# Gestion de la Concurrence

César COLLÉ

Rémy KALOUSTIAN

POLYTECH NICE-SOPHIA

SI4

10/10/2016

1. Introduction

2. Algorithme de déplacement

3. Java vs Posix

4. Algorithme de création des threads fils

5. Algorithme de terminaison de l'application

6. Comparaison des scénarios

7. Conclusion

# 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). En effet il ne s'agit que de déplacement d'entité sur la carte. L'emploie de tels processus sont ici jugé bien trop important pour une telle application. De plus le langage C est un langage de référence pour la programmation système

# 2. Algorithme de déplacement

**Précondition** :

- Toutes les entités de la carte sont situées dans une zone libre.
- P appartient à l'ensemble définit d'un Thread.

**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

Lmin et Lmax décrivent respectivement les lignes max et min de la zone d'arrivée. Cmax correspond à la collone la plus à gauche de l'ensemble d'arrivé

```
SI ( Y < Lmin)
        SI ( La zone bas_gauche est libre)
              bouger_bas_gauche(P) ;

        Sinon SI(La case à gauche est libre)
              bouger_gauche(P) ;

        else if (La case du bas est libre)
              bouger_bas(P) ;

    Sinon SI( Y > Lmax)
        Si (La case en haut à gauche est libre)
              bouger_haut_gauche(P) ;

        Sinon SI(La case a gauche est libre)
              bouger_gauche(P) ;

        Sinon SI(La case du haut est libre)
              bouger_haut(P) ;

   Sinon SI (Y > Lmin && Y < Lmax && X > Cmax )
        if(La case de gauche est libre)
              bouger_gauche(P) ;
```

Figure 1 : algorithme de déplacement

Ici nous mettons en lumière le fait qu'il est plus rapide pour une entité d'aller en premier temps sur sa diagonale gauche qui la rapproche le plus de l'ensemble final.

# 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 implémenter 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 a terminé 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() d'un objet Thread.

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.

# 3. Algorithme de création des threads fils

```
Invariant : k valeurs de 0 à N

Variables : threads, tableau de threads

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
associées i.e. le nombre de thread nécessaire aux version 0 et 1 respectivement 1 ou 4.

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.

Post condition : le programme principal attend la fin de ses threads fils.


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

*Figure 2 : algorithme de création de thread*

Nous remarquons ici que nous attribuons a chaque algorithme un ensemble de donnée nous permettant au niveau de l'implémentation de pouvoir réutiliser la function de déplacement de t0 facilitant les tests.

## 4. Algorithme de terminaison de l'application

```
Précondition : Il y a une personne qui est disponible

Variables :   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.

DEBUT

Tant qu'il y a des personnes qui n'ont pas terminé FAIRE

           mouvement(P) ;

