Inge5SE – Programmation parallèle 15/10/2020 – TP3 : Synchronisation

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

Synchronisation des threads

Durant cette séance, vous allez être confrontés à des programmes qui semblent fonctionner cependant ce n'est pas le cas. Sur chacun des fichiers, vous aurez besoin de trouver le ou les problèmes, les résoudre avec les pthread puis avec openmp.

Comme vous nous donnerez 2 fichiers sources, faites-en sorte que `pthread` et `omp` soient présentes dans les noms des fichiers afin que l'on puisse les identifier lors de la correction.

Nous avons beaucoup travaillé avec les images durant les TP précédents maintenant nous allons passer sur des données plus dans un style dataflow. Ici nous allons générer un ou plusieurs fichiers en sortie de nos programmes.

Afin d'avoir un effet plus visuel, je vous propose d'installer gnuplot ou alors vous pouvez toujours utiliser Excel ;) Sinon voici la commande pour installer gnuplot sous Ubuntu : `sudo apt-get install gnuplot`. Pour tracer les courbes, vous pouvez taper la commande suivante : `gnuplot --persist -e 'plot [t=0:100] "data.txt" with lines'`. Vous pouvez également préciser la fenêtre en x que vous souhaitez en mettant la fourchette de valeur de t. Si vous ne mettez pas cette option, toute la courbe sera tracée.

1.1 Synchronisation simple

Nous allons dans un premier temps travailler avec un simple exercice. Il y a un producteur de données et un seul consommateur. Le programme sur lequel nous allons travailler dans un premier temps est `sine_sync.c`.

Q2 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

Q3 (0,5pt) : Quel est le problème avec ce programme ?

Q4 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

Q5 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.

Q6 (0,5pt) : Mesurer les latences de vos programmes modifiés.

1.2 Synchronisation multi consommateur

Le schéma utilisé précédemment est assez rare. En effet, dans la plupart des cas dataflow, il y a plusieurs consommateurs des données générées. C'est ce que nous allons voir dans le programme `sine_multi_sync.c`.

Q7 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

Q8 (0,5pt) : Quel est le problème avec ce programme ?

Q9 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

Q10 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.

Q11 (0,5pt) : Mesurer les latences de vos programmes modifiés.

1.3 Synchronisation multi producteur

Il est assez rare qu'il y ait plusieurs producteurs d'une même donnée et d'ailleurs dans le prochain programme vous verrez que les données produites sont différentes en fonction des threads et l'objectif est d'avoir un seul consommateur par producteur. Vous aurez donc à synchroniser les threads pour le programme `sine_full_sync.c`.

Q12 (0,5pt) : Compilez le programme, mesurer sa latence et observez le fichier de sortie.

Q13 (0,5pt) : Quel est le problème avec ce programme ?

Q14 (2pt) : Modifiez ce programme afin de résoudre le problème avec les pthreads (mutex ou sémaphore).

Q15 (3pt) : Modifiez ce programme afin de résoudre le problème avec OpenMP.

Q16 (0,5pt) : Mesurer les latences de vos programmes modifiés.

Q17 (1pt) : Maintenant que vous avez mesuré de nombreuses fois les latences de tous ces programmes, avez-vous une remarque particulière ? Comment expliquer ce phénomène ?

Etranges phénomènes

Dans ce dernier exercice, nous vous proposons de faire l'analyse du code ensemble dans ce sujet. Et vous allez devoir résoudre le problème en ajouter des memory barriers. Pour ajouter une barrière mémoire, il faut ajouter dans le code C la fonction : `asm volatile("mfence" ::: "memory");`. Cette fonction permet d'activer la barrière du compilateur ainsi que celle du CPU afin de ne pas réorganiser l'instruction du code et faire des optimisations.

