Shared Memory Programming with Pthreads - TPC2 -

Carlos Sá - A59905
carlos.sa01@gmail.com

March 15, 2016

Abstract

Este relatório é o resultado de um estudo sobre programação em memória partilhada com pthreads (POSIX threads). Neste estudo serão exploradas três técnicas bastante conhecidas de exclusão mútua: Mutex, Semáforos e Busy-waiting.

O algoritmo que usa cada uma destas técnicas como objecto de estudo é o algoritmo de cálculo da regra trapezoidal. Para cada uma das técnicas, serão feitas diversas execuções numa máquina multi-core para um número variável de fios de execução. Irei explorar o que acontece a nível de tempos de execução e ao nível da sincronização de threads por forma a perceber em que medida isso se reflete no resultado final do cálculo da aproximação.

Este trabalho está divido essencialmente em duas partes fundamentais. Uma primeira parte que corresponde a um primeiro exercicio em que apenas irei estudar o tempo médio de criação e terminação de um (e de vários) fios de execução. Na segunda parte irei estudar as três técnicas de exclusão mútua (utilizando como caso de estudo o cálculo da aproximação da regra trapezoidal) e discutir as vantagens e desvantagens de ambas as abordagens.

1. Introdução

Um dos modelos de programação explorados em computação paralela é o modelo de memória partilhada. Num modelo de programação em memória partilhada é possível construir programas que possuam num mesmo processo, várias threads em execução.

Existe uma API para threads que executam num mesmo processo comumamente designadas como **pthreads** - *POSIX®* Threads. Esta API é muito conhecida em UNIX, Linux e Solaris. Na utilização de um ambiente paralelo com pthreads, existem alguns problemas associados a sincronização de threads e no acesso a variáveis partilhadas. Quando uma variável é partilhada por várias threads, estas podem simultaneamente querer actualizar o valor desta, produzindo resultados incorretos. A este fenómeno designa-se por **data races**. É por isso necessário utilizar técnicas que assegurem um acesso ordeiro das threads a este tipo de variáveis garantindo que não existirá incoerências no resultado final.

Este conjunto de técnicas asseguram **exclusão mútua** das threads no acesso a essas variáveis. Neste estudo irei abordar três técnicas de exclusão mutua: *Mutex*, *Semáforos* e *Busy-Waiting* num programa que calcula a aproximação de um integral num intervalo [a,b] utilizando a **regra trapezoidal**[1] com n trapezoides.

Este trabalho está dividido em 2 exercicios essenciais.

Um primeiro exercicio mais simples onde o objectivo é encontrar o tempo médio necessário para o sistema criar e terminar um fio de execução.

No segundo exercício irei fazer o estudo das diferentes técnicas de exclusão mútua utilizando a regra trapezoidal descrita anteriormente como objecto de estudo.

2. Metodologia

2.1. Medição de tempos

Todos os tempos medidos neste estudo estão em **ms** (milisegundos). Uma vez que as funções *GET_TIME* realizam as medições em segundos, tive que converter esses tempos. O resultado em segundos é convertido para ms no printf. O código completo pode ser consultado no anexo em C.

Como iremos ver ao longo deste estudo, foram feitas várias execuções para um número variável de threads. Para cada número de threads utilizadas em cada execução foram retiradas várias amostras de tempos (5 amostras). Para chegar ao valor final da execução para um determinado número de threads, e é selecionado o melhor tempo obtido.

Como era suposto, a medição de tempos foi feita com recurso à macro **timer.h** cujo código pode ser encontrado em anexo em B que é baseado no **/sys/time.h** e utiliza ***gettimeofday(t, NULL)*** para as medições.

[1] `https://en.wikipedia.org/wiki/Trapezoidal_rule`

2.2. Contrução do programa em C com Pthreads

A nível de construção do programa, foi criado um **único programa (um único ficheiro .c)**. Como o algoritmo é o mesmo e apenas se pretende mudar a técnica de exclusão mútua, utilizo um único ficheiro .c e com **ifdefs** restrinjo a parte do código que quero utilizar consoante a técnica de exclusão mútua. Para melhor entender a forma como foi feito o programa recomendo a consulta do código completo no anexo C.

Construí uma makefile que apartir do mesmo código fonte .c passo na makefile a referência a cada uma das macros e são gerados 3 executáveis distintos cada um deles com uma versão do programa com uma técnica de exclusão mútua: trap_mutex, trap_semaphore e trap_busywaiting. O conteúdo da makefile pode ser consultado em anexo em D.

