Faculdade de Engenharia da Universidade do Porto
Faculdade de Ciências da Universidade do Porto

Licenciatura em Engenharia Informática e Computação

Computação Paralela e Distribuída - L.EIC028

março de 2024

Projeto 1

Avaliação de *Performance* da Hierarquia de Memória

Produto de Duas Matrizes

Turma 2 - Grupo 17
António Augusto Brito de Sousa - up202000705@up.pt
Pedro de Almeida Lima - up202108806@up.pt
Pedro Simão Januário Vieira - up202108768@up.pt

Índice

Problema e Algoritmos **1**
Medidas de performance **2**
Resultados e análise **2**
Parte 1: avaliação da performance sequencial (core único) 2
Parte 2: avaliação da performance paralela (multi-core) 4
Conclusões **5**
Referências **5**

Problema e Algoritmos

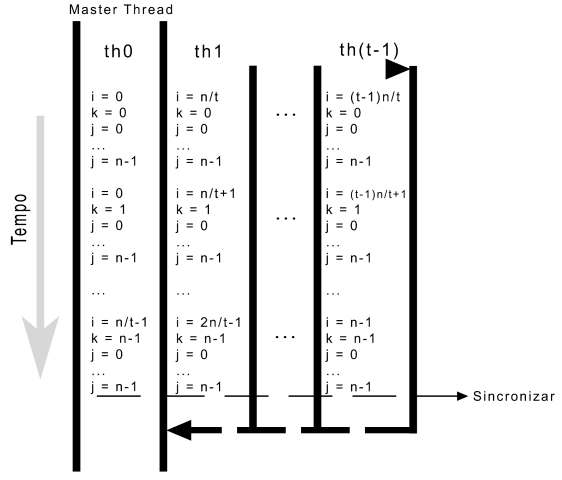
No projeto a que se refere o presente relatório, é pedido para implementarmos alguns algoritmos para calcular o produto de matrizes, avaliando a sua *performance*, com recurso à ferramenta Performance API (PAPI), que permite consultar os valores de contadores da CPU, e retirando conclusões sobre os resultados obtidos.

O projeto está dividido em 2 partes.
A primeira consiste na implementação de três algoritmos que calculam o produto de duas matrizes, utilizando um só *core* do processador. Todos os algoritmos foram implementados em C/C++, dos quais os dois primeiros foram também em Java.
O primeiro é o mais simples para abordar o problema. Para cada posição na matriz resultante, calcula o produto escalar entre a linha correspondente a esta posição na primeira matriz pela coluna correspondente da segunda matriz.
O segundo algoritmo é o algoritmo de multiplicação de linha, que multiplica um elemento da primeira matriz pela linha correspondente da segunda matriz. É mais eficiente do que o primeiro, pois os valores estão mais próximos uns dos outros, o que leva a um melhor aproveitamento da *cache*.
O terceiro é o algoritmo de multiplicação por blocos, que divide as matrizes em blocos (submatrizes mais pequenas) e multiplica-os recorrendo ao método de multiplicação de linha, obtendo, assim, o produto das matrizes maiores. A principal diferença em relação a uma multiplicação de matrizes sem a divisão em blocos é que se as matrizes forem demasiado grandes, torna-se impossível de manter uma linha inteira da matriz em cache. A divisão em blocos resolve este problema, tornando, assim, o algoritmo mais eficiente.

