Inge5SE – Programmation parallèle 01/10/2020 – TP1 : PThreads

Alexandre Berne, Etienne Hamelin

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

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.

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

Organisation mémoire

Q2 (2pt) : Dans le code memory_organization.c, identifier pour chaque variable ci-dessous, à quel segment mémoire elles appartiennent.

	Static (code+data)	Stack	Heap
x			
y			
Z			
str[0…99]			
cpy_str[0…99]			
loop			

Introduction aux performances

1.1 Performances séquentielles

La commande `time` vous permet de réaliser des mesures de latence d'un programme.

- Exécutez la commande : `/usr/bin/time -p sleep 1`.

Q3(1pt) : Que signifient les temps `real`, `user` et `sys` ?

- Compilez le programme `ln2_approx.c` (Annexe B), et analysez le temps d'exécution avec : `/usr/bin/time -p ./ln2_approx`. Faites plusieurs mesures.

Q4(0.5pt) : Donnez les valeur de real, user, sys, $\frac{user+sys}{real}$.

Q5(1pt) : Que représente le ratio $\frac{user+sys}{real}$ *?*

- Compilez le programme `julia_fractal.c` (Annexe C) ; exécutez-le avec les données d'entrée : `./julia_fractal < julia_parameters.txt` et analysez les valeurs retournées par la commande time. Le programme calcule des images fractales ; vous pouvez les visualiser avec par exemple la commande `display {image}.pgm &`[4]

Q6(0.5pt) : Donnez les valeur de real, user, sys, $\frac{user+sys}{real}$.

*Q7(1pt) : Pourquoi observe-t-on une différence de la valeur **sys** entre les deux programmes ?*

1.2 Performances du programme parallèle

Le code `ln2_approx.c` présent en Annexe B est parallélisable. A l'aide de la bibliothèque pthread que nous avons vue durant le cours :

Q8(4pts) : Modifiez ce programme afin de paralléliser son exécution, et qu'il puisse prendre le nombre de thread en paramètre (argc, argv).

Vérifiez que le résultat est le même que le code séquentiel !

- Comme durant l'exercice précédent, mesurer les temps d'exécution de vos programmes parallélisés avec la commande `time -p`. Le nombre de thread variera entre 1 et 10.

[4] Au besoin, installez imagemagick `(sudo apt-get install imagemagick)` : ce paquet contient des outils pour afficher, convertir, transformer des images entre plein de formats dont pdf, jpeg, etc.

*Q9(1pt) : Tracez la courbe des **real** obtenus en fonction du nombre de threads.* $real = f(n_{th})$

Q10(1pt) : Tracez le ratio $\frac{real(1)}{real(n)}$ *en fonction du nombre de threads. Quelle est sa valeur max ?*

Comme le code `ln2_approx.c`, le code `julia_fractal.c` (Annexe C) est également parallélisable !

Q11(5pts) : Modifiez ce programme afin de paralléliser son exécution, et qu'il puisse prendre le nombre de thread en paramètre (argc, argv)

Comme durant l'exercice précédent, mesurez les temps d'exécution de vos programmes parallélisés avec la commande **time**. Le nombre de thread variera entre 1 et 10 afin de voir les limites de vos machines.

*Q12(1pt) : Tracez la courbe des **real** obtenus en fonction du nombre de threads.*

Q13(1pt) : Tracez le ratio $\frac{real(1)}{real(n)}$ *en fonction du nombre de threads. Quelle est sa valeur max ?*

Q14(2pt) : Au vu des courbes tracées pour les deux exemples (questions Q10 et Q12), que représente le ratio $\frac{real(1)}{real(n)}$ *? Comparez (commentez) la valeur max de ce ratio dans les deux cas.*

Synchronisation de threads

Le code `thread_synch.c` (Annexe D) contient un code parallélisé dans lequel chaque thread affiche son numéro de création.

Exécutez (plusieurs fois) ce programme.

Q15 (1pt) Comment expliquez-vous l'ordre de l'affichage ? Modifiez ce code, pour que l'affichage dans chaque thread se passe dans l'ordre. Expliquez votre choix d'implémentation.

