Conception des Systèmes d'Exploitation
Rapport sur les mesures expérimentales du tri

Line POUVARET, Mickaël TURNEL

2015-2016

1 Réponses aux Questions du TP

Question 1

Il est judicieux de placer les appels à gettimeofday juste avant et après l'exécution de l'algorithme de tri.

En particulier, il est nécessaire d'inclure la création de threads et la synchronisation finale dans la mesure du temps car il nous faut savoir combien de temps met le programme à créer ses threads et à les synchroniser.

Question 2

Oui l'ordonnanceur a une influence sur le programme car gettimeofday nous permet de mesurer un temps absolu alors que l'ordonnanceur a peut être donné la main à un autre programme pendant la mesure du temps.

Question 3

Oui le nombre de threads influe sur le temps reporté par getrusage puisque cette fonction calcule le cumul du temps passé par chaque thread du programme.

Donc, plus on créera de threads, et plus le temps reporté par getrusage sera grand.

Si on crée plus de threads que de nombre de processeurs (ou coeurs) alors les threads ne seront pas tous exécuté en parallèle mais cela n'influence pas plus ou moins sur le temps reporté par getrusage.

En revanche, cela influe énormément sur le temps écoulé depuis le début du lancement du programme.

Question 4

Puisque l'algorithme s'éxécute de façon parallèle grâce aux threads, le temps correspondant au cumul de l'ensemble des threads du processus obtenu avec getrusage sera forcément supérieur au temps obtenu avec gettimeofday (sauf si l'on exécute un seul thread uniquement, dans ce cas, le temps obtenu avec getrusage est légèrement inférieur à celui obtenu avec gettimeofday).

Question 5

Influences sur l'accélération et l'efficacité d'un programme parallèle :

- algorithmes utilisés → influe sur l'accélération et l'efficacité car cela va dépendre de la complexité de l'algorithme, des accès mémoires, opérations arithmétiques, etc...L'accélération va diminuer plus l'algorithme sera complexe et utilisera des instructions coûteuses en temps. Donc l'efficacité va diminuer aussi.

- nombre de threads utilisés → influe sur l'accélération et l'efficacité car un programme parallèle qui utilise un nombre de threads inférieur à celui d'un autre programme parallèle sera plus long. Par contre utiliser un nombre de thread supérieur au nombre de coeurs du processeur donnera lieu à un temps d'exécution plus long.
- bibliothèque de threads utilisée → influe probablement puisqu les bibliothèques de threads ont des fonctionnement différents
- le système d'exploitation → ne doit pas influer sur ces deux critères
- le nombre de processeurs de la machine → influe sur l'accélération et l'efficacité car si l'algorithme utilise par exemple un trop grand nombre de threads, le processeur va passer beaucoup de temps à passer d'un thread à un autre
- la vitesse des processeurs → influe, cela va de soit plus les processeurs seront rapides plus vite ils vont exécuter un algorithme

2 Présentation du plan d'expérience

Nous avons effectué nos tests sur la machine MandelBrot qui possède 4 processeurs.

Nous avons utilisé un script bash pour automatiser nos expériences et permettre de changer les paramètres (comme la taille du vecteur, le nombre de threads, le nombre de tests à effectuer...)

Nous avons effectué ce script bash avec un nombre de tests égal à 20 et récupéré les moyennes générées pour réaliser nos courbes.

