Inge5SE – Programmation parallèle 08/10/2020 – TP2 : OpenMP

Alexandre Berne, Etienne Hamelin

aberne@inseec-edu.com, ehamelin@inseec-edu.com

1 Introduction des TPs

Vous allez vous rendre compte que les résultats d'exécutions sont dépendant de l'architecture de votre machine et ils défèrent donc de binômes en binômes. De plus, une fois un groupe constitué restez ensemble de semaines en semaines (même si vous êtes fâchés).

Si vous avez des questions n'hésitez pas à nous interpeler. Egalement si vous voulez nous montrer votre travail.

Rendu du TP : vous déposerez sur http://campus.ece.fr avant mercredi soir un dossier ZIP, portant le nom **TPx-Nom1-Nom2.zip**, contenant :

- Le rapport, portant le nom **TPx-Nom1-Nom2.pdf**
  Vos codes modifiés (codes source, Makefile, fichiers d'entrées, etc., mais pas les binaires compilés ni les images produites). Votre code doit compiler et s'exécuter sans erreur sur notre machine Linux.

2 Connaître sa machine

Vous allez mesurer les performances de plusieurs programmes sous Linux. Vous pouvez utiliser de préférence une machine Linux native, sinon un Mac, si vous avez Windows10, préférez le Windows Subsystem for Linux (WSL)[1], en dernier recours une machine virtuelle Linux (exemple Virtualbox[2]).

Pour que ces mesures soient exploitables, pensez bien à :

- exécuter tous les exercices sur la même machine,
- attribuer au moins 2 ou 4 cœurs à la machine virtuelle, si vous utilisez Virtualbox,
- brancher votre portable sur le secteur[3],
- arrêter tous les programmes « gourmands » en parallèle.

La première étape est de trouver le nombre de cœur disponible sur votre machine. Sous Linux, vous trouverez ces informations en tapant : `lscpu` (ou `cat /proc/cpuinfo`). Faites de même pour trouver la quantité de RAM sur votre système (`lsmem` ou `cat /proc/meminfo`).

Q1 (0pts) : Combien votre machine a-t-elle de cœurs physiques ? de cœurs logiques ? de RAM ?

Quel système utilisez-vous ? (e.g. Linux natif, WSL/Windows10, Mac, Machine virtuelle/Windows 8, etc.)

[1] Comment installer & utiliser : voir https://doc.ubuntu-fr.org/wsl
[2] Installez les extensions invité, pour pouvoir partager des fichiers avec le système hôte Windows
[3] Sinon, l'OS est susceptible d'ajuster les performances (fréquence CPU et RAM) selon le niveau de batterie

3 Analyses temporelles pthread

Les programmes ln2_approx_base.c et ln2_approx_special.c sont fonctionnellement identiques. Comme vous pouvez le remarquer, ils ont quelques différences.

Q2 (1pt) : Compilez les deux programmes. Relevez les latences de ces deux programmes.

Q3 (2pts) : Tentez d'expliquer la différence entre ces deux valeurs.

Q4 (1pts) : Selon vous, quelles sont les avantages et inconvénients de chacun des deux programmes ?

4 Introduction à OpenMP

4.1 Performances parallèles ln2

La semaine précédente vous avez travaillé avec le code ln2_approx.c et vous deviez utiliser les pthreads pour paralléliser le code.

Aujourd'hui nous allons utiliser OpenMP pour réaliser la parallélisation. Pour cela, installez-le package `libomp-dev` sur votre machine. Lorsque vous compilez un programme qui utilise OpenMP vous devez ajouter `-fopenmp` dans votre commande de compilation.

De plus lorsque vous souhaitez modifier le nombre de thread d'OpenMP, vous devez modifier la variable d'environnement de votre shell `OMP_NUM_THREADS` afin qu'elle contienne le nombre de threads souhaité.

Q5 (0.5pt) : Mesurez à nouveau la latence de ce programme sans parallélisation.

