Memória Partilhada: Programação com PThreads

Sérgio Caldas
Universidade do Minho
Escola de Engenharia
Departamento de Informática
Email: a57779@alunos.uminho.pt

***Resumo*—Este trabalho baseia-se essencialmente na programação em memória partilhada, recorrendo a *POSIX Threads*, mais conhecidas como *PThreads*, estas são muito populares no ambiente *UNIX*. O trabalho divide-se em duas partes, na primeira parte medi o tempo médio de criação de uma *Thread* e a segunda parte desenvolvi um algoritmo (*Trapezoidal Rule*) com recurso a *PThreads*.**

1. Introdução

Este trabalho, desenvolvido no âmbito da disciplina de Engenharia Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de MIEI[1], consiste no desenvolvimento de programas com recurso a *PThreads*. Este trabalho é dividido em duas partes, na primeira parte desenvolvi um pequeno programa (tipo *Hello World*) em que me media o tempo médio da criação de um fio de execução (*Thread*). Na segunda parte desenvolvi o algoritmo da regra trapezoidal, com recurso a *PThreads*, em que media o tempo de execução para um dado intervalo, no algoritmo existe uma variável que irá ser partilhada por todas as *Threads* e para evitar *Data Races* implementei as técnicas que estudamos para forçar a exclusão mutua, mais precisamente *Busy Waiting, Mutex* e Semáforos.

Neste relatorio encontra-se a análise dos resultados obtidos, tanto para a primeira parte do trabalho, como para a segunda.

2. Ambiente de testes

Depois de desenvolvidos os dois algoritmos (relativos a primeira e segunda parte) efetuei testes na minha máquina pessoal e no *Search*, mais precisamente numa máquina 431. A metodologia de testes por mim escolhida foi a de *k-best solution*, esta metodologia baseia-se essencialmente na execução do *Kernel* de *k* vezes escolhendo a melhor solução dessas *k* vezes. O *k* por mim escolhido foi de 5. De referir que foram feitos testes para 1, 2, 4, 8, 16, 32 e 64 *Threads*, sendo que para cada um destes números de *Threads* o *Kernel* foi executado 5 vezes (*k-best solution*).

1. Mestrado Integrado em Engenharia Informática

System	MacBook Pro (15-inch, Mid 2010)
# CPUs	1
CPU	Intel® Core™ I5-520M
Architecture	Arrandale
# Cores per CPU	2
# Threads per CPU	4
Clock Freq.	2.53 GHz
L1 Cache	64 KB 32 KB por core
L2 Cache	512 KB 256 KB por core
L3 Cache	3 MB
Inst. Set Ext.	SSE4.1/4.2
#Memory Channels	2
Memory BW	17.1 GB/s

Tabela 1. CARACTERIZAÇÃO DA MÁQUINA PESSOAL

System	Máquina 431
# CPUs	2
CPU	Intel® Xeon® X5650
Architecture	Nehalem
# Cores per CPU	6
# Threads per CPU	12
Clock Freq.	2.66 GHz
L1 Cache	192 KB 32 KB por core
L2 Cache	1536 KB256 KB por core
L3 Cache	12 MB
Inst. Set Ext.	SSE4.2 e AVX
#Memory Channels	3
Memory BW	32 GB/s

Tabela 2. CARACTERIZAÇÃO DA MÁQUINA 431

Na tabela 1 e na tabela 2, encontra-se a caracterização das duas máquinas de teste, isto é, a caracterização da minha máquina e da máquina 431 do *Search*, respetivamente.

3. 1ª Parte - Tempo Médio de Criação de um Fio de Execução

Nesta primeira parte do trabalho, desenvolvi um pequeno algoritmo que cria um determinado número de *Threads*, executa uma função (que não faz nada) por cada *Thread*, depois faz *join* a todas as *Threads* e termina o programa. O objetivo deste algoritmo é calcular o tempo médio de criação de um fio de execução (*Thread*), de forma a poder comparar o comportamento do tempo, quando se aumenta o número de *Threads*.