Contenu de tri.sh

```
1  #!/bin/bash
2
3  # On supprime les fichiers si il existe deja
4  rm -f resultat_sequentiel.txt resultat_thread.txt log_sequentiel.txt log_thread.txt
5
6  # Si on a pas le bon nombre d'arguments on indique comment utiliser la commande
7  if [ $# != 5 ]; then
8          echo 'Usage : ./tri.sh <Nombre de test> <Taille du vecteur> <Min> <Max> <
       Nombre de threads>'
9          exit
10 fi
11
12 # On cree notre vecteur
13 ./creer_vecteur --size $2 --min $3 --max $4 > vecteur.txt
14
15 # On recupere le(s) resultat(s) du tri sequentiel dans le fichier log_sequentiel.
       txt
16 touch log_sequentiel.txt
17 for i in `seq 1 $1`;
18 do
19         ./tri_sequentiel --quiet --time --rusage < vecteur.txt >> log_sequentiel.
       txt
20 done
21
22 # on recupere le(s) resultat(s) du tri avec threads dans le fichier log_thread.txt
23 touch log_threads.txt
24 for i in `seq 1 $1`;
```

```
25 do
26         ./tri_threads --quiet --time --rusage --parallelism $5 < vecteur.txt >>
       log_thread.txt
27 done
28
29 # Puis on calcule la moyenne et on stocke le resultat dans le fichier
       resultat_sequentiel.txt
30 touch resultat_sequentiel.txt
31 i=1
32 while read aLine;
33 do
34         tab[$i]=$aLine
35         i=$(($i+1))
36 done < log_sequentiel.txt
37
38 count=$(($i-1))
39
40 counttime=1
41 countcpu=1
42 for i in `seq 1 $count`;
43 do
44         if (( $i%2 != 0 )); then
45                 tabtime[$counttime]=${tab[$i]}
46                 counttime=$(($counttime+1))
47         fi
48         if (( $i%2 == 0 )); then
49                 tabcpu[$countcpu]=${tab[$i]}
50                 countcpu=$(($countcpu+1))
51         fi
52 done
53
54 counttime=$(($counttime-1))
55 countcpu=$(($countcpu-1))
56
57 echo 'Temps global' >> resultat_sequentiel.txt
58
59 moyenne=0
60 for i in `seq 1 $counttime`;
61 do
62         echo "Test␣$i␣:␣${tabtime[$i]}␣micro␣secondes" >> resultat_sequentiel.txt
63         moyenne=$(($moyenne+${tabtime[$i]}))
64 done
65 moyenne=$(($moyenne/$counttime))
66
67 echo "Moyenne␣(sur␣$counttime␣tests)␣:␣$moyenne" >> resultat_sequentiel.txt
68
69 echo 'Temps CPU' >> resultat_sequentiel.txt
70
71 moyenne=0
72 for i in `seq 1 $countcpu`;
73 do
74         arr=(${tabcpu[$i]})
75         echo "Test␣$i␣:␣${arr[0]}" >> resultat_sequentiel.txt
76         moyenne=$(($moyenne+${arr[0]}))
77 done
78 moyenne=$(($moyenne/$countcpu))
79
80 echo "Moyenne␣(sur␣$countcpu␣tests)␣:␣$moyenne" >> resultat_sequentiel.txt
81
```

```
82  # Puis on calcule la moyenne et on stocke le resultat dans le fichier
        resultat_thread.txt
83  touch resultat_thread.txt
84  i=1
85  while read aLine;
86  do
87          tab[$i]=$aLine
88          i=$(($i+1))
89  done < log_thread.txt
90
91  count=$(($i-1))
92
93  counttime=1
94  countcpu=1
95  for i in `seq 1 $count`;
96  do
97          if (( $i%2 != 0 )); then
98                  tabtime[$counttime]=${tab[$i]}
99                  counttime=$(($counttime+1))
100         fi
101         if (( $i%2 == 0 )); then
102                 tabcpu[$countcpu]=${tab[$i]}
103                 countcpu=$(($countcpu+1))
104         fi
105 done
106
107 counttime=$(($counttime-1))
108 countcpu=$(($countcpu-1))
109
110 echo "Avec␣$5␣threads" >> resultat_thread.txt
111 echo 'Temps global' >> resultat_thread.txt
112
113 moyenne=0
114 for i in `seq 1 $counttime`;
115 do
116         echo "Test␣$i␣:␣${tabtime[$i]}" >> resultat_thread.txt
117         moyenne=$(($moyenne+${tabtime[$i]}))
118 done
119 moyenne=$(($moyenne/$counttime))
120
121 echo "Moyenne␣(sur␣$counttime␣tests)␣:␣$moyenne" >> resultat_thread.txt
122
123 echo 'Temps CPU' >> resultat_thread.txt
124
125 moyenne=0
126 for i in `seq 1 $countcpu`;
127 do
128         arr=(${tabcpu[$i]})
129         echo "Test␣$i␣:␣${arr[0]}" >> resultat_thread.txt
130         moyenne=$(($moyenne+${arr[0]}))
131 done
132 moyenne=$(($moyenne/$countcpu))
133
134 echo "Moyenne␣(sur␣$countcpu␣tests)␣:␣$moyenne" >> resultat_thread.txt
135
136 # Enfin on affiche les resultats
137 cat resultat_sequentiel.txt
138 cat resultat_thread.txt
```

Exemple d'exécution du script

bash tri.sh 20 100000 1 5000 4

Cette ligne de commande exécutera le script avec 20 tests pour chaque algorithme, avec un vecteur de taille 100000 comportant des valeurs comprises entre 1 et 5000, et 4 threads pour l'algorithme parallèle.

3 Résultats obtenus

Nous avons mesuré l'accélération et l'efficacité d'abord pour un nombre de threads fixe (pour 2, 4 et 8) avec les tailles de vecteur suivantes : 1000, 5000, 10000, 50000, 100000, 500000, 1000000

Nous avons mesuré ensuite l'accélération et l'efficacité pour une taille de vecteur fixe avec un nombre de threads allant de 1 à 16.

