TD3 : Multithreading with Posix Threads

Master M1 MOSIG, Grenoble Universities

Lenka Kuníková Lina Marsso

26/10/2014

1 Introduction

In this project we have to learn about POSIX Pthread primitives.

We are two sick woman in our team (angine :(), it's why the begining of this report is in english and the end in french. The code is very interessing, they are a Readme and many comment for understand.

2 Exercise 1

2.1 Question 1

The variable in the main function is an array of ID of threads. The space of this variable is allocated thanks to a malloc and initialized thanks to a pthread_create. We first have to do pthread_join so that we wait for the end of all threads. Then, we free at end the array.

2.2 Question 2

The threads are created thanks to the pthread_create function. This function needs at least 4 arguments :

- We need one pthread_t, the id of the thread.
- We can give attributes to the thread we are creating. If you put NULL, it take the default attributes.
- We need a pointer to the function that should be run by the thread.
- All the needed arguments to this function.

2.3 Question 3

When usleep is called, the program will paused/blocked for a random microsecondes. When this function is called, the thread is in a waiting state. The order of the messages printed by the supporter threadsis not the same as the order of the creation of the threads.

2.4 Question 4

The pthread_join() function waits for the thread specified to terminate. That means we have to wait the end of all threads to free the tids array. If the developer forgets the last loop in the main function, all threads won't have enough time to finish its execution. For example, in the program, we free the array before the end of the print of the song.

3 Exercise 2

```
1 //Structure arguments
2 typedef struct Arg_thread {
3         char* chant;
4         int freq;
5 } Arg_thread;
```

We decideded to create a structur, for write the frequence. And the structure can be the last argument in the pthread_create().

4 Exercise 3

4.1 Question 3

Grce notre programme (qu'on lance avec ./time.sh), nous constatons que selon le nombre de thread utilise une diffrence se creuse avec le programme lanc sequenciellement. Effectiviement plus il y a de thread moins de diffrence se creuse. Il peut arriver dans certains cas que le programme lanc sequentiellement soit plus rapide que le programme lanc thread (un trs grand nombre de thread). Nous en avons conclus, que l'utilisation des thread est performante lorsque la charge de travail est equitablement paralis, ni trop ni pas assez.

5 Exercise 4

```
1 //Global variables
2 int global_sum;
3 pthread_mutex_t lock;
```

Nous avons alors mis en place un buffer global pour stocker les valeurs retourns. Pour gerer les acces concurents cette mmoir partag entre les diffrents buffer, nous avons mis en place une gestion d'accs cette variable, les mutex. Le resultat dans notre buffer est alors finalement les valeurs dans l'ordre des terminaison des threads. Cette ordre est la plus part du temps diffrent que l'ordre de cration des threads.

6 Exercise 5

6.1 Question 3

En thorie si il y a deux coeurs, deux thread peuvent s'execute en meme temps. Ce serait deux fois plus rapide. Mais en pratique ce n'est pas le cas. Effectivement, il y a d'autres coup gestion des threads, gestion des mutex .. Autant de threads autant que de coeur, on ne peux pas faire mieux en performance.

6.2 Question 4

We can see like a best speedup 2. We can see also, the thread is usefull when we have a big size of the array in the next graph and next array. Because for a small array, the cost of thread management is more expensive then the sequential time. The theoritical speedup is two, but in reality is two minus time of threads management.

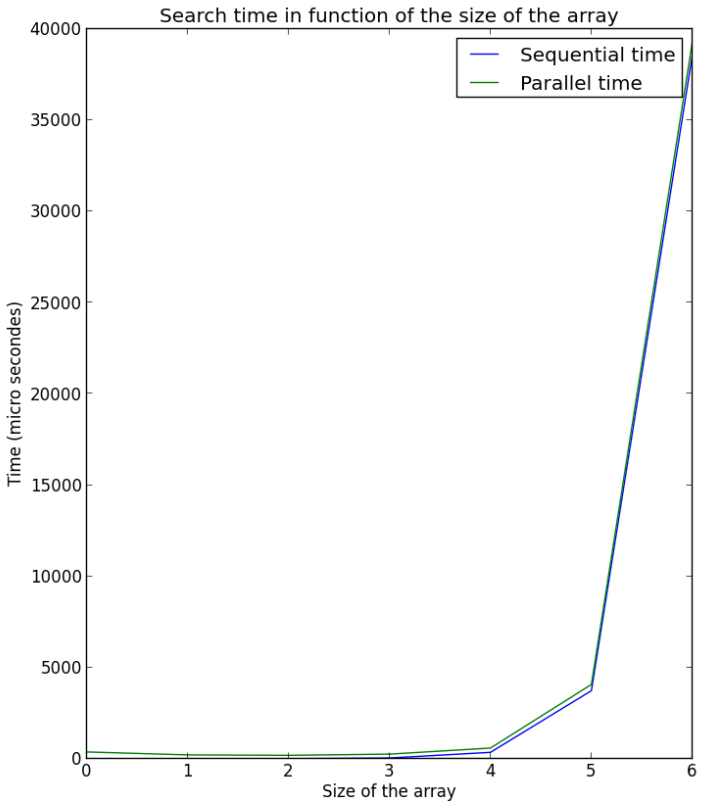

Figure 1: Search time in function of the size of the array

6.3 Table with search time in function of the size of the array

Time sequential	Time parallel	Size Array
1	248	10
2	214	100
5	216	1000
38	240	10000
353	559	100000
3677	4068	100000O
37058	37918	100000O0

Nous avons dans cette exercice, utilise l'algorithme de shuffler, pour pouvoir evaluer correctement le programme en sequentiel et en paralleliser. Effectivement, nous avons pas de graine aleatoire genere a chaque lancement de programme, mais un tableau melange aleatoirement. Dans nos test aussi, on ne teste qu'une valeur qui n'existe pas dans le tableau, pour que la chance ne corrompt pas nos evaluation de performances.

7 Conclusion

The goal of this project was to get familiar with the thread managment and I think we succeeded. We managed to implement all small program.