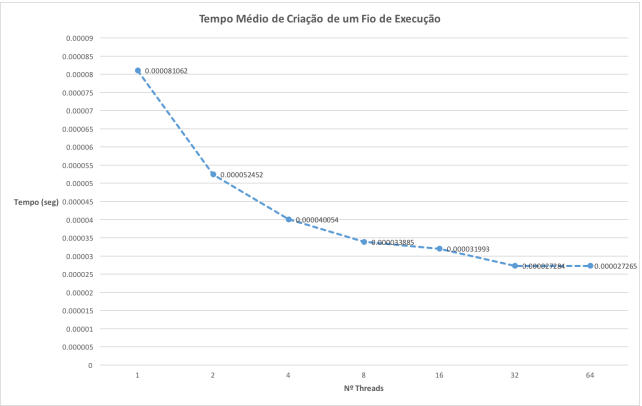

Figura 1. Tempo Médio da Criação de um Fio de Execução

No gráfico da figura 1 podemos encontra os resultados obtidos na minha máquina. Como podemos ver pela figura, o tempo médio, quando se cria uma única *Thread* é de 0.000081062 segundos, à medida que vamos aumentando o número de *Threads* o tempo médio vai diminuindo, atingindo 0.000027265 segundos com 64 *Threads*, sendo portanto, um ganho de aproximadamente 3.

Relativamente aos resultados do *Search*, o gráfico da figura 2 ilustra os resultados obtidos. A semalhança do que acontece na minha máquina pessoal, os tempos médios diminuem à medida que se aumenta o número de *Threads*, sendo que com 16 *Threads* é quando é atingido o menor tempo médio, subindo ligeiramente a partir dai até às 64 *Threads*.

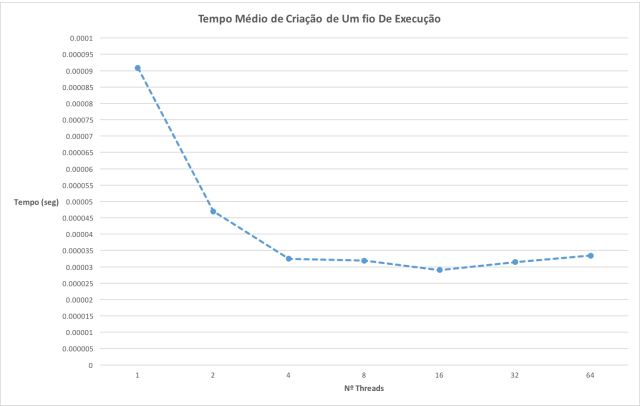

Figura 2. Tempo Médio da Criação de um Fio de Execução

Em resposta as perguntas efectuadas pelo professor no enunciado, posso dizer que o número de fios de execução (*Threads*) criados, afeta o tempo médio, como podemos constatar pela figura, quanto mais se aumenta o número de *Threads* menor é o tempo médio para a criação de uma *Thread*.

4. 2ª Parte - Regra Trapezoidal [1]

4.1. Caso de Estudo

A **Regra Trapezoidal** é uma integração numérica, cujo principal objetivo é aproximar o valor de um integral definido, de uma função, em que não é necessário usar a expressão analítica para a primitiva da função.

Este método usa três fases:

1) Decompõe-se o domínio em sub-intervalos;
2) Calcula-se uma integração aproximada para cada sub-intervalo;
3) Soma de todos os resultados obtidos.

Este método é usado, porque nem todas as funções tem uma primitiva de forma explícita, é difícil avaliar a primitiva de uma função e sobretudo quando não é possível ter uma expressão analitica para o integral, mas conhece-se os seus valores num conjunto de valores de um intervalo.

A formula matemática para a Regra Trapezoidal é dada por:

$$\int_a^b f(x)dx \simeq (b-a)\frac{f(a)+f(b)}{2}$$

Para se perceber melhor a Regra Trapezoidal, suponhamos que temos:

$$x = a$$
$$x = b$$

e

$$x_{n+1} = x_n + h$$

em que

$$h = \frac{(b-a)}{n}$$

em que *n* é número de sub-intervalos então a Regra Trapezoidal composta é:

$$\int_a^b f(x)dx \simeq \frac{h}{2}\left[f(x_1) + 2f(x_2) + 2f(x_3) + \cdots + f(x_n)\right]$$

4.2. Análise dos Resultados

O algoritmo usa *PThreads*. Depois de criadas *x Threads*, cada *Thread* calcula uma aproximação para um sub-intervalo e no fim soma aos outros resultados das outras *Threads*, de referir que é nessa operação que a variável é partilhada pelas *Threads* (região crítica). Na região critica apliquei as diferentes formas de forçar a exclusão mutua que estudamos (Busy Waiting, mutex e semáforos) de maneira a evitarmos *Data Races*. Depois de desenvolvido o algoritmo, procedi a realização de vários testes, como referi em cima foram feitos 5 execuções para cada número de *Threads* (1, 2, 4, 8 ,16, 32 e 64) e extrai o melhor tempo dessas medições. Os

meus valores de entrada foram: a=0, b=1024, sendo que n varia, correspondendo ao número de *Threads*.

O resultado do algoritmo para estes valores é de 357914112. Os resultados obtidos na minha máquina podem ser consultados no gráfico da figura 3.

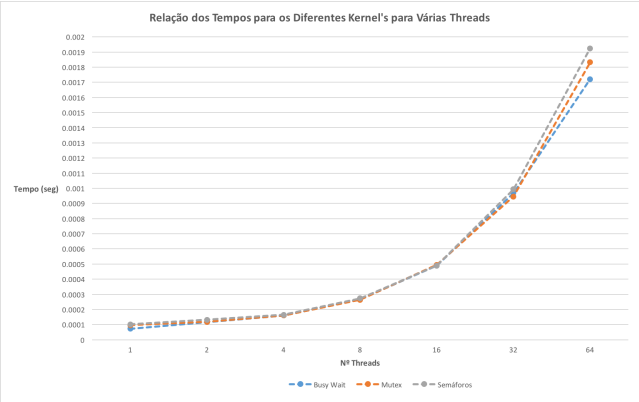

Figura 3. Relação dos Tempos para os Diferentes *Kernel's* para Várias *Threads*

Ao analisarmos o gráfico da figura 3, podemos constatar que o comportamento dos métodos de exclusão mutua, à medida que se aumenta o número de *Threads* é praticamente semelhante para os 3. De referir que com um número de *Threads* de 64 o *Busy Waiting* é o mais eficiente seguido do *Mutex* e por fim dos Semáforos. Com os resultados que obtive na minha máquina não posso concluir que haja um método que seja melhor que os outros todos, uma vez que há alguma variação, por exemplo, com 32 *Threads* o método *Mutex* é melhor que os outros mas com 64 *Threads* já é o método *Busy Waiting* e com 16 *Threads* já é o método dos semáforos (apesar da diferença ser mesmo muito reduzida em relação aos outros).

