U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Universidade do Porto

Faculdade de Engenharia

Computação Paralela e Distribuída

2021/2022

Relatório - Projeto 1

PROFESSOR:

Prof. Jorge Manuel Gomes Barbosa

ESTUDANTES:

Camilo Melgaço (202110918)

Renato Leite (201908633)

Shirley Fortes (201808614)

Março de 2022

Introdução

Este projeto tem como objetivo o estudo do desempenho do processador da hierarquia de memória quando se tenta acessar a grandes quantidades de dados. Para este estudo serão utilizados alguns algoritmos de multiplicação de matrizes e o Performance API (PAPI) para coletar indicadores de desempenho relevantes da execução do programa.

Descrição do problema e explicação dos algoritmos

Problema nº 1:

Para o problema nº1 foi disponibilizado um programa em C++ que faz a multiplicação de duas matrizes, multiplicando as linhas da primeira pelas colunas da segunda matriz.
É pedido que se implemente o mesmo algoritmo de multiplicação de matrizes agora numa linguagem à escolha, que neste caso foi o Java.

O algoritmo em Java segue exatamente o mesmo raciocínio, tendo apenas alguns ajustes nos detalhes em relação à sintaxe correspondente à linguagem.
Para cada um dos programas (em C++ e em Java), deve-se registrar o tempo de processamento para as duas linguagens para matrizes de entrada de 600x600 a 3000x3000 elementos com incrementos em ambas as dimensões de 400.

Problema nº 2:

Para resolver o problema nº2, foi implementado uma nova versão que faz a multiplicação por linhas, ou seja, que multiplica cada elemento da primeira matriz pela linha correspondente da segunda matriz. Para implementar esta versão basicamente tivemos apenas de inverter os ciclos *for*, em que o último ciclo passa a ser o penúltimo e vice-versa. Esta nova versão foi implementada tanto em C++ como em Java.

```
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];
        }
    }
}
```

Problema nº 3:

Uma nova versão do algoritmo foi implementada para resolver o problema nº3. Este algoritmo divide as matrizes em n blocos e depois executa a mesma sequência de cálculos apresentada na resolução do problema 2. Exemplo:

$$AB = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{22}B_{12} + A_{22}B_{22} \end{array}\right]$$

```
for(int jj=0; jj<m_ar; jj+=bkSize){
    for(int kk=0;kk<m_br; kk+=bkSize){
        for(int i=0;i<m_ar;i++){
            for(int j = jj; j<((jj+bkSize)>m_ar?bkSize:(jj+bkSize)); j++){
                temp = 0;
                for(int k = kk; k<((kk+bkSize)>m_ar?m_ar:(kk+bkSize)); k++){
                    temp +=  pha[i*m_ar+k] * phb[k*m_br+j];
                }
                phc[i*m_ar+j] += temp;
            }
        }
    }
}
```