2.3. Valores de teste

Por forma a poder obter resultados comparativos, todos os resultados deste estudo foram feito para o cálculo do integral em que o valor de a (limite inferior) é 2 e o limite superior b é 242. Desta forma conseguimos fazer a divisão em 12 sub-intervalos e até um número de threads igual a 6 (inclusivé) é possível fazer um correto balanceamento de carga do trabalho pelas threads. Apartir das 6 threads, já não existe um número de sub-intervalos múltiplo do número de threads. Este resultado será explorado na parte experimental deste estudo em mais detalhe na seccção 9.2.

2.4. Script criada para tratamento resultados

Todos os resultados são obtidos utilizando uma script que executa automaticamente todos os resultados, para os diferentes números de threads pretendidos para as diferentes versões do programa com diferentes técnicas de exclusão mutua. Todos os resultados são exportados para ficheiros CSV utilizados para tratamento estatistico dos resultados.

A script utilizada para execução de todos os programas e obtenção de todos os resultados pode ser encontrada em anexo em A.1.

3. Caracterização da Máquina de teste

A máquina de teste utilizada para os testes foi o meu computador pessoal MacBook Air modelo Early 2014. Abaixo segue a tabela descritiva das caracteristicas de hardware desta máquina:

Manufacturer	Intel® Corporation
Processor	Core™ i5-4260U
Microarchitecture	Haswell
Processor's Frequency	1.40 GHz
Intel® Turbo-Boost	2.7 GHz
#Cores	2
#Threads	4
Lithography	22nm
Peak FP Performance	44.8 GFlops/s
ISA extensions	AVX 2.0 and SSE4.2
CPU technologies	Hyper-Threading Turbo Boost 2.0
Cache L1 **Cache L2** **Cache L3**	2x 32KB + 2x 32KB 2x 256KB 3MB
Associativity (L1/L2/L3)	8-way / 8-way / 12-way
Memory access bandwidth	13.2 GB/s
RAM Memory	2x 4GB 1600Mhz DDR3
Memory Channels	2 (0 free)

Table 1: *Caracteristicas Macbook Air Early 2014*

4. Parte 1: Tempo de criação de fios de execução com PTH Hello

Para o primeiro exercicio, o objectivo era determinar o tempo médio necessário para o sistema criar e terminar um fio de execução. No final pretendia-se fazer um pequeno estudo da forma como o número de fios de execução criados afecta o tempo médio de execução. Como poderemos ver já de seguida, utilizei uma versão do **Hello World** com pthreads para realizar as medições.

4.1. Implementação Hello World com PThreads

Para este primeiro exercicio utilizei um programa **Hello World** com pthreads medindo os tempos desde que são criadas e terminadas todas as threads. Assim sendo, os tempos são medidos no **main** e dizem respeito ao intervalo de tempo (em ms) que vai desde que é feito o ***pthread_create***, até que é feito o join das threads com: ***pthread_join***.

```
(.....)

GET_TIME(start);

for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread],
                   NULL,
                   Hello,
                   (void*) thread);
```

```
    //printf("Hello from the main thread\n");

for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

GET_TIME(finish);
elapsed = finish - start;
(.....)
```

No código utilizei as funções **GET_TIME** e **END_TIME** para obter o intervalo de tempo gasto desde a criação até ao término das threads.

4.2. Análise de tempos pth_hello

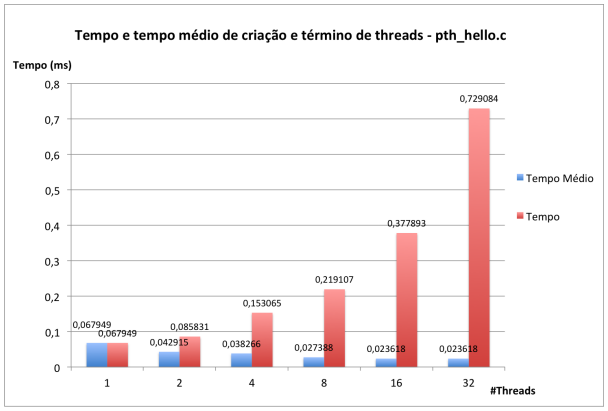

Figure 1: *Tempo e tempo médio de criação e término de threads*

Pela observação do gráfico 1 conseguimos verificar facilmente que o número de fios de execução criados afeta o tempo global que vai desde a criação com **pthread_create** e **pthread_join**.

Podemos verificar que quantas mais threads pretendemos criar, maior é o tempo demora a criar e a terminar todas as threads. Já em relação ao tempo **médio** que demora a iniciar e a terminar o diferente número de threads, este diminui quanto maior for o número de threads.

Podemos então concluír que existe um overhead inicial de criação das threads, mas esse efeito diminui quanto maior for o número de threads criadas.