Q6 (3.5pt) : Faites la modification de ce code afin de le paralléliser avec OpenMP. Concentrez-vous sur une approche par parallélisme de données.

Q7 (0.5pt) : Quelle est la latence une fois la parallélisation réalisée ? Faites varier le nombre de threads de 1 à 10.

Q8 (1pt) : Tracez la courbe du speedup en fonction du nombre de threads : $S(n) = \frac{real(1)}{real(n)}$

Q9 (0.5pt) : Tracez la courbe du rapport suivant : $\frac{S(n)}{n}$ *en fonction du nombre threads*

Q10 (1pt) : Que représente le ratio $\frac{S(n)}{n}$ *?*

4.2 Performances parallèles gradient

Le gradient est un vecteur permettant de montrer la variation de la valeur d'un point par rapport aux points qui l'entoure. Dans notre cas, la direction du gradient en un point est orientée vers le point de plus clair et sa norme indique l'intensité de la variation.

Ce programme un peu plus complexe permet, à partir d'une image, d'extraire les contours et de générer une nouvelle image avec nos fameux gradients. Le code va récupérer une image et réalise le calcul de gradient dans la direction x et y puis génère l'image de sortie. La couleur indiquant l'angle du vecteur et la saturation varie en fonction de l'intensité de la variation.

Rien que pour vos yeux, voici les fichiers générés lorsque l'on fait les calculs de gradients seulement dans une seule direction et des niveaux de gris sont utilisés pour indiquer l'intensité de la variation :