   end

   Destruction des threads et fin du programme

FIN
```

*Figure 3 algorithme de fin d'application*

## 5. 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.

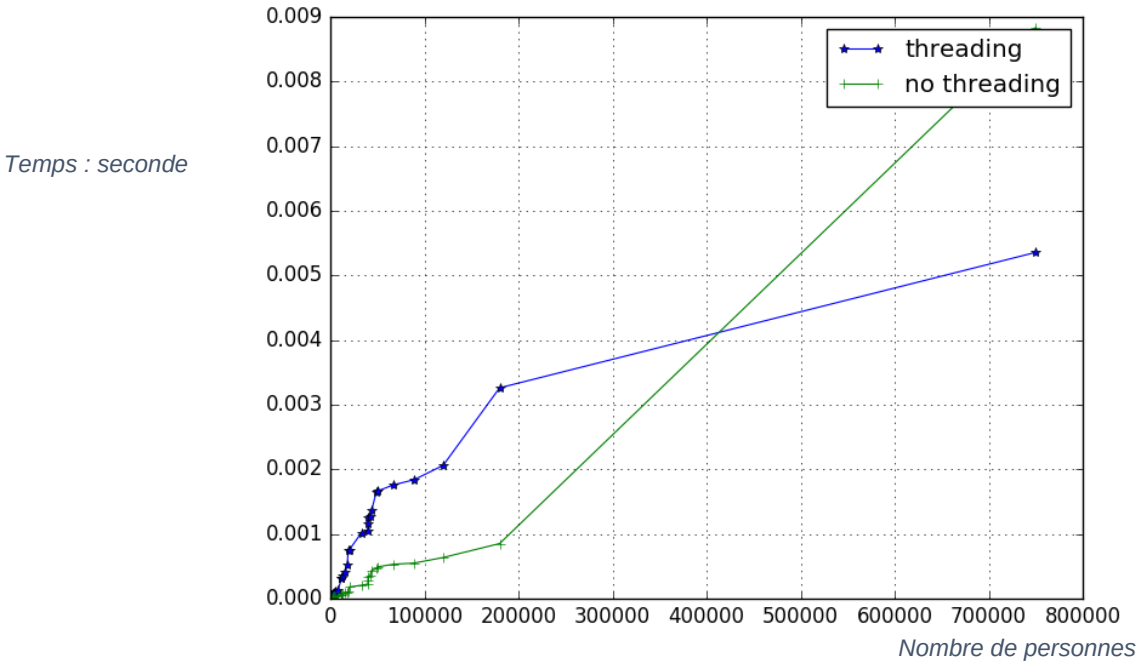


*Figure 4 : graphe d'analyse du nombre de fibonacci à calculer en fonction du temps. Le calcul se fait ici sur de grandes données*

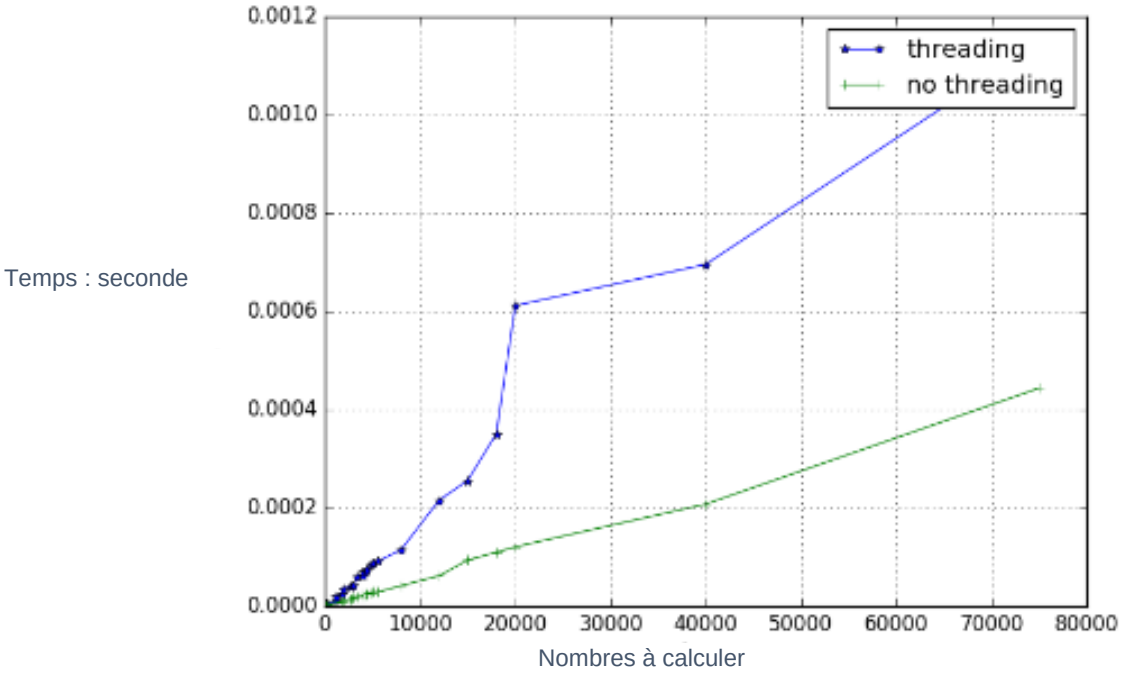


*Figure 5 : graphe d'analyse du nombre de fibonacci à calculer en fonction du temps.Le calcul ici s'effectue sur de petites données*

Cette première analyse nous a permis de comprendre l'intérêt de l'utilisation de la programmation concurrente. Et son apport en gain de temps notamment selon la situation que nous rencontrons. En effet nous constatons que l'utilisation de calcules parallèle sur des données trop petites, la création des threads est couteuse et donc on obtient de moins bon temps d'exécution qu'un calcul sans thread. Néanmoins sur des données importantes il permet d'optimiser le temps de calcule.

Nous essayerons donc dans la comparaison du temps d'exécution de notre application de trouver le même résultat que l'étude précédente.

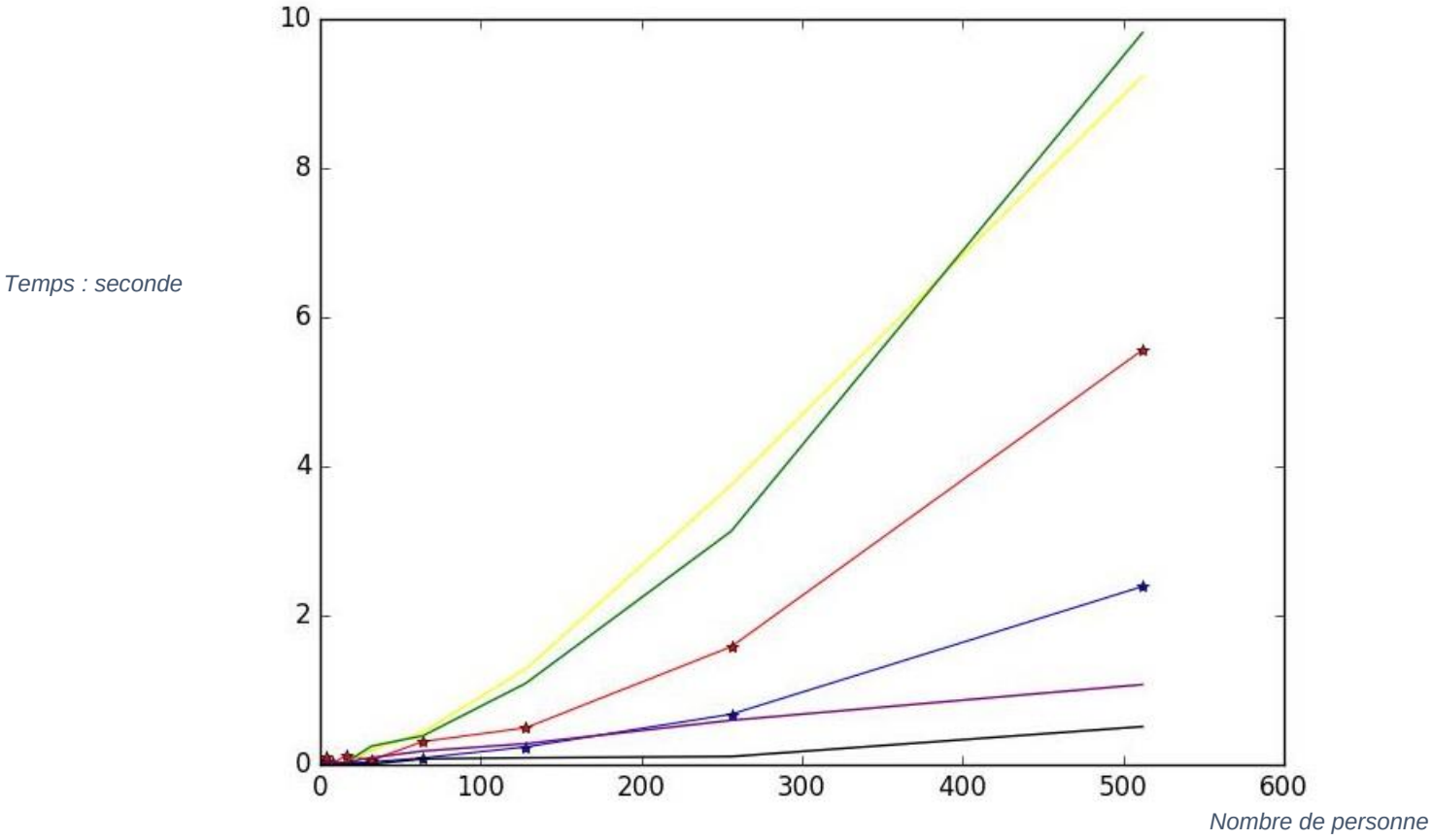


*Figure 6 Graphe temps exécution du nombre de personne en fonction du temps et des différents scénarios*

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

**Constatation générale :** On constate dans tous les cas que le temps CPU est inférieur au temps utilisateur.

## Analyse du scénario le plus rapide

De plus le scénario le moins chronophage est celui ayant un thread par personne, Malgré une couteuse création, il n'y a pas de mise en attente et tous les déplacements s'effectues « simultanément » en accord avec la gestion de processus round-robin du système UNIX. De plus si nous comparons notre implémentation, il n'y a pas tous les tests à effectuer pour savoir si une entité est située dans l'ensemble d'un thread. En effet chaque thread est indépendant dans son déplacement.