5. Parte 2: Caso de estudo - Regra Trapezoidal

O presente caso de estudo para o segundo exercicio do trabalho é um programa em **C** que implementa a regra trapezoidal num intervalo $[a,b]$ utilizando n trapezoides. Se a e b forem valores reais e $f(x)$ uma função bem comportada é possível aproximar o integral:

$$\int_a^b f(x)dx$$

Dividindo o intervalo $[a,b]$ em n sub-intervalos e aproximar a àrea de cada um pela àrea de um trapézio. Se todos os intervalos forem iguais e se verificar:

$$h = \frac{b-a}{n} \wedge x_i = a + i \times h \tag{1}$$

em que $i = 0, 1, ..., n$.

Então, a aproximação será obtida por:

$$h\left[\frac{f(x_0)}{x_2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right] \tag{2}$$

5.1. Algoritmo Regra Trapezoidal em C

Nesta secção podemos ver a versão em C da regra trapezoidal que foi utilizada neste estudo e fornecida no enunciado.

Este algoritmo calcula uma aproximação ao integral utilizando várias threads em que $f(x)$ é a função de aproximação utilizada:

```
/* Algorithm for one aproximation interval */
double trapezoidal_rule(double a_thread, double b_thread, int n_thread, double h) {

    double approx;
    double x_i;
    int i;

    approx = (f(a_thread)+f(b_thread))/2.0;
    for (i = 1; i <= n_thread-1; i++) {
        x_i = a_thread + i*h;
        approx += f(x_i);
    }
    approx = approx*h;

return approx;
}

/*------ f(x) ------*/
double f(double x) {
    double return_val;

    return_val = x*x;
    return return_val;
}
```

Esta função, é uma função chamada dentro da função **job_thread**:

```
void *job_thread(void* rank)
```

Esta função é a função que representa o trabalho que as threads terão que realizar para calcular o resultado final. Por essa razão, esta função têm então que ser passada na função **pthread_create** quando se realiza a criação da team de threads :

```
pthread_create(&thread_handles[thread_id], NULL, job_thread, (void*) thread_id);
```

Como não utilizamos **pthread_attr_t***, apenas passamos o endereço NULL. Cada thread possuirá um id thread_id.

5.2. Função job_thread executada pelas várias POSIX threads

Vejamos agora em mais detalhe a função: **job_thread**:

```
void *job_thread(void* rank)
```

Cada thread que executa esta função, executa também a função **trapezoidal_rule** já descrita anteriormente. Esta função sendo chamada por cada uma das threads que executa a função **job_thread**, produz um resultado que é guardado numa variável my_integral:

```
void *job_thread(void* rank) {
    long my_rank = (long) rank;
    double a_thread;
    double b_thread;
    double my_integral;

    a_thread = a + my_rank*n_thread*h;
    b_thread = a_thread + n_thread*h;

    my_integral = trapezoidal_rule(a_thread, b_thread, n_thread, h);

//(.....UPDATING all_approx variable should be done now carefully!!....)
```

Após cada thread calcular o valor da aproximação, é agora necessário actualizar o valor de all_approx. Esta variável é a variável que utilizo no programa para guardar o valor do resultado final global da aproximação. Como será de esperar, é necessário que a atualização desta variável seja feita de uma forma cuidadosa apartir do momento que utilizamos mais que uma thread. Essa atualização caracteriza por isso uma secção critica. Todas as threads irão querer actualizar esta variável, e por isso é necessário garantir exclusão mútua no acesso e na actualização da variável **all_approx**. No presente trabalho irei estudar três técnicas de exclusão mútua: Mutex, semáforos e busy-wait. O estudo feito será feito com base na discussão de vantagens e desvantagens de cada uma das técnicas. O código de todo este trabalho, para uma consulta mais completa está disponível em anexo em: C.

6. Exclusão Mútua: Utilização de Mutex

Como foi explicado anteriormente na secção onde apresentei a metodologia adotada, foi criado um único ficheiro .c e com macros, utilizo as partes necessárias para cada técnica de exclusão mútua. A primeira técnica explorada foi utilizando um mutex de acordo com o código abaixo:

```
(.....)

#ifdef D_MUTEX
    /******* Initiate MUTEX ******/
    pthread_mutex_init(&mutex, NULL);
#endif

 /* Create Pthreads*/
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
        pthread_create(&thread_handles[thread_id], NULL, job_thread, (void*) thread_id);
    }

    /* Waiting threads to join */
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
        pthread_join(thread_handles[thread_id], NULL);
    }

    GET_TIME(end);
    elapsed = end - start;

    /* All threads finished their work. Main print the result */
    printf("%s,%d,%f,%f,%d,%f,%f\n",prog_name,thread_count,a,b,n,all_approx,(elapsed)*pow(10,3));

#ifdef D_MUTEX
    /**** Destroy the mutex ***/
    pthread_mutex_destroy(&mutex);
#endif

    free(thread_handles);

return 0;
```

```
}
```

Na função **job_thread** podemos consultar no código completo em C que a implementação do mutex é feita da seguinte forma:

```
#ifdef D_MUTEX
    /* Entering the critical region and LOCK the mutex */
    pthread_mutex_lock(&mutex);
    /* Update all_approx safely */
    all_approx += my_integral;
    pthread_mutex_unlock(&mutex);
    /* Exit critical region and UNLOCK de mutex for other thread can use */
#endif
```

O mutex é um tipo de lock. Este mutex é uma variável especial que restringe o acesso das threads à secção critica. Desta forma, pelo código acima podemos verificar que apenas uma thread de cada vez incrementa o valor de all_approx de forma segura.

