FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Relatório do 1º Projeto de Computação Paralela e Distribuída

Turma 3LEIC6, Grupo 14
Daniel José de Aguiar Gago (up202108791)
Daniel Moreira Carneiro (up202108832)
José Miguel Sereno Santos (up202108729)

Índice

1. Introdução

O seguinte relatório de Computação Paralela e Distribuída, do curso de Licenciatura em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, tem como objetivo estudar o efeito que a hierarquia da memória tem na performance do processador em 3 métodos de multiplicação de matrizes.
Foi usada a Performance API (PAPI) para recolher algumas métricas de desempenho durante a execução dos programas.
Para a realização da 2ª parte, foi também necessário utilizar OpenMP como também adicionar a sua flag ao método de compilação `-fopenmp`.

2. Descrição do problema e explicação dos algoritmos

2.1. Descrição do problema

O desempenho de um sistema computacional é influenciado por vários fatores. A hierarquia da memória do sistema é um desses fatores cruciais, visto que o acesso à memória de disco é uma operação custosa e que requer a ajuda de memórias cache, menores em termos de espaço de armazenamento mas mais rápidas no seu acesso. De forma a estudar o impacto que o uso eficiente desta hierarquia da memória tem no desempenho de um sistema computacional, usámos 3 algoritmos de multiplicação de matrizes. O segundo algoritmo foi testado tanto numa implementação *single core* como numa *multi core*.

2.2. Explicação dos algoritmos

Os algoritmos foram desenvolvidos em C++ em conjunto com o PAPI (Performance API), e em Java.

2.2.1. Algoritmo inicial de multiplicação de matrizes

O algoritmo inicial de multiplicação de matrizes é uma implementação bastante simples e não muito eficiente para realizar a multiplicação entre 2 matrizes, iterando cada linha da 1ª matriz e cada coluna da 2ª matriz para obter o produto entre cada vetor, acumulando o resultado na matriz final. Revela-se um algoritmo com complexidade O(n^3).

```
    For each i from 0 to matrix_size:
        For each j from 0 to matrix_size:
           temp = 0
           For each k from 0 to matrix_size:
               temp += pha[i* matrix_size +k] * phb[k* matrix_size +j]
           phc[i*matrix_size+j] = temp
```

2.2.2. Multiplicação linha a linha

A segunda implementação, apesar de possuir a mesma complexidade que a primeira, é uma melhoria pois multiplica um elemento da 1ª matriz pela linha correspondente da 2ª matriz. Como a matriz é guardada na memória sequencialmente (1º linha, seguida da 2º linha, etc), haverá maior chance de encontrar o elemento seguinte na cache quando comparado ao primeiro algoritmo, que, no loop interior, itera sobre a 2º matriz por colunas, ao invés de linhas.

```
For each i from 0 to matrix_size:
    For each j from 0 to matrix_size:
        For each k from 0 to matrix_size:
            phc[i* matrix_size + k] = pha[i* matrix_size +j] * phb[j*
matrix_size +k]
```

Para a realização da segunda parte, foi possível otimizar ainda mais este algoritmo ao utilizar multi-core. A primeira alteração pedida foi a introdução da seguinte linha de código:

```
#pragma omp parallel for num_threads(8)
```

na linha anterior à inicialização do primeiro for loop. O objetivo desta implementação é paralelizar o primeiro for loop em 8 threads, fazendo com que cada uma execute uma porção do loop exterior **i**, melhorando o tempo de execução. Os restantes loops **j** e **k** são executados sequencialmente em cada thread.
Na segunda alteração da segunda parte foi nos pedido para colocar as seguintes linhas:

Antes do loop **i**:

```
#pragma omp parallel private(i,j,k) num_threads(8)
```

Antes do loop **k**:

```
#pragma omp for
```

O objetivo da utilização destas linhas de código é criar uma região paralela onde os loops serão todos executados em várias threads. **private(i,j,k)** garante que cada thread tem uma cópia das variáveis o que ajuda a evitar race conditions e overheads. **#pragma omp for**, como é utilizado numa região paralelizada irá distribuir as instruções do loop interior ,**k,** pelas threads.

