Faculdade de Engenharia da Universidade do Porto

Avaliação de desempenho de implementações *single core* e *multi-core*

Computação Paralela e Distribuída - Projeto 1

Licenciatura em Engenharia Informática e Computação

Turma 12 - Grupo 15

João Filipe Oliveira Ramos (up202108743@edu.fe.up.pt)

Marco André Pereira da Costa (up202108821@edu.fe.up.pt)

Tiago Filipe Castro Viana (up201807126@edu.fe.up.pt)

Índice

Descrição do Problema

Este projeto, realizado no âmbito da unidade curricular de Computação Paralela e Distribuída, tem como objetivo o estudo do efeito da hierarquia da memória do sistema no desempenho do processador durante o acesso a grandes quantidades de dados.

Algoritmos

Como caso de estudo para este projeto, foi utilizado o processo de multiplicação de matrizes quadradas, estando os elementos de cada matriz alocados em memória de forma contígua. Foram analisados diversos algoritmos que implementam a referida operação, mas que diferem no acesso aos elementos da matriz e no processo de produção do resultado, a fim de analisar a sua influência nas diferentes métricas de medição de desempenho.

No que diz respeito a programas sequenciais (single core), foram implementados três algoritmos que realizam a operação pretendida com diferentes níveis de consideração pela alocação de memória para este processo. Estes algoritmos correspondem à multiplicação simples de matrizes, à multiplicação de matrizes por linha e à multiplicação de matrizes por bloco. Todos estes algoritmos apresentam uma complexidade temporal de $O(n^3)$ e foram implementados em *C++*. Para efeitos de avaliação de desempenho, os dois primeiros algoritmos também foram implementados em Java.

Multiplicação simples de matrizes

A implementação para o algoritmo de multiplicação simples de matrizes em *C++* foi fornecida com o enunciado do projeto, e corresponde ao método algébrico de multiplicação de matrizes, em que cada entrada da matriz resultante é computada diretamente através do produto da linha correspondente da primeira matriz com a coluna correspondente da segunda.

Multiplicação de matrizes por linha

O algoritmo de multiplicação de matrizes por linha baseia-se na incrementação iterativa das entradas na matriz resultante, inicializadas com valor zero, com os respetivos valores da multiplicação de cada elemento da primeira matriz pela linha correspondente da segunda.

Multiplicação de matrizes por bloco

Por fim, o algoritmo de multiplicação de matrizes por bloco consiste na divisão das matrizes que se pretende multiplicar em matrizes mais pequenas, denominadas de blocos, em que o produto entre blocos é realizado através da multiplicação por linha de matrizes. Cada bloco da matriz resultante é calculado seguindo a multiplicação simples de matrizes entre a respetiva fila de blocos da primeira matriz e a respetiva coluna de blocos da segunda.

Implementação *multi-core*

Além da implementação dos algoritmos sequenciais mencionados e da avaliação do seu desempenho, foram implementadas duas versões em **paralelo** do algoritmo de multiplicação de matrizes por linha, usufruindo da capacidade de *multi-core* do processador. Estas duas versões foram implementadas e avaliadas exclusivamente em *C++*.

A primeira versão deste algoritmo em paralelo apresenta a seguinte estrutura:

```
#pragma omp parallel for
for (int i=0; i<n; i++)
      for (int k=0; k<n; k++)
            for (int j=0; j<n; j++)
```

Nesta implementação, a derivativa ***#pragma omp parallel for*** é utilizada para criar uma região de paralelização e especificar que as iterações do *loop* devem ser distribuídas pelas diferentes *threads* em execução, aumentando significativamente o desempenho do algoritmo.

Por sua vez, a segunda versão do mesmo algoritmo segue uma estrutura que especifica a distribuição das diferentes iterações pelas *threads* em execução no ciclo mais interior:

```
#pragma omp parallel
for (int i=0; i<n; i++)
      for (int k=0; k<n; k++)
            #pragma omp for
            for (int j=0; j<n; j++)
```

Desta forma, a diretiva ***#pragma omp parallel*** cria as múltiplas *threads* para esta região de paralelização e é a diretiva ***#pragma omp for*** que especifica a atribuição de diferentes iterações do ciclo mais interior às *threads* em execução.