Também foi necessário declarar a variável do mutex como global para que seja visivel por todas as threads:

```
#ifdef D_MUTEX
pthread_mutex_t mutex; /* The "Special" type for mutexes - Slide 35 */
#endif
```

O mutex, comparativamente com o busy-waiting que irei explorar na secção 8, tem uma grande vantagem de eliminar a espera activa causada por um controlo de acesso à região critica baseada em busy-waiting. Em mutex uma thread que tenha que esperar para aceder a secção critica, é "marcada" não sendo escalonada pelo CPU e por isso, não gastando ciclos de clock sem realizar trabalho útil. Contudo, os mutex tem o inconveniente do escalonamento ser deixado a cargo do sistema. Pelo meu código podemos ver que uma thread invoca a chamada de **pthread_mutex_unlock** e no final de executar o código da secção critica, a thread invoca **pthread_mutex_unlock** para a libertar. É feita uma correta inicialização do mutex e no final, o mutex é destruído.

7. Exclusão Mútua: Utilização de Semáforos

A segunda técnica de exclusão mútua utilizada foi a técnica de utilização de semáforos. Á semelhança dos mutex foi necessário declarar uma variável global que possa ser acessivel apartir de todas as threads:

```
#ifdef D_SEMAPHORE
sem_t sem;   /* Declaration of semaphore */
#endif
```

Abaixo podemos ver a correspondente versão com implementação de semáforos:

```
(.....)

#ifdef D_SEMAPHORE
    /******* Initiate SEMAPHORE ******/
    sem_init(&sem, 0, 1); /** 0 - semaphore shared by all threads, 1 - Semaphore initialized as unlocked **/
#endif

    /* Create Pthreads*/
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
        pthread_create(&thread_handles[thread_id], NULL, job_thread, (void*) thread_id);
    }

    /* Waiting threads to join */
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
        pthread_join(thread_handles[thread_id], NULL);
    }

 (.....)


#ifdef D_SEMAPHORE
    /*** Destroy the semaphore ***/
    sem_destroy(&sem);
#endif

(.....)
```

Os semáforos não são uma parte integrante do POSIX® Threads e como tal também necessitei de fazer *include* do .h **semaphore.h**. Os semáforos utilizados neste programa não é mais do que a utilização clássica de um semáforo. A codificação de acesso à seccção critica é feita no meu programa de acordo com:

```
(......)

    my_integral = trapezoidal_rule(a_thread, b_thread, n_thread, h);


```

```
#ifdef D_SEMAPHORE
 /* If sem is zero thread waits until is!= zero */
 /* If sem if not zero, thread enter critical region */

 /* Entering the critical region and LOCK the semaphore sem */
    sem_wait(&sem);
    /* Update all_approx safely */
    all_approx += my_integral;
    /* Exit the critical region and RELEASE the semaphore by incrementing sem */
    sem_post(&sem);
#endif

(......)
```

Se a variável sem for zero, significa que existe uma thread a executar código da seccção critica e como tal, uma thread que leia $sem = 0$, espera até que a variável seja diferente de zero. Se a variável possui um valor diferente de zero, então a thread executa o código da secção critica e quando sai, incrementa o semáforo.

A implementação do semáforo desta forma, estabelece uma *Barrier* que comparativamente com busy-wait faz com que os semáforos não consumam ciclos de CPU quando ficam bloqueados em *sem_wait(sem)*.

8. Exclusão Mútua: Utilização de Busy-Waiting

A última técnica de exclusão mútua explorada foi Busy-Waiting. Esta técnica revelou-se ser a pior técnica de todas as que foram feitas neste estudo uma vez que utiliza a noção de **espera ativa**. Isto significa que uma thread testa repetidamente uma condição até que o estado de uma variável de condição mude e se possa continuar com a execução para aceder ao código da região critica. Os ciclos de CPU gastos a consultar repetidamente o estado da variável faz com que se desperdicem grandes quantidades de ciclos de clock sem fazer trabalho útil para a execução do algoritmo. Tal penalização é absolutamente inaceitável a nível de performance e por essa razão esta técnica de exclusão mútua não deve ser utilizada. Contudo também fazia parte deste estudo explorar esta técnica. O código que se segue ilustra a minha implementação de busy-wait no programa deste estudo.