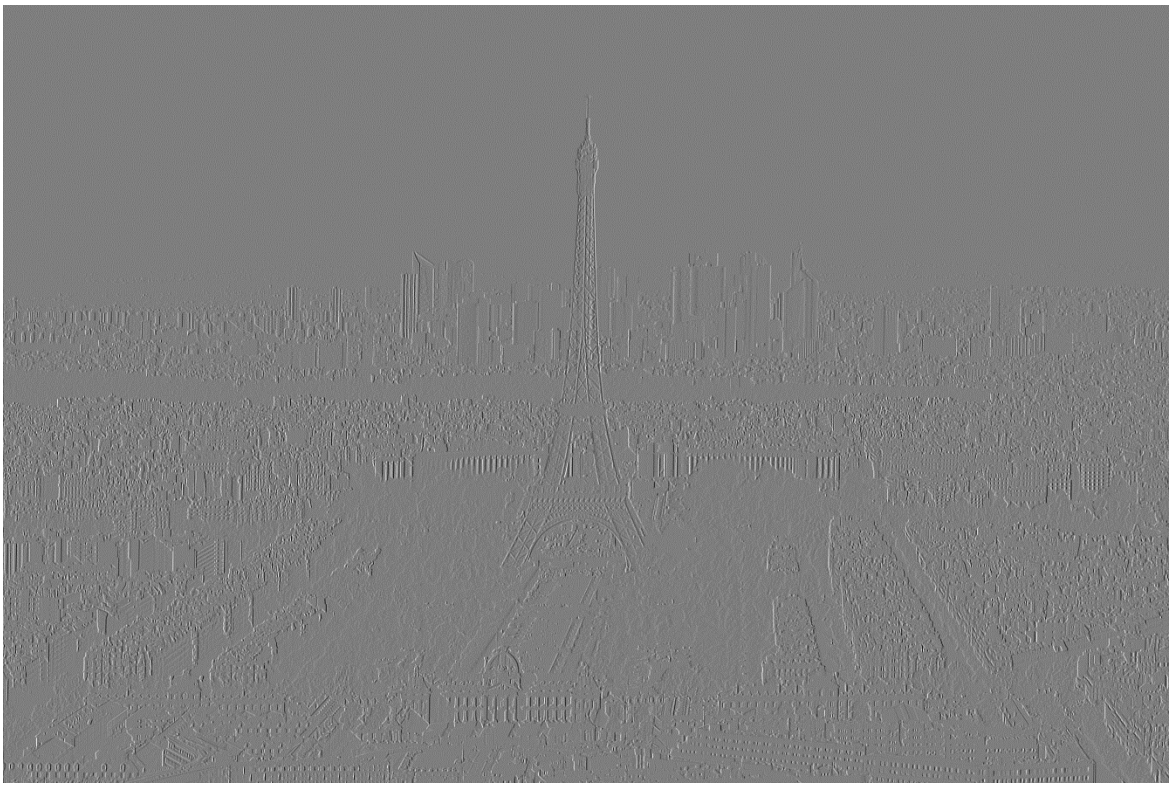

Image 1 : sortie calcul gradient direction x

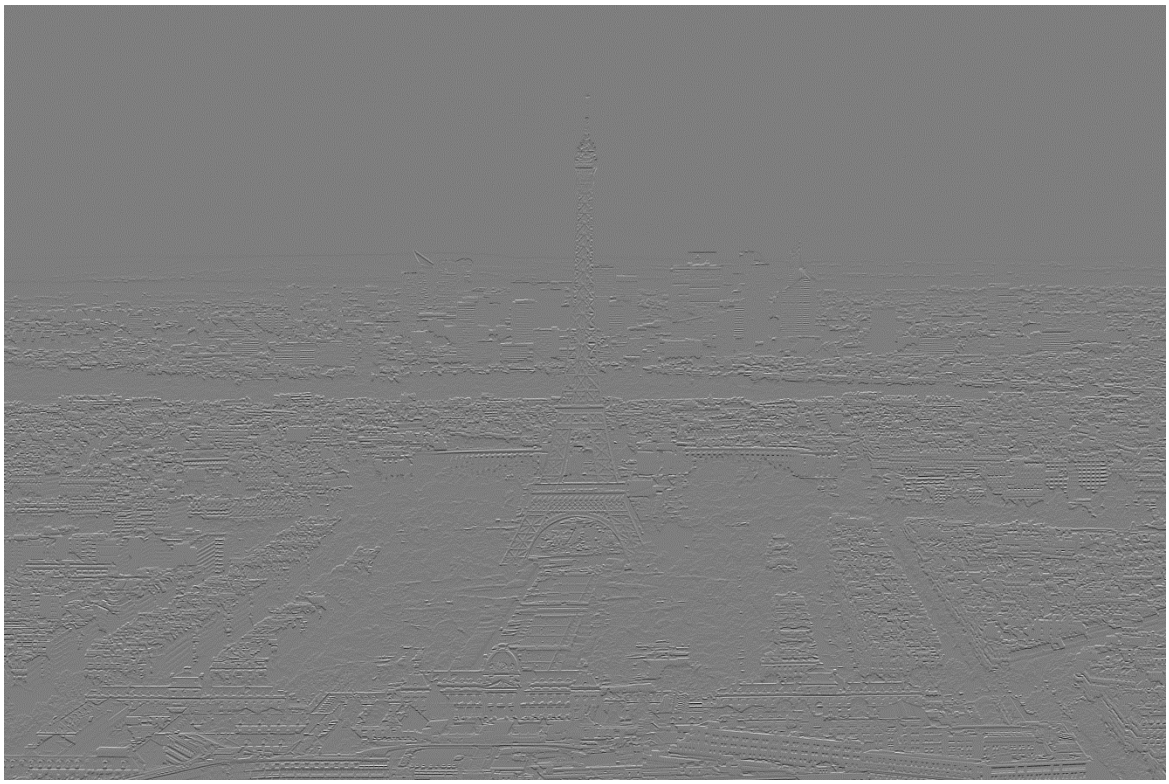

Image 2 : sortie calcul gradient direction y

Cet exercice génèrera une image de sortie composant les calculs de gradient dans les deux directions. Comme pour les exercices précédents :

Q11 (0.5pt) : Mesurez à nouveau la latence de ce programme sans parallélisation.