Métricas de Desempenho

Nos testes realizados em *C++* foram recolhidos o **tempo de execução** do produto de matrizes (não incluindo a alocação de memória para variáveis e a preparação de cada matriz a multiplicar) e o número de ***cache misses*** às caches **L1** e **L2**, captado através da biblioteca **Performance API** (**PAPI**). Uma ***cache miss*** gera uma sobrecarga adicional e, por isso, a análise desta métrica é extremamente relevante para avaliar a qualidade da implementação. O programa em *C++* foi sempre compilado com a *flag* de otimização **-O2** para que a performance do código gerado fosse mais elevada.

Nos testes realizados em *Java* foi apenas recolhido o **tempo de execução** do produto de matrizes, seguindo as mesmas condições.

Para a comparação dos algoritmos paralelos, foram também calculadas as seguintes métricas de desempenho:

$$Speedup = T_{seq} / T_{par}$$

T_{seq} é *o tempo necessário para realizar a tarefa numa implementação sequencial*

T_{par} é *o tempo necessário para realizar a tarefa numa implementação paralela*

$$Efficiency = Speedup / N_{cores}$$

N_{cores} é *o número de cores do processador utilizado*

$$MFlops = 2n^3 / (T * 1000000)$$

n é *a dimensão de uma linha/coluna da matriz quadrada*

Note-se que $2n^3$ é o número de operações realizadas na multiplicação de matrizes. Para o produto de duas matrizes quadradas com linhas de tamanho n são realizadas n^3 multiplicações e $n^2 * (n - 1)$ somas para calcular a matriz resultante. Os testes foram realizados nas máquinas das salas de aula da FEUP, cujo o processador é um **Intel® Core™ i7-9700**, pelo que o N_{cores} corresponde a 8. Por fim, foram utilizados os resultados da implementação do algoritmo sequencial de multiplicação de matrizes por linha como valores para o T_{seq}.

O ***speedup*** permite-nos comparar a versão paralela do algoritmo com a versão sequencial. Já a **eficiência** permite-nos analisar o aumento do desempenho real em relação ao máximo teórico da versão paralela do algoritmo. Finalmente, os **MFlops** permitem-nos ter uma métrica universal de comparação entre todas as versões dos algoritmos, visto estarmos a calcular a quantidade de operações realizadas.

Análise de Resultados

Os resultados obtidos correspondem à média calculada a partir de uma amostra de três execuções de cada teste. Em cada gráfico, a **dimensão** refere-se ao número de linhas/colunas das matrizes quadradas, e o eixo vertical apresenta o resultado da medição efetuada. O ***speedup***, a **eficiência** e os **MFlops** foram calculados a partir das fórmulas apresentadas na secção anterior.

Comparação do desempenho entre a multiplicação simples e a multiplicação por linha

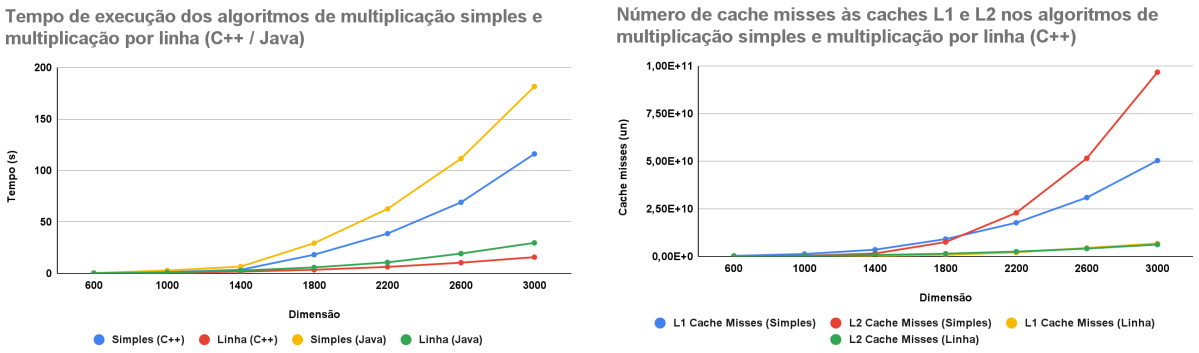

Figura 1 - Comparação dos tempos de execução e do número de *cache misses* em L1 e L2 dos algoritmos de multiplicação simples e por linha