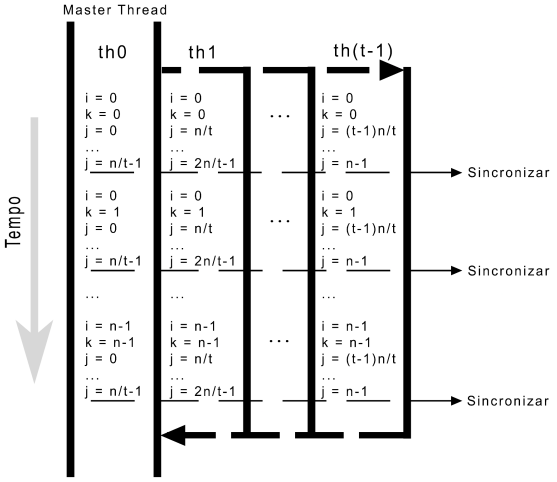
A segunda parte do projeto consiste na implementação de duas versões paralelas do segundo algoritmo (multiplicação de linha), e na análise da *performance* das mesmas.
Doravante, seja *n* o número de linhas/colunas da matriz e *t* o número *threads* criadas.
Na primeira versão, utilizamos a diretiva "`#pragma omp parallel for`", para paralelizar o ciclo mais externo do algoritmo. Esta diretiva cria *threads* consoante o número de *cores* disponíveis no processador e distribui as iterações do ciclo mais externo de multiplicação pelas mesmas. Nesta versão, o código apenas precisa de sincronizar valores quando o ciclo mais externo estiver concluído, ou seja, quando todas as iterações forem concluídas, ao fim de n^3/t iterações.
Na segunda versão, utilizamos a diretiva "`#pragma omp parallel`" antes de começarmos qualquer iteração, para criar as *threads* de acordo com o número de *cores* disponíveis no processador, e a diretiva "`#pragma omp for`" para paralelizar o ciclo mais interno, ou seja, para distribuir as iterações deste ciclo pelas *threads* criadas anteriormente. Nesta versão os dois ciclos mais externos são realizados em todas as *threads.* Esta versão tem que sincronizar sempre que um ciclo mais interno termina, ou seja, de *n/t* em *n/t* iterações, o que adiciona um *overhead* ao processo.
Em ambas as versões, cada *thread* executa n^3/t iterações.

A divisão da computação pelas *threads* pode ser mais bem compreendida no diagrama seguinte. Tal divisão resulta do facto de, por omissão, o *scheduling* do OpenMP ser *static*.

Na primeira versão:
th0: i ∈ [0, n/t[
  k ∈ [0, n[
  j ∈ [0, n[
th1: i ∈ [n/t, 2n/t[
  k ∈ [0, n[
  j ∈ [0, n[
…
th(t-1): i ∈ [(t-1)n/t, n[
  k ∈ [0, n[
  j ∈ [0, n[



Na segunda versão:
th0: i ∈ [0, n[
  k ∈ [0, n[
  j ∈ [0, n/t[
th1: i ∈ [0, n[
  k ∈ [0, n[
  j ∈ [n/t, 2n/t[
…
th(t-1): i ∈ [0, n[
  k ∈ [0, n[
  j ∈ [(t-1)n/t, n[



Medidas de *performance*

Nas medições de *performance* dos algoritmos em C++ utilizamos a API disponibilizada pela unidade curricular, Performance API (PAPI), que nos dá acesso a parâmetros da CPU. Para cada exercício, foram executados 3 testes numa mesma máquina equipada com um processador Intel Core i7-9700, no SO Ubuntu 22.04. A compilação do programa foi feita utilizando a flag -O2.

Levamos em consideração 3 níveis de *cache* do processador (L1,L2 e L3), com o intuito de comprovar a eficiência de cada versão do algoritmo de multiplicação de matrizes, no que toca à gestão da memória. Além disso, também levamos em consideração o número de dupla precisão de números de vírgula flutuante (MFlops) e o número de instruções por segundo (MIPS) para o cálculo do speedup e eficiência. Vale também salientar que utilizamos o MIPS para analisar a melhoria na aplicação do algoritmo em relação ao tempo, nomeadamente, para a segunda e terceira versão do algoritmo.Comparámos também o tempo de execução entre a linguagem C++ e Java para as duas primeiras versões de aplicação do algoritmo.

Para calcular a eficiência das soluções paralelas, de modo a estabelecer uma comparação com o algoritmo de multiplicação de linhas sequencial, consultámos o número de *threads* disponíveis no processador utilizado: 8 (conforme referência indicada no final do presente documento).

Resultados e análise

Parte 1: avaliação da *performance* sequencial (*core* único)

Seguem-se os tempos médios de execução (em segundos) das implementações em C/C++ e Java da primeira e segunda soluções, acompanhados de uma comparação entre os tempos de execução de uma linguagem para outra:

Dimensão da matriz	Multiplicação básica (C/C++)	Multiplicação básica (Java)	T(C) / T(Java)	Multiplicação por linha (C/C++)	Multiplicação por linha (Java)	T(C) / T(Java)
600 x 600	0,198	0,217	91,24%	0,110	0,164	67,07%
1000 x 1000	1,286	1,550	82,97%	0,495	0,782	63,30%
1400 x 1400	3,530	4,600	76,74%	1,818	2,735	66,47%
1800 x 1800	18,325	19,033	96,28%	3,580	5,736	62,41%
2200 x 2200	38,472	40,213	95,67%	6,449	10,592	60,89%
2600 x 2600	69,156	70,738	97,76%	10,765	17,291	62,26%
3000 x 3000	117,085	117,092	99,99%	16,36	26,651	61,39%
Média			91,52%			63,40%

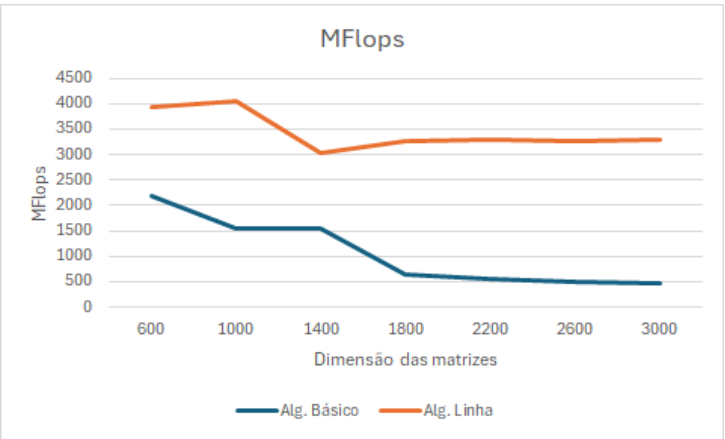
É notória a superioridade das soluções em C/C++: no algoritmo *naïve*, as execuções em C demoraram, em média, 91,52% do tempo que demoraram as execuções em Java; já no caso do segundo algoritmo, a diferença é ainda mais expressiva, com as execuções em C a tomarem, em média, apenas 63,40% das execuções em Java. Tal observação indica que uma boa gestão de memória tem maior tradução em C/C++ do que em Java.

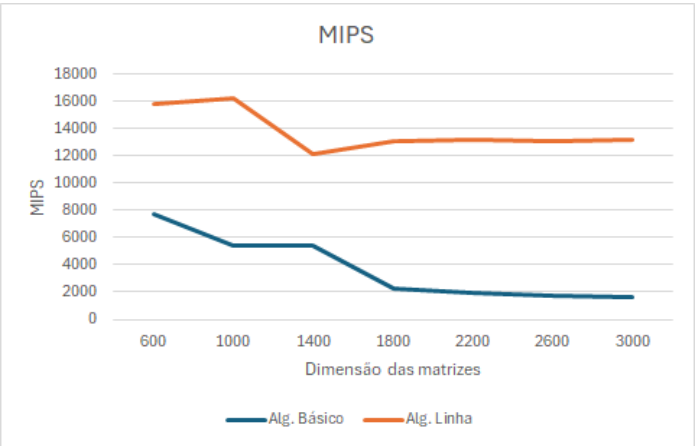
Para as execuções em C/C++ das mesmas soluções, obtivemos, também, os valores médios dos contadores desejados e, consequentemente, calculámos MFlops e MIPS:

Sol.	Dim.	Tempo	L1 DCM	L2 DCM	L3 LDM	DP OPS	TOT INS
Básica	600	0,198	2,45E+08	3,95E+07	1,07E+05	4,32E+08	1,52E+09
	1000	1,286	1,23E+09	3,05E+08	8,21E+06	2,00E+09	7,02E+09

	1400	3,530	3,50E+09	1,38E+09	1,39E+08	5,49E+09	1,92E+10
	1800	18,325	9,09E+09	7,92E+09	5,04E+08	1,17E+10	4,09E+10
	2200	38,472	1,77E+10	2,29E+10	8,61E+08	2,13E+10	7,46E+10
	2600	69,156	3,09E+10	5,13E+10	1,28E+09	3,52E+10	1,23E+11
	3000	117,085	5,03E+10	9,56E+10	1,81E+09	5,40E+10	1,89E+11
Linha	600	0,110	2,71E+07	5,71E+07	1,16E+04	4,32E+08	1,74E+09
	1000	0,495	1,26E+08	2,63E+08	2,57E+05	2,00E+09	8,02E+09
	1400	1,818	3,46E+08	6,84E+08	6,21E+06	5,49E+09	2,20E+10
	1800	3,580	7,45E+08	1,43E+09	1,80E+07	1,17E+10	4,67E+10
	2200	6,449	2,07E+09	2,56E+09	3,97E+07	2,13E+10	8,53E+10
	2600	10,765	4,41E+09	4,17E+09	7,46E+07	3,52E+10	1,41E+11
	3000	16,360	6,78E+09	6,35E+09	1,21E+08	5,40E+10	2,16E+11
	4096	44,432	1,75E+10	1,60E+10	4,59E+08	1,37E+11	5,50E+11
	6114	143,651	5,91E+10	5,40E+10	1,60E+09	4,64E+11	1,86E+12
	8192	345,084	1,40E+11	1,29E+11	3,88E+09	1,10E+12	4,40E+12
	10240	654,307	2,73E+11	2,58E+11	7,62E+09	2,15E+12	8,59E+12

Na tabela acima, é possível observar a demora induzida pela má gestão da memória *cache* por parte do algoritmo básico, bem como a superioridade no nº de *data cache misses* em comparação com o algoritmo de multiplicação por linha. Uma vez que dificilmente é mantida em *cache* mais do que uma linha de uma matriz e que o 1º algoritmo acede consecutivamente a linhas diferentes da matriz B, a probabilidade de *cache miss*, para encontrar o elemento de B que corresponda ao elemento de A em cada momento, é mais alta.
Consequentemente, uma *cache miss* implica acesso a camadas seguintes de *cache* ou à memória principal, operação que consome ciclos de relógio sem que, durante tal intervalo de tempo, seja efetuado algum cálculo ou execução de instrução útil.



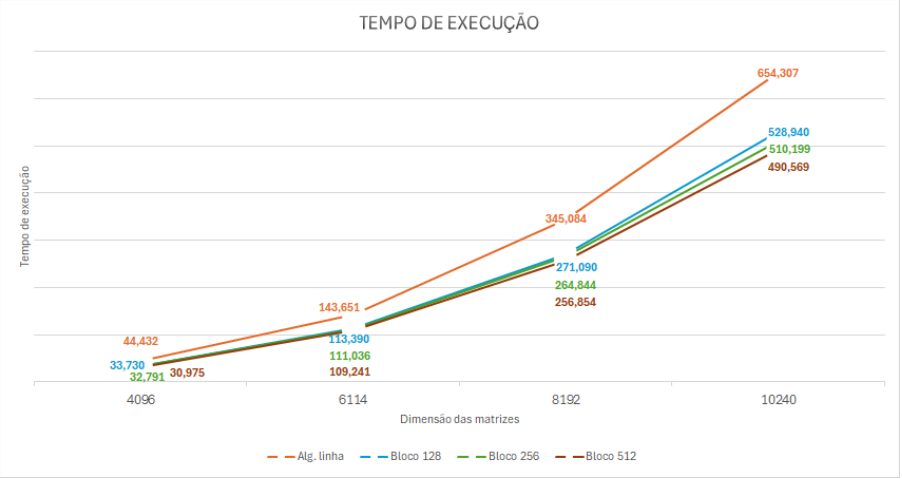


Ora, tal fator explica, para o primeiro algoritmo, o decréscimo no nº de MFlops e de MIPS (como demonstram os gráficos acima) com o aumento da dimensão das matrizes: muito tempo é gasto em acessos derivados de *cache misses*, sem que esteja a ser efetuada alguma operação, o que faz aumentar o tempo de execução em maior medida do que o nº de operações de vírgula flutuante e de instruções executadas (amplitudes de 1720,62 MFlops e de 6062,56 MIPS). Por outro lado, no caso do segundo algoritmo, graças à melhor gestão de memória *cache*, o nº de MFlops e o de MIPS não sofrem discrepâncias tão grandes (amplitudes de 1020,60 e de 4100,81, respetivamente, considerando apenas matrizes de dimensão entre 600 e 3000, para igualar as condições em que o primeiro algoritmo é executado).

