Computação Paralela e Distribuída

Primeiro Projeto

3LEIC03 - Grupo 11:

Beatriz Bernardo - up202206097
Beatriz Sonnemberg - up202206098
Daniel Silva - up201909935

Licenciatura em Engenharia Informática e Computação

2024/2025

Índice

1. Introdução

1.1. Descrição do Problema

Este projeto foi desenvolvido no âmbito da unidade curricular de Computação Paralela e Distribuída (CPD), com o objetivo de analisar o efeito da hierarquia de memória no desempenho do processador ao lidar com operações de grande volume de dados, tendo como caso de estudo o produto de duas matrizes. Para este fim, foram implementadas e comparadas diferentes abordagens de multiplicação de matrizes, tanto em computação *single-core* como *multi-core*. Utilizou-se a *Performance API* (PAPI) para recolher indicadores de desempenho relevantes durante a execução do programa.

Embora o número de operações necessárias seja o mesmo para os diferentes algoritmos analisados, a forma como cada um acede à memória pode ter um impacto significativo no desempenho. Isto deve-se ao *layout* dos dados e à hierarquia de cache, tornando alguns algoritmos mais eficientes no aproveitamento da cache do que outros.

1.2. Explicação dos Algoritmos

1.2.1 Implementações single core

1.2.1.1. “Simple Matrix Multiplication”

Neste algoritmo básico a função ***OnMult()*** multiplica cada linha da primeira matriz por todas as colunas da segunda matriz para calcular os elementos da matriz resultante. Apesar da sua simplicidade (3 ciclos for aninhados), este algoritmo não é eficiente em termos de cache, pois as matrizes são armazenadas em memória linha a linha. Como consequência, ao aceder às colunas da segunda matriz, são feitos saltos entre posições de memória distantes, resultando num elevado número de *cache misses* e tendo um impacto negativo no seu desempenho.

```
for (i = 0; i < m_ar; i++)
   {
       for (j = 0; j < m_br; j++)
       {
           temp = 0;
           for (k = 0; k < m_ar; k++)
           {
               temp += pha[i * m_ar + k] * phb[k * m_br + j];
           }
           phc[i * m_ar + j] = temp;
       }
   }
```

1.2.1.2. “Line Matrix Multiplication”

Neste algoritmo a função ***OnMultLine()*** multiplica cada elemento de uma linha da primeira matriz por toda a linha correspondente da segunda matriz, acumulando os valores na linha correspondente da matriz resultante (inicializada com zeros). Desta forma, os acessos à memória ocorrem de forma sequencial, aproveitando a localidade espacial da cache, pois a probabilidade dos próximos elementos de uma linha já estarem na cache é alta. Isso ocorre porque, quando um valor na memória principal é acedido, outros valores adjacentes também são carregados na cache. Assim,

é possível tirar melhor proveito da hierarquia de cache, e como consequência, este algoritmo é significativamente mais eficiente do que o anterior.

```
    for (i = 0; i < m_ar; i++)
  {
      for (k = 0; k < m_ar; k++)
      {
          for (j = 0; j < m_br; j++)
          {
              phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
          }
      }
  }
```

1.2.1.3. “Block Matrix Multiplication”

Neste algoritmo a função ***OnMultBlock()*** divide as matrizes em blocos menores e, em seguida, multiplica esses blocos tal como no método anterior. Esta abordagem permite aproveitar melhor a localidade espacial da cache, já que estamos a multiplicar blocos menores das matrizes, que têm maior probabilidade de caber na cache. Note-se que o tamanho do bloco não precisa de ser divisor do tamanho da matriz, por exemplo, numa matriz 4x4 o tamanho do bloco poderia ser 3. Como iremos ver mais à frente, este algoritmo é o que tira melhor proveito da cache e produz os melhores resultados.

```
for (ii = 0; ii < m_ar; ii += bkSize)
      for (kk = 0; kk < m_ar; kk += bkSize)
          for (jj = 0; jj < m_br; jj += bkSize)
              for (i = ii; i < min(ii + bkSize, m_ar); i++)
                  for (k = kk; k < min(kk + bkSize, m_ar); k++)
                      for (j = jj; j < min(jj + bkSize, m_br); j++)
                          phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
```