Voici les courbes que nous avons tracé à partir des données recueillies :

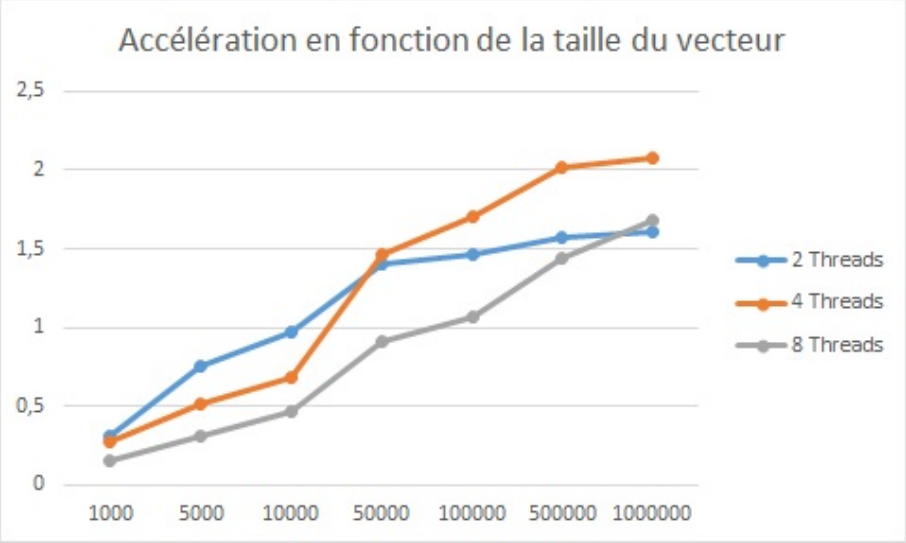

FIGURE 1

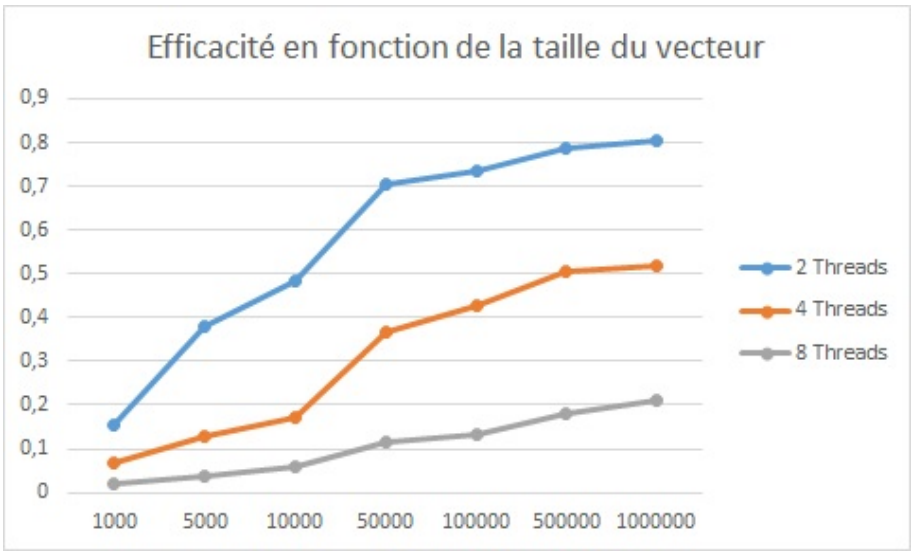

FIGURE 2

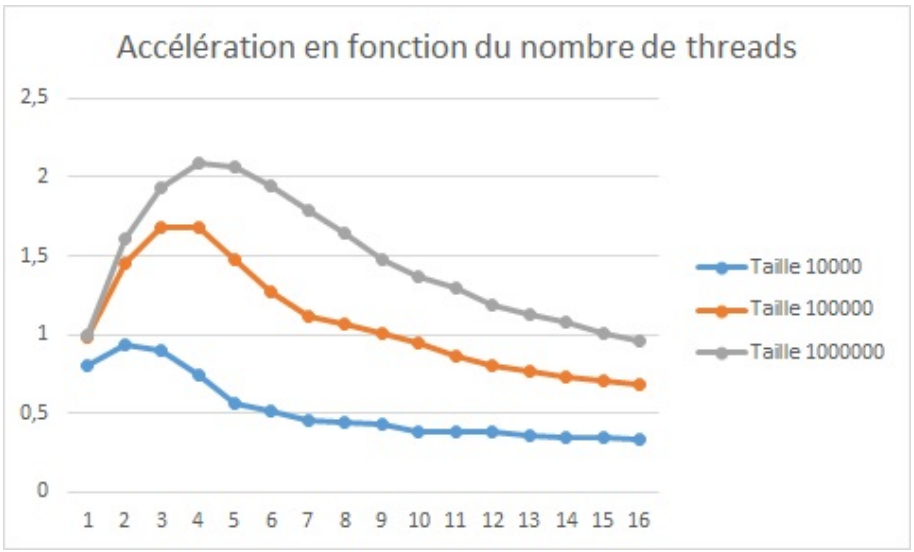

FIGURE 3

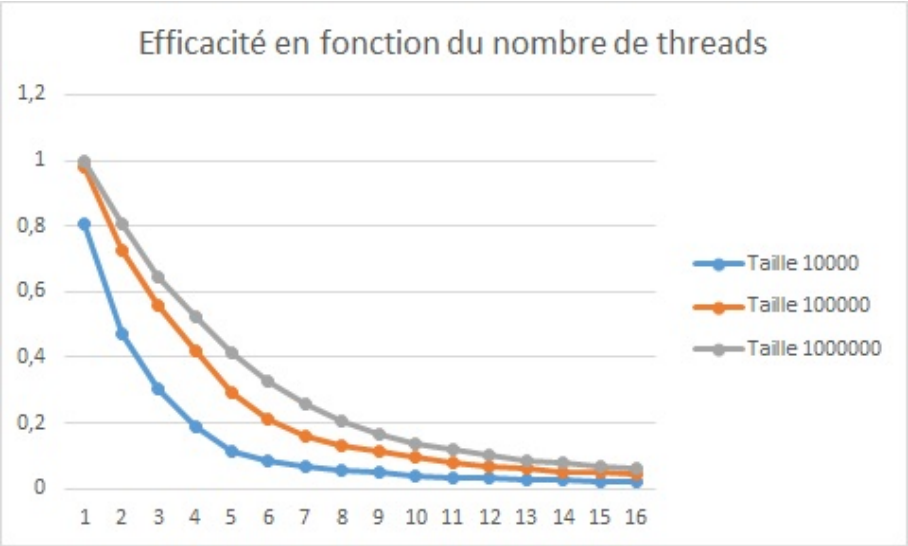

FIGURE 4

4 Analyse des résultats

Pour un nombre de threads fixé

Figure 1

On remarque que plus la taille du vecteur est grande et plus l'accélération est grande.
En effet, l'algorithme séquentiel va mettre de plus en plus de temps plus le vecteur est grand.
Alors que l'algorithme utilisant les threads va mettre de plus en plus de temps plus le vecteur est grand mais ce temps sera réparti entre les threads.

C'est donc pour cela que l'algorithme utilisant les threads sera plus rapide par rapport à l'algorithme séquentiel quand la taille du vecteur augmente.
On remarque également sur le graphique que pour une taille de vecteur inférieure à 50000, le tri sera plus rapide avec 2 threads puis après 50000, l'algorithme utilisant 4 threads a une meilleure accélération (En effet, la machine sur laquelle nous avons effectué nos tests possède 4 processeurs).

L'algorithme utilisant les threads ne devient vraiment plus rapide qu'à partir d'une taille de vecteur qui dépasse les 10000 (pour 2 threads), environ 20000 (pour 4 threads) et environ 100000 (pour 8 threads).

Figure 2