## Analyse du scénario moyen

Nous observons sur le graphe que le cas moyen est celui où la création de thread est absente. En effet l'absence de création de threads qui est couteux pour le système amène une performance sur des valeurs faibles mais conduis à un ralentissement dû a la « file d'attente » créé lorsque l'application s'exécute avec de nombreux processus.

De ce fait on a un temps d'exécution qui si situe tout de même plus proche du scénario 2 que du scénario 1. En effet « la file d'attente » est moins couteuse l'effet d'attente que nous observons pour le scénario t1 expliqué ci-après.

## Analyse du scénario le plus lent

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. De plus si on a 4 ensembles stricts alors, si toutes les entités s'alignent sur la ligne menant à la sortie alors ce sont les deux processus partageant cette ligne qui auront à T/2 le plus d'entités à déplacer provoquant une mise en attente des entités a déplacer qui seront majoritaires a T/2 dans ces ensembles en plus d'une boucle d'attente pour les deux threads qui n'auront donc pas d'entité a déplacer.

Nous pensons qu'avec une approche statistique nous pouvons remarquer que les configurations pour lesquels le scénario 1 serait optimale ne peut se produire que très peu souvent avec un placement aléatoire des entités sur la carte en vertu des propriétés liées à la théorie des grands nombres.

### Analyse général

# 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 déplacements 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.