FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório: Efeito na performance do processador com acesso a grande quantidade de dados dentro de uma hierarquia de memória.

4º Ano do Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Aluno:
Cleverson Veloso Nahum - 201710783 - up201710783@fe.up.pt

12 de março de 2018

1 Introdução

O presente trabalho tem o objetivo de avaliar o impacto da hierarquia de memória no desempenho do processador ao acessar uma grande quantidade de dados na memória, além de estabelecer comparações de desempenho de algoritmos executados com ou sem paralelismo utilizando threads (OpenMP - Open Multi-Processing) e obter as métricas desses testes utilizando a interface PAPI (Performance Application Programming Interface) para obter informações do sistema como processamento, memória e outras que foram consideradas relevantes para o entendimento do trabalho. Outra características levadas em consideração é a diferença de custo na execução de programas escritos em linguagens com níveis de abstrações diferentes, no caso utilizando C++ e JAVA.

O problema proposto para a avaliação das características citadas acima, foi a multiplicação de matrizes através da implementação de 2 algoritmos diferentes como solução.

2 Descrição do problema

O problema sugerido é a multiplicação de matrizes, matematicamente para a realização da operação é necessário que hajam duas matrizes e que o número de colunas da primeira matriz seja exatamente igual ao número de linhas da segunda matriz. De forma que considerando A e B matrizes distintas com dimensões de $A\,(i\,x\,j)$ e $B\,(k\,x\,l)$ onde j e k teriam que necessariamente apresentar o mesmo valor, isso é $j = k$. A multiplicação de duas matrizes descritas resulta em uma matriz com a quantidade de linhas da matriz A e a quantidade de colunas da matriz B, de forma que $A\,(i\,x\,j) * B\,(k\,x\,l) = C(i\,x\,l)$. Onde cada elemento da matriz C é definido pela seguinte fórmula:

$$C_{i,j} = \sum_{k=0}^{j-1} A_{i,k} * B_{k,j} \qquad (1)$$

No problema proposto para esse trabalho as matrizes A e B são matrizes quadradas de mesma dimensões, para facilitar a implementação e execução.

3 Algoritmos propostos

Os dois algoritmos propostos são bem parecidos e apresentam ordem de complexidade $O(n^3)$ por utilizarem 3 laços de execução para realizar a multiplicação das duas matrizes.

Para utilizar a biblioteca de C++ chamada Vector (permite a alocação de memória de forma dinâmica) o algoritmo foi adaptado de forma a converter matrizes bidimensionais em matrizes unidimensionais através do sequenciamento das diversas linhas presentes na matriz bidimensional enfileirando-as, por exemplo uma matriz com dimensões (n,m) vira um vetor de n*m elementos. O controle da dimensão do vetor gerado representando a matriz C se dá através da manipulação dos índices desse vetor.

3.1 Algoritmo 1

O algoritmo 1 realiza a multiplicação da matriz A e B através da multiplicação sucessiva de elementos nas linhas da matriz A pelos elementos em colunas da Matriz B. No trecho de código abaixo percebemos que a variável i é associada às linhas da matriz, a variável j as colunas e k é a variável que vai permitir iterar pelos elementos da matriz fazendo a soma acumulada, portanto o algoritmo lê uma linha da matriz A e as colunas da matriz B a cada interação da variável i de forma que no final de um ciclo tenhamos uma linha da matriz C escrita .

```
for(int i = 0; i < d; ++i)
        for(int j = 0; j < d; ++j){
                for( int k = 0; k < d; ++k)
                  {
                        mult[i*d+j]+=m1[i*d+k]*m2[k*d+j];
                  }
          }
```

Código Fonte 1: Trecho de código do algoritmo 1

3.2 Algoritmo 2

O algoritmo 2 apresenta um algoritmo um tanto similar ao algoritmo 1, diferenciando que ao invés de fazer a multiplicação de linhas da matriz A pelas colunas

de B, é realizada a multiplicação sucessiva de elementos de uma linha da matriz A por todas as linhas da matriz B. Analisando o código abaixo percebemos que a cada interação da variável i, a linha i da matriz A é lida e todos os elementos de cada linha da matriz B são lidos e consequentemente é escrito uma linha na matriz C.

```
for(int i = 0; i < d; ++i)
        for(int k = 0; k < d; ++k){
                for( int j = 0; j < d; ++j)
                 {
                        mult[i*d+j]+=m1[i*d+k]*m2[k*d+j];
                 }
          }
```

Código Fonte 2: Trecho de código do algoritmo 2

4 Metodologia e Métricas

Para realizar a avaliação das implementações propostas serão utilizadas diferentes métricas de acordo com as características a serem levadas em consideração de forma a avaliar cada cenário proposto.

Os experimentos foram realizados em um notebook com CPU Intel Core i3-5005U de frequência de 2.0 GHz com 4 cores, apresentando 4 tipos de memória Cache, sendo 32K de L1d, 32K de L1i, 256K de L2 e 3072K de L3. Além de permitir a execução de até 2 threads por core.

4.1 Comparação C++ e Java

Nesse cenário foi implementado o algoritmo 1 nas linguagens de programação C++ e JAVA. Foram levados em consideração o tempo de execução dos algoritmos em cada linguagem para matrizes quadradas com dimensões de 500x500 até 3000x3000 em intervalos de 500 em 500 elementos.

Para realizar a coleta de informações de tempo de execução em C++ foi utilizado a ferramenta PAPI e em JAVA foi utilizado a função da biblioteca “System” denominada “nanotime()”.

4.2 Comparação dos algoritmos 1 e 2

Nesse cenário foram implementados o algoritmo 1 e 2 em C++, e foi utilizado a ferramenta PAPI para obter as seguintes métricas de execução dos programas: Tempo real de execução, tempo de processamento, total de instruções executadas, instruções por ciclo de clock (IPC), quantidade de cache misses na memória L2. Por limitações do PAPI na CPU do notebook onde foram realizados os testes, não foram obtidas métricas das cache L1 e L3 ou mesmo a quantidade de cache hits e acessos a L2.

Cada algoritmo foi executado inserindo matrizes quadradas com dimensões 500x500 até 3000x3000 em intervalos de 500 em 500 elementos, e também com matrizes quadradas de 5000x5000 até 10000x10000 em intervalos de 1000 em 1000 elementos.

4.3 Comparação dos algoritmos 1 e 2 usando Threads

Neste terceiro cenário foram implementados os algoritmos 1 e 2 utilizando threads com a biblioteca OpenMP [2]. Foram avaliadas as seguintes métricas obtidas utilizando o PAPI: Tempo real de execução, tempo de processamento, total de instruções executadas, instruções por ciclo de clock (IPC).

Cada algoritmo foi executado inserindo matrizes quadradas com dimensões de 500x500 até 3000x3000 em intervalos de 500 em 500 elementos, onde a cada dimensão avaliadas também variam a quantidade de threads executando o algoritmo, variando de 1 a 8 threads.

