Projeto 1 CPD

Turma 3

Manuel Ramos Leite Carvalho Neto up202108744

Matilde Isabel da Silva Simões up202108782

Pedro Vidal Marcelino up202108754

Licenciatura em Engenharia Informática e Computação

Computação Paralela e Distribuída

2023/2024

Índice

Descrição do Problema

O problema abordado neste projeto consiste na avaliação do desempenho de algoritmos de multiplicação de matrizes quadradas. Assim, através da análise de métricas de desempenho como o tempo de execução, as *cache misses*, o *speedup* e a eficiência, conseguimos compreender o impacto da hierarquia da memória e da paralelização do código na *performance* dos programas. Inicialmente, são comparadas implementações sequenciais em diferentes linguagens de programação (C++ e Java), seguidas pela análise de duas implementações paralelizadas. O projeto abrange a variação do tamanho das matrizes e a divisão das mesmas em blocos, permitindo uma compreensão abrangente do comportamento dos algoritmos em diferentes tipos de processamento. O objetivo é identificar e compreender as melhores práticas/estratégias para otimizar o desempenho temporal e espacial.

Explicação dos Algoritmos

Na primeira parte do projeto, não exploramos a computação paralela, mas concentramo-nos apenas na otimização do código sequencial através da manipulação de memória. Foram implementados três algoritmos distintos para avaliar o desempenho de um único processador, diferenciando-se fundamentalmente na forma como se gerem os acessos a memória, tendo conhecimento das estruturas de *hardware* que lhe estão subjacentes. Esses algoritmos são multiplicação simples de matrizes, multiplicação de matrizes por linha e multiplicação de matrizes por blocos.

Ao implementá-los em C++ e Java (para os dois primeiros algoritmos) e apenas em C++ para o terceiro, pretende enfatizar-se a semelhança entre as duas linguagens no que diz respeito à forma direta de aceder à memória, demonstrando que o uso eficiente da mesma pode aprimorar significativamente o desempenho de código sequencial, mesmo sem alterar a complexidade temporal dos algoritmos. O retorno de cada um dos programas inclui o resultado da multiplicação e o tempo de execução, bem como, no caso de C++, os respetivos padrões de acesso à memória, traduzidos através do número de *cache misses*, que permitem comparar as diferentes implementações. Note-se que, nas análises subsequentes da complexidade do código, *n* refere-se ao número de linhas/colunas das matrizes quadradas a tratar.

A implementação do primeiro algoritmo consiste numa abordagem direta para a multiplicação de matrizes. A escolha deste algoritmo justifica-se pela simplicidade e clareza do código, tornando-o mais compreensível e intuitivo de analisar, por seguir o método normal para o produto de matrizes. O algoritmo utiliza três ciclos para percorrer as linhas e as colunas das matrizes, calculando os elementos da matriz resultante e preenchendo-a, através da multiplicação de cada linha da primeira matriz por cada coluna da segunda matriz. O código utilizado encontra-se nos ficheiros *part1.cpp* e *part1.java*, na função *OnMult*, tendo uma complexidade temporal e espacial de $O(n^3)$.

Na implementação do algoritmo de multiplicação de matrizes por linha, foi adotada uma abordagem aprimorada para a multiplicação de matrizes. Neste caso, a otimização ocorre porque se multiplica um elemento da primeira matriz pela linha correspondente da segunda matriz, acumulando em cada célula da matriz resultante os valores dos cálculos intermédios, em vez de a preencher imediatamente com o valor final, como anteriormente. Apesar de se manter a complexidade temporal e espacial de $O(n^3)$, a reorganização dos ciclos – e, consequentemente, da ordem dos acessos a memória – melhora o desempenho do programa ao diminuir as *cache misses*, dado que as linhas das matrizes são colocadas na cache quando um valor delas é lido, devido ao princípio de localidade espacial. O código respetivo apresenta-se no ficheiro *part1.cpp* e *part1.java*, na função *OnMultLine*.

Na abordagem da multiplicação por blocos, as matrizes são subdivididas em blocos menores, que podem ser processados separadamente (multiplicados por linha) e, posteriormente, os seus resultados são somados e colocados nas posições corretas. Assim como no algoritmo anterior, esta estratégia mantém uma complexidade temporal e espacial de $O(n^3)$, mas tem um desempenho ainda melhor. A inovação desta estratégia permite otimizar o acesso à memória, possibilitando que os dados sejam armazenados com regularidade em níveis de memória mais rápidos, como as caches L1 e L2. Ao processar blocos menores de dados de forma independente, o algoritmo minimiza os atrasos associados à procura dos valores em níveis mais lentos da memória, maximizando, assim, os recursos da memória hierárquica. Assim, esta divisão em blocos garante uma utilização mais eficiente da memória cache, reduzindo a necessidade de procurar repetidamente dados na memória principal, o que é naturalmente mais custoso. O código desenvolvido situa-se no ficheiro *part1.cpp*, na função *OnMultBlock*.

Na segunda parte do projeto, já nos concentramos na paralelização do código, tendo em vista uma ainda melhor eficiência temporal. Assim, implementamos duas versões do algoritmo de multiplicação de matrizes por linha, com diferentes níveis de paralelismo. Em ambas as implementações, o paralelismo do código foi obtido através de diretivas *pragma*, da API *OpenMP*, que oferece suporte para multiprocessamento com memória partilhada em C++.

Na primeira versão, recorremos à diretiva *#pragma omp parallel for* para paralelizar o ciclo *for* mais externo, ou seja, aquele que determina o índice da linha da primeira matriz a aceder para realizar a multiplicação, bem como o índice da linha da matriz resultante onde colocar o valor obtido. Assim, combinando as diretivas *#pragma omp parallel* e *#pragma omp for* numa única linha, instruímos o compilador a distribuir as iterações do ciclo da variável *i* pelas *threads* disponíveis, paralelizando o mesmo. Deste modo, cada uma das *k threads* fica responsável por n/k iterações, conseguindo, assim, diminuir o tempo de execução do programa em cerca de *k* vezes. O código está disponível no ficheiro *part2.cpp*, na função *OnMultLine1*.

Na segunda implementação, utilizamos duas diretivas separadamente: *#pragma omp parallel* e *#pragma omp for*. A primeira diretiva foi colocada antes do ciclo *for* que faz variar a variável *i* (que determina a linha a ser acedida, conforme explicado anteriormente), com a cláusula *private (i, k)*, de maneira a paralelizar o bloco de código seguinte, mas tornando as variáveis *i* e *k* (dos dois ciclos mais externos, em que *k* determina a coluna a aceder na primeira matriz e a linha a aceder na segunda matriz) privadas para cada *thread*, isto é, assegurando que cada uma das *threads* criadas só acede a uma cópia de *i* e de *k*, sem modificar a memória partilhada. A segunda diretiva faz com que as iterações do ciclo mais interno sejam divididas pelas *threads* disponíveis. Assim, as diferentes *threads* encarregam-se de diferentes colunas das matrizes, o que contraria a ideia do algoritmo de multiplicação por linhas e obriga a uma sincronização de *threads* adicional, não aproveitando o paralelismo tanto quanto possível. O código encontra-se no ficheiro *part2.cpp*, na função *OnMultLine2*,

Métricas de Desempenho

Para avaliarmos o desempenho dos vários algoritmos implementados em C++, medimos o tempo de execução e utilizamos a ferramenta *Performance API* (PAPI) que permite medir ciclos do CPU, instruções executadas e *cache misses,* entre outras métricas. O recurso a esta API foi essencial para conseguir analisar o tipo de acessos a memória e a forma como são feitos (interpretados através das *cache misses*), comparando-os e extraindo conclusões sobre a sua importância na execução/eficiência dos programas. Para tal, foi necessário compilar o código com a *flag -fopenmp*.

Assim, nos algoritmos em C++ da primeira parte do trabalho, foram registados os tempos de execução, as *caches misses* L1 e L2 e foi calculado o número de FLOPS (*Floating-point Operations Per Second*), usando a fórmula $\frac{2n^3}{t}$ (por se tratarem de operações de multiplicação de duas matrizes quadradas de dimensão *n*), sendo t o tempo de execução. Nos algoritmos em Java, só foi medido o tempo de execução e foi determinado o número de FLOPS, pela mesma fórmula. Na segunda parte do projeto, foi também medido o tempo de execução dos programas para, mais tarde, calcular o *speedup* $\left(\frac{t_{sequencial}}{t_{paralelo}}\right)$, a eficiência $\left(\frac{speedup}{n_{cores}}\right)$ e o número de FLOPS, através do método anterior.

De maneira a sustentar as conclusões posteriores, é necessário compreender a organização hierárquica de memória, nomeadamente das memórias cache. Sabe-se que, quanto mais próxima estiver uma unidade de memória do processador, mais rápido é aceder-lhe, pelo que um acesso à memória cache é mais rápido do que um acesso à memória principal. Deste modo, se o valor pretendido não estiver na cache de nível 1, dá-se uma *cache miss* e ele é procurado na cache de nível 2 e, se ocorrer nova *cache miss*, então só aí é feito um acesso à memória principal.