1.2.2 Implementações multi-core

Nas duas versões paralelas da segunda implementação do produto de matrizes existe um único *fork* e um único *join* de todas as *threads*. Um dos desafios destas versões paralelas é o problema da concorrência (*race conditions*). Este ocorre quando múltiplas *threads* tentam aceder e modificar a mesma variável simultaneamente, causando resultados imprevisíveis. Uma vez que, as variáveis declaradas fora da região paralela, por defeito, são partilhadas por todas as *threads*, podem surgir *race conditions*. Para evitar este problema, em alguns casos foi necessário torná-las privadas dentro da região paralela, utilizando a cláusula *private()*. Desta forma, cada *thread* possui cópias independentes destas variáveis, evitando acessos concorrentes que poderiam comprometer a correção do algoritmo.

A variável *phc* é partilhada entre todas as *threads*, mas não apresenta problemas de concorrência, uma vez que cada *thread* atualiza índices distintos de *phc[i * m_br + j]*. No entanto, se múltiplas *threads* pudessem modificar o mesmo elemento de *phc* simultaneamente, seria necessário usar *#pragma omp critical* ou *atomic* para sincronização, o que iria impactar o desempenho.

1.2.2.1. “Listing 1: Parallel Nested Loops with OpenMP”

Na primeira versão - ***OnMultLineOMP1()*** - o *loop* externo é paralelizado. A cláusula ***private()*** foi utilizada para tornar as variáveis j e k privadas, uma vez que, por defeito, estas eram partilhadas entre as *threads*. A expressão ***#pragma omp parallel for*** divide as iterações do *loop* externo pelos processadores disponíveis, criando uma *thread* para cada processador (e torna a variável i privada).

1.2.2.2. “Listing 2: Parallel Nested Loops with OpenMP”

Na segunda versão - ***OnMultLineOMP2()*** - o *loop* interno é paralelizado. A cláusula ***private()*** foi utilizada para tornar as variáveis i e k privadas, uma vez que, por defeito, são partilhadas. A diretiva **#pragma omp parallel** cria um grupo de *threads* que executam todo o código dentro dela, ou seja, cada *thread* executa os *loops i* e *k*. A outra expressão, **#pragma omp for**, divide o *loop* interno *j* pelas *threads* já criadas. No entanto, espera-se que esta versão apresente um desempenho inferior à primeira. Isto acontece porque **#pragma omp for** força a sincronização entre todas as *threads*, o que significa que todas devem estar no início do *loop* mais interno antes que qualquer uma possa avançar na sua execução. Isto é, existe trabalho duplicado (na execução dos primeiros 2 ciclos), sendo neste caso preferível o código sequencial.

2. Métricas de desempenho

De forma a avaliar o desempenho das diferentes implementações e linguagens de programação, mediu-se o **tempo de execução**. Em C++, foi usada a função ***clock()*** para a versão sequencial e ***omp_get_wtime()*** para a versão paralela, e em Java recorreu-se a ***System.nanoTime()***.

Outras métricas importantes para avaliar o desempenho do programa são o **número de falhas na cache L1** *(L1 Data Cache Misses, **L1 DCM**)* e o **número de falhas na cache L2** *(L2 Data Cache Misses, **L2 DCM**),* obtidas com o PAPI. Como o acesso à L1 é extremamente rápido, um algoritmo otimizado para cache melhora significativamente a performance. Quando ocorre uma falha na L1, os dados podem ser encontrados na L2 (apesar de ter um custo maior).

Para além disso, utilizaram-se as seguintes métricas:

- $MFLOPS = (\ 2\ *\ (Tamanho\ da\ Matriz)\text{^}3) \div (Texecução\ *\ 10\text{^}6\ )$;
- $Speedup = Tempo\ Sequencial \div Tempo\ Paralelo$;
- $Eficiência = Speedup \div Nº\ Threads$

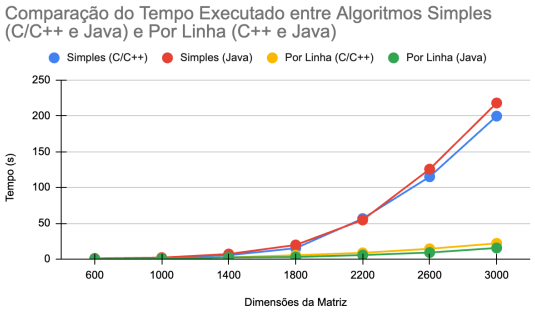
Para garantir a precisão dos resultados, utilizamos a flag de otimização **-O2** durante a compilação, o que ajuda a melhorar o desempenho, especialmente em operações intensivas como a multiplicação de matrizes. Além disso, foram utilizados diferentes tamanhos de matrizes e os testes foram repetidos múltiplas vezes, a fim de minimizar variações nos resultados e garantir maior confiabilidade.

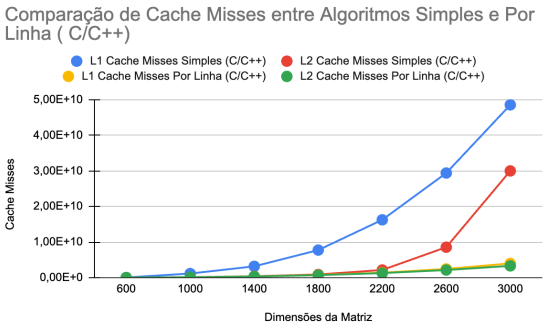
Todos os testes foram realizados no mesmo computador para evitar que diferenças de hardware influenciassem os resultados. O computador utilizado possui as seguintes características: Processador Intel Core i3-1005G1 CPU @ 1.20GHz **[2]**, com 2 núcleos físicos e 4 núcleos lógicos, cache L2 de 1 MiB, e cache L1 de 96 KiB. Verificou-se que existiam **4 threads disponíveis**, utilizando a função **omp_get_max_threads()**.

3. Resultados e Análises

3.1. Análise da “Simple Matrix Multiplication” e “Line Matrix Multiplication” em C++/Java

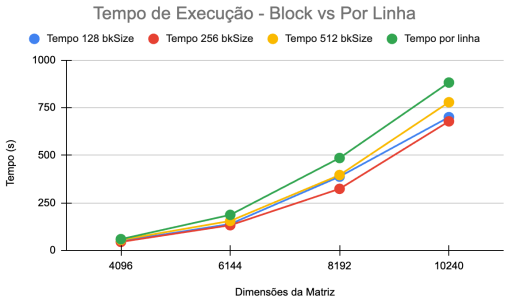
Embora se esperasse que a implementação em Java fosse mais lenta (possivelmente devido ao *overhead* da máquina virtual e à verificação dos limites dos arrays), os resultados mostraram que, para diferentes tamanhos de matrizes, as implementações em C++ e Java apresentaram tempos de execução semelhantes, tanto para o algoritmo mais básico quanto para o *line x line.* Através de uma análise mais minuciosa, verifica-se que no caso do algoritmo mais simples, em C++ existe uma ligeira vantagem (apesar de não se verificar no *line x line*). Contudo, é razoável concluir que a diferença de desempenho entre os algoritmos é independente da linguagem de programação utilizada.





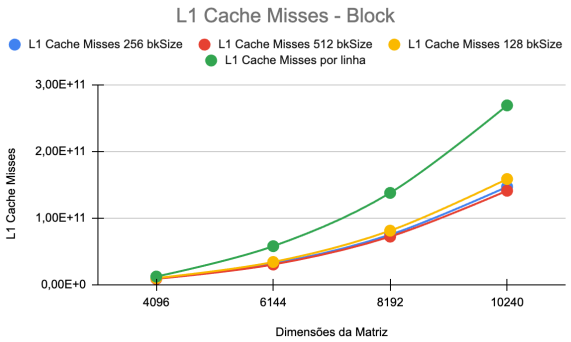
Através do gráfico à esquerda, é evidente que o algoritmo *line x line* é mais eficiente do que o algoritmo simples. Essa superioridade deve-se ao facto do algoritmo aproveitar melhor a localidade espacial da memória cache, conforme explicado anteriormente. Como era de se esperar, essa abordagem resulta em menos falhas na cache (gráfico à direita), o que contribui significativamente para o seu melhor desempenho.

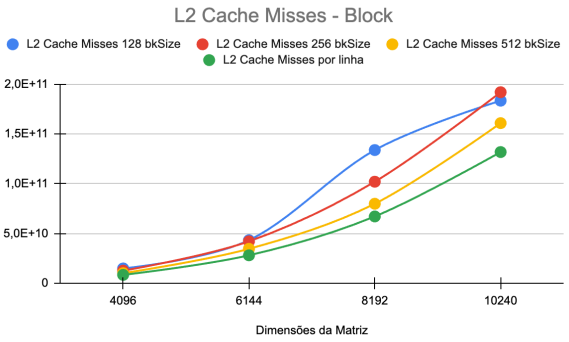
3.2. Análise da “Block Multiplication” vs “Line Matrix Multiplication”



Como é possível analisar através do gráfico acima, a multiplicação por blocos (com 3 tamanhos de bloco distintos) apresenta um tempo de execução inferior à multiplicação por linha.

Apesar de não ser uma vantagem destacada, esta vai aumentando à medida que o tamanho da matriz aumenta, sendo notória a partir do tamanho 6144.





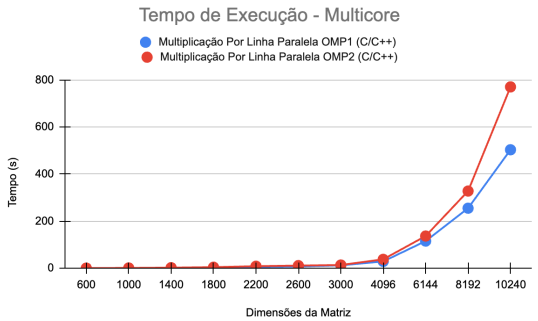
Apesar da multiplicação por blocos ter um tempo de execução inferior, esta implementação não apresenta um menor número de falhas na cache L2 (gráfico à direita), apenas na L1 (gráfico à esquerda), daí a vantagem não ser tão destacada.

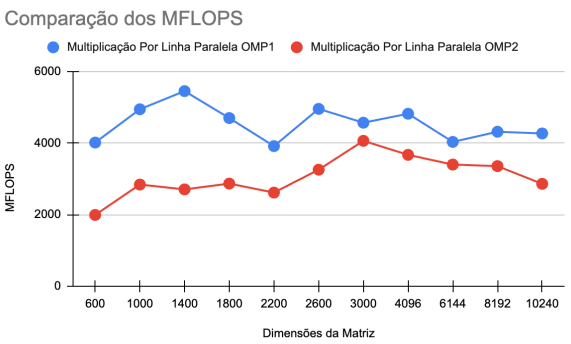
No caso da multiplicação por linha, ao lidar com matrizes grandes, uma única linha ocupa praticamente (ou até excede) a capacidade da cache L1, resultando em falhas nessa cache. Ainda assim, essa mesma linha pode caber completamente na cache L2, levando a um menor número de falhas na L2 para este algoritmo.

No caso da multiplicação por blocos, ao lidar com matrizes/blocos mais pequenos (e consequentemente linhas mais pequenas), resulta num menor número de falhas na cache L1. Porém, ao dividir a matriz em blocos menores, só processamos pequenos segmentos de uma linha de cada vez, e como só avançamos para o próximo segmento da linha depois de processarmos completamente um bloco, nunca conseguimos armazenar uma linha completa na cache L2. Daí este algoritmo apresentar mais falhas na L2.

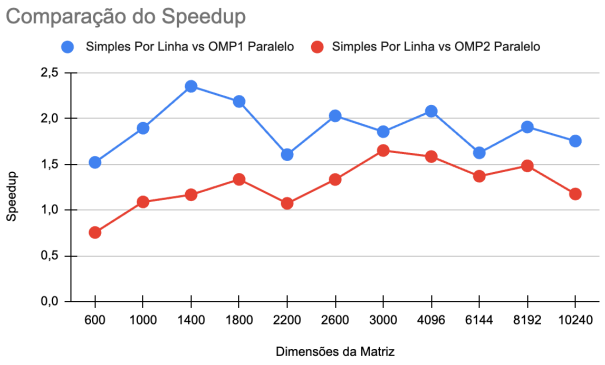
Observando o gráfico à direita, é possível deduzir que de forma geral quanto maior o tamanho do bloco, menor é o número de falhas na L2. Ao aumentar o tamanho do bloco, esta abordagem tende a aproximar-se da multiplicação por linha, logo tende a aproximar-se do seu número de falhas na L2.

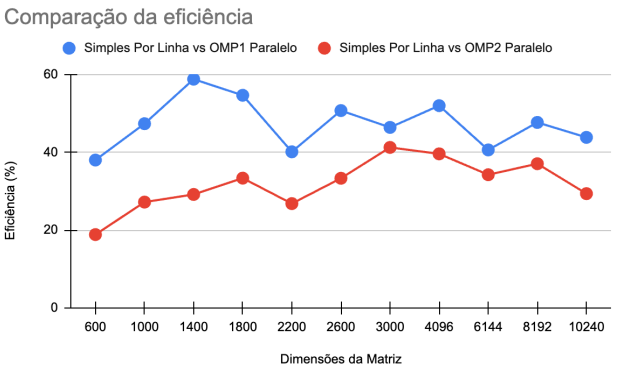
3.3. Análise da “Line Matrix Multiplication” multi-core





Como esperado, a primeira versão (*loop* externo paralelizado) tem um desempenho significativamente melhor. Embora a segunda versão (*loop* interno paralelizado) ainda seja mais eficiente do que o código sequencial, à exceção de um *outlier* (tamanho 600) e especialmente para matrizes maiores, esta acaba por desperdiçar poder de processamento quando comparada à primeira versão. Isto deve-se ao facto da sincronização imposta pela diretiva **#pragma omp for** no loop interno ter impacto no desempenho, tal como já foi explicado acima.





Por um lado, nas duas versões, verifica-se uma subida no *speedup*/eficiência até aos tamanhos 1400-1800. O problema parece estar relacionado com *false sharing*, que ocorre quando várias threads acedem a diferentes variáveis que estão na mesma linha de cache.

Por outro lado, a partir de 1400-1800 nota-se uma queda no desempenho, que pode estar ligado à concorrência entre múltiplas threads no acesso à memória. Como a cache L2 é partilhada pelas *threads* do mesmo core, uma matriz suficientemente grande pode explicar este acontecimento, pois torna-se impossível guardar todos os valores necessários.

4. Conclusão

Na primeira parte, ao avaliar o desempenho em *single-core*, verificou-se que o algoritmo *line x line* superou a abordagem mais simples, devido à melhor utilização da cache. Contudo, o algoritmo de multiplicação por blocos apresentou um desempenho ainda melhor do que o do algoritmo *line x line*, especialmente em matrizes grandes. Observando-se assim, que pequenas variantes do mesmo algoritmo podem gerar performances bastante diferentes devido à forma de como os dados são acessados na memória.

Na segunda parte, na avaliação do desempenho multi-core, foram analisadas duas versões paralelas do algoritmo *line x line*. Observou-se que uma das versões apresentava um desempenho superior à outra, devido a uma melhor distribuição da carga de trabalho e a uma menor sobrecarga de sincronização entre threads. A análise permitiu identificar os principais fatores que influenciaram a eficiência de cada abordagem, destacando a importância de uma boa gestão do acesso à memória e da redução da concorrência entre threads, de forma a otimizar o desempenho em soluções paralelas.

5. Referências

[1] Slides e documentos disponibilizados pela Unidade Curricular;

[2] Intel Core i3 Análise

6. Anexos

- Resultados e Gráficos Obtidos