuxième façon de procéder à la création d'un thread en java. En effet on peut aussi effectuer l'implémentation d'une interface Runnable que l'on passera au constructeur de la classe Thread. Cette méthode est la plus générale car elle permet en outre de pouvoir hériter d'une autre classe. Pour l'utiliser il faut la donner en paramètre à un constructeur de la classe Thread.

  Nous pensons qu'utiliser l'interface Runnable est donc un choix plus judicieux, pour sa capacité de réutilisation.

Démarrage :

- En C : L'appel à pthread_create() crée et démarre le thread. Cependant, si nous ne faisons pas rejoindre le thread créé et le programme principal, il y a de fortes chances que ce dernier se termine sans que le thread créé n'ait pu s'exécuter en entier. Pour éviter cela et assurer le bon déroulement du programme, nous utilisons pthread_join(), en passant le thread en paramètre.

- En Java, il suffit dans les deux cas d'appeler la méthode start() de l'objet Thread, qui produit un appel à la méthode run() par la JVM. On peut aussi ajouter la méthode join() permettant au thread appelant d'attendre la fin de ses fils.

Arrêt :

- En C, on peut appeler pthread_exit() qui va arrêter l'exécution du thread et revenir au thread principal. Cette fonction est par ailleurs appelée implicitement à chaque fois qu'un thread à terminer son exécution.
- En Java, lorsque le thread termine son traitement il se finit. On peut aussi le stopper avec l'appel de la méthode stop().

Destruction :

- En C, on peut utiliser pthread_cancel() qui envoie une requête de destruction au thread. Il se peut que cela ne détruise pas le thread, la destruction dépendra de deux facteurs, l'état et le type de destruction qui peuvent être choisis avant la destruction.
- Java étant un langage à pile. Le garbage collector de la JVM détruit le thread ainsi que tous les composants qu'il aurait alors créés.

4. Algorithme de création des threads fils

```
Invariant : k valeurs de 0 à N

Variables : pthread[] threads, tableau de threads

boundary[] boundaries, tableau contenant la liste des limites pour
les threads

Version : version dénote la version de l'application (0, 1, 2) et
prend les valeurs associés i.e. le nombre de thread nécessaire à
chaque version.

Précondition : Chaque thread contient des limites de traitements,
que cela soit un quart de la carte pour le t1 ou toute la carte pour
le t0. Nous n'avons pas de limites de traitements pour le t2. N > 1
```

Création des threads fils

```
create_boundaries() ;
for k allant de 0 à version do
      create_thread(boundaries[k]);
      join(threads[k]);
end
```

5. Algorithme de terminaison de l'application

Pré-condition : Il y a une personne qui est disponible

Variables : person * population, liste des personnes sur la grille
int nbpeople, nombre de personnes sur la grille
P, liste des personnes disponibles

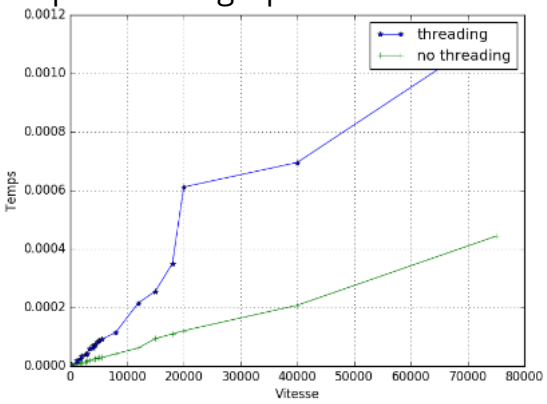
Invariant : Étant donné la post condition du déplacement (on doit automatiquement se déplacer à gauche) et la position finale où s'arrête le thread (à gauche), alors il existe un chemin C tel que pour tout thread p, p prend C et peut finir. En finissant la personne devient indisponible.

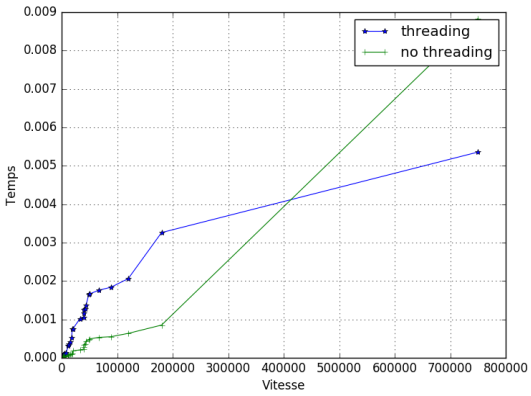
Terminaison de l'application

```
tant qu'il y a des personnes disponibles
        execute_thread_movement(P) ;
end
terminate_program() ;
```