Dans un premier temps, les explications du code `consume_sync.c`:

Nous avons une première phase d'initialisation de 4 variables. Nous affectons la valeur 0 dans chacune d'elles :

Initialisation

`a[0] = 0;`
`b[0] = 0;`
`x[0] = 0;`
`y[0] = 0;`

Puis nous avons deux sections de code OpenMP qui vont s'exécuter en parallèle avec les codes suivants :

Thread 1	Thread 2
`a[0] = 1;` `x[0] = b[0];`	`b[0] = 1;` `y[0] = a[0];`

Et pour finir, nous récupérons les valeurs du couple x et y. En sachant qu'il n'y a théoriquement que 4 valeurs possibles : 00, 01, 10 et 11.

Récupération des valeurs
`++results[x[0] + 2*y[0]];`

Considérons que l'ordre des affectations n'est pas interchangeable. C'est-à-dire que les valeurs de x et y ne peuvent pas être attribuées avant a et b pour chacun des threads. Cela signifie qu'en théorie, le couple XY ne peut jamais prendre la valeur 00.

Q18 (0,5pt) : Compilez et exécutez le programme. Quelle remarque faites-vous ?

Q19 (1pt) : Grace aux memory barriers, vous devez corriger ce programme afin que le cas 00 ne se produise jamais.

ANNEXES

1.4 Sine_sync.c

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

#define N_MAX 10000000LL
#define PI 3.14159265

int sine_value = 0;

void *sine_producer (void *thread_arg)
{
    int phase = 0;
    int amplitude = 10000;
    float x = 0;

    for (phase = 0; phase < N_MAX; ++phase)
    {
        x = 40 * 0.001 * phase;
        sine_value = (int)(amplitude * sin(x));
    }

    return NULL;
}

void *sine_writer (void *thread_arg)
{
    int nb_write = 0;
    FILE *file = NULL;
    file = fopen ("sine.txt" , "w");
    if (file == NULL)
    {
        printf("ERROR while opening file\n");
    }

    for (nb_write = 0; nb_write < N_MAX; ++nb_write)
    {
        fprintf(file, "%d\t%d\n", nb_write, sine_value);
    }

    fclose(file);

    return NULL;
}

int main (int argc, char **argv)
{
    int n_threads;
    int rc;
    pthread_t *my_threads;
    void *thread_return;
```

```
    n_threads = 2;
    my_threads = calloc(n_threads, sizeof(pthread_t));

    pthread_create(&my_threads[0], NULL, sine_writer, NULL);
    pthread_create(&my_threads[1], NULL, sine_producer, NULL);

    for (int i = 0; i < n_threads; i++)
    {
        rc = pthread_join(my_threads[i], &thread_return);
        if (rc < 0)
            printf("pthread_join ERROR !!! (%d)\n", rc);
    }

    free (my_threads);

    return (0);
}
```

AnnexeA : sine_sync.c

1.5 Sine_multi_sync.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <math.h>

#define N_MAX 10000000LL
#define PI 3.14159265

typedef struct {
  int thread_id;
} thread_args_t;

int sine_value = 0;

void *sine_producer (void *thread_arg)
{
    int phase = 0;
    int amplitude = 10000;
    float x = 0;

    for (phase = 0; phase < N_MAX; ++phase)
    {
        x = 40 * 0.001 * phase;
        sine_value = (int)(amplitude * sin(x));
    }

    return NULL;
}

void *sine_writers (void *thread_arg)
{
    thread_args_t *my_args = (thread_args_t *)(thread_arg);
    int nb_write = 0;
    FILE *file = NULL;
    char filename[20] = {'\0'};

    sprintf(filename, "sine_%d.txt", my_args->thread_id);
    file = fopen (filename , "w");
    if (file == NULL)
    {
```

```
        printf("ERROR while opening file\n");
    }

    for (nb_write = 0; nb_write < N_MAX; ++nb_write)
    {
        fprintf(file, "%d\t%d\n", nb_write, sine_value);
    }

    fclose(file);

    return NULL;
}

int main (int argc, char **argv)
{
    int n_threads;
    int rc;
    pthread_t *my_threads;
    thread_args_t *my_args;
    void *thread_return;

    n_threads = 4;

    my_threads = calloc(n_threads, sizeof(pthread_t));
    my_args = calloc(n_threads, sizeof(thread_args_t));

    for (int i = 0; i < n_threads; i++)
    {
        my_args[i].thread_id = i;
        if (i == (n_threads - 1))
            pthread_create(&my_threads[i], NULL, sine_producer, (void
*)&my_args[i]);
        else
            pthread_create(&my_threads[0], NULL, sine_writers, (void
*)&my_args[i]);
    }

    for (int i = 0; i < n_threads; i++)
    {
        rc = pthread_join(my_threads[i], &thread_return);
        if (rc < 0)
            printf("pthread_join ERROR !!! (%d)\n", rc);
    }

    free (my_threads);

    return (0);
}
```