Métricas de Performance

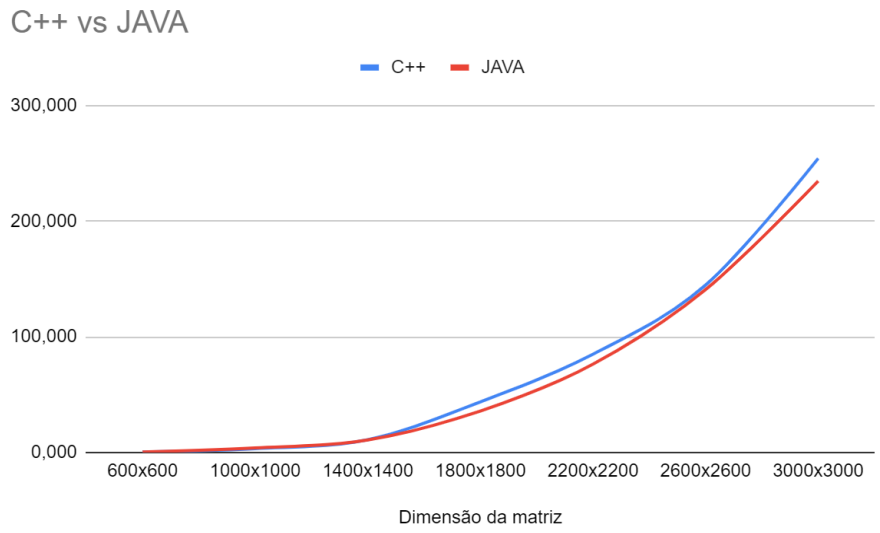
De modo a avaliar a performance dos programas em C++ foi usado a Performance API (PAPI) para a análise dos *caches misses* do processador. Para o cálculo de matrizes é necessário acessar a uma determinada célula da matriz, o que significa estar a aceder a certas posições da memória. De forma a tornar as operações mais rápidas, alguns dados mais importantes são guardados na cache do processador. Uma *cache miss* ocorre quando os dados que estão sendo solicitados por um sistema ou aplicativo não são encontrados na memória cache. Em outras palavras, um *cache miss* é uma falha na tentativa de acessar e recuperar os dados solicitados. Para se ter programas mais rápidos e eficientes é necessário tentar reduzir ao máximo os *cache misses*. Veremos que as diferentes versões de algoritmo de multiplicação de matrizes vão ter diferentes valores no *cache miss.*

Para comparar o desempenho do programa escrito em C++ em relação ao programa escrito em Java foi usado como métrica de performance o tempo de execução do programa para cada um dos algoritmos.

Resultados e análises

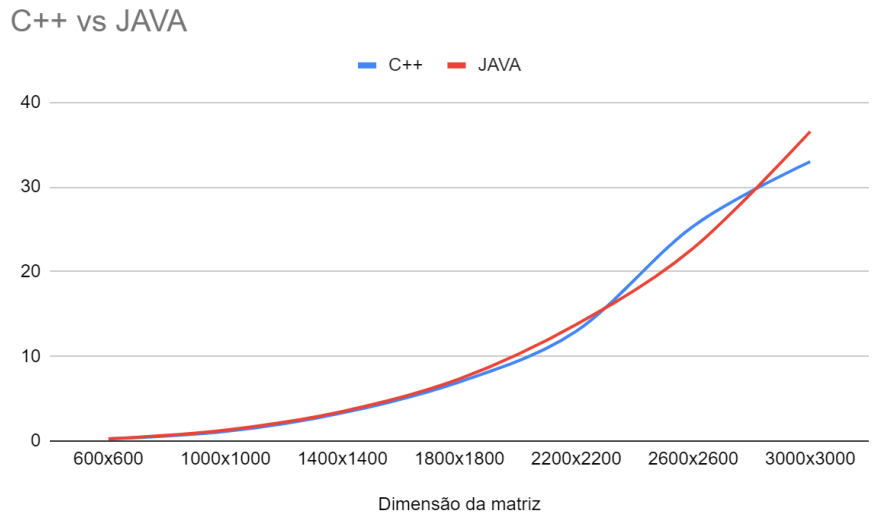
Multiplicação Básica de Matrizes: C++ vs JAVA (Tempo de execução em segundos)

Dimensão da matriz	C++	JAVA
600x600	0,374	0,410
1000x1000	3,330	3,934
1400x1400	11,053	10,820
1800x1800	43,759	35,868
2200x2200	84,984	76,374
2600x2600	144,692	140,833
3000x3000	254,166	234,608

