Faculdade de Engenharia da Universidade do Porto
Unidade Curricular - Computação Paralela e Distribuição

Relatório do 1º Projeto

Avaliação do Desempenho de Um Único Core

Contribuidores
Henrique Sousa, up201906681
Mateus Silva, up201906232
Melissa Silva, up201905076

Descrição do Problema

O problema dado para análise no presente relatório prende-se com a operação de produto de matrizes, utilizando três variações diferentes para a resolução deste problema em código. Como parte da investigação, era esperado o estudo das duas primeiras variações para duas linguagens, uma a nosso critério e a outra sendo C/C + +. Só esta última é esperada na terceira variação da resolução.

A multiplicação de matrizes é uma operação com um raciocínio mais prático e mecânico quando feita de forma manual - isto é, com papel e material de escrita -, mas também pode ser feita também utilizando programação. Para efeitos de melhor compreensão, temos abaixo um exemplo da operação manual que se espera implementar.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \mathrm{x} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Para calcularmos a matriz, resultado, precisamos de fazer 8 operações de multiplicação e o mesmo número de somas: para o primeiro elemento, na posição (1, 1), faremos $1 * 5 + 2 * 7$, cujo resultado é 19. Multiplicamos então a primeira linha da matriz esquerda pela primeira coluna da linha direita. Segue-se multiplicar a primeira linha pela segunda coluna - elemento na posição (1, 2) da matriz produto) -, por exemplo.

Já em programação, temos alguns desafios a estudar, nomeadamente, os algoritmos utilizados, explorados com mais detalhe mais tarde. Estes procuram, por pressuposto, aumentar a velocidade da operação.

Descrição dos Algoritmos

Primeira Versão

A primeira versão da resolução é a que possui o menor grau de paralelismo. O código possui três níveis diferentes, distinguíveis pelo cabeçalho do seu respetivo ciclo *for*. Todos os ciclos *for* utilizam iteradores (*i*, *j* e *k*) percorrendo toda a dimensão da matriz para ter em conta todas as operações necessárias, que seriam $2n^3$(o 2 é devido à multiplicação e soma dos elementos).

Cada posição (*i, j*) é mantida durante as *n* operações do iterador *k*, e sendo este o iterador mais profundo, é usado para corresponder uma posição numa linha de uma matriz a uma posição numa coluna da outra, indicando o progresso no produto vetorial. Percebemos assim que percorremos todas as posições de uma coluna antes de avançarmos na linha.

Utilizando cada posição da matriz produto como um acumulador para todas as operações relativas a este, torna-se possível obter o resultado esperado, a fórmula para conseguir tal é a seguida no interior do ciclo *for* da variável k:

$$M_p[i][j] = M_p[i][j] + A[i][k] * B[k][j]$$

Sendo M_p a matriz produto, indexada com um par de coordenadas (linha, coluna).

Este algoritmo é o mais próximo do cálculo manual, mas acaba por ter a particularidade de aceder repetidamente às mesmas colunas, gastando-se então o triplo do espaço para guardar as matrizes necessárias (os dois operandos e o produto).

Além disso, os acessos a memória, devido ao funcionamento dos ciclos *for*, incrementam em uma unidade por iteração, acedendo a posições sequenciais na matriz ao longo de uma coluna ou linha. Embora isto tenda a ajudar na resolução manual, em programação há que ter em conta que posições sequenciais na matriz não serão guardadas sequencialmente em memória, pelo que ter de aceder à posição seguinte à corrente terá os seus custos de processamento.

Por exemplo, para se obter uma posição necessária, aceder-se-ão a posições que podem ser irrelevantes para a operação de então.

Segunda Versão

A segunda versão da resolução tenta responder a uma das características da versão anterior que afetava o desempenho, nomeadamente, a repetição de acessos a elementos das matrizes. O código é muito semelhante, mas troca-se a ordem dos dois últimos ciclos *for* - mantendo a mesma nomenclatura usada anteriormente, a ordem fica então (i, k, j).

A fórmula utilizada para acumulação de resultados é a mesma da versão anterior, pondo-se, portanto, a questão de perceber qual o efeito que a alteração feita teve. O efeito é que o acesso aos elementos das matrizes é feito em linha e não em coluna, o que reduz o número de acessos repetidos, visto que, avançando através de uma linha, para cada novo elemento, utilizaremos uma nova coluna.