Tal como os métodos anteriores, é também necessário declarar uma variável global acessivel apartir de todas as threads:

```

(....)

#ifdef D_BUSYWAITING
long flag=0;
#endif

(....)
```

No **job_thread** é também utilizado o seguinte código para controlo de acesso à região critica:

```

(.....)
my_integral = trapezoidal_rule(a_thread,b_thread, n_thread, h);

#ifdef D_BUSYWAITING
    while(flag != my_rank)
     ;
    all_approx += my_integral;
    flag = (flag+1) \% thread_count;
#endif

(.....)
```

O ciclo while utilizado acima sem qualquer instrução é constantemente executado até que a variável **flag** (que foi declarada como global e acessivel por qualquer thread) mude de valor e o código que realiza o incremento em **all_approx** possa ser executado pela thread em espera.

9. Mutex VS Semáforos VS Busy-Wait

No final da codificação das três técnicas realizei alguma experimentação por forma a perceber os impactos de cada uma das técnicas de exclusão mútua ao nível de resultado e de tempo de CPU gasto.

9.1. Tabela de tempos e resultados da aproximação

- Resultados obtidos com Mutex

Mutex @ MacbookAir Early 2014		
Nº Threads	Time(ms)	Result approx
1	0,068903	4740160
2	0,081062	4740160
4	0,117064	4740160
6	0,178099	4740160
8	0,200987	1427840

Table 2: *Tempos e resultados da aproximação obtidos com Mutex em Macbook Air Early 2014*

- Resultados obtidos com Semáforos

Semaphores @ MacbookAir Early 2014		
Nº Threads	Time(ms)	Result approx
1	0,066042	4740160
2	0,085831	4740160
4	0,12207	4740160
6	0,164986	4740160
8	0,197172	1427840

Table 3: *Tempos e resultados da aproximação obtidos com Semáforos em Macbook Air Early 2014*

- Resultados obtidos com Busy-Waiting

Busy-Waiting @ MacbookAir Early 2014		
Nº Threads	Time(ms)	Result approx
1	0,06485	4740160
2	0,08893	4740160
4	0,106812	4740160
6	0,176191	4740160
8	0,197887	1427840

Table 4: *Tempos e resultados da aproximação obtidos com Busy-Waiting em Macbook Air Early 2014*

9.2. Interpretação dos resultados da aproximação e tempos

Para verificar que os valores das várias threads estavam corretos, realizei os diferentes testes para um número variável de threads[2]. Como a minha máquina pessoal possui apenas 2 cores fisicos (com Hyper-Threading, 4 no total) fiz as medições até um total de 8 threads. O resultado da aproximação calculada pelos 3 programas é o mesmo à excepção de quando corremos para 8 threads. Note-se no entanto que este resultado é propositado.

Ao longo da criação do programa o que reparei é que fixando o valor de sub-intervalos (n), sem considerar o número de fios de execução criados conduz a soluções em que o resultado deixa de estar correto. Isto acontece porque o balanceamento de carga pelas threads deixa de ser feito corretamente pois não é possivel distribuír um número de sub-intervalos uniformemente pelas threads criadas. O número de retângulos deixa de ser múltiplo do número de threads. Então, dado o algoritmo fornecido para estudo tem que ser assumido como pré-condição que o número de sub-intervalos tem que ser sempre múltiplo do número de threads utilizadas e por isso não pode ser fixado para um número variável de threads.

A nível de tempos de execução, podemos verificar que em todos os testes se obtêm tempos mais curtos quando o número de fios de execução criados é igual ao número de cores fisicos. Apartir do momento em que se utiliza o *Hyper-Threading*, a performance degrada-se.

Aínda dentro dos melhores tempos (obtidos para um número de threads igual ao número de cores fisicos) obtêm-se melhores tempos para 1 Thread. Neste algoritmo em especifico como apartir de 2 threads, os tempos aumentam, podemos concluir que o overhead de criação de mais que uma thread não compensa a nível de desempenho. Não existem ganhos na criação de mais que uma thread para calcular a aproximação para uma àrea de integração na ordem dos limites inferiores e superiores estabelecidos neste estudo (respectivamente $a = 2$, $b = 242$).