5 Análises e Resultados

5.1 Comparação C++ e Java

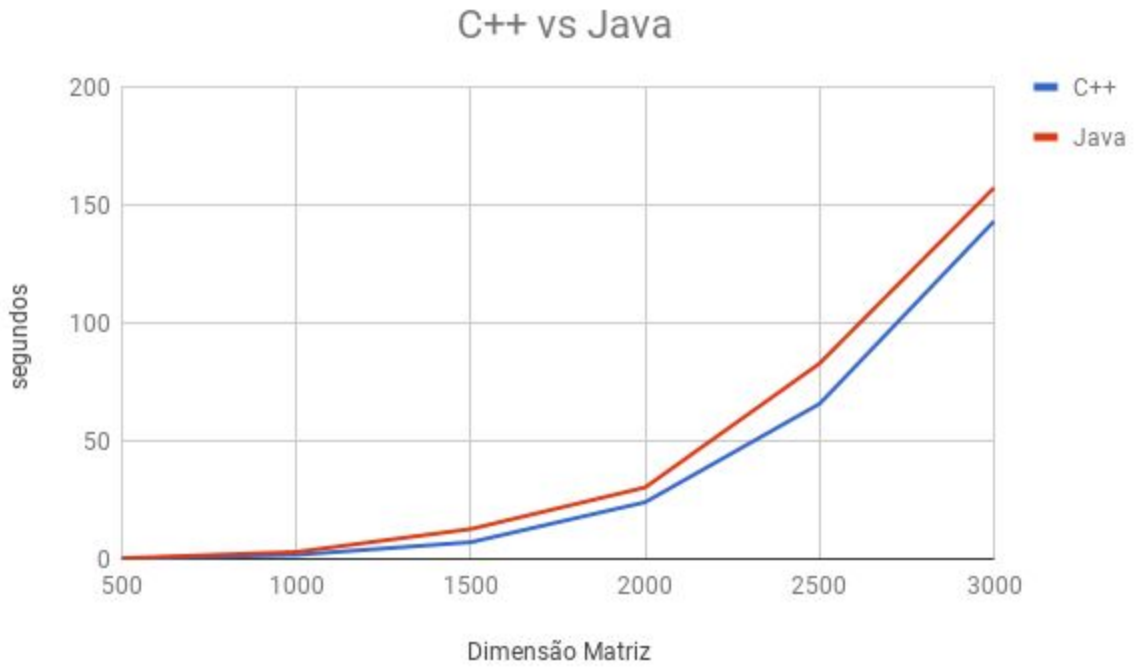

Figura 1: Comparativo do tempo de execução do algoritmo 1 implementado em C++ e Java

Analisando ambos os gráficos das implementações dos diferentes algoritmos podemos perceber que em todos os cenários a implementação na linguagem C++ demonstrou vantagem com relação ao tempo de execução do algoritmo completo, o que se trata de um comportamento esperado devido às características das duas linguagens.

Java é uma linguagem interpretada que apresenta maior portabilidade, mas ao ser executada em uma máquina virtual acaba por perder bastante em desempenho se comparada a linguagem C++ que é uma linguagem compilada e de mais baixo nível, contando ainda com várias otimizações que podem ser incluídas no compilador usado [3].