Esta versão possui melhor gestão de memória do que a anterior, permitindo a redução do tempo passado a fazer a operação.

Terceira Versão

A terceira versão procura otimizar o uso da memória obtendo uma maior taxa de *hits* na cache.

Com o cálculo do produto matricial com base em blocos, cada par de blocos é usado de forma independente e final. Se a isto se aliar um tamanho de bloco ideal, um ciclo (em que se mantém o mesmo par de blocos) que vá à memória para obter um conjunto de dados não irá repetir esse processo, porque conseguirá, eventualmente, guardar em cache toda a informação necessária para o ciclo.

Além disso, utiliza-se também a multiplicação em linha para beneficiar da estratégia da segunda versão. Quando todos os pares forem percorridos, o produto matricial estará completo.

Esta versão leva ao melhor desempenho entre todas as versões uma vez que possui a melhor gestão de memória e o menor tempo de execução.

Métricas de Desempenho

As métricas de desempenho que usamos foram: $FLOPS$ e tempo (em segundos). Apontamos ainda os valores "Data Cache Misses" de L1 e L2, abreviado como DCM. Todos os valores exceto os $FLOPS$ vieram da *PAPI*. Os FLOPS foram calculados utilizando a seguinte fórmula:

$$FLOPS = \frac{2n^3}{t}$$

$$n - dimensão\ da\ matriz \qquad t - tempo\ de\ execução$$

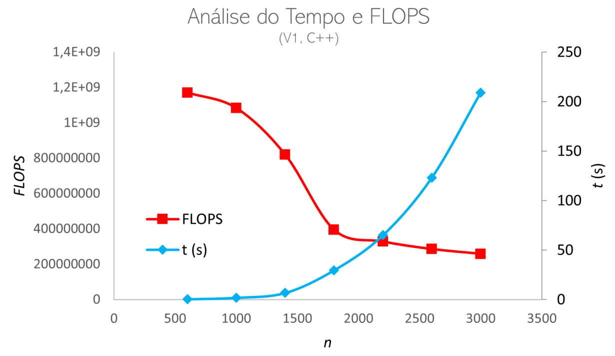
Para os programas em Java, não foi possível utilizar a PAPI, pelo que apenas se obteve o tempo usando a função *System.nanotime()*.

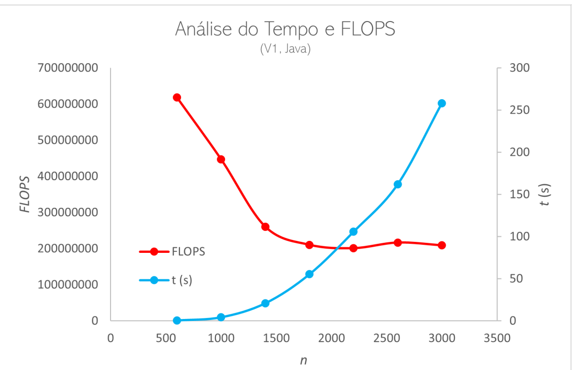
Resultados e Análise

Por uma questão de facilidade, apresentaremos os nossos resultados em gráficos, mas as tabelas com os dados utilizados para a sua elaboração na secção de anexos, incluindo nestas os valores dos contadores L1 e L2 DCM (*Data Cache Misses*) para as versões em C++. Além disto, consideramos importante mencionar os seguintes dados do computador utilizado para as medições: 1.99 GHz de frequência base com cache de tamanho 256 KB.

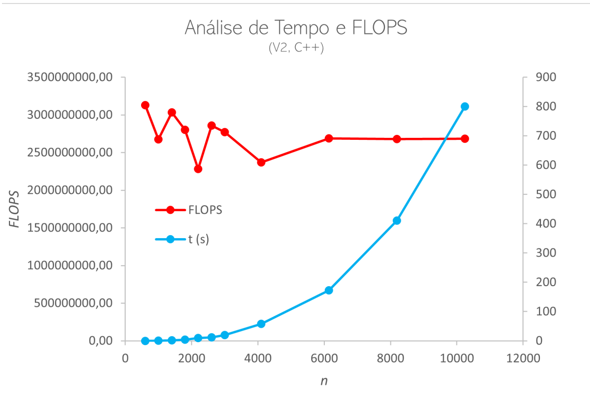
Para começar, os estudos feitos nas duas linguagens propostas, C++ e Java, não apresentam diferenças significativas que nos permitam tirar alguma conclusão fora do esperado.

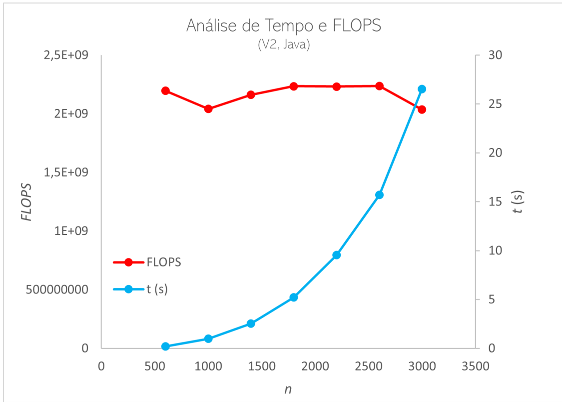
Na primeira versão, em cada ciclo do *loop*, é necessário aceder-se a um valor numa linha da segunda matriz diferente. Torna-se então expectável que esse valor não esteja em cache, pelo que se tem de recorrer a dados que apenas estão na memória (e não na cache). Tal aumenta significativamente o tempo gasto em obtenção de dados, relativamente àquele gasto em cálculo.





Na segunda versão, é possível obtermos o mesmo resultado com uma execução mais rápida, uma vez que, em iterações subsequentes do mesmo ciclo, utilizam-se valores na mesma linha que já estão em cache. Desta forma, reduzir-se-ão os “Data Cache Misses”, pelo que o programa executará mais rápido. Isto é observável para ambas as linguagens.

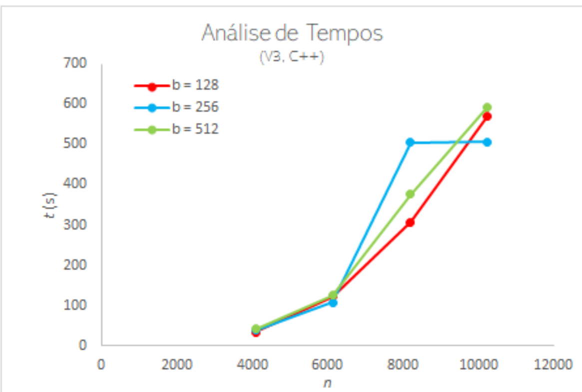


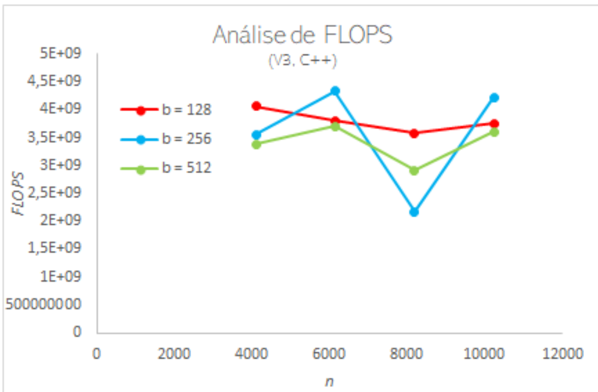


Tanto na primeira versão como na segunda, podemos observar que ambas as linguagens apresentam valores na mesma ordem de grandeza (sendo que C++ apresenta valores ligeiramente mais rápidos e é por isto que usamos esta linguagem para obter valores com números mais elevados de n).

A terceira versão otimiza ainda mais o uso da cache, devido a cada ciclo lidar com um conjunto independente e menor de dados, o que diminui a necessidade de recurso à memória principal e, por sua vez, o tempo gasto nesse acesso.

De facto, comparando as versões 2 e 3, verificamos, como esperado, que a última obteve tempos de execução mais baixos. Verificamos que a causa para tal se deve à melhor utilização da memória cache, uma vez que a métrica L1 DCM para a versão 3 é inferior à da versão 2. Quer isto dizer que, durante a execução da versão 3, não foi tão frequente o acesso à memória, que é temporalmente custoso, visto que os dados estariam já guardados em cache.