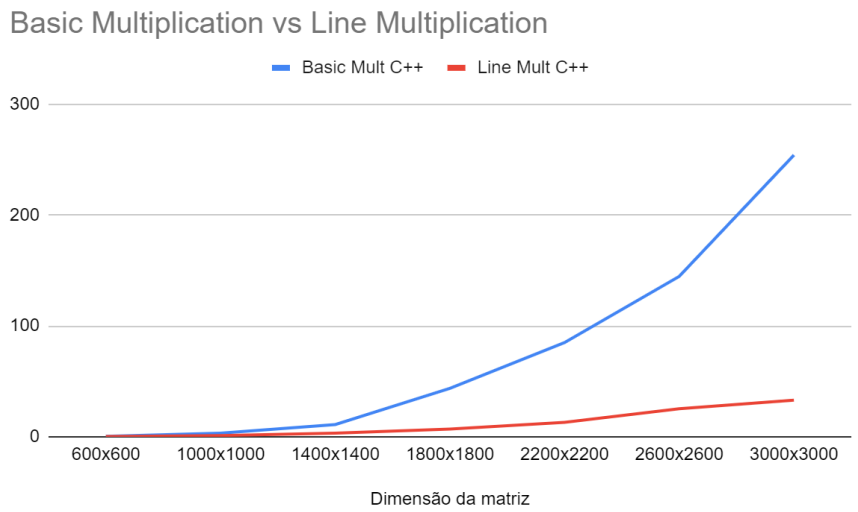


Multiplicação por linhas: C++ vs JAVA (Tempo de execução em segundos)

Dimensão da matriz	C++	JAVA
600x600	0,257	0,26
1000x1000	1,126	1,285
1400x1400	3,316	3,493
1800x1800	6,98	7,336
2200x2200	12,982	13,766
2600x2600	25,374	22,777
3000x3000	32,988	36,557



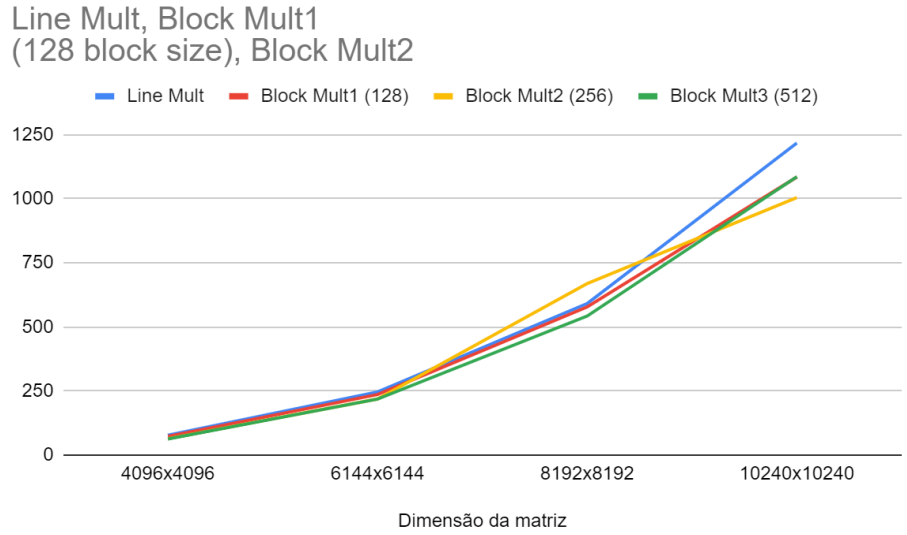
Através da interpretação dos gráficos acima podemos afirmar que os valores não variam muito quando se trata de usar a linguagem C ou Java. A linguagem neste caso não determina um papel importante na classificação da eficiência ou ineficiência do programa.



Com este gráfico podemos notar que o tempo de execução quando se utiliza o algoritmo de multiplicação por linhas é muito menor do que no primeiro caso onde se faz a multiplicação básica de matrizes. O tempo diminui drasticamente tanto no programa em C como em Java. Sendo assim pode-se afirmar que o algoritmo desempenha um papel decisivo no que diz respeito à eficiência do programa. Logo, o algoritmo de multiplicação por linhas é muito mais eficiente do que o algoritmo que faz a multiplicação simples de matrizes. Melhores algoritmos geram melhores resultados e menores tempos de execução.