Apresentamos os tempos médios de execução, valores médios dos contadores obtidos (arredondados à unidade por excesso) e MFlops/MIPS com a terceira solução:

Bloco	Dim.	Tempo	L1 DCM	L2 DCM	L3 LDM	DP OPS	MFlops	TOT INS	MIPS
128	4096	33,730	9,47E+09	2,14E+09	3,15E+07	1,38E+11	4091,31	5,59E+11	16572,78
	6114	113,390	3,20E+10	7,28E+09	9,21E+07	4,66E+11	4109,71	1,89E+12	16668,14
	8192	271,090	7,58E+10	1,71E+10	4,19E+08	1,10E+12	4057,69	4,47E+12	16488,99
	10240	528,940	1,48E+11	3,37E+10	5,28E+08	2,16E+12	4083,64	8,74E+12	16523,61
256	4096	32,791	5,34E+10	1,92E+10	5,20E+07	1,38E+11	4208,47	5,54E+11	16894,88
	6114	111,036	3,07E+10	6,52E+10	1,50E+08	4,65E+11	4187,83	1,87E+12	16841,38
	8192	264,844	7,27E+10	1,55E+11	4,70E+08	1,10E+12	4153,39	4,43E+12	16726,83
	10240	510,199	1,42E+11	3,02E+11	6,84E+08	2,15E+12	4214,04	8,66E+12	16973,77
512	4096	30,975	8,85E+09	1,87E+10	2,53E+07	1,38E+11	4455,21	5,52E+11	17820,82
	6114	109,241	2,99E+10	6,32E+10	9,21E+07	4,64E+11	4247,49	1,40E+12	12815,70
	8192	256,854	7,08E+10	1,50E+11	2,13E+08	1,10E+12	4282,59	3,09E+12	12030,18
	10240	490,569	1,38E+11	2,93E+11	4,26E+08	2,15E+12	4382,67	8,63E+12	17591,82

Tal como exposto anteriormente, a divisão em blocos torna-se vantajosa no caso de matrizes com maior dimensão, uma vez que deixa de haver capacidade em *cache* para manter linhas inteiras. Assim, pode observar-se, comparando o algoritmo de multiplicação por blocos com o de linha, uma discrepância nos tempos de execução, tanto maior quanto a dimensão das matrizes a multiplicar, comprovando a utilidade e melhoria introduzida pela solução. Analogamente, a escolha de tamanho de bloco maior ganha mais relevância quanto maior for a dimensão das matrizes.



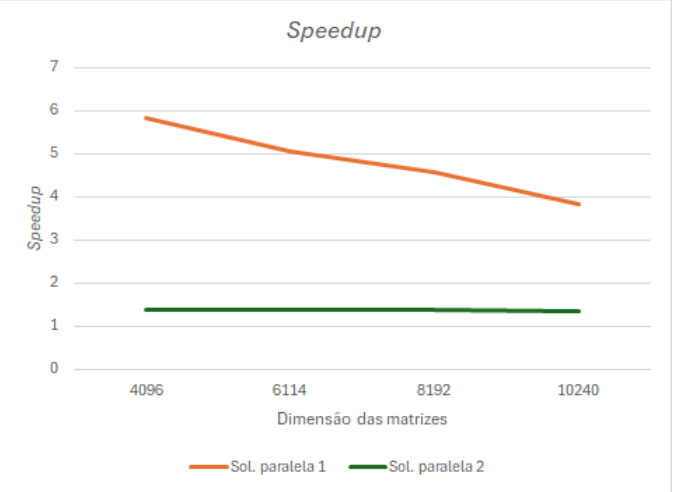
Parte 2: avaliação da *performance* paralela (*multi-core*)

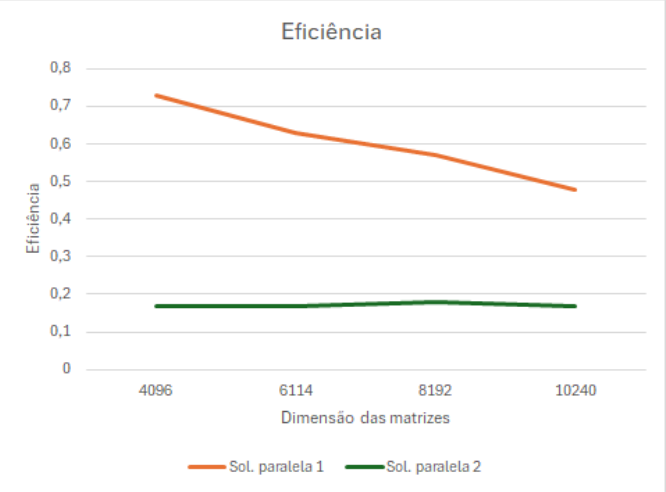
Seguem-se as habituais medições relativas às soluções paralelas, sendo apresentados, o *speedup* e a eficiência, estabelecendo uma comparação com a solução algoritmicamente igual (multiplicação por linha), mas sequencial.

(Nota: por questões de economia de espaço na folha, não é indicado na tabela a qual das soluções paralelas cada linha corresponde. Assim sendo, faz-se saber que a divisória horizontal mais espessa encarrega-se da referida distinção, sendo que as linhas da tabela acima da mesma correspondem às medidas das execuções da primeira solução paralela e, consequentemente, as linhas abaixo, à segunda).

Dim.	Tempo	L1 DCM	L2 DCM	L3 LDM	DP OPS	MFlops	TOT INS	MIPS	T. seq
4096	7,613	2,20E+09	2,10E+09	8,07E+06	1,72E+10	2259,29	6,89E+10	9050,31	44,432
6114	28,317	7,43E+09	7,13E+09	3,15E+07	5,80E+10	2048,24	2,32E+11	8192,96	143,651

8192	75,495	1,76E+10	1,68E+10	1,34E+08	1,37E+11	1814,69	5,50E+11	7285,25	345,084
10240	170,949	3,43E+10	3,20E+10	2,75E+08	2,68E+11	1567,72	1,07E+12	6259,18	654,307
4096	32,035	1,22E+09	2,32E+09	1,37E+07	1,72E+10	536,91	6,37E+10	1988,45	44,432
6114	104,546	4,08E+09	7,86E+09	1,63E+08	5,80E+10	554,78	2,11E+11	2018,25	143,651
8192	246,199	9,20E+09	1,48E+10	4,08E+08	1,37E+11	556,46	4,96E+11	2014,63	345,084
10240	490,017	1,82E+10	2,93E+10	9,29E+08	2,68E+11	546,92	9,71E+11	1981,56	654,307





Decorrente do já detalhado anteriormente neste documento, não surpreende que a segunda solução paralela traga uma vantagem muito reduzida em relação à solução equivalente sequencial, uma vez que, recordando, a cada iteração da multiplicação, há que sincronizar valores, fazendo com que o trabalho efetivamente paralelo seja pontual.

Por último, analisemos o *speedup* e eficiência conseguidos com a adoção da primeira solução.Tal como na solução sequencial algoritmicamente equivalente, o nº de MFlops e de MIPS diminui com o aumento da dimensão das matrizes, dado o aumento de *cache misses*, devido às razões já apresentadas no que respeita à solução sequencial. Ora, uma *cache miss* (concretamente, das *caches* L1 e L2, dado que, no processador utilizado, cada núcleo tem *cache* L1 e L2 próprias, sendo L3 comum) por parte de um núcleo individual implica acesso a memória partilhada, o que implica que as vantagens do paralelismo desvaneçam.

Conclusões

Em primeiro lugar, a realização deste primeiro Trabalho Prático e consequente redação do presente documento permitiram-nos consolidar, aplicando, na prática, os conhecimentos adquiridos ao longo das aulas teóricas da UC, no que diz respeito à programação paralela.

Com o desenrolar do projeto, pudemos observar as implicações que a gestão de memória tem na eficiência dos algoritmos, bem como as vantagens e limitações da paralelização de algoritmos.

Referências

Intel Core i7-9700, CPU Grade (https://cpugrade.com/db/intel/desktop/core-i7/i7-9000/i7-9700/, consultado a 14 de março de 2024)