ANNEXES

1.3 memory_organization.c

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

int x;

int function1 (void)
{
    static int z;
    z = x;
    return (z);
}

int main(void)
{
    int y;
    int loop = 0;
    char *str;
    char cpy_str[] = "Ce cours est genial! Maintenant je sais reconnaitre
les zones memoires ou sont situees mes variables";

    y = 4;
    x = y;

    str = malloc(100*sizeof(char));
    for (loop = 0; loop < 100; ++loop)
        str[loop] = cpy_str[loop];
    printf("str : %s\n", str);
    free(str);
    loop = function1 ();
    return 0;
}
```

AnnexeA : memory_organization.c

1.4 ln2_approx.c

```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;
}
```

AnnexeB : ln2_approx.c

1.5 Julia_fractal.c

```
#include <stdio.h>
#include <stdlib.h>

#define X_SIZE 4096
#define Y_SIZE 4096
#define IT_MAX 255

// Prototypes
void compute_set ( char *raster, double cx, double cy);
long compute_point (double zx, double zy, double cx, double cy);
int read_seed (double *cx, double *cy);
void write_pgm( char *raster, char *name);

int main (void)
{
    char *raster;
    char name [256];
    double cx, cy;
    raster = (char *) malloc (Y_SIZE * X_SIZE);
    while (read_seed(&cx, &cy))
    {
        compute_set (raster, cx, cy);
        sprintf (name, "julia_%f_%f.pgm", cx, cy);
        write_pgm(raster, name);
    }
    free(raster);
    return 0;
}

void compute_set(char *raster, double cx, double cy)
{
    long x, y;
    double zx, zy;
    for (y = 0; y < Y_SIZE; y++)
    {
        zy = 4.0 * (double) y / (double) (Y_SIZE - 1) - 2.0;
        for (x = 0; x < X_SIZE; x++)
        {
            zx = 4.0 * (double) x / (double) (X_SIZE - 1) - 2.0;
            raster[y * X_SIZE + x ] = compute_point ( zx, zy, cx, cy);
        }
    }
}

long compute_point (double zx, double zy, double cx, double cy)
{
    double zx_temp, zy_temp;
    long it = 0;
    while ((it < IT_MAX) && ((zx * zx) + (zy * zy) < 4.0))
    {
        zx_temp = zx * zx - zy * zy + cx;
        zy_temp = 2 * zx * zy + cy;
        zx = zx_temp;
        zy = zy_temp;
        it++;
    }
    return it;
}

int read_seed (double *cx, double *cy)
```

```
{
    if (scanf ("%lf %lf\n", cx, cy) == EOF)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

void write_pgm( char *raster, char *name)
{
    FILE *fp;
    fp = fopen (name, "wb");
    fprintf (fp , "P5 %d %d %d\n", X_SIZE, Y_SIZE, IT_MAX);
    fwrite (raster , 1 , X_SIZE * Y_SIZE, fp);
    fclose (fp);
}
```

AnnexeC : julia_fractal.c

1.6 Julia_parameters.txt

```
0.28500 0.00000
0.28500 0.01000
0.29500 0.55000
0.30028 0.48857
0.32450 0.04855
0.45000 0.14280
0.49610 0.54320
-0.40000 0.60000
-0.52000 0.57000
-0.62400 0.43500
-0.70176 -0.38420
-0.80000 0.15600
-0.81000 -0.17950
-0.82400 -0.17110
-0.83500 -0.23210
-1.03700 0.17000
```

Params_AnnexeC : julia_parameters.txt

1.7 Threads_synch.c

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

/*thread function definition*/
void *threadFunction(void* args)
{
    printf("Thread %ld\n", (long) args);

    return NULL;
}

int main(void)
{
    // Creating thread ids
    pthread_t thread_id[20];
    int ret = 0;
```

```
    long loop = 0;

    for (loop = 0; loop < 10; ++loop)
    {
        // Creating threads
        ret = pthread_create(&thread_id[loop],NULL, &threadFunction,
(void *)loop);
        if (ret)
        {
            printf("Error during thread creation !\n");
            return (-1);
        }
    }

    // Waiting end of every threads
    for (loop = 0; loop < 10; ++loop)
        pthread_join (thread_id[loop], NULL);

    return 0;
}
```

AnnexeD : threads_synch.c