O código foi compilado com a *flag* de otimização *-O2*, que permite diminuir o tempo de execução do código, através da realização de mais otimizações, mas aumentando, consequentemente, o tempo de compilação. De modo a garantir que o ambiente de execução não interfere nos resultados, foi utilizado o mesmo computador para medir todos os valores, repetindo cada medição cinco vezes e utilizando a média dos resultados obtidos. Salvo exceções pontuais (com valores de incerteza relativa ligeiramente superiores a 5%), a incerteza de medição rondou 1%. O sistema operativo usado foi Linux Ubuntu 22.04, instalado num computador com um processador Intel i7 7700HQ com frequência de 2.8 GHz, 4 núcleos (*cores*) e 8 processadores lógicos. A memória física/RAM disponível é de 15,9 GB e cada núcleo contém uma cache L1 de 64KB e outra (L2) de 256KB. Existe ainda uma cache L3 partilhada entre todos os núcleos, totalizando 6MB de memória cache.

Resultados e Análise

Conforme é possível observar nos dois gráficos da Figura 1 (F. 1) e no primeiro da Figura 2 (F. 2), o tempo de execução varia apenas de acordo com o algoritmo utilizado, mas não com a linguagem de programação, nem com o tamanho dos blocos (de forma relevante). O primeiro gráfico (F. 1) – que apresenta os tempos das implementações dos dois primeiros algoritmos, em C++ e em Java – demonstra dois aspetos dignos de destaque: em primeiro lugar, que o tempo de execução do programa não depende da linguagem utilizada, dado que se verifica uma quase total coincidência entre os valores obtidos para ambas as linguagens; em segundo lugar, que o algoritmo de multiplicação de matrizes por linha é significativamente mais eficiente do que o algoritmo de multiplicação simples de matrizes, apresentando tempos de execução consideravelmente inferiores. O segundo gráfico (F. 1), que compara todas as variações de algoritmos implementados em C++, evidencia um aspeto decorrente do anterior – que não só o algoritmo de multiplicação por linha é mais rápido do que o algoritmo simples, mas também o algoritmo de multiplicação por blocos é mais rápido do que o algoritmo de multiplicação de matrizes por linha. O primeiro gráfico (F. 2) detalha o impacto do tamanho dos blocos no tempo de execução do algoritmo de multiplicação de matrizes por blocos, evidenciando que, apesar de as diferenças não serem muito significativas, quanto maior for o tamanho dos blocos, menor é o tempo de execução.

Por um lado, as semelhanças entre os valores obtidos para C++ e Java devem-se ao facto de ambas as linguagens derivarem de C, tendo, por isso, compiladores/interpretadores idênticos e serem parecidas nos acessos a memória, embora C++ seja ligeiramente mais rápido, por ter um acesso mais direto aos recursos do sistema. Por outro lado, as diferenças entre os tempos justificam-se pelas diferentes abordagens na gestão de memória utilizadas pelos três algoritmos. O primeiro algoritmo segue a teoria mais intuitiva para a multiplicação de matrizes, que não se preocupa em aceder à memória de forma eficiente, limitando-se a percorrer as linhas de uma matriz e as colunas de outra, o que não permite armazenar valores em cache a longo prazo e beneficiar disso mesmo, dado que cada célula da segunda matriz só é acedida de *n* em *n* iterações. O segundo algoritmo, ao aceder sequencialmente às linhas das matrizes e preencher a matriz resultante com o valor acumulado até ao momento, consegue tirar proveito das memórias cache e da sua maior velocidade de acesso, dado que os valores são lidos linha a linha e a cache consegue armazená-los. O terceiro algoritmo, ao seguir a mesma ideia do que o segundo, mas ainda dividir as matrizes em blocos menores, acaba por simplificar as operações sobre as matrizes a multiplicar, subdividindo-as em matrizes menores, sendo, por isso, mais rápido.

O gráfico que traduz o número de FLOPS (em MFLOPS para simplificar a visualização dos dados) evidencia três conclusões. Ao comparar linguagens de programação, constata-se que o número de FLOPS é superior em C++, em comparação com Java, o que traduz alguma maior eficiência/velocidade da primeira em relação à segunda, dado que os acessos à memória em C++ são mais diretos do que em Java. Ao comparar algoritmos, verifica-se que (em consonância com o observado no gráfico dos tempos de execução), o número de FLOPS aumenta do primeiro algoritmo para o segundo e do segundo para o terceiro. No terceiro algoritmo, ao comparar o tamanho dos blocos, observa-se que quanto maior for o tamanho dos blocos, mais operações de vírgula flutuante são feitas por segundo, logo, mais rápida é a execução.

Estas conclusões poderiam ser igualmente deduzidas pela análise da fórmula de cálculo do número de FLOPS $\left(\frac{2n^3}{t}\right)$ e dos gráficos dos tempos de execução, sendo uma consequência direta das variações dos tempos, já explicitadas.

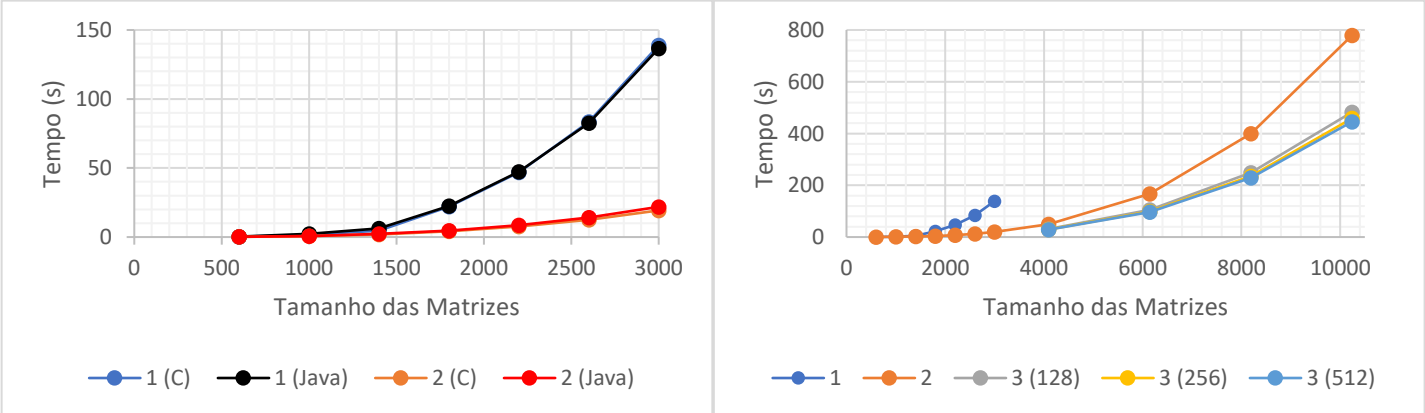

Figura 1 – Gráficos de Tempo de Execução (Parte 1)

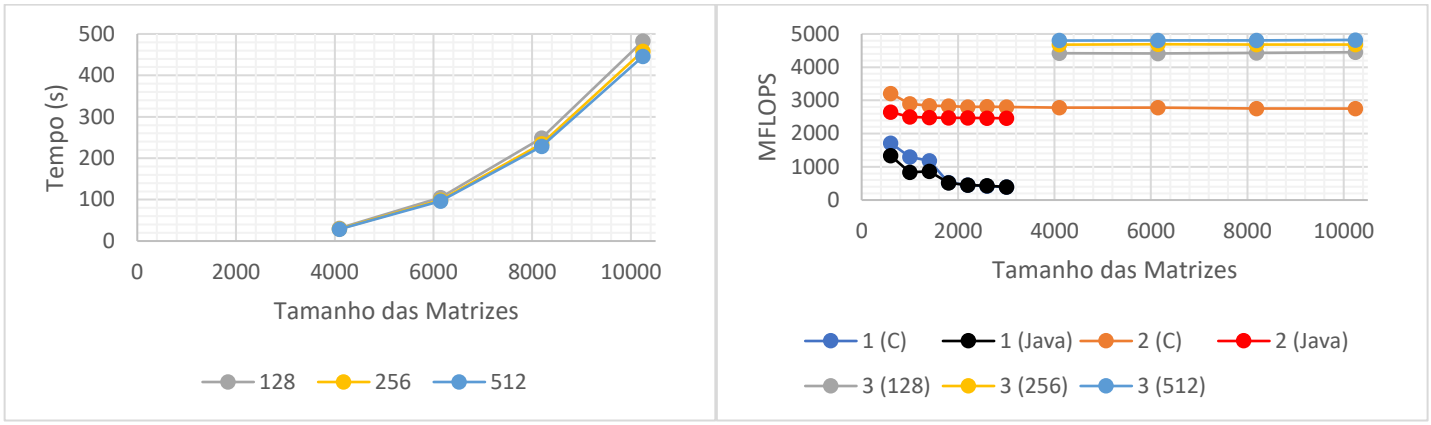

Figura 2 – Gráficos de Tempo de Execução e FLOPS (Parte 1)

Os dois pares de gráficos seguintes expõem as diferenças no número de *cache misses* de cada algoritmo implementado em C++, para as memórias cache nível 1 (L1) e nível 2 (L2), respetivamente. Através da observação e da análise dos quatro gráficos, extraem-se duas principais conclusões. Em primeiro lugar, conclui-se – por comparação do número de *misses* L1 com o número de *misses* L2 – que estes valores estão correlacionados, na medida em que um aumenta quando o outro aumenta e vice-versa (ainda que não aumentem na mesma proporção). Em segundo lugar, observa-se que o número de *cache misses* (L1 e L2) diminui do primeiro algoritmo para o segundo e do segundo para o terceiro, tal como já podia ter sido deduzido anteriormente, bem como diminui com o aumento do tamanho dos blocos.

Estas duas conclusões são sustentadas devido às diferentes formas de gestão de memória efetuadas nos algoritmos implementados. Tal como já foi descrito, o algoritmo de multiplicação de matrizes por linha segue uma estratégia que tira proveito da estrutura hierárquica de memória e dos princípios de localidade temporal/espacial basilares às memórias cache, dado que, ao processar todas as matrizes linha a linha, acede a posições próximas de memória (numa linha das matrizes) em intervalos de tempo muito próximos, reduzindo, assim, o número de *cache misses*. A relação entre o número de *misses* L1 e L2 deve-se simplesmente à organização da memória de forma hierárquica e ao maior tamanho da cache L2 em relação à cache L1, que faz com que uma *miss* na cache de nível 1 leve a uma pesquisa/procura na cache de nível 2 que, por ser maior, tem maior probabilidade de conter o valor procurado, mas, se não o contiver, obriga a aceder à memória principal, como já anteriormente explicado.

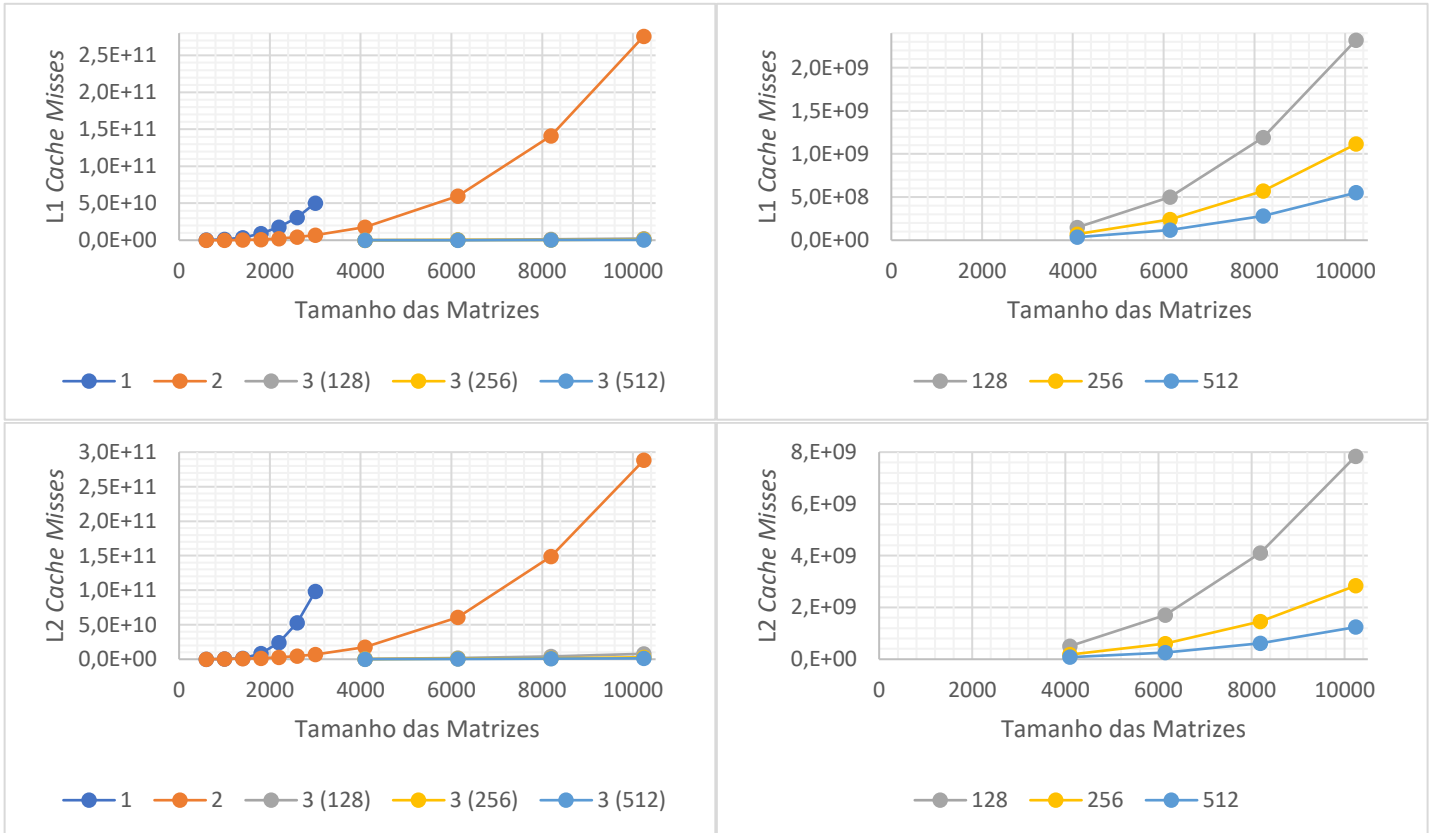

Figura 3 – Gráficos de Cache Misses *L1 e L2 (Parte 1)*

Tal como é visível no gráfico dos tempos de execução para as implementações paralelas, a primeira implementação (legendada como "A") é mais rápida do que a segunda (legendada como "B"), sendo ambas mais rápidas do que a implementação sequencial de multiplicação de matrizes por linha. A diferença nas velocidades de execução entre as duas implementações é explicada pelo facto de a primeira aproveitar melhor as potencialidades do paralelismo do que a segunda, dado que a primeira paraleliza o ciclo externo e divide todas as suas iterações pelas *threads* disponíveis, enquanto a segunda, apesar de paralelizar o bloco de código correspondente aos dois ciclos mais externos, apenas divide as iterações do ciclo mais interno pelas *threads*, o que se reflete num aproveitamento parcial das potencialidades do paralelismo. Apesar desta diferença entre elas, ambas as versões paralelas demoram menos tempo a executar do que a implementação sequencial, pois fazem uso do paralelismo do processador, conseguindo, assim, reduzir os tempos de execução, ao fazer mais operações em simultâneo.

Conclusões semelhantes podem ser inferidas a partir do gráfico de MFLOPS. Comparando as três implementações, resulta claramente que a primeira versão paralela realiza um número amplamente superior de operações de vírgula flutuante por segundo e que a versão sequencial é aquela que efetua o menor número de FLOPS, estando a segunda versão paralela entre elas. Numa análise análoga à efetuada para os códigos sequenciais, estes valores decorrem diretamente da fórmula de cálculo do número de FLOPS, isto é, o paralelismo permite efetuar mais cálculos em simultâneo, o que reduz o tempo de execução e, consequentemente, aumenta o número de FLOPS (mantendo o número de operações, mas diminuindo o tempo em que são efetuadas). Para o caso das versões paralelas, aquela que paraleliza e divide as iterações do ciclo mais externo, ao repartir mais trabalho entre as *threads* e manter as variáveis partilhadas de forma eficiente, sem necessidade de as privatizar, consegue que sejam realizadas mais multiplicações por segundo em comparação com o código que cria as *threads* no bloco de código externo, mas apenas divide as iterações do ciclo interno.

Assim, a análise conjunta deste par de gráficos permite confirmar o fundamento teórico assente em cada uma das implementações paralelas. A primeira implementação, ao paralelizar o bloco de código mais externo e dividir as suas iterações pelas *threads* criadas, consegue tirar maior proveito do paralelismo e, com isso, aumentar o número de operações por segundo, diminuindo o tempo de execução. A segunda implementação, ao criar *threads* no ciclo externo, mas apenas dividir as iterações do ciclo interno, desperdiça algum paralelismo potencial, sendo, por isso, mais lenta.

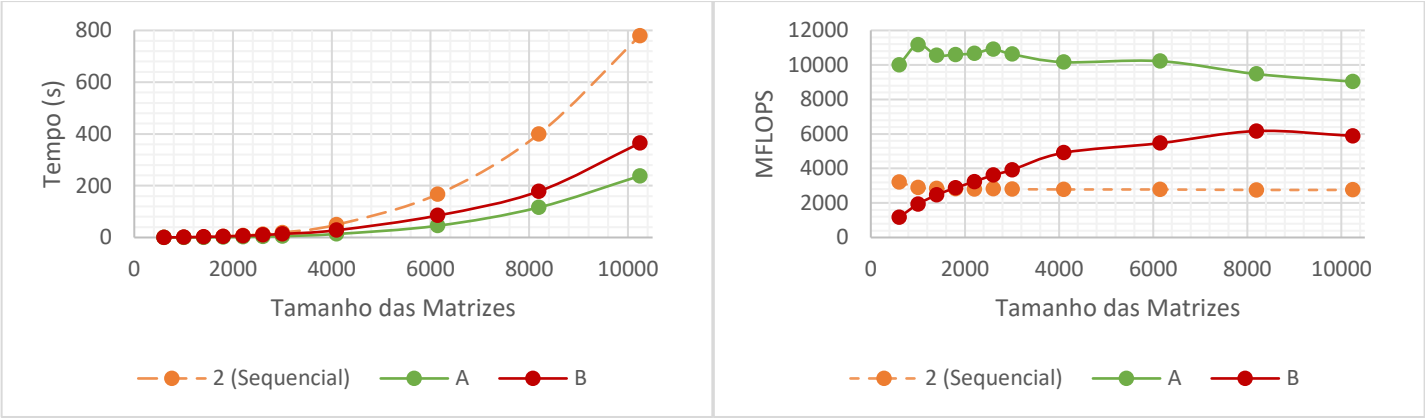

Figura 4 – Gráficos de Tempo de Execução e FLOPS (Parte 2)

No que concerne ao número de *cache misses* (nível 1 ou nível 2), apresenta-se, por um lado, uma elevada coincidência entre o número de *cache misses* das duas implementações paralelas e, por outro lado, uma redução significativa deste número nas versões paralelas, em relação ao código sequencial. Ambas estas conclusões devem-se ao facto de, na máquina utilizada, as caches L1 e L2 serem únicas para cada *core* do processador (enquanto a memória principal é, evidentemente, partilhada), o que faz com que o código paralelo, por tirar proveito de todos os quatro *cores* disponíveis no computador, tire também proveito das quatro memórias cache (de cada nível) para reduzir o número de *cache misses*, dado que cada um dos *cores* armazena nas respetivas caches os valores de que mais provavelmente necessitará, de acordo com os princípios de localidade temporal e espacial já referidos. Ora, ao haver, efetivamente, mais memória cache a ser utilizada (por serem utilizados mais *cores* em simultâneo), mantendo os padrões de acesso a memória, é evidente que o número de *cache misses* diminui.