2.2.3. Algoritmo orientado por blocos

Por fim, o algoritmo orientado por blocos tem como objetivo dividir as operações de multiplicação de matrizes em blocos mais pequenos, realizando a computação nesses blocos diretamente. Semelhantemente ao anterior, isto também aumentará a probabilidade de encontrar elementos seguidos na cache, já que, ainda que a matriz inteira possa não caber na cache, os blocos mais pequenos cabem.

Como os algoritmos anteriores, tem complexidade O(n^3).

```
For each i in the range [0, matrix_size) with step block_size:
    For each j in the range [0, matrix_size) with step block_size:
        For each k in the range [0, matrix_size) with step block_size:
```

```
            For each i2 in the range [i, i + block_size):
                For each j2 in the range [j, j + block_size):
                    For each k2 in the range [k, k + block_size):
                        phc[i2*matrix_size + k2] += pha[i2*matrix_size +
j2] * phb[j2 * matrix_size + k2];
```

3. Métricas de desempenho

Para avaliar a performance dos algoritmos desenvolvidos em C++, além do tempo de execução de cada, usamos três métricas disponibilizadas pelo PAPI, o número de cache misses L1 e L2 (PAPI_L1_DCM e PAPI_L2_DCM) e o número de operações de vírgula flutuante (PAPI_DP_OPS).
Uma operação de falha na cache implica uma sobrecarga no processamento dos nossos algoritmos e, por isso, a recolha dessa informação é importante para avaliarmos a eficiência do código.
Fazendo uma média dos tempos de execução de cada algoritmo (20 tempos medidos para cada métrica pedida), de forma a combater as pequenas diferenças obtidas entre medições, e dividimo-los pelo número de operações de vírgula flutuante, obtendo os FLOPS (operações de vírgula flutuante por segundo).

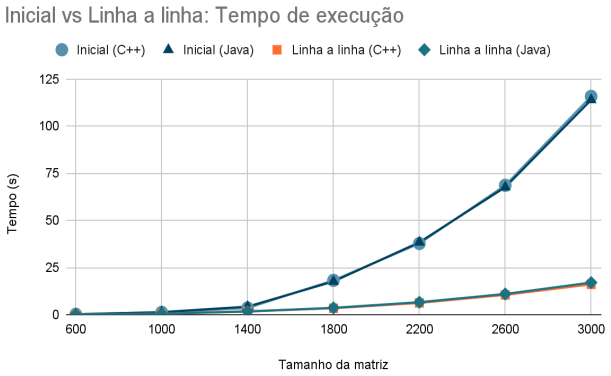
4. Metodologia

Todos os testes foram realizados no mesmo computador que possui uma CPU Intel R CoreTm i7-9700 CPU @ 3.00GHz x 8 e com um sistema operativo Ubuntu 22.04.
Para cada processador, a cache L1 tem 32 KB de tamanho, a cache L2 tem 256 KB e por fim a cache L3 tem 12MB de tamanho.
Os dois primeiros algoritmos (inicial e linha a linha) foram desenvolvidos e testados em C++ e em Java, enquanto o resto foi testado exclusivamente em C++.
Cada ponto dos gráficos apresentados abaixo nos resultados provêm de uma média de 20 execuções, à parte dos casos em que o tempo de execução era demasiado elevado, em que tivemos de reduzir o número de execuções para 10-15 (verificar apêndice).

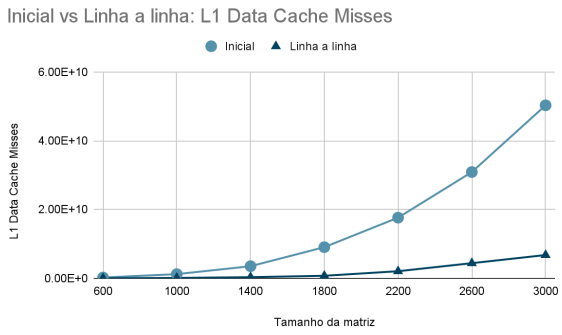
5. Resultados e análise

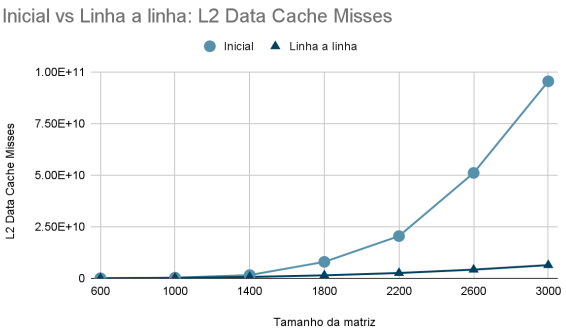
A métrica no eixo das abscissas será sempre a dimensão da respetiva matriz, enquanto que o eixo das ordenadas vai variar entre o tempo de execução, o número de cache misses e o número de operações de vírgula flutuante.