No geral, e como esperado, percebemos que da versão 1 à 3 aumenta a eficiência do uso da memória cache, o que possibilita uma execução mais rápida.

Conclusões

Este projeto teve como objetivo investigar os fatores que influenciam o tempo de execução de um programa utilizando-se apenas um único núcleo. Trivialmente, observamos que o tamanho dos dados provoca um aumento do tempo de execução do programa. Contudo, as três versões estudadas confirmaram que programas concebidos de maneiras diferentes, mesmo mantendo-os sequenciais, podem ter desempenhos díspares.

Neste sentido, o acesso à memória é o fator mais importante e aceder à mesma é computacionalmente dispendioso. Assim sendo, uma melhor gestão da memória cache resultará num programa mais rápido, visto que se reduz o custo de uma gestão de memória menos indicada. As versões com multiplicação por linha e bloco maximizam o uso da cache, procurando fazer operações que apenas utilizem dados que já lá se encontrem.

Em suma, compreendemos que é responsabilidade do programador estar ciente de todos os fatores que possam afetar o desempenho do seu programa, sendo essencial ter-se uma visão global da arquitetura de um computador para melhor orientar os esforços realizados tendo em vista os melhores resultados possíveis.

Anexos

Primeira Versão, C++

n	*t* (*s*)	*FLOPS*	L1 DCM	L2 DCM
600	0.369	1170731707.32	244743280	40296664
1000	1.846	1083423618.63	1227008470	270359476
1400	6.692	820083682.01	3526406801	1348693438
1800	29.539	394867801.89	9055691330	7380468162
2200	64.955	327857747.67	17624043570	23439473173
2600	123.104	285547179.62	30875560054	52772858727
3000	208.961	258421427.92	50299757356	97786479155

Segunda Versão, C++

n	*t* (*s*)	*FLOPS*	L1 DCM	L2 DCM
600	0.138	3130434782.61	27111738	56823221
1000	0.748	2673796791.44	125936682	254160753
1400	1.809	3033720287.45	346153172	703088972
1800	4.162	2802498798.65	750904823	1443437372
2200	9.325	2283753351.21	2082359854	2650312591
2600	12.304	2856957087.13	4412616187	4392240165
3000	19.486	2771220363.34	6779253122	6758659504
4096	58.017	2368942783.53	17719158666	17255791763
6144	172.643	2686795688.03	59715347656	58443830589
8192	410.192	2680480428.13	141449802183	138610761163
10240	799.755	2685176895.42	276322229754	277908208574

Terceira Versão, C++

n	Nº de Blocos	*t* (*s*)	FLOPS	L1 DCM	L2 DCM
4096	128	33.774	4069371513.76	9730770336	31561415341
4096	256	38.562	3564103352.32	9093497347	2198313837
4096	512	40.645	3381447987.99	8765591495	18686603989
6144	128	121.752	3809846803.08	32861074340	106719256669
6144	256	106.876	4340136868.60	30650951963	75838366402
6144	512	125.055	3709219687.08	29629467890	62931635706
8192	128	307.004	3581424436.74	77923544491	252001586754
8192	256	504.990	2177293862.80	73133496656	163905749669
8192	512	376.890	2917327675.92	70305408921	145937900358
10240	128	571.010	3760851207.51	152079980076	493380857047
10240	256	507.584	4230794603.45	141890942516	345029577045
10240	512	594.583	3611747473.44	137001544940	288619562855

Primeira Versão, Java

n	*t* (*s*)	*FLOPS*
600	0.6985596	618415379.30
1000	4.4721911	447208081.07
1400	21.0427352	260802597.56
1800	55.3692335	210658505.86
2200	105.8207027	201246064.87
2600	162.0726782	216890350.62
3000	258.0806974	209236880.34

Segunda Versão, Java

n	*t* (*s*)	*FLOPS*
600	0.1966419	2196886828.29
1000	0.979317	2042239642.53
1400	2.5388417	2161615669.07
1800	5.21805	2235317791.13
2200	9.5404427	2232181531.79
2600	15.7125223	2237196506.64
3000	26.5210152	2036121151.20