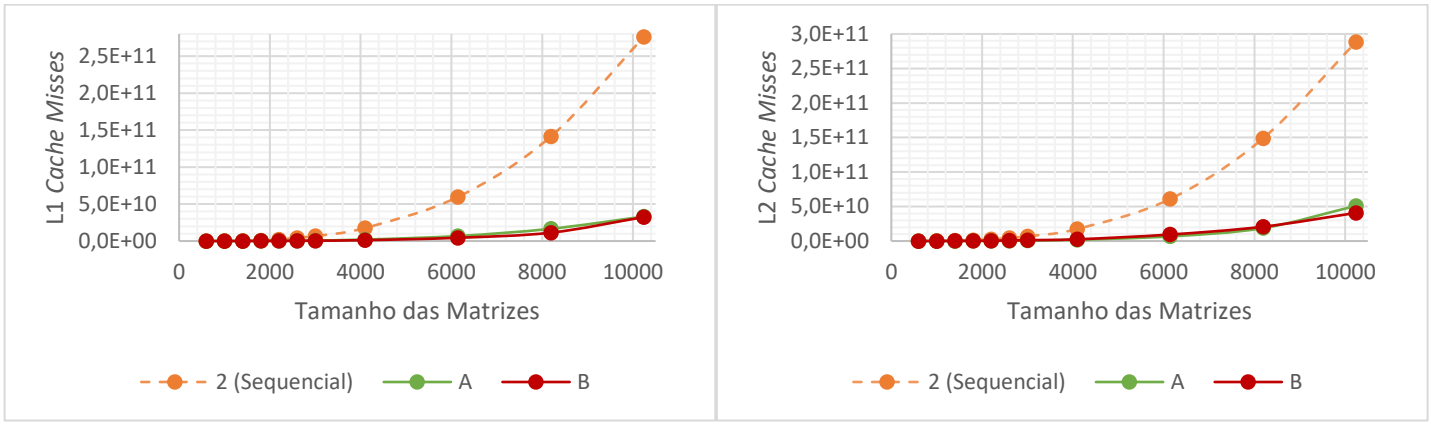

Figura 5 – Gráficos de Cache Misses *L1 e L2 (Parte 2)*

Por último, resta tecer considerações sobre o *speedup* e a eficiência dos códigos paralelos. Interpretando os valores calculados para estes dois parâmetros, conclui-se, que, no caso da primeira implementação (legendada como "A"), o *speedup* é ligeiramente inferior ao número de *cores* (pelo que a eficiência está mais próxima dos 100%) e, no caso da segunda implementação (legendada como "B"), o *speedup* e a eficiência aumentam com o aumento do tamanho das matrizes, tomando valores de quase 2,5 e 60%, respetivamente. Ambas estas observações resultam diretamente das fórmulas de cálculo utilizadas, traduzindo, por isso, os ganhos das implementações paralelas relativamente às implementações sequenciais. De facto, o *speedup* da versão paralela mais otimizada aproxima-se do número de *cores* (4) do computador, o que se traduz numa eficiência – ou seja, aproveitamento dos *cores* e do seu potencial de paralelização – bastante elevada, enquanto a implementação menos ótima tem um *speedup* que tende para aproximadamente metade do número de *cores* (portanto, 2), o que se reflete numa eficiência à volta dos 50%.

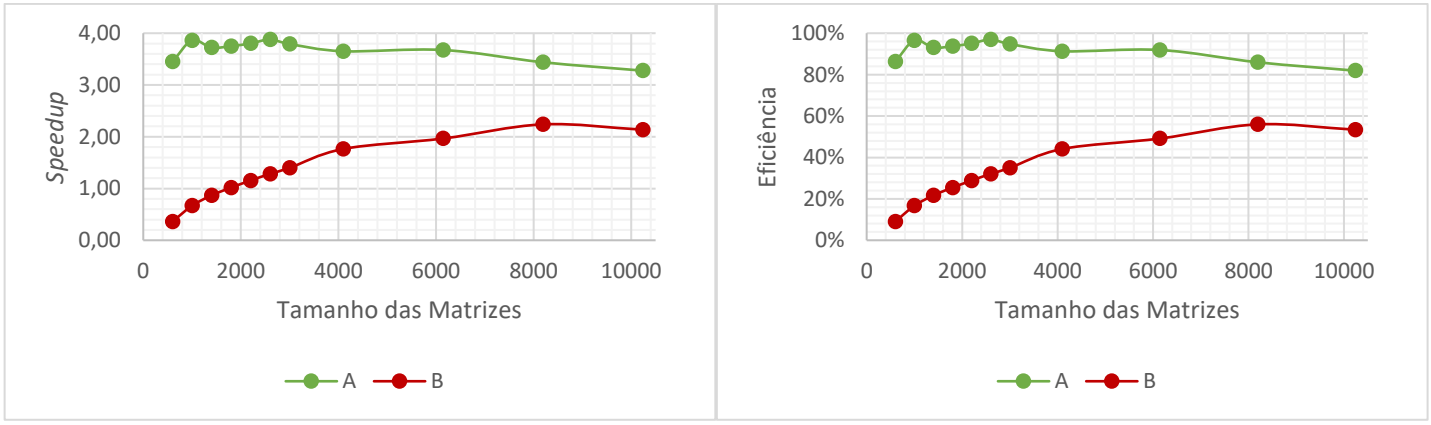

Figura 6 – Gráficos de Speedup *e Eficiência (Parte 2)*

Conclusões

Em suma, existem três conclusões principais que conseguimos retirar através da realização deste trabalho prático. Em primeiro lugar, principalmente pela parte 1, compreendemos a importância de efetuar uma gestão correta e eficiente dos acessos a memória aquando da elaboração de código, sendo isto preponderante para conseguir um bom desempenho, mesmo sem alterar a complexidade algorítmica do programa. Em segundo lugar, com a parte 2, aprendemos a escrever código paralelo e como funcionam as diferentes diretivas *pragma* da biblioteca *OpenMP*, conseguindo, através delas, construir programas que correm paralelamente e, por isso, são mais eficientes, sem necessitar de alterações significativas no código. Por último, devido a ambas as partes do projeto, percebemos quão relevante é escolher adequadamente as métricas para analisar diferentes implementações de forma correta, permitindo a extração de conclusões válidas e bem fundamentadas. Deste modo, consideramos que a elaboração deste projeto foi muito bem sucedida, na medida em que nos permitiu assimilar os conhecimentos essenciais da Unidade Curricular.