Multiplicação por linhas e blocos (bloco de 128) em C++:

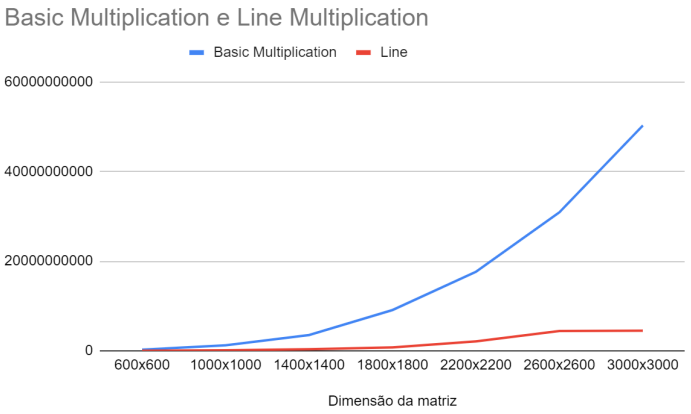
Dimensão da matriz	Line Mult	Block Mult1 (128)	Block Mult2 (256)	Block Mult3 (512)
4096x4096	77,839	72,603	63,612	62,816
6144x6144	246,248	236,276	218,619	218,385
8192x8192	590,83	577,838	669,385	542,515
10240x10240	1217,277	1084,693	1004,297	1086,017



Com a análise do gráfico acima, podemos ver que os valores do tempo de execução na utilização do algoritmo da multiplicação em linhas ou em blocos não varia muito. Apesar da multiplicação por blocos tornar a operação matemática mais rápida, em termos de acesso à memória este algoritmo usa a mesma técnica da multiplicação em linhas dentro de cada bloco, tendo um comportamento de acesso à cache muito semelhante. Por esta razão não há grande otimização.

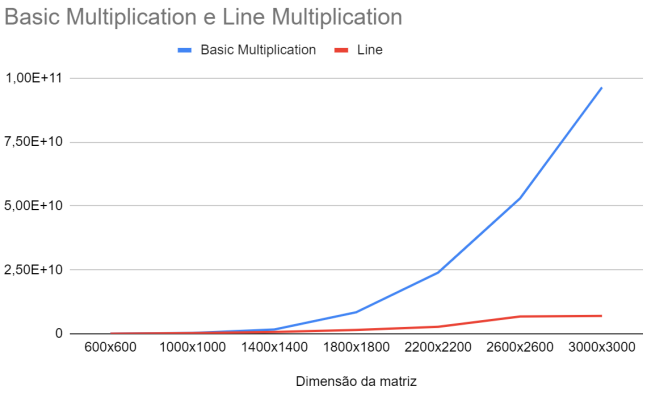
Cache Misses no Nível 1 (L1):

Dimensão da matriz	Basic Multiplication	Line Multiplication
600x600	244725079	27252315
1000x1000	1227154607	125963774
1400x1400	3510560902	351742581
1800x1800	9090432043	758461191
2200x2200	17637942830	2100002388
2600x2600	30910728853	4409853503
3000x3000	50294550070	4492739685



Cache Misses no Nível 2 (L2):

Dimensão da matriz	Basic Multiplication	Line Multiplication
600x600	38658433	55948188
1000x1000	308395352	256069773
1400x1400	1657042638	699120382
1800x1800	8415786199	1496708641
2200x2200	23944312861	2747884028
2600x2600	52987128785	6778754115
3000x3000	96367956476	6979010451



Analisando os dados obtidos com a biblioteca PAPI (Performance API), percebe-se que o método de multiplicação por linhas reduz significativamente o número de erros na memória cache, o que aumenta a eficiência do programa ao se aproveitar do princípio de localidade.

Conclusões

Em resumo, podemos concluir com este estudo que a linguagem de programação escolhida não afeta significativamente a eficácia desses programas, pois são programas que realizam operações matemáticas simples, portanto, o fator linguagem de programação não desempenha papel decisivo nesse contexto. Em suma, um bom algoritmo torna um programa mais eficiente, reduzindo assim seu tempo de execução. A eