deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

Computação em Larga Escala – Trabalho 1

P3 G1

Diogo Carvalho – 92969 – MEI

Rafael Baptista – 93367 – MEI

Abril de 2022

Programa 1

O primeiro programa consiste na leitura de ficheiros de texto, cujo o nome são passados como argumentos ao programa, e no seu processamento de modo a realizar uma contagem das palavras totais, das palavras que começam com uma vogal e das palavras que acabam com uma consoante. É importante referir que, no processamento do texto, todas os caracteres com acentos são convertidos para caracteres sem acentos e os ç/Ç para c/C, contando assim para as contagens.

O programa está desenvolvido em multi-thread e por isso o seu funcionamento está referido nos seguintes pontos:

1. Ao executar o programa, o primeiro passo da main thread é guardar os nomes dos ficheiros e inicializar os seus contadores a 0 numa região partilhada. Esta região partilhada está implementada através de um monitor e trata-se de um array de elementos do tipo struct FileCounters em que contêm o nome do ficheiro, e os valores dos três contadores.

2. De seguida a main thread vai gerar todas as threads invocando-lhes a função **worker**:
   - Funcionamento função **worker**: pedir um chunk acedendo a uma região partilhada que irá ser explicada no ponto 3. Caso não haja chunks disponíveis, a thread vai esperar até que seja colocado um chunk pela main thread. Após obter o chunk, vai verificar se é o chunk especial que significa que não existem mais chunks para processar (se for, a thread simplesmente termina), de seguida vai processar o chunk de dados, liberar a memória utilizada pelo chunk uma vez que já não é necessária e guardar os contadores deste chunk na região partilhada referente ao seu ficheiro referida no ponto 1. Importante referir que o processamento do chunk, isto é, a contagem dos três contadores foi implementada seguindo a solução do professor.

3. Após criar as threads, a main thread vai criar os chunks e guardá-los numa região partilhada implementada também através de um monitor referida no ponto anterior. Esta região partilhada consiste num FIFO de tamanho definido no ficheiro probConst.h, que irá guardar elementos struct ChunkInfo que contêm um identificador do ficheiro a que o chunk pertence, o tamanho do chunk (em bytes), e um ponteiro para o começo do chunk. Para criar os chunks, a main thread vai iterar todos os ficheiros, e tentar criar chunks de tamanho 4000 bytes. Caso o tamanho do ficheiro atual ou o que resta do ficheiro atual seja inferior a 4000 bytes, o chunk tem o tamanho do ficheiro até ao final. Caso seja superior a 4000 bytes, por forma a garantir que o chunk acaba num caractér de corte, isto é, que não divida uma palavra a meio e que não divida um carater multibyte a meio, a main thread é responsável por verificar se o último caractér do chunk é um caractér de corte, e caso não seja vai aumentar o tamanho do chunk até que acabe num caracter seguro de corte. Após encontrar este caracter de corte, o chunk está pronto a ser guardado na região partilhada, para isso a main thread vai ler o chunk do ficheiro e guardá-lo num buffer incluindo também o último caracter de corte, guardando-o de seguida na região partilhada para que as threads consigam processar o chunk. Caso o FIFO esteja cheio, a main thread vai esperar até que uma thread obtenha um chunk para haver espaço para guardar este novo chunk no FIFO.

4. Por último, após ter criado todos os chunks, a main thread vai guardar um número igual ao número de threads existentes de elementos do tipo struct ChunkInfo no FIFO com identificador do ficheiro igual a -1, por forma a informar todas as threads que já não existem mais chunks para serem processados.

5. Finalmente, a main thread vai esperar que as threads terminem e no fim, vai aceder ao monitor que contém as contagens dos ficheiros e vai mostrar os resultados.

Resultados programa 1

ficheiro \ contador	Palavras totais	Palavras que começam com vogal	Palavras que acabam com consoante
text0.txt	14	10	4
text1.txt	1184	381	365
text2.txt	11027	3648	3220
text3.txt	3369	1004	1054
text4.txt	9914	3095	3175

Nº Threads	1	2	4	8
Tempo de execução (s)	0.008113 0.009000	0.006314 0.006798	0.004738 0.005604	0.004315 0.004390

Nota: os resultados foram obtidos utilizando dois computadores:

- Azul – Hp Omen, intel core i7 8750H, com 6 cores
- Vermelho – Asus ROG, intel core i7 67000HQ, com 4 cores

Programa 2

O segundo programa consiste na leitura de um ficheiro binário que contém matrizes, cujo o nome é passado como argumento ao programa, e no seu processamento de modo a calcular o determinante de cada matriz presente no ficheiro. É importante referir que, no início do ficheiro está presente o número de matrizes que o ficheiro contém, e a sua ordem, e só após isso é que estão presentes as matrizes.

O programa está desenvolvido em multi-thread e por isso o seu funcionamento está referido nos seguintes pontos:

1. Ao executar o programa, o primeiro passo da main thread após processar os argumentos do programa é abrir o ficheiro e gerar as todas as threads invocando-lhes a função **worker**:
    - Funcionamento função **worker**: pedir uma matriz acedendo a uma região partilhada que está implementada através de um monitor que irá ser explicada no ponto 2. Caso não haja matrizes disponíveis, a thread vai esperar até que seja colocada uma matriz pela main thread. Após obter a matriz, vai verificar se é a matriz especial que significa que não existem mais matrizes para processar (se for, a thread simplesmente termina), de seguida vai calcular o determinante, liberar a memória utilizada pela matriz uma vez que já não é necessária, e guardar o determinante desta matriz na posição correspondente de um array responsável por guardar os determinantes de todas as matrizes.
2. Após criar as threads, a main thread vai ler o número de matrizes que o ficheiro contém, e com esta informação, alocar memória para o array responsável por guardar todos os determinantes. De seguida vai ler a ordem das matrizes e vai começar a ler matriz por matriz para um buffer e guardar a matriz no FIFO acedendo a uma região partilhada implementada através de um monitor. Caso o FIFO esteja cheio, a main thread vai esperar até que uma thread obtenha uma matriz, libertando assim espaço para a main thread colocar uma nova matriz. Na região partilhada, o FIFO irá guardar elementos do tipo struct MatrixInfo que contêm o identificador da matriz, a ordem da matriz e um ponteiro para o inicio da matriz.
3. Por último, após ter guardado todas as matrizes no FIFO, a main thread vai guardar um número igual ao número de threads existentes de elementos do tipo struct MatrixInfo no FIFO com identificador da matriz igual a -1, por forma a informar todas as threads que já não existem mais matrizes para serem processadas.
4. Finalmente, a main thread vai esperar que as threads terminem e no fim, vai percorrer o array que contém os determinantes e mostrar os resultados calculados.

Resultados programa 2

Relativamente aos resultados dos cálculos dos determinantes das matrizes comparámos com os resultados que o professor forneceu e verificámos que os resultados eram os mesmos.

Tempos de execução (s) :

ficheiro \ threads	1	2	4	8
mat128_32.bin	0.005972 0.005709	0.004100 0.004295	0.003528 0.004647	0.004770 0.004893
mat128_64.bin	0.010229 0.012675	0.007821 0.008581	0.006336 0.007568	0.004398 0.006057
mat128_128.bin	0.046244 0.058642	0.026757 0.032949	0.016982 0.022934	0.012305 0.019511
mat128_256.bin	0.377998 0.478656	0.202911 0.244555	0.109930 0.163160	0.098092 0.120338

ficheiro \ threads	1	2	4	8
mat512_32.bin	0.013098 0.011224	0.007539 0.008951	0.007286 0.008158	0.013276 0.006864
mat512_64.bin	0.028139 0.035191	0.017207 0.021493	0.010745 0.016513	0.010122 0.015714
mat512_128.bin	0.171736 0.211053	0.090833 0.115352	0.046733 0.079815	0.041724 0.061688
mat512_256.bin	1.502593 1.807789	0.762295 0.958790	0.408227 0.511060	0.353288 0.451839

Nota: os resultados foram obtidos utilizando dois computadores:

- Azul – Hp Omen, intel core i7 8750H, com 6 cores
- Vermelho – Asus ROG, intel core i7 67000HQ, com 4 cores

Conclusões

Por forma a retirar melhores conclusões, decidimos testar os dois programas em duas máquinas distintas e efetuar comparações relativas aos resultados que obtivemos:

1. Hp Omen, com 16GB de memória RAM. Processador Intel core i7 8750H 2.2GHz, com 6 cores e sistema operativo Xubuntu 20.02.
2. Asus ROG, com 16GB de memória RAM. Processador Intel core i7 6700HQ 2.6GHz, com 4 cores e sistema operativo Ubuntu 20.04.

Relativamente ao **primeiro exercício**:

- Ambos os computadores têm o melhor desempenho quando se corre o programa com 8 threads, e pior com 1.
- Comparando os dois computadores, apesar de desempenhos semelhantes verifica-se que o Hp Omen tem o melhor desempenho com 8 threads.
- Apesar de não estar demonstrado nos resultados, os tempos que apresentamos são o valor médio de várias experiências, e verificámos que o aumento do desempenho não é bastante significativo quando se aumenta de 4 threads para 8 threads sendo que por vezes este acabou por ser pior em algumas experiências.

Face a estes resultados, conseguimos concluir que, uma vez que o computaor HP Omen 15 é um computador mais potente, este acaba por ter melhores resultados. O facto de por vezes o desempenho não ser melhor quando executamos com mais threads deve-se ao facto do problema ser simples e pouco exigente em termos computacionais, pelo que por vezes quantas mais threads existirem, o tempo para as gerir acaba por piorar o desempenho do programa.

Relativamente ao **segundo exercício**:

- De uma forma geral, ambos os computadores têm o melhor desempenho quando se corre o programa com 8 threads, e pior com 1.
- Quanto maior é o ficheiro, pior é o desempenho do programa, mas verifica-se uma melhoria cada vez mais significativa ao correr o programa com várias threads do que com apenas 1.
- Os restantes resultados apontados no primeiro exercício também se verificam neste segundo exercício.

Face aos resultados, ao contrário do primeiro exercício, neste caso já conseguimos verificar melhor as diferenças no desempenho para ficheiros maiores, e posto isto quando se aumenta de 1 para 2 threads, o tempo de execução reduz para cerca de 50%, de 2 para 4 o mesmo se verifica, contudo, de 4 para 8 esta diferença já não se nota tanto. Quanto maior for o ficheiro, maior é a complexidade do problema, pelo que é possível verificar melhor as diferenças no desempenho do programa para o diferente número de threads.

Concluímos, portanto, que a implementação de programas multithread pode oferecer uma melhoria bastante significativa no seu desempenho relativamente aos programas singlethread, quando aplicada corretamente. Por sua vez, concluímos mais uma vez, que este desempenho está bastante ligado à máquina em que o programa é executado, pelo que quanto maior for o poder computacional da máquina, melhor será o desempenho do programa.