5.2 Comparação dos algoritmos 1 e 2

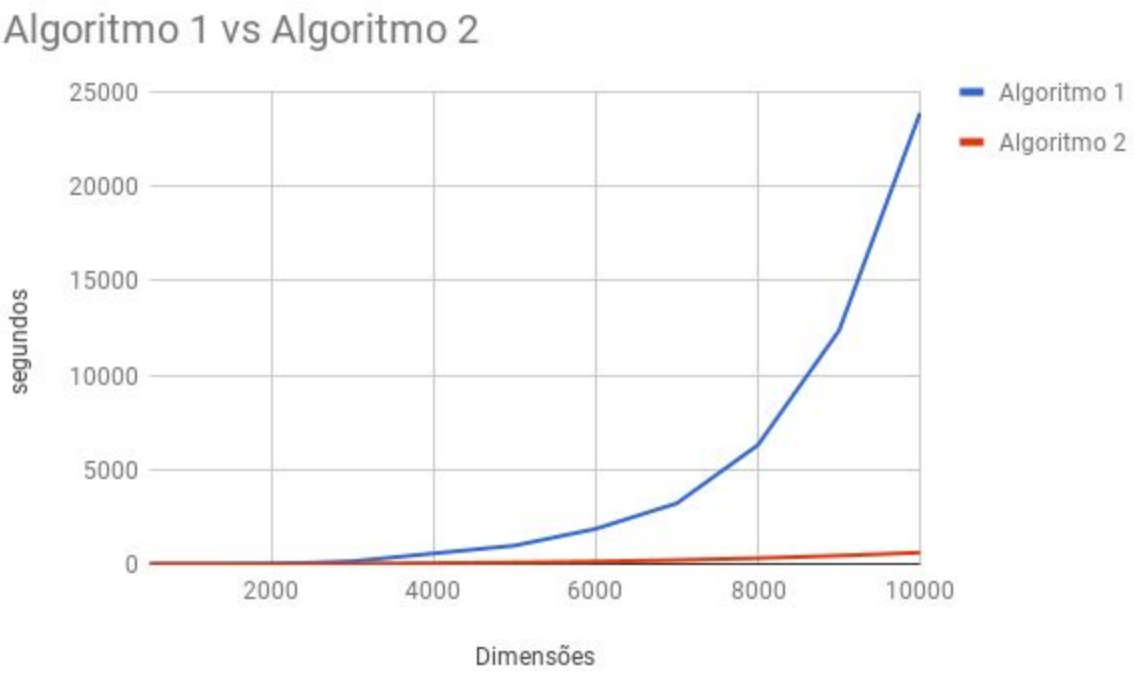

Figura 2: Comparativo tempo de execução algoritmo 1 e 2 implementados em C++ e Java

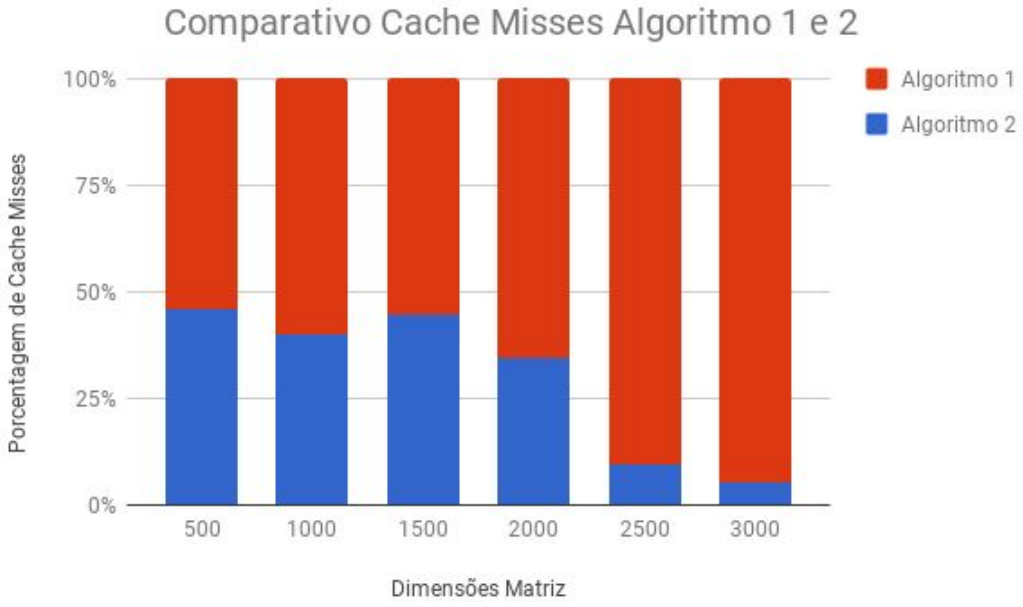

Figura 3: Comparativo de Cache Misses dos algoritmo 1 e 2 implementados em C++

Observando os gráficos comparando o desempenho do algoritmo 1 e o algoritmo 2, podemos perceber que em todos os cenários o algoritmo 2 sempre foi executado mais rapidamente, mesmo que aparentemente os dois tenham a mesma quantidade de linhas de códigos e a única diferença entre eles seja a ordem com que as matrizes são acessadas.

Após perceber a vantagem relacionada ao tempo de execução que o algoritmo 2 leva sobre o algoritmo 1, podemos analisar o gráfico demonstrando as informações referentes a razão de cache misses na memória cache L2 do processador. Percebe-se que o algoritmo 2 apresenta melhores índices com relação ao aproveitamento das informações da memória Cache, isso acontece devido a lógica “row-major order” utilizada por padrão na linguagem C++ para armazenamento das matrizes na memória RAM.

A lógica “row-major order” define que os elementos de uma mesma linha deverão ser armazenados em áreas de região contígua na memória RAM, como as matrizes foram implementadas em vetores nos algoritmos, os blocos acabam por seguir a mesma ordem do vetor na memória, de forma de que os elementos de uma linha estão sempre em sequência e próximos uns aos outros.