AnnexeB : sine_multi_sync.c

1.6 Sine_full_sync.c

```
 #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <math.h>

#define N_MAX 10000000LL
#define PI 3.14159265

typedef struct {
```

```
  int thread_id;
} thread_args_t;

int sine_value[3];

void *sine_producers (void *thread_arg)
{
    thread_args_t *my_args = (thread_args_t *)(thread_arg);
    int phase = 0;
    int amplitude = 10000;
    float x = 0;

    for (phase = 0; phase < N_MAX; ++phase)
    {
        x = 40 * 0.001 * phase + my_args->thread_id;
        sine_value[my_args->thread_id%3] = (int)(amplitude * sin(x));
    }

    return NULL;
}

void *sine_writers (void *thread_arg)
{
    thread_args_t *my_args = (thread_args_t *)(thread_arg);
    int nb_write = 0;
    FILE *file = NULL;
    char filename[20] = {'\0'};
    sprintf(filename, "sine_%d.txt", my_args->thread_id);
    file = fopen (filename , "w");
    if (file == NULL)
    {
        printf("ERROR while opening file\n");
    }

    for (nb_write = 0; nb_write < N_MAX; ++nb_write)
    {
        fprintf(file, "%d\t%d\n", nb_write, sine_value[my_args-
>thread_id%3]);
    }

    fclose(file);

    return NULL;
}

int main (int argc, char **argv)
{
    int n_threads;
    int rc;
    pthread_t *my_threads;
    thread_args_t *my_args;
    void *thread_return;

    n_threads = 6;

    my_threads = calloc(n_threads, sizeof(pthread_t));
    my_args = calloc(n_threads, sizeof(thread_args_t));

    for (int i = 0; i < n_threads; i++)
    {
        my_args[i].thread_id = i;
```

```
        if (i%2 == 0)
            pthread_create(&my_threads[i], NULL, sine_producers, (void
*)&my_args[i]);
        else
            pthread_create(&my_threads[0], NULL, sine_writers, (void
*)&my_args[i]);
    }

    for (int i = 0; i < n_threads; i++)
    {
        rc = pthread_join(my_threads[i], &thread_return);
        if (rc < 0)
            printf("pthread_join ERROR !!! (%d)\n", rc);
    }

    free (my_threads);

    return (0);
}
```

AnnexeC : sine_full_sync.c

1.7 consume_sync.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // usleep

#ifndef N
    #define N 10000000
#endif

#define PADDING 64

int main(int argc, char **argv)
{
    volatile int a[PADDING];
    volatile int b[PADDING];
    volatile int x[PADDING];
    volatile int y[PADDING];

    int results[4] = {0, 0, 0, 0};


    for (int i = 0; i < N; i++)
    {
        a[0] = 0;
        b[0] = 0;
        x[0] = 0;
        y[0] = 0;

        #pragma omp parallel shared(a, b, x, y) num_threads(2)
        {
        /* Start parallel region */
            #pragma omp sections
            {
                #pragma omp section
                {
                    a[0] = 1;
                    x[0] = b[0];
                }
                #pragma omp section
```

```
                {
                    b[0] = 1;
                    y[0] = a[0];
                }
            }

      } /* end of parallel section */

      ++results[x[0] + 2*y[0]];

    }

    for (int vx = 0; vx < 2; vx++) {
        for (int vy = 0; vy < 2; vy++) {
            printf("(x = %d, y = %d) : %9d times (%.3f%%)\n", vx, vy,
results[vx + 2*vy], (100.0d*results[vx + 2*vy])/N);
        }
    }

}
```

AnnexeD : consume_sync.c