TP 1 - Protocole experimental

Hugo Bolloré[1] and Mohammed Salah Ibnamar[2]

[1]hugo.bollore@uvsq.fr
[2]mohammed-salah.ibnamar@uvsq.fr

L'analyse de performances requiert un protocole exprérimental robuste afin de garantir des mesures stables et reproductibles.

Il existe plusieurs métriques de performance (puissance utilisée, nombre d'instructions exécutées, nombre d'accès aux caches/disques, etc.) mais nous nous intéresserons ici au temps, qui est la métrique la plus utilisée et, dans beaucoup de cas, nécessaire en corrélation des autres métriques (45 accès disques ne signifient pas la même chose sur un programme qui dure 1 sec que sur un programme qui dure 2h).

Le but de ce TP est de construire, en C, un environnement d'évaluation de performances. Nous identifierons les problèmes liés à la mesure de temps et aux trasnformations du compilateur.

Nous utiliserons une boucle DAXPY (Double Alpha time X plus Y) qui s'écrit comme suit :

```
for (i = 0; i < N; i++)
   Y[i] = alpha * X[i] + Y[i];
```

Les tableaux X et Y contiennent N réels de type double, N est un entier positif et alpha est un réel de type double.

Ce type de noyaux de calculs (kernels) est la base de beaucoup de programmes scientifiques comme la multiplication matrice-vecteur ou le calcul d'un gradient.

Pour l'ensemble des exercices nous utiliserons le compilateur GNU (GCC) fournit sur un bon nombre de distributions Linux.

Exercice 1.1 Implémentation

Programmez le code DAXPY en prenant en argument du programme la taille du vecteur (N) ainsi que la valeur de la variable alpha.

Ainsi la commande *./daxpy 10 2.4* permettra d'exécuter le code DAXPY avec N=10 et alpha=2.4.

Les tableaux C et Y devront être alloués dynamiquement (via la fonction calloc).

Question 1.1.1

Quel est l'avantage de prendre les valeurs en paramètre d'exécution ? Quelles autres solutions seraient envisageables ?

Question 1.1.2

Pourquoi est-il nécessaire de bien vérifier les arguments du programme ? Considérez que vous allez exécuter votre programme de nombreuses fois.

Exercice 1.2 Mesures

Il existe plusieurs façons de prendre les mesures :

1. La commande time : mesure le temps pris par un programme pour s'exécuter (voir man time),

2. La fonction time : effectue le même calcul mais cette fois ci via un appel de fonction,

3. Le compteur RDTSC (ReaD Time Stamp Counter) : donne le nombre de cycles depuis le dernier démarrage. Vous pouvez récupérer la valeur de ce registre facilement en assembleur ou à l'iade de la fonction *getticks()* du fichier http://www.fftw.org/cycle.h,

4. Les compteurs matériels : monitore un certain nombre d'événements liés au comportement de l'architecture pendant l'exécution d'un programme.

Question 1.2.1

Lancez la *commande* time sur le programme */bin/sleep 5*. Expliquez les trois valeurs renvoyées par time.

Question 1.2.2

Evaluez le temps pris par le code DAXPY à l'aide de la *commande* time pour alpha = 2,5 et :

1. N allant de 1 à 100 par pas de 10

2. N allant de 10000 à 100000 par pas de 10000.

Quelle est l'évolution du temps d'exécution ?

Question 1.2.3

Utilisez les commandes *perf record ./votre-programme* puis *perf report* pour obtenir le nombre de cycles que prend le programme pour s'exécuter.

Question 1.2.4

Répétez plusieurs fois l'opération. Quel problème apparaît ?

Question 1.2.5

Pour résoudre le problème observé précédemment, on se propose d'ajouter une boucle de répétition et de calculer la moyenne (ou la médiane) des mesures effectuées. Le nombre de répétitions devra être un argument du programme.

Relancez la dernière expérience avec un nombre de répétition *"suffisant"*.

Exercice 1.3 Compilateur

La programmation d'un environnement d'expériences doit éviter les pièges liés â la compilation. En effet, le code qui sera exécuté ne sera pas forcément celui qui a été écrit en C. Il faut alors garantir que ce que nous mesurons est exactement ce que nous pensons mesurer.

Question 1.3.1

A la place de la boucle DAXPY, on considère à présent le produit scalaire suivant :

```
s = 1.0;
for (i = 0; i < N; i++)
{
   s += Y[i] * X[i];
}
```

Programmez ce bout de code et compilez le avec l'option *-O3*.

Question 1.3.2

Mesurez les performances avec les commandes de *perf*. Que remarquez-vous ?

Question 1.3.3

Ajoutez un affichage de la variable *s* et mesurez à nouveau. Que remarquez-vous ? Quelle explication peut-on fournir ?

Question 1.3.4

Comparez le temps d'exécution en compilant avec -O0 ou -O3, quelle remarque pouvez-vous faire ?

Question 1.3.5

Maintenant on se propose de coder une version pondérée du produit scalaire :

```
alpha = XX;
s = 0;
for (i = 0; i < N; i++)
{
   s += alpha * Y[i] * X[i];
}
printf("Valeur de s : %g\n", s);
```

Implémenter ce code dans le même environnement que les questions précédentes en remplaçant XX par les valeurs 0.0, 1.0 et 2.0 (il faut donc recompiler à chaque fois). Utilisez un N assez grand.

Compilez avec *-O3 -ffast-math*. Qu'observe-t-on ? Ce cas arrive-t-il si alpha est donné en paramètre du programme à l'exécution ?

Question 1.3.6

Ce problème peut aussi se régler en utilisant la compilation séparée. On compile la fonction dans un autre fichier de sorte que le compilateur ne puisse plus savoir ce qu'il advient de s.

Sortez la boucle de calcul dans une fonction dont le prototype est :

```
void daxpy(double * Y, double * X, double alpha, int N);
```

Implémentez cette fonction dans un autre fichier que votre fonction main(). On compilera alors avec :

```
gcc -O3 -Wall -ffast-math -c daxpy.c
gcc -O3 -Wall -ffast-math -c driver.c
gcc daxpy.o driver.o -o tp1
```

Vérifiez que le problème ne se pose plus même si l'on n'affiche plus s.

Exercice 1.4 Granularité de la mesure

Les exercices précédents ont ajouté beaucoup de code afin d'éviter les pièges ainsi qu'à la préparation du code à mesurer, mais inutile pour la mesure de la performance du calcul.

Question 1.4.1

Utilisez la fonction *getticks()* fournit par le fichier http://www.fftw.org/cycle.h afin d'accéder au compteur RDTSC. Utilisez cette dernière pour ne mesurer que le temps de notre boucle de calcul.