INTRODUCTION

Dans le cadre de notre projet de Théorie des Systèmes d'exploitation 1, nous devons développer deux programmes en C, l'un en séquentiel, l'autre en parallèle. Ceux-ci permettant de remplir un tableau de taille N, variable à définir lors de l'exécution du programme par l'utilisateur, par des nombres aléatoires. Par la suite le programme doit calculer la moyenne arithmétique, la moyenne absolue, la somme des carrés ainsi que la somme des cubes et donner le temps d'exécution. Nous devons ensuite à partir des données obtenues, générer à l'aide de GNUPLOT, un logiciel qui sert à produire des représentations graphiques, une courbe montrant l'évolution du temps d'exécution en fonction de N.

L'objectif de ce projet est d'apprendre l'utilisation des threads, ainsi que l'utilité de ceux-ci dans un programme. Le fait de comparer les temps d'exécution permet de ce rendre compte de l'utilité d'un programme en parallèle par rapport à un programme en séquentielle, et donc savoir utilisé les threads à bon escient.

L'utilisation des threads et les programmes parallèles sont-ils toujours une bonne alternative aux programmes séquentiels ?

I – Programmation Séquentiel

A – Création des fonctions

Nous avons quatre calculs à faires dans le cadre du programme, nous avons décidé pour des raisons de lisibilité et de maintenabilité de créer une fonction pour chaque calcul.

```c
int N;

void MARIT(int* pT)
{
    double somme=0;
    for(int i=0;i<N;i++)
    {
        somme+=pT[i];
    }
    printf("\n\nLa moyenne arithmetique est %f \n",somme/N);
}
```

Fonction MARIT permettant le calcul de la moyenne arithmétique.

Nous avons déclaré N, variable déterminant la taille du tableau, en global pour pouvoir y accéder plus facilement depuis n'importe quelle partie du programme. La fonction MARIT prend en paramètre un pointeur du tableau T permettant d'accéder aux données du tableau pour faire les différends calculs. La variable « somme » est déclarée en double car initialement nous retournions la variable. Nous nous sommes rendu compte que c'était inutile. La boucle for parcours la totalité du tableau, de i à N, pour sommer toutes ses valeurs. Enfin nous affichons la somme totale diviser par le nombre totale de valeur N du tableau T.

Une seconde fonction, MABSO, reproduit le même procédé à une différence près. Durant la somme des valeurs, nous utilisons la fonction abs() pour obtenir la somme des valeurs absolues.

```
void SCUB(int* pT)
{
    int somme=0;
    for(int i=0;i<N;i++)
    {
        somme+=pT[i]*pT[i]*pT[i];
    }
    printf("La somme des cube est %d \n",somme);
}
```

Fonction SCUB permettant le calcul de la somme des cubes.

La fonction SCUB prend, comme les fonctions MARIT et MABSO, en paramètre un pointeur vers le tableau T pour accéder à ses valeurs. Nous avons déclarer la variable somme en integer car une somme d'entier ne peut être qu'entière. Une fois encore une boucle for parcours le tableau T dans sa totalité, puis nous sommons le produit de la multiplication de trois fois la même valeur, donc un cube. Enfin nous affichons la somme.

Une dernière fonction CARRE, qui prend les mêmes paramètres que les fonctions précédentes, suit le même procédé que la fonction SCUB. Cependant ici le produit n'est pas de trois fois mais deux fois la même valeur, donc un carré.

B – Fonction main et timer

Dans un premier temps nous déclarons et remplissons un tableau T d'entier.

```
while((N<=0)||(N>200))
{
    printf("Donnez une valeur comprise entre 0 et 200 \n");
    scanf("%d",&N);
}
```

Boucle permettant de demander une valeur temps que celle-ci n'est pas comprise entre les bornes.

Pour se faire nous demandons une valeur N à l'utilisateur. Cette valeur sera utilisée à l'initialisation du tableau T. Ici les bornes sont [0,200] car dans le cadre du projet, les valeurs à tester vont jusqu'à 200. On initialise donc le tableau T.

```
int T[N];
for(int i=0;i<N;i++)
{
    T[i]= rand() % (200+100)-100;
    printf("[ %d ] ",T[i]);
}
```

Initialisation du tableau T, Remplissage du tableau T avec des nombres aléatoires.