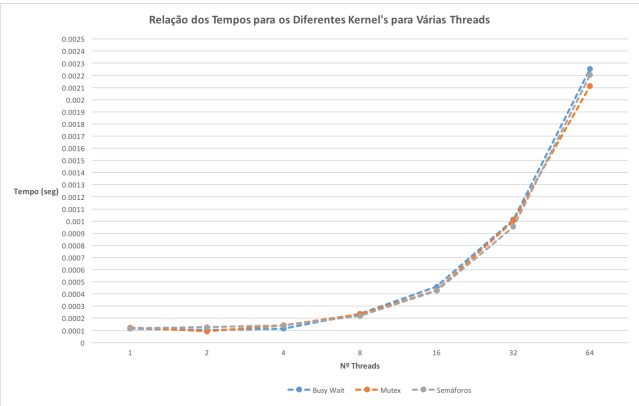

Figura 4. Relação dos Tempos para os Diferentes *Kernel's* para Várias *Threads*

Os resultados obtidos na máquina 431 do *Search* estão ilustrados no gráfico da figura 4. Ao analisarmos o gráfico, podemos constatar que o comportamento dos diferentes *Kernel's* é semelhante ao comportamento destes quando executados na minha máquina, ou seja, á medida que se aumenta o número de *Threads* o tempo de execução dos *Kernel's* também tende a aumentar. Mais uma vez, e também à semelhança do que aconteceu na minha máquina, com estes resultados não podemos concluir que um *Kernel* é melhor que o outro, pois o comportamento dos 3 é muito semelhante. Por exemplo o *Kernel Busy Waiting* é melhor com 4 *Threads*, já o *Kernel* Semáforos é o melhor com 16 e 32 *Threads* e por fim o *Kernel Mutex* é o melhor com 64 *Threads*, posto isto não posso afirmar com certeza qual é o melhor método para forçar a exclusão mutua.

4.3. Busy Waitinging

- **Vantagens**
  - Os algortítmos de *Busy Waiting* são fáceis de implementar em qualquer máquina.
- **Desvantagens**
  - Estes algoritmos são da inteira responsabilidade do programador;
  - Como existem vários processos á espera, a ocupação do *CPU* acaba por ser um desperdício de recursos. Estes processos poderiam ser bloqueados.

4.4. Semáforos

- **Vantagens**
  - Em *POSIX* os semáforos podem sincronizar processos com ou sem memória partilhada.
- **Desvantagens**
  - Uma má programação pode levar a resultados inesperados;
  - Obriga o programador a inserir explicitamente instruções *sem_wait* ou *sem_post*.

4.5. Mutexes

- **Vantagens**
  - Versão simplificada de um semáforo (não precisa de contar);
  - Só necessita de 1 bit para representar a variáverl mutex (livre,ocupado);
  - São eficientes e fáceis de implementar;
  - Se bem implementados as hipóteses de haver erros é quase 0;
  - Há um melhor aproveitamento do processador.
- **Desvantagens**
  - São adequados apenas para organizar a exclusão mútua de algum recurso ou parte de código partilhada

5. Conclusão

As *PThreads* ou *POSIX Threads*, como já foi dito em cima, são muito utilizadas em ambientes *Unix*, para criação de *Threads*, com estas podemos facilmente paralelizar o código, de forma a tirar o melhor partido dos recursos que temos disponíveis. Como vimos nas análises dos resultados obtidos para a primeira parte deste trabalho quanto mais *Threads* criámos, menor é o seu tempo de criação.

Para se paralelizar um código usando *PThreads*, é necessário ter em atenção as regiões críticas, estas não podem ser acedidas por mais que uma *Thread* ao mesmo tempo, para isso é necessário implementar mecanismos de exclusão mútua (*Busy Waiting, Mutex* e Semaforos). Estes 3 mecanismos, como vimos, foram implementados e testados em duas máquinas, a minha máquina pessoal e a máquina 431 do *Search*, das quais obtive os resultados em cima apresentados, resultados esses que não me permitem concluir qual é o mecanismo/método mais eficiente para implementar exclusão mutua.

Apesar de não ter obtido dados muito esclarecedores, penso que a implementação de exclusão mutua será mais eficiente/segura se usarmos semáforos ou *Mutex*, embora o *Mutex* seja apenas adequado para uma parte de código ou algum recurso partilhado. Penso que estes dois métodos são os mais eficientes/seguros pois são os que tem menor margem de erro e como tal a ocorrência de *Data Races* é pouco provável, para além disso, permite-nos tirar um maior partido dos recursos que dispomos.

Na realização deste trabalho não me deparei com grandes dificuldades, uma vez que era um trabalho simples, e curto. Contudo confesso que não estou satisfeito com os meus resultados, isto porque, não era bem estes resultados que estava à espera. Esperava que os tempos de execução dos 3 *Kernel's* fossem mais esclarecedores do que aqueles que obtive, posto isto como trabalho futuro pretendo debruçar-me melhor sobre o algoritmo, apesar de achar que este está bem implementado, mas gostava de perceber o porquê de estes resultados não terem sido os que esperava. Quanto ao restante trabalho faço um balanço bastante positivo.

Referências

[1] Integração numérica. https://pt.wikipedia.org/wiki/Integracao_numerica.