Comparando os tempos de execução dos algoritmos de multiplicação de matrizes simples e por linha, podemos observar uma diferença que se mostra cada vez mais significativa para matrizes com dimensão superior a 1400 linhas/colunas. O segundo algoritmo revela um desempenho mais favorável quer em *C++* quer em *Java*, o que resulta do número mais reduzido de *cache misses* no acesso às *caches* L1 e L2 neste algoritmo. Sendo que o acesso às entradas das matrizes é realizado linha por linha e que os elementos de uma linha estão alocados sequencialmente em memória, os valores necessários para o

cálculo do produto estão mais frequentemente disponíveis em níveis mais baixos de memória. Isto deve-se ao facto de cada *cache miss* ocorrido no acesso a um elemento em memória carregar para a respetiva *cache* um bloco de memória que contém o elemento em questão e elementos contíguos. Estes elementos estarão disponívies em níveis de memória mais próximos do processador para os cálculos posteriores.

Também é possível constatar que o tempo de execução de ambos o algoritmos é ligeiramente inferior em *C++*, em relação a *Java*. Este efeito deve-se ao facto de *C++* ser uma linguagem compilada, enquanto que *Java* introduz alguma sobrecarga associada à *Java Virtual Machine* que carrega e executa o *bytecode* produzido.

Comparação do desempenho entre a multiplicação por linha e a multiplicação por bloco

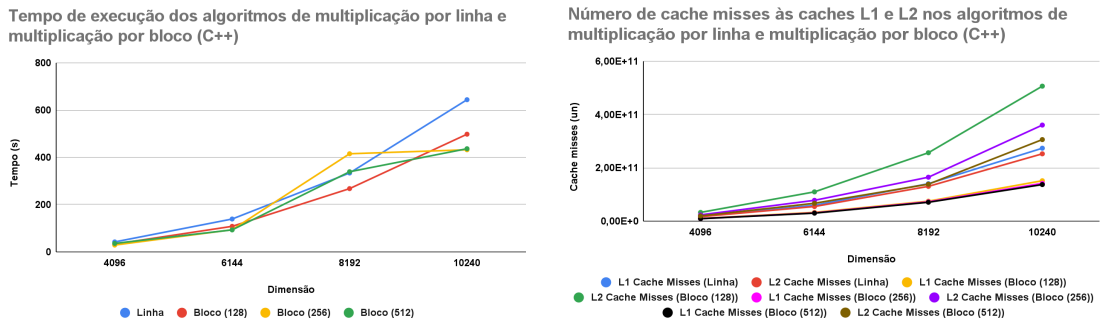

Figura 2 - Comparação dos tempos de execução e do número de *cache misses* em L1 e L2 dos algoritmos de multiplicação de matrizes por linha e por bloco

Os tempos de execução obtidos para estes algoritmos permitem-nos observar que a multiplicação de matrizes por bloco apresenta uma vantagem clara sobre a multiplicação de matrizes por linha, com a única exceção dos dois casos em que foram utilizados blocos de 256x256 e blocos de 512x512 para matrizes de 8192x8192.

A melhoria do tempo de execução deve-se ao número inferior de *cache misses* em L1. Isto sucede-se, pois em multiplicação por linha, quando aplicada a matrizes de dimensões significativamente superiores, os elementos iniciais da segunda matriz, que são os primeiros a serem carregados para L1 e L2, acabam por ser substituídos antes de percorrer a segunda matriz por completo. Isto resulta na impossibilidade de manter os elementos em memória persistentemente para que estes venham a ser reutilizados.

A multiplicação por bloco resolve este problema subdividindo as matrizes em blocos de menores dimensões, o que permite que uma porção maior dos elementos que serão reutilizados durante a multiplicação persista em L1 e L2.