Outro conceito que combinado com o “row-major order” justifica a diferença dos resultados do algoritmo 1 e 2, é o “Princípio da Localidade” [1], no qual ao se fazer um acesso a memória RAM, os valores correspondentes àquela localidade do acesso (vizinhança) são copiados para a memória cache do processador, se supõe uma maior probabilidade estatística de que os dados da vizinhança venham a ser necessários posteriormente e tê-los na cache tornam a execução ainda mais rápida.

Os dois fatores explicados anteriormente ajudam a entender os resultados superiores de performance obtidos pelo algoritmo 2 em relação ao algoritmo 1, já que nele o acesso a diversas linhas é feito logo nas primeiras instruções do algoritmo e os valores da Cache acabam por ser melhor utilizados, o que é mostrado pela Figura 3.

4.3 Comparação dos algoritmos 1 e 2 usando Threads

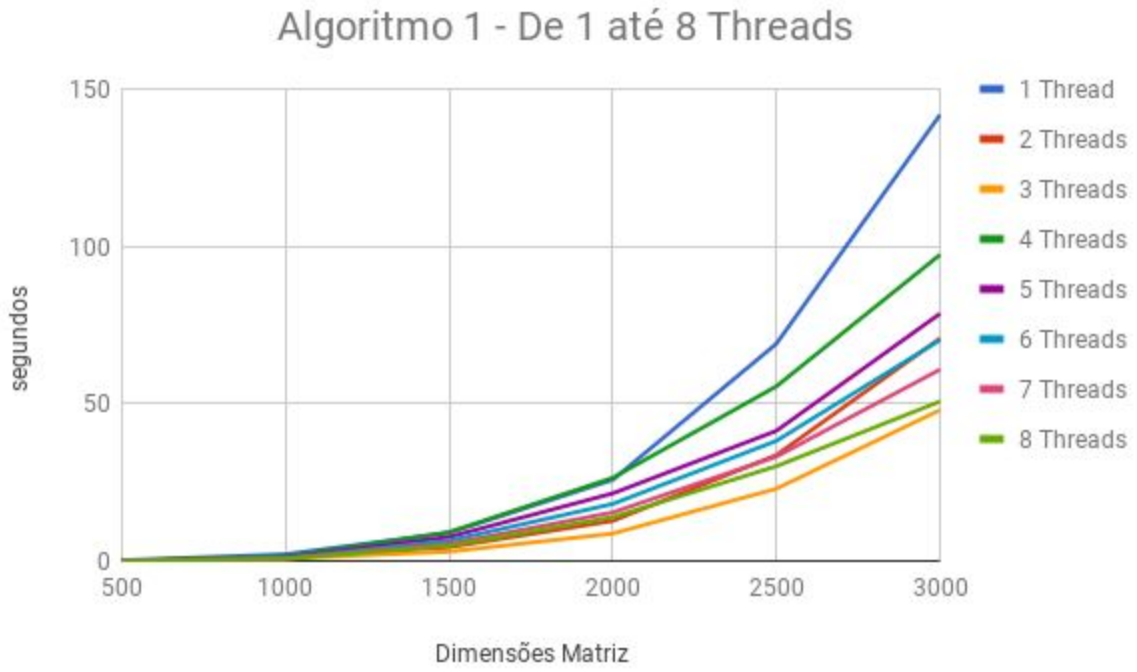

Figura 4: Comparativo tempo de execução algoritmo 1 implementado em C++ variando a quantidade de Threads utilizadas