On observe que plus la taille du vecteur augmente, plus l'efficacité augmente pour un nombre de threads fixe mais les courbes ont tendance à se stabiliser au dessus de 1000000 (L'efficacité tend vers un nombre différent selon le nombre de threads).

Pour une taille de vecteurs assez petite, l'efficacité des algorithmes utilisant les threads n'est pas notable.
En effet, le temps de la création des threads, de l'exécution de l'algorithme et de la synchronisation des threads, l'algorithme séquentiel a généralement déjà effectué son tri.

Pour une taille de vecteur fixé

Figure 3

On remarque un pic dans l'accélération. En effet on peut apercevoir que pour plusieurs tailles de vecteur fixes, il y a un nombre de threads pour lesquels l'algorithme est plus rapide.

Ici pour une taille de 10000, l'algorithme a une meilleure accélération avec 2 threads. Mais on remarque que peu importe le nombre de threads, l'algorithme séquentiel est plus rapide car aucune accélération ne dépasse 1.

Puis pour des tailles de vecteur plus conséquentes, on peut apercevoir que le "pic" est presque toujours vers 4 threads. Cela peut s'expliquer par le fait que la machine n'a que 4 processeurs et donc que 4 tâches peuvent s'éxécuter parallèlement.

Figure 4

On remarque que plus on utilise de threads plus l'efficacité diminue car comme la machine utilisé n'a que 4 processeurs plus on va passer du temps à créer et synchroniser les processus. Le processeur va donc passer son temps à "switcher" les threads.