Q12 (6pts) : Faites la modification de ce code afin de le paralléliser avec OpenMP. Concentrez-vous sur une approche par parallélisme de données.

Q13 (1pt) : Quelle est la latence une fois la parallélisation réalisée ? Faites varier le nombre de threads de 1 à 10.

Q14 (0.5pt) : Tracez la courbe du speedup en fonction du nombre de threads : $S(n) = \frac{real(1)}{real(n)}$

Q15 (1pt) : Tracez la courbe du rapport suivant : $\frac{S(n)}{n}$ *en fonction du nombre threads.*

5 ANNEXES

5.1 ln2_approx_thread.c

```c
#include <pthread.h>
#include <stdio.h>

#define N_MAX 1000000000LL
#define THREAD_MAX 10

void *thread_for (void *sum);

int main(void)
{
    pthread_t ids [THREAD_MAX];
    double sums [THREAD_MAX];
    int i;
    double sum = 0.0;
    for (i = 0; i < THREAD_MAX; i++)
    {
        sums [i] = (double) i;
        pthread_create (&ids[i], NULL, thread_for, (void *)&sums[i]);
    }
    for (i = THREAD_MAX - 1; i >= 0; i--)
    {
        pthread_join (ids[i], NULL);
        sum += sums [i];
    }
    printf ("sum: %.12f\n" , sum);
    return 0;
}

void *thread_for (void *sum)
{
    long long n, n_min, n_max;
    n_min = N_MAX / THREAD_MAX * *(double *)sum;
    n_max = N_MAX / THREAD_MAX * (*(double *)sum + 1) ;
    *(double *)sum = 0.0 ;
    for (n = n_max ; n > n_min ; n--)
    {
        if (n % 2 == 0)
        {
            *(double*)sum -= 1.0 / (double )n;
        }
        else
        {
            *(double *)sum += 1.0 / (double )n;
        }
    }
    return (void *)0;
}
```

AnnexeA : ln2_approx_thread.c

5.2 ln2_approx_special_thread.c

```c
#include <pthread.h>
#include <stdio.h>

#define N_MAX 1000000000LL
#define THREAD_MAX 10

void *thread_for (void *sum);
```

```
int main(void)
{
    pthread_t ids [THREAD_MAX];
    double sums [THREAD_MAX * 64];
    int i;
    double sum = 0.0;
    for (i = 0; i < THREAD_MAX; i++)
    {
        sums [i * 64] = (double) i;
        pthread_create (&ids[i], NULL, thread_for, (void *)&sums[i *
64]);
    }
    for (i = THREAD_MAX - 1; i >= 0; i--)
    {
        pthread_join (ids[i], NULL);
        sum += sums [i * 64];
    }
    printf ("sum: %.12f\n" , sum);
    return 0;
}

void *thread_for (void *sum)
{
    long long n, n_min, n_max;
    n_min = N_MAX / THREAD_MAX * *(double *)sum;
    n_max = N_MAX / THREAD_MAX * (*(double *)sum + 1) ;
    *(double *)sum = 0.0 ;
    for (n = n_max ; n > n_min ; n--)
    {
        if (n % 2 == 0)
        {
            *(double*)sum -= 1.0 / (double )n;
        }
        else
        {
            *(double *)sum += 1.0 / (double )n;
        }
    }
    return (void *)0;
}
```

AnnexeB : ln2_approx_special_thread.c

5.3 ln2_approx.c

```
#include <stdio.h>

#define N_MAX 100000000LL

int main (void)
{
    double sum = 0.0;
    long long n;
    for (n = N_MAX; n > 0; n--)
    {
        if (n % 2 == 0)
        {
            sum -= 1.0 / (double)n;
        }
        else
        {
            sum += 1.0 / (double)n ;
```

```
        }
    }
    printf ("sum: %.12f\n",sum);
    return 0;
}
```

AnnexeC : ln2_approx.c