Comparação do desempenho entre os algoritmos de multiplicação em paralelo

Analisando as diferenças entre os tempos de execução dos algoritmos em paralelo e o algoritmo de multiplicação por linha, é notório que as implementações em paralelo possuem uma vantagem clara em relação à implementação sequencial. Entre as versões *multi-core*, a primeira versão apresenta uma prestação muito superior no que diz respeito ao tempo de execução. Por sua vez, a segunda versão

apresenta menos *cache misses* nas *caches* L1 e L2, pelo que existe outro fator que torna esta implementação menos eficiente. Comparando as duas versões em paralelo apresentadas, era previsível que a segunda tivesse um desempenho inferior, apesar do número total de iterações distribuídas pelas diferentes *threads* ser idêntico em ambas. A segunda versão implica que se realize uma distribuição de iterações por cada vez que o ciclo intermédio é executado e, por consequente, que o programa aguarde que todas as *threads* acabem de executar antes de proceder para a próxima iteração do ciclo intermédio. Isto resulta numa sobrecarga substancial que pode ser evitada se tanto o processo de distribuição das iterações como o de sincronização de *threads* forem feitos de forma singular. A primeira versão evita este problema, uma vez que distribui as iterações do ciclo mais exterior por cada *thread* apenas uma vez, e, subsquentemente, todas as iterações nelas contidas. Da mesma forma, também aguarda uma única vez pela sincronização das diferentes *threads*, antes de proceder com a execução do resto do programa, reduzindo a sobrecarga. Uma abordagem que permitiria melhorar a eficiência da segunda implementação seria desligar a sincronização no fim de cada bloco ***#pragma omp for*** através da clásula ***nowait***. Esta solução não apresentaria um desempenho tão favorável como a primeira versão do algoritmo em paralelo, mas teria uma melhoria significativa.

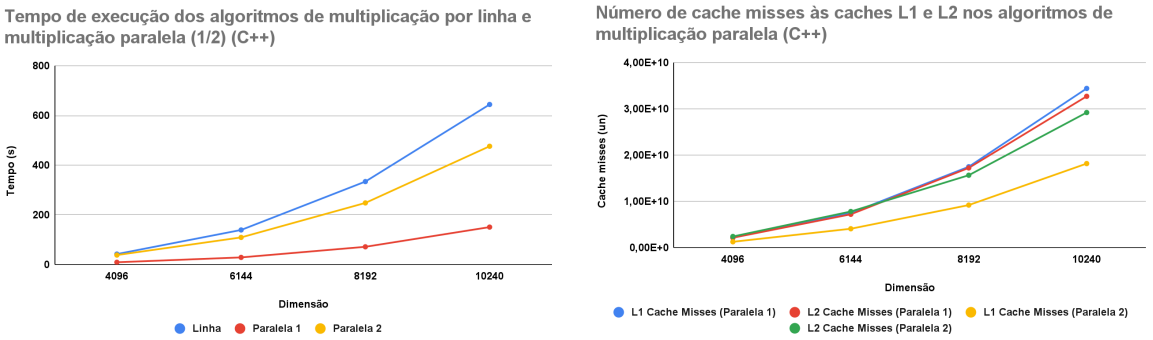

Figura 3 - Comparação dos tempos de execução e do número de *cache misses em* L1 e L2 dos algoritmos de multiplicação de matrizes em paralelo

Análise do *speedup*, da eficiência e dos MFlops dos algoritmos de multiplicação de matrizes em paralelo

Comparando o ***speedup*** calculado para os algoritmos, constata-se que a primeira implementação revela um aumento da velocidade de realização da tarefa muito superior à segunda implementação, o que se traduz numa **eficiência** no uso dos *cores* do processador mais elevada. Quanto aos **MFlops**, apesar da segunda implementação ter um número inferior de *cache misses* às *caches* L1 e L2, a sobrecarga criada na sinconização das *threads* em cada iteração intermédia permite, em comparação, que a primeira implementação execute um número muito superior de operações por segundo.