9.3. Análise Gráfica de Tempos e Ganhos: Mutex VS Semáforos VS Busy-Waiting

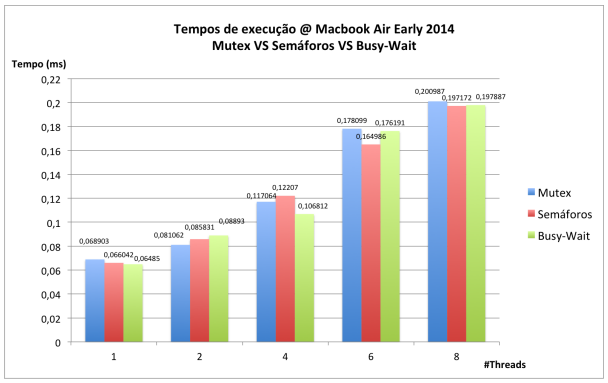

Figure 2: *Tempos obtidos para as 3 técnicas para número variável de threads*

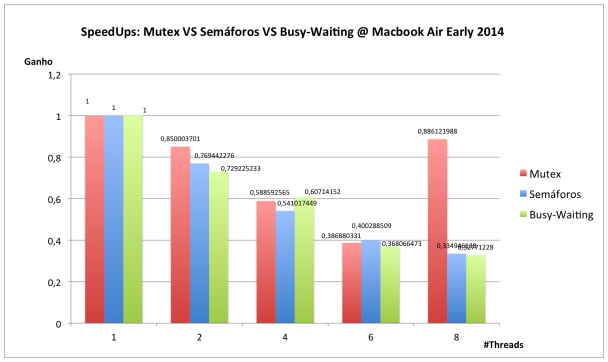

Figure 3: *Speedups obtidos para as 3 técnicas de exclusão mútua*

Após a criação do gráfico de *Speedups* e de tempos podemos concluir que para este caso de estudo e para o intervalo de integração estabelecido, as diferentes técnicas de exclusão mutua *per si* não trazem ganhos de desempenho. O aumento do número de

[2]O resultado final deve ser o mesmo por isso utilizei uma calculadora da regra trapezoidal para verificar que os resultados estavam corretos: `http://www.emathhelp.net/calculators/calculus-2/trapezoidal-rule-calculator/`

threads neste caso, provoca um overhead que penaliza o algoritmo a nível de desempenho. O facto do código possuir secções criticas onde é necessário assegurar exclusão mútua, a criação de um maior número de threads penaliza aínda mais o desempenho. Contudo, podemos inferir que os mutexes e os semáforos são melhores técnicas de exclusão mútua. O gráfico apresenta os SpeedUps até às 8 threads mas pelos motivos que foram já explicados em 9.2 o resultado para este número de threads não deve ser considerado.

10. Conclusão

Após a realização dos diferentes testes pude comprovar que os semáforos e os mutex são as duas melhores técnicas de garantir exclusão mútua em pthreads. A utilização de Busy-Waiting requer uma verificação sistemática do valor de uma flag o que resulta num elevado número de ciclos de clock sem que seja realizado trabalho útil. Graças a este trabalho consegui desenvolver conhecimentos de programação com pthreads em Unix e perceber um pouco melhor os cuidados necessários quando se programa em multi-threading com memória partilhada.

[1]

References

[1] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

A. Appendix

A.1. Script de testes utilizada

```
#!/bin/bash

max_threads=16
max_sample=5
numthreads=1

a=2
b=242
n=12

echo "Executing Ex1 for pth_hello..."

echo "NThreads,Average_Time_Per_Thread(ms),Time(ms)" >> CSV/pth_hello_output.csv

echo "Sequential pth_hello..."
for ((sample=1; sample <= $max_sample; sample++ ))
do
 ./pth_hello 1 >> CSV/pth_hello_output.csv
done

echo "Parallel pth_hello using threads...."
for (( numthreads=2; numthreads <= $max_threads; numthreads+=2 ))
do
 for ((sample=1; sample <= $max_sample; sample++ ))
 do
  ./pth_hello $numthreads >> CSV/pth_hello_output.csv
 done
done




echo "Executing Ex2..."

echo "MutualExclusion,NThreads,val_a,val_b,nRectangles/Sub-Intervals,Result,Time(ms)" >> CSV/trap_mutex_output.csv
echo "MutualExclusion,NThreads,val_a,val_b,nRectangles/Sub-Intervals,Result,Time(ms)" >> CSV/trap_semaphore_output.csv
echo "MutualExclusion,NThreads,val_a,val_b,nRectangles/Sub-Intervals,Result,Time(ms)" >> CSV/trap_busywaiting_output.csv

echo "Sequential MUTEX | SEMAPHORE | BUSY-WAITING..."
for (( sample=1; sample <= $max_sample; sample++ ))
do
 ./trap_mutex 1 $a $b $n >> CSV/trap_mutex_output.csv
 ./trap_semaphore 1 $a $b $n >> CSV/trap_semaphore_output.csv
 ./trap_busywaiting 1 $a $b $n >> CSV/trap_busywaiting_output.csv
done

echo "Parallel MUTEX | SEMAPHORE | BUSY-WAITING...."
for (( numthreads=2; numthreads <= $max_threads; numthreads+=2 ))
do
 for ((sample=1; sample <= $max_sample; sample++ ))
 do
  ./trap_mutex $numthreads $a $b $n >> CSV/trap_mutex_output.csv
  ./trap_semaphore $numthreads $a $b $n >> CSV/trap_semaphore_output.csv
```

```
  ./trap_busywaiting $numthreads $a $b $n >> CSV/trap_busywaiting_output.csv
 done
done
echo "Done! All CSVs generated!...."
```