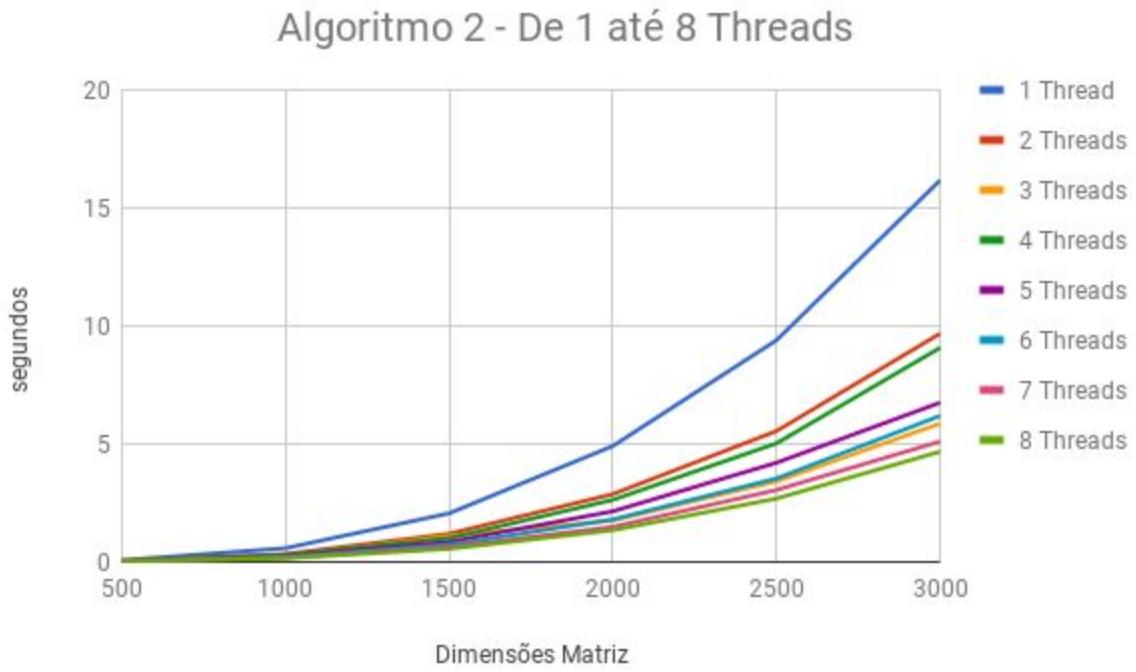

Figura 5:: Comparativo do tempo de execução algoritmo 2 implementado em C++ variando a quantidade de Threads utilizadas

Analisando os gráficos referentes ao desempenho dos algoritmos 1 e 2 utilizando paralelismo com a quantidade de threads variando de 1 a 8, podemos perceber que o paralelismo contribui bastante para a melhoria da performance de execução, tal fato se deve ao melhor aproveitamento dos recursos computacionais ao promover o uso dos diferentes núcleos do processador através da paralelização do processamento das multiplicações executadas por cada algoritmo.

É possível perceber que conforme o número de threads utilizadas vai aumentando, ocorre uma melhora no tempo de execução de ambos os algoritmos, mas que essa escala não é propriamente linear pois fatores como a alocação de blocos de memória na cache se tornam mais complexas à medida que se possui matrizes de maiores dimensões e diversas threads atuando ao mesmo tempo, aumentando a variabilidade de conteúdo que necessita ser alocado na cache.

Outro fator importante a ser observado, é que o computador utilizado na execução do algoritmo possui 4 núcleos de processamento e permite a execução de até 2 threads por núcleo, significando que 2 threads em execução no mesmo núcleo acabam por compartilhar recursos computacionais contribuindo ainda mais para que o aumento do número de threads não reflita de forma linear na melhora no tempo de execução dos algoritmos.

6 Conclusão

O presente trabalho propôs a análise de desempenho de dois algoritmos diferentes para a solução da multiplicação de matrizes em diferentes linguagens, analisando fatores como tempo de execução, acesso a memória e paralelismo. Concluiu-se que a linguagem C++ teve maior eficiência do que Java, e em todos os cenários o algoritmo 2 apresenta vantagens devido se aproveitar de conceitos como o “Princípio de Localidade” e “row-major order” da alocação de memória cache, além de que os dois algoritmos tiveram seus desempenhos melhorados em até 4 vezes após a utilização de threads.

Referências

[1] Barria, Javier A. Communication networks and computer systems: a tribute to Professor Erol Gelenbe. World Scientific, 2006.

[2] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." *IEEE computational science and engineering* 5.1 (1998): 46-55.

[3] Moreira, José Eduardo, Samuel P. Midkiff, and Manish Gupta. "A comparison of Java, C/C++, and FORTRAN for numerical computing." *IEEE Antennas and Propagation Magazine* 40.5 (1998): 102-105.