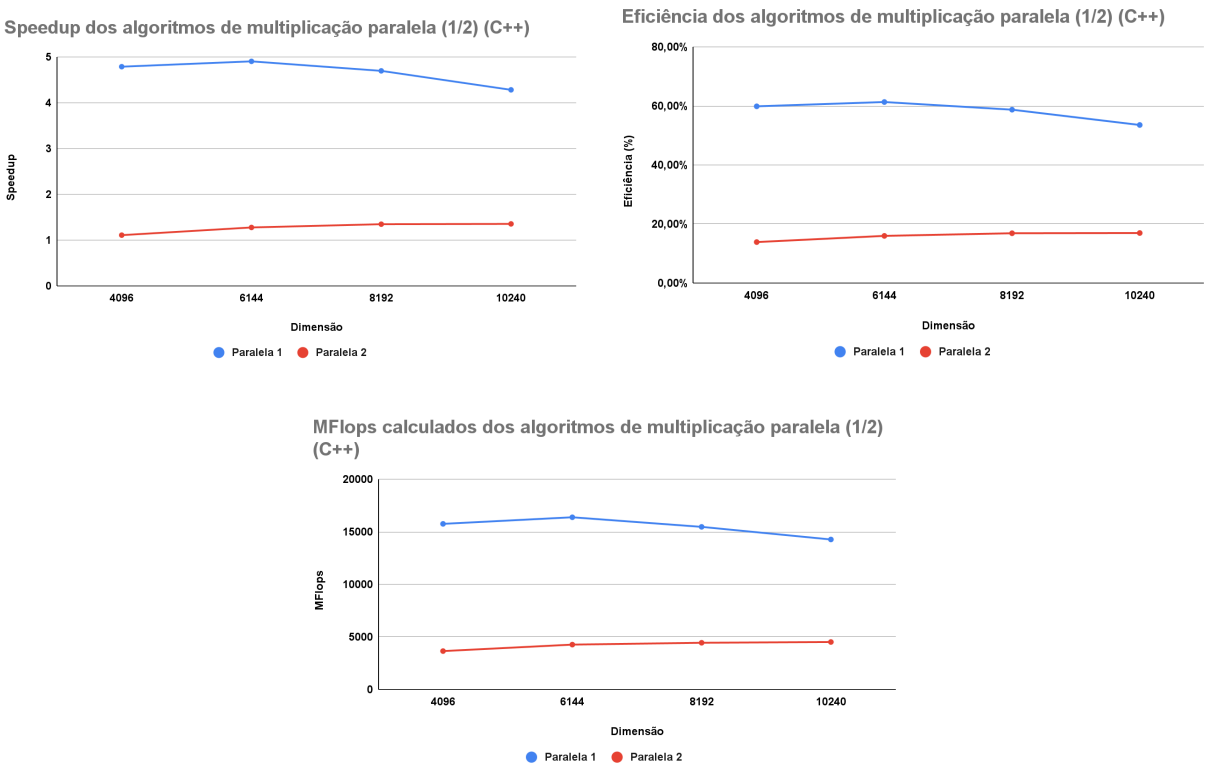

Figura 4 - Comparação do *speedup*, da eficiência e dos MFlops dos algoritmos de multiplicação de matrizes em paralelo

Conclusões

Ao longo deste projeto, todos os objetivos propostos foram alcançados. O desenvolvimento e a análise do desempenho dos diferentes algoritmos permitiu-nos aprofundar o nosso conhecimento na gestão e manipulação de memória, e entender a sua importância na implementação de um programa sequencial. Além disto, este estudo também nos proporcionou um melhor entendimento de como um programa sequencial pode ser paralelizado de forma eficiente, de como se deve analisar o desempenho de uma implementação em pararelo, e de como é essencial planear a distribuição de tarefas pelas diferentes *threads* de forma a que seja retirado o máximo proveito da capacidade do processador.

Referências Bibliográficas

- Enunciado do projeto na página Moodle da unidade curricular. Disponível em https://moodle2324.up.pt/pluginfile.php/195675/mod_resource/content/10/CPD_ex1.pdf. Acedido a 14 de março de 2024.
- Intel. Intel® Core™ i7-9700 Processor Specifications. Disponível em https://www.intel.com/content/www/us/en/products/sku/191792/intel-core-i79700-processor-12m-cache-up-to-4-70-ghz/specifications.html. Acedido a 14 de março de 2024.
- Chapman, B., Jost, G., & Van Der Pas, R. (2008). Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press.
- Notas de aula do curso da Universidade de Aalto CS-E4580 Programação de Computadores Paralelos. https://ppc.cs.aalto.fi/. Acedido a 13 de março de 2024.