Une fois le tableau initialisé avec une taille N. On le remplis à l'aide d'une boucle for, allant de i à N, puis nous utilisons la fonction rand() pour générer des nombres aléatoires allant de -100 à 200. Pour utiliser la fonction rand(), il faut initialiser srand() avec le temps actuel soit time(NULL).

Il ne reste plus qu'à appeler les fonctions effectuant les différends calculs.

```
//Calcul de la moyenne arithmetique
MARIT(T);

//Calcul de la moyenne absolue
MABSO(T);

//Calcul de la somme des carre
CARRE(T);

//Calcul de la somme des cube
SCUB(T);
```

Appel des fonctions.

Pour finir nous devons obtenir le temps d'exécution.

```
//on initialise le timer
gettimeofday(&tvBegin,&tzBegin);
printf("temps du debut  %d \n\n",tvBegin.tv_usec);

//timer final
gettimeofday(&tvEnd,&tzEnd);
printf("temps final %d \n",tvEnd.tv_usec);

//temps d'execution du programme
printf("Le programme a mis %d microsecondes \n",tvEnd.tv_usec-tvBegin.tv_usec);
```

Initialisation des timers, calcul et affichage du temps total.

Pour se faire nous utilisons la fonction gettimeofday() qui nécessite deux structures créées au préalable. Les structures tvBegin, tzBegin, tvEnd, tzEnd sont de type timeval et timezone. Elles permettent de récupérer le temps en seconde et en microseconde pour timeval et le type de changement d'horaire pour timezone.

Une fois la fonction appelée, on utilise les structures timeval et leur variable tv_usec de type suseconds_t pour récupérer le temps en microseconde. Il ne reste plus qu'a faire la différence entre le temps au début de l'exécution et celui à la fin de l'exécution et à l'afficher.

II – Programmation Parallèle

A – Passage du Séquentiel au Parallèle

Au passage du séquentiel au parallèle certaines modifications sont obligatoires pour permettre l'utilisation des threads. Tel que les paramètres des fonctions ainsi que l'accès à celle-ci.

```
void *MARIT(void* arg) //Fonction qui calcul la moyenne arithmetique
{
    int *tab = (int *) arg;
    double somme = 0;
    for(int i=0;i<N;i++)
    {
        somme += *tab;
        tab++; //parcour le tableau
    }
    printf("\n\nLa moyenne arithmetique est %f \n",somme/N);
    pthread_exit(NULL); //ferme le thread
}
```

Fonction MARIT du programme parallèle

Nous passons en paramètres de chaque fonction, un pointeur nommé « arg ». Nous sommes obligé de procéder ainsi car lors de la création d'un thread, que nous verrons après, la fonction pthread_create ne peut prendre en quatrième paramètre que l'adresse de la variable, ici un tableau, que l'on passe dans la fonction exécutée.

De plus, cette même fonction pthread_create ne peut prendre en troisième paramètre qu'un pointeur vers la fonction, MARIT dans le cas présent. C'est pourquoi nous avons créer la fonction de la manière suivante : void *MARIT(…).

```
int *tab = (int *) arg;
double somme = 0;
for(int i=0;i<N;i++)
{
    somme += *tab;
    tab++; //parcour le tableau
}
```

Contenu de la fonction MARIT

Pour utiliser le tableau passé en paramètre sous la forme d'un pointeur, nous créons un pointeur « *tab » qu'on faire pointer vers le paramètre « arg » qui lui-même pointe vers le tableau.

Pour parcourir le tableau nous utilisons une boucle for, héritée du programme séquentiel. Nous aurions cependant pu utiliser une boucle while. Pour sommer le contenu du tableau, nous parcourons le tableau avec un simple tab++, nous avançons de cette manière d'une case mémoire à chaque itération de la boucle for. Il suffit ensuite de sommer chaque case mémoire avec une addition entre chaque *tab, qui est la valeur à l'adresse désignée par le pointeur tab.