B. Código fonte timer.h

```
/* File:      timer.h
 *
 * Purpose:   Define a macro that returns the number of seconds that
 *            have elapsed since some point in the past.  The timer
 *            should return times with microsecond accuracy.
 *
 * Note:      The argument passed to the GET_TIME macro should be
 *            a double, *not* a pointer to a double.
 *
 * Example:
 *    #include "timer.h"
 *    . . .
 *    double start, finish, elapsed;
 *    . . .
 *    GET_TIME(start);
 *    . . .
 *    Code to be timed
 *    . . .
 *    GET_TIME(finish);
 *    elapsed = finish - start;
 *    printf("The code to be timed took %e seconds\n", elapsed);
 *
 * IPP:  Section 3.6.1 (pp. 121 and ff.) and Section 6.1.2 (pp. 273 and ff.)
 */
#ifndef _TIMER_H_
#define _TIMER_H_

#include <sys/time.h>

/* The argument now should be a double (not a pointer to a double) */
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}

#endif
```

C. Código completo - trap.c

```c
/*
 * Purpose:  Compute a integral with the trapezoidal rule using POSIX Threads.
 *
 * Input:    One value for inferior limit -> a
 *          One value for the superior limit -> b
 *         The number of sub-intervals -> n
 *
 * Output:   Estimate the integral from a to b of f(x) using Pthreads for the trapezoidal rule and n sub-intervals
 *
 *
 */
#include <stdio.h>      /* For I/O needed: scanf and printf **/
#include <stdlib.h>
#include <pthread.h>
#include "timer.h" /* For measuring time. gettimeofday and sys/time.h based*/
#include <math.h>


#ifdef D_SEMAPHORE
#include <semaphore.h>  /* Semaphores need to be included because is not part of Pthreads */
#endif

const int MAX_THREADS = 64;

/* Global variable:  accessible to all threads */
/* Global variables can introduce subtle and confusing bugs! */
/* The global variables are shared among all the threads. */
int thread_count;
double a;
double b;
double h;
int n;
int n_thread;
double all_approx;


#ifdef D_SEMAPHORE
sem_t sem;   /* Declaration of semaphore */
#endif

#ifdef D_MUTEX
pthread_mutex_t mutex; /* The "Special" type for mutexes - Slide 35 */
#endif

#ifdef D_BUSYWAITING
long flag=0;
#endif


/*** Global Functions ***/
void Usage (char* prog_name);
void *job_thread(void* rank);         /* Thread function */
double trapezoidal_rule(double a_thread, double b_thread, int n_thread, double h); /* Calculate integral for thread */
double f(double x);

```

```
/*-------------------- MAIN ------------------------------------------*/
int main(int argc, char** argv) {
    long thread_id;
    pthread_t* thread_handles;
    all_approx = 0.0; /* For being used with semaphore */
    //int get_out = 0; /* Flag USER-FRIENDLY Interface I/O while */
    double start, end, elapsed;       /* For measuring time with Professor Antonio Pina timer.h */


    /* Get number of threads from command line */
    char* prog_name = argv[0];
    if (argc < 5) Usage(prog_name);
    thread_count = strtol(argv[1], NULL, 10);
    if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);

    /****************** I/O Script VERSION *****************************/

    /** Interval [a,b] **/
    a = strtol(argv[2], NULL, 10);
    b = strtol(argv[3], NULL, 10);

    /** Number of Sub-intervals/rectangles **/
    n = strtol(argv[4], NULL, 10);

    /*****************************************************************/



    /**************** USER-FRIENDLY VERSION ****************************
    while(!get_out && 1){
     puts("Insert the values of [a,b] interval");
     printf("Inferior limit (a): \n");
     scanf("%lf", &a);
     printf("Superior limit (b): \n");
     scanf("%lf", &b);
     if (a > b) printf("a should be smaller than b!");
     else get_out = 1;
    }
    get_out = 0;
    while(!get_out && 1){
     printf("Insert the number of Sub-intervals: \n");
 scanf("%d", &n);
 if (n <= 0) puts("Not a valid number of Sub-intervals\n");
 else get_out = 1;
    }
    *****************************************************************/
    h = (b-a)/n;

    /*** Pre-condition! We should divide the number of Sub-intervals EQUALLY to the number of threads we re executing the program! Otherwise, the result will not be correct ***/
    n_thread = n/thread_count;

    thread_handles = malloc (thread_count*sizeof(pthread_t));

    GET_TIME(start);

#ifdef D_MUTEX
    /******* Initiate MUTEX ******/
    pthread_mutex_init(&mutex, NULL);
```

```
#endif


#ifdef D_SEMAPHORE
    /******* Initiate SEMAPHORE ******/
    sem_init(&sem, 0, 1); /** 0 - semaphore shared by all threads, 1 - Semaphore initialized as unlocked **/
#endif

    /* Create Pthreads*/
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
         pthread_create(&thread_handles[thread_id], NULL, job_thread, (void*) thread_id);
    }

    /* Waiting threads to join */
    for (thread_id = 0; thread_id < thread_count; thread_id++) {
        pthread_join(thread_handles[thread_id], NULL);
    }

    GET_TIME(end);
    elapsed = end - start;

    /* All threads finished their work. Main print the result */
    printf("%s,%d,%f,%f,%d,%f,%f\n",prog_name,thread_count,a,b,n,all_approx,(elapsed)*pow(10,3));


    /*********  PRINT USER-FRIENDLY ******
    printf("From [%f,%f] the integral estimate with trapezoide-rule with %d sub-intervals is: %f\n", a, b, n, all_approx);
   ************************************/


#ifdef D_SEMAPHORE
    /*** Destroy the semaphore ***/
    sem_destroy(&sem);
#endif


#ifdef D_MUTEX
    /**** Destroy the mutex ***/
    pthread_mutex_destroy(&mutex);
#endif

    free(thread_handles);

return 0;
}

/*---------------- Each thread sould run this job ----------------------------------*/
void *job_thread(void* rank) {
    long my_rank = (long) rank;
    double a_thread;
    double b_thread;
    double my_integral;

    a_thread = a + my_rank*n_thread*h;
    b_thread = a_thread + n_thread*h;

    my_integral = trapezoidal_rule(a_thread, b_thread, n_thread, h);
```

```
#ifdef D_SEMAPHORE
     /* If sem is zero thread waits until is != zero */
     /* If sem if not zero, thread enter critical region */
    sem_wait(&sem);       /* Entering the critical region and LOCK the semaphore sem */
     all_approx += my_integral; /* Update all_approx safely */
    sem_post(&sem);       /* Exit the critical region and RELEASE the semaphore by incrementing sem */
#endif



#ifdef D_MUTEX
    pthread_mutex_lock(&mutex);  /* Entering the critical region and LOCK the mutex */
     all_approx += my_integral; /* Update all_approx safely */
    pthread_mutex_unlock(&mutex); /* Exit critical region and UNLOCK de mutex for other thread can use */
#endif



#ifdef D_BUSYWAITING
    while(flag != my_rank)
         ;
    all_approx += my_integral;
    flag = (flag+1)%thread_count;
#endif

return NULL;
}


/*-------------- Algorithm for one aproximation interval ----------------------*/
double trapezoidal_rule(double a_thread, double b_thread, int n_thread, double  h) {

    double approx;
    double x_i;
    int i;

    approx = (f(a_thread)+f(b_thread))/2.0;
    for (i = 1; i <= n_thread-1; i++) {
        x_i = a_thread + i*h;
        approx += f(x_i);
    }
    approx = approx*h;

return approx;
}
/*---------------- f(x) ----------------*/
double f(double x) {
    double return_val;

    return_val = x*x;
    return return_val;
}
/*--------------------------------------------------------------------*/
/* Usage */
void Usage(char* prog_name) {
 fprintf(stderr, "Make sure you re passing %s NTHREADS a b n as arguments", prog_name);
 fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
 fprintf(stderr, "0 < number of threads <= %d\n", MAX_THREADS);
exit(0);
}
```

D. Makefile utilizada

```
all: pth_hello trap_mutex trap_semaphore trap_busywaiting

pth_hello : pth_hello.c
 gcc-5 -Wall -g -o pth_hello pth_hello.c -lpthread

trap_mutex : trap.c
 gcc-5 -Wall -g -DD_MUTEX -o trap_mutex trap.c -lpthread

trap_semaphore : trap.c
 gcc-5 -Wall -g -DD_SEMAPHORE -o trap_semaphore trap.c -lpthread

trap_busywaiting : trap.c
 gcc-5 -Wall -g -DD_BUSYWAITING -o trap_busywaiting trap.c -lpthread

.PHONY: clean

clean:
 rm -f *.o
 rm -f *.out
 rm -rf *.dSYM
 rm pth_hello
 rm -f trap_*
```