5.1. Comparação entre o algoritmo inicial e o algoritmo linha a linha

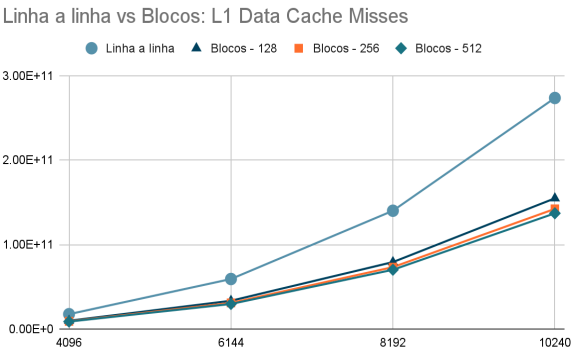


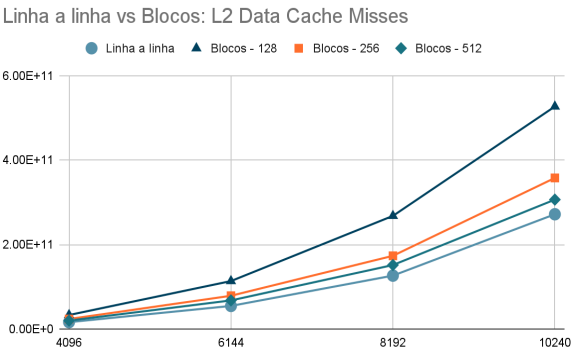
Comparando algoritmos iguais, nas duas linguagens, podemos concluir que o tempo de execução é praticamente o mesmo, já que em ambas as linguagens mantemos a mesma abordagem de resolução e compilação.
Podemos também concluir que o segundo algoritmo (Line Multiplication) é mais eficiente que o primeiro (Simple Multiplication) já que, quer o tempo de execução quer o número de cache misses é menor na segunda implementação, tal como era esperado, tendo em conta a forma como este acede à memória. Como a matriz é guardada na memória sequencialmente (como explicado anteriormente), haverá maior chance de encontrar o elemento seguinte na cache quando comparado ao primeiro algoritmo, que, no loop interior, itera sobre a 2º matriz por colunas, ao invés de linhas. Isto é evidente na medição das cache misses de ambos os algoritmos:

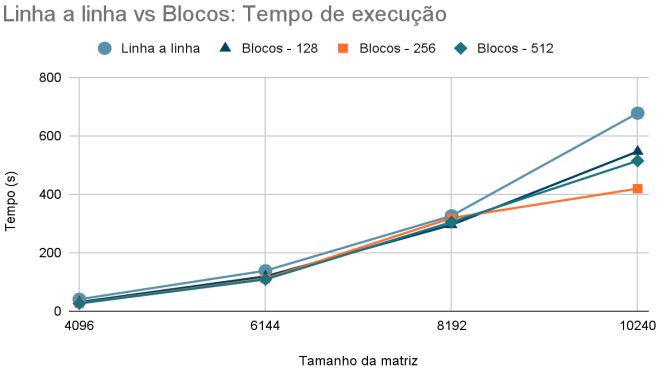




5.2. Comparação do tempo de execução e das data cache misses entre o algoritmo linha a linha e o algoritmo de multiplicação em bloco



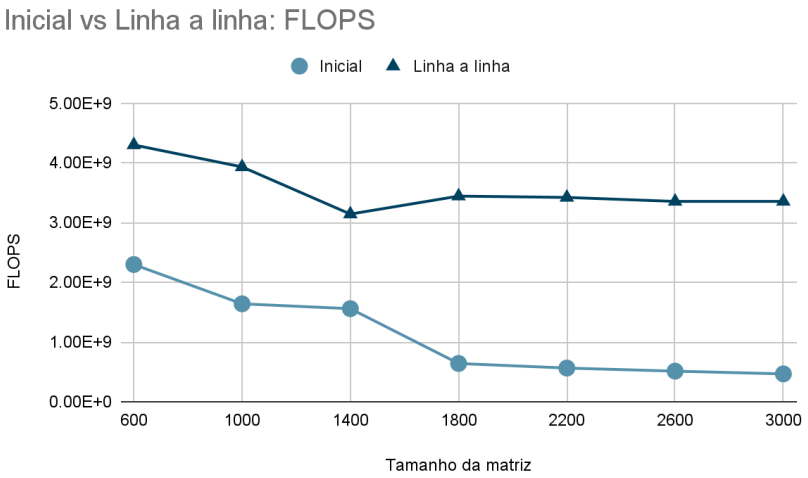




Entre estes algoritmos, conseguimos verificar que existe um maior número de L1 cache misses para o algoritmo linha a linha, mas um número menor de L2 cache misses comparativamente com os algoritmos de multiplicação em bloco. Podemos concluir que, apesar deste último número ser maior para o terceiro algoritmo implementado, as falhas na cache L1 têm mais influência no tempo de execução, como verificamos no terceiro gráfico deste subcapítulo.

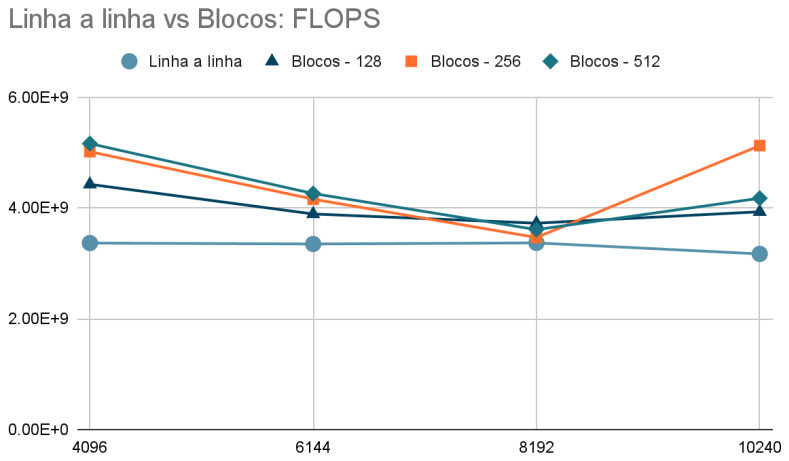
Também podemos concluir que o tamanho dos blocos não influenciou de forma significativa o tempo de execução e a performance das caches. Ao usar blocos de 128x128, apenas a cache L1 (de tamanho 32KB) é toda ocupada, ao contrário da cache L2 (de tamanho 256KB), onde sobra memória a ser mapeada, provocando mais L2 cache misses, afetando o seu tempo de execução.

5.3. Comparação do FLOPS entre o algoritmo simples e o algoritmo de multiplicação em linha



Como esperado, o número de operações de vírgula flutuante (FLOP) manteve-se igual entre algoritmos.
O algoritmo de multiplicação em linha acabou por ter melhores valores de FLOPS comparativamente com o algoritmo simples, correspondendo a menos falhas de cache e consequentemente a uma melhor performance e eficiência do segundo algoritmo.
No algoritmo simples, à medida que aumentamos o tamanho da matriz, o algoritmo torna-se cada vez mais lento, comprovado pela descida de FLOPS.
Já no algoritmo de multiplicação de linha, a variação não é assim tão notória, já que existe uma melhor otimização dos recursos e a minimização das falhas no acesso às caches.

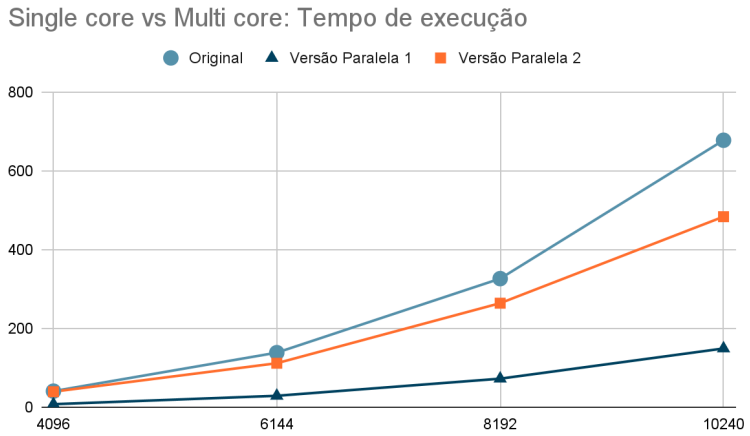
5.4. Comparação do FLOPS entre o algoritmo de multiplicação em linha e o algoritmo de multiplicação em bloco



Como esperado, o número de operações de vírgula flutuante (FLOP) manteve-se igual entre algoritmos.

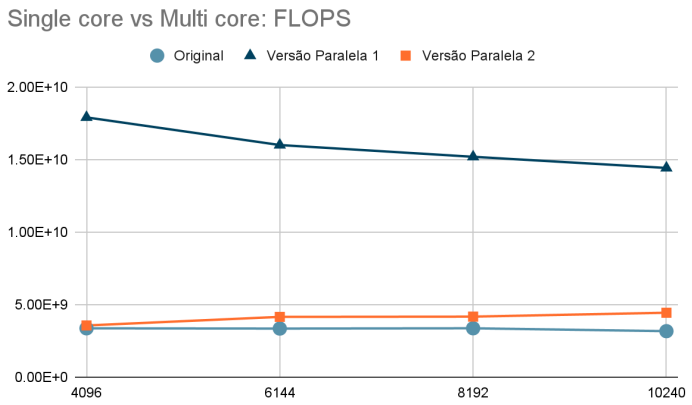
Pelos resultados obtidos, conseguimos concluir que o algoritmo em de multiplicação em bloco obteve FLOPS mais elevados, correspondendo a um menor número de falhas de acesso a cache.
Importante ressalvar que o algoritmo com blocos de 128x128 apresenta, em media, um número menor de FLOPS, já que não usufrui totalmente da cache L2, ao contrário dos outros das outras duas execuções com blocos maiores.

5.5 Comparação do tempo de execução e das data cache misses entre o algoritmo multiplicação em linha e as versões multi-core



A partir dos resultados obtidos, é possível concluir que a paralelização, em ambos os casos, resultou numa melhoria do tempo de execução. No entanto, podemos observar que a versão paralela 1 é mais rápida que a versão paralela 2, devendo-se isto principalmente à diminuição do overhead que acontece na atribuição de iterações às threads e à sincronização das threads. Na 2º versão, este overhead é maior, visto que têm de ser atribuídas tarefas mais frequentemente no loop interior. Podemos então chegar à conclusão que a paralelização do loop exterior é mais benéfica que a paralelização do loop interior em termos de tempo de execução,

5.6 Comparação do FLOPS entre o algoritmo de multiplicação em linha e as versões multi-core



Verifica-se que para as versões paralelizadas possuem um maior número de FLOPS comparado à versão original desse mesmo código. No entanto, devido ao método de paralelização, a versão paralela 1 mostra-se mais eficiente (maior número de flops), tal como esperado, dado os resultados anteriores.

5.7 Speedup e eficiência das versões multi core

Em relação à primeira versão paralela do algoritmo, o speedup observado, comparativamente ao algoritmo da primeira parte do projeto foi, em média, de 4.78 e a sua eficiência foi de 59.79%.
Já em relação à segunda implementação, o speedUp observado foi, em média, de 1.23 e a sua eficiência foi de 15.35%.

6. Conclusões

Com base no que foi apresentado, podemos concluir que o uso eficiente da hierarquia de memória é um aspeto de extrema importância no que diz respeito ao desempenho de um processador, visto que os algoritmos de multiplicação linha a linha e por blocos constituíram um aumento de eficiência significativo quando comparados ao algoritmo inicial, sendo que este último possui um elevado número de cache misses, o que leva a latência originada no acesso à memória de disco.

7. Apêndices

Dados, gráficos e cálculos :
https://docs.google.com/spreadsheets/d/1X9BNJZr2bRrsS3CKDOaIBxExzRX-pDYmV4giuhMNxzI/edit?usp=sharing