B – Ajout des threads

Pour créer et utiliser des threads, plusieurs étapes sont nécessaires. Dans une premier tant nous initialisons le threads.

```
pthread_t tMARIT, tMABSO, tCARR, tSCUB; //Initialisation des threads
```

Initialisations des threads

Nous créons des identifiants de threads de type pthread_t, un pour chaque threads que nous allons créer, soit un pour chaque fonction. Une fois les identifiants créer il faut créer les threads à l'aide de la fonction pthread_create.

```
if (pthread_create(&tMARIT, NULL, MARIT, &T)) //création du thread
{
    perror("pthread_create");
    exit(1); //quitte le programme si la création du thread échoue
}
```

Fonction pthread_create

Nous utilisons la fonction pthread_create pour créer le thread et lui associer une fonction ainsi qu'un paramètre de cette fonction. La premier paramètre de la fonction est un

pointeur de l'identifiant du thread. Le second paramètre désigne les attributs du thread, généralement on met NULL. Le troisième paramètre et un pointeur vers la fonction exécuté dans le thread. Enfin le quatrième paramètre est un argument qui sera utilisé dans la fonction.

Nous utilisons la fonction pthread_create dans un if, car si la fonction échoue alors le programme s'arrête avec un exit(1).

```
if(pthread_join(tMARIT, NULL)) //rejoint le thread
{
    perror("pthread_join");
    exit(1);
}
```

Fonction pthread_join

Nous utilisons la fonction pthread_join pour lancer le thread. Le premier paramètre est l'identifiant du thread, la second paramètre permet de récupérer la valeur retournée par la fonction, ici NULL car la fonction ne retourne rien.

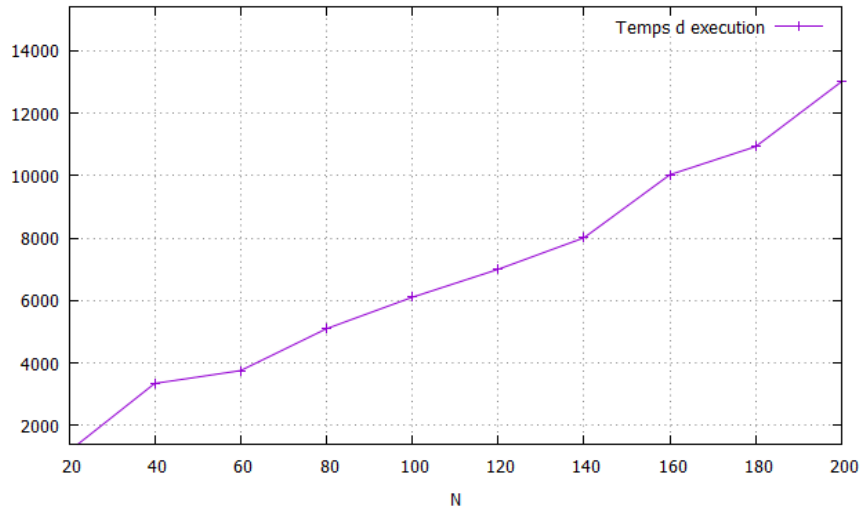
Nous utilisons la fonction dans un if car dans le cas ou elle échoue alors le programme s'arrête avec un exit(1).

Enfin pour fermer un thread à la fin de son exécution, on utilise un pthread_exit. Nous le plaçons à la fin de la fonction exécutée par le thread, ici MARIT. La fonction pthread_exit prend en paramètre la valeur qui doit être retournée par la fonction, ici NULL car la fonction ne retourne rien.

III – Analyse et Comparaison des temps d'exécution

A – Temps d'exécution du programme séquentiel

Pour analyser le temps d'exécution, nous avons exécuté le programme plusieurs avec différentes tailles de tableau, de 20 à 200. Nous avons ensuite recueilli les temps d'exécution pour chaque essaie et grâce à GNUPLOT nous avons réalisé une courbe d'évolution du temps d'exécution en fonction de la taille du tableau.

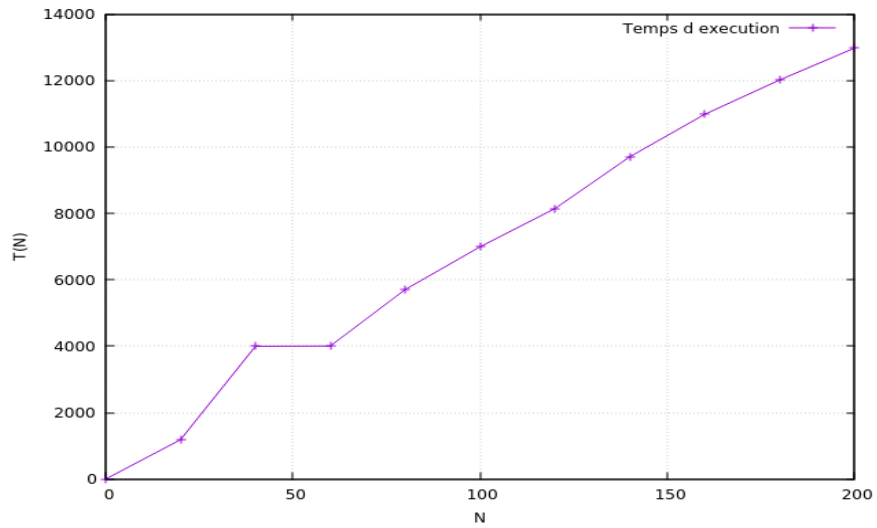


Evolution du temps d'exécution (en µs) du programme séquentiel en fonction de la taille du tableau

Cette courbe permet de voir le temps d'exécution du programme séquentiel en fonction de la taille du tableau. On constate donc que plus le tableau est grand plus le programme met de temps à s'exécution. Ces valeur pourront êtres comparer avec celles du programme parallèle pour une analyse pour approfondie.

B – Temps d'exécution du programme parallèle

Pour analyser le temps d'exécution nous avons utilisé le même procédé que pour le programme séquentiel. Nous avons exécuté le programme pour différente taille de tableau, de 20 à 200 encore une fois. Puis nous avons recueilli les données avec lesquelles nous avons créons une courbe d'évolution.

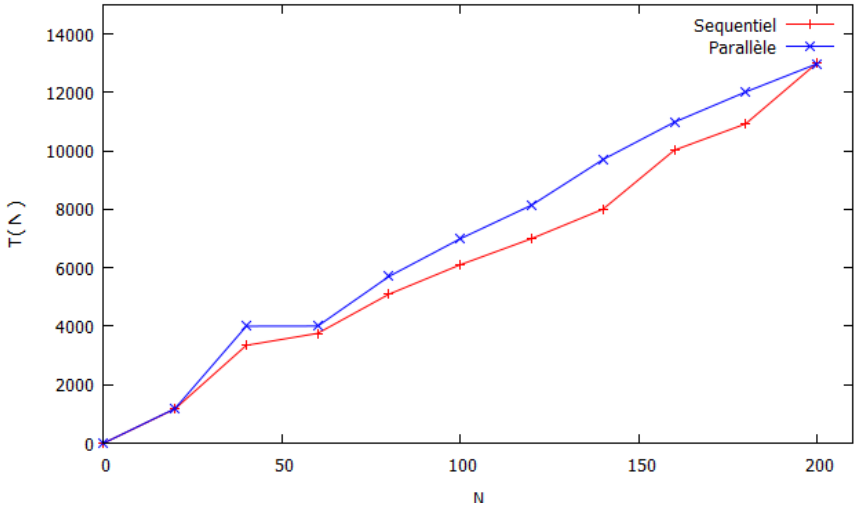


Evolution du temps d'exécution (en µs) du programme parallèle en fonction de la taille du tableau

Cette courbe permet de voir l'évolution du temps d'exécution du programme parallèle en fonction de la taille du tableau, de 20 à 200. On constate donc que plus la taille augmente plus le programme est long à s'exécuter. Maintenant que nous possédons un jeu de données pour les deux programmes nous pouvons les comparer.

C – Comparaisons des temps d'exécution

Maintenant que nous possédons des données pour chaque programme nous pouvons les comparer et en tirer une analyse.

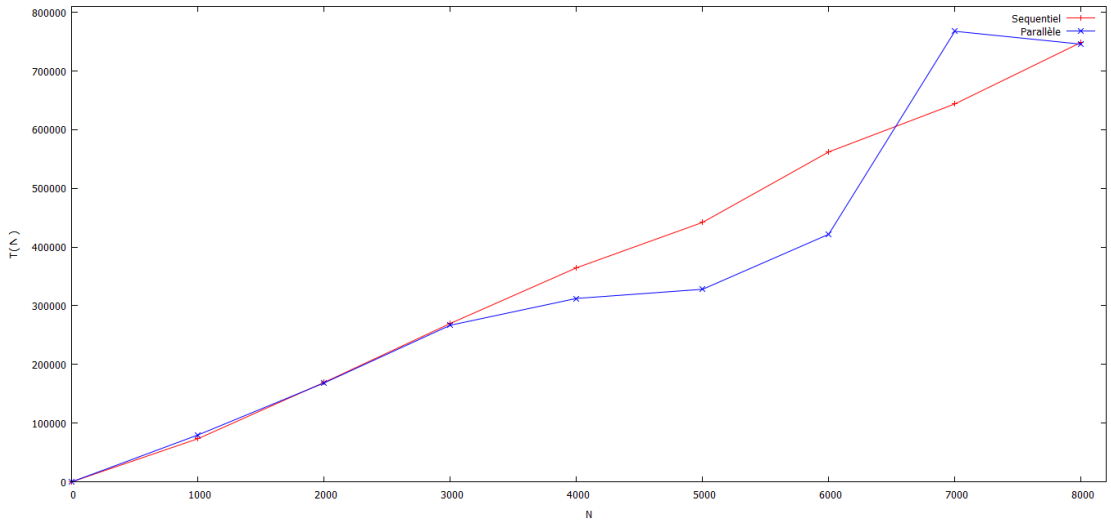


Courbe d'évolution du temps d'exécution (en µs) des programmes parallèle et séquentiel en fonction de la taille du tableau.

Ces courbes nous permettent de constater l'évolution du temps d'exécution des programmes séquentiel et parallèle en fonction de la taille du tableau, de 20 à 200. Nous pouvons ainsi comparer les temps et juger de l'efficacité et de l'utilité des threads.

Ici on constate qu'au-delà d'une taille de tableau égale à 20 le programme parallèle mets plus de temps à s'exécuter. Cependant pour une taille de tableau égale à 200 on constate que le temps d'exécution est à peu de chose près le même. Cela nous permet d'émettre l'hypothèse que les threads seront plus efficace pour de grande valeur, ici au-delà de 200, car les différentes fonctions s'exécute en parallèle. On suppose que si le programme parallèle est plus long que le séquentiel pour des valeurs inferieur à 200, c'est parce que la création et l'utilisation des threads prend un temps non négligeable pour des opérations aussi courte.

Pour vérifier notre hypothèse nous avons exécuté les programmes avec des tailles de tableau allant jusqu'à 8000.



Courbe d'évolution du temps d'exécution (en µs) des programmes parallèle et séquentiel en fonction de la taille du tableau

Ces courbes nous confortent dans notre hypothèse, l'utilisation des threads réduit le temps d'exécution d'un programme, mais seulement si les opérations effectuée ne sont pas trop courte.

CONCLUSION

Ce projet nous à permis d'apprendre à utiliser et surtout quand utiliser les threads en programmation C. On constate grâce à la comparaison des courbes qu'en utilisant les threads le temps d'exécution peut être diminué car les différentes tâches s'exécutent alors en parallèle. Cependant l'utilisation des threads ne doit pas être automatique car d'en certain cas, le temps d'exécution d'un programme séquentiel est déjà très court et l'ajout de threads n'est pas nécessaire car le temps de création des threads n'est plus négligeable.

Ce projet nous a permis de mieux appréhender la programmation parallèle et sont application dans différent contexte. Cela nous à aussi permis de mieux comprendre d'autre aspect de la programmation tel que les différent passage de paramètre ou les pointeurs.