6. Comparaison des scénarios

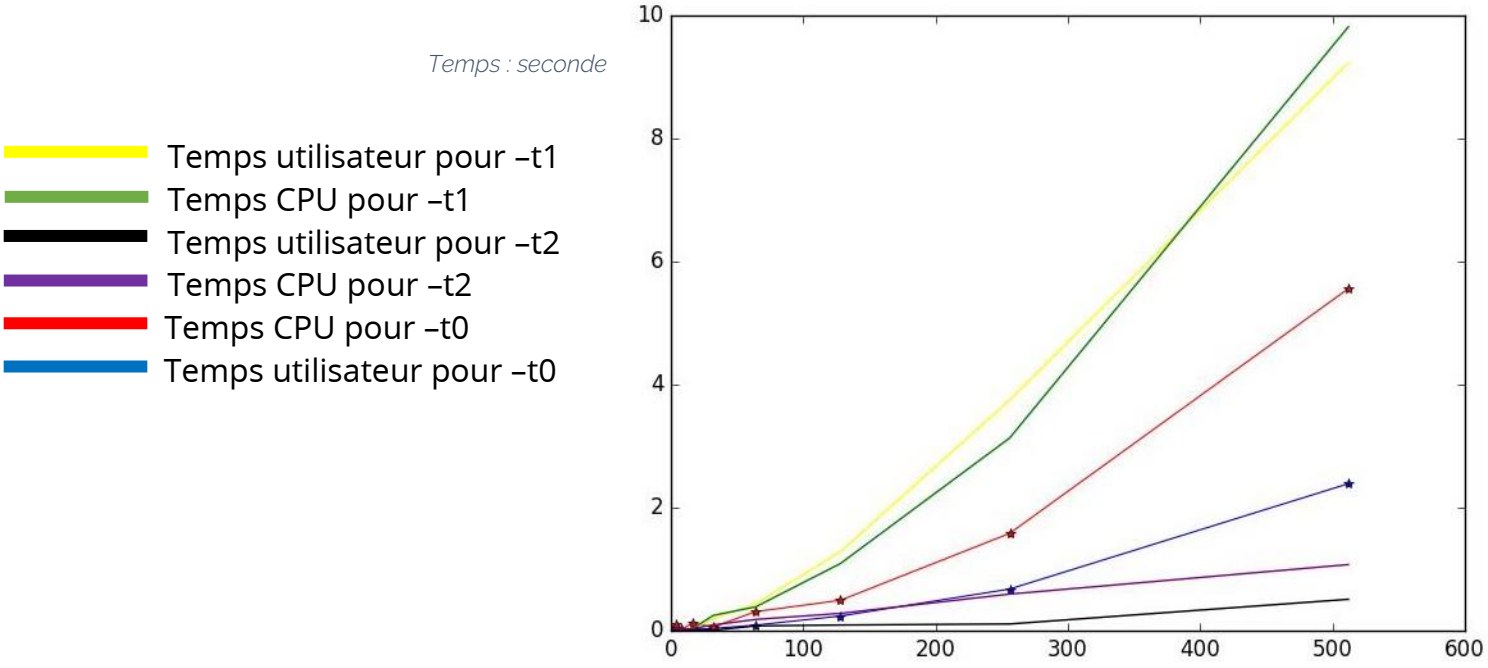
Nous avons étudié en premier temps le calcule de nombre de Fibonnaci sur différent scénarios pour prédire le comportement des threads sur notre application. Le potentiel ralentissement ou pas. On a constaté ainsi que l'utilisation de thread sur des petites données était inefficace alors que sur des données importantes nous avions de meilleurs résultats. Comme le précise les graphes ci-dessous.





Graphe temps exécution

Temps : seconde



Nombre de personnes

Ce graphe représente l'évolution du temps d'exécution de notre application en fonction des scénarios et des personnes créées.

On constate dans tous les cas que le temps utilisateur est inférieur au temps CPU. De plus le scénario le moins chronophage est celui ayant un thread par personne, ce qui est logique car au cours d'une exécution, les prochains déplacements à effectuer ne sont pas placés en attente, car ils peuvent être effectués en parallèle. On remarque d'ailleurs que le scénario avec quatre threads (un par section de la grille) est celui qui coûte le plus de temps à cause de l'attente créée par les parcours de zone.

7. Conclusion

Au niveau des choses à améliorer, nous pourrions peut-être optimiser notre code afin d'effectuer des opérations en moins et ainsi consommer moins de temps.

D'après nos mesures, nous avons remarqué que le mode -t2 consomme le moins de temps CPU et utilisateur. Cela est dû au fait que plusieurs déplacement sont effectués en parallèle, ainsi, il n'y a pas de thread à attendre pour passer au prochain déplacement, tout se fait donc plus rapidement.

Cette option se révèle alors être la plus adaptée au niveau de la minimisation du temps d'exécution.

Nous remarquons alors l'utilité et la puissance des threads qui permettent de réduire le temps d'exécution d'un programme en effectuant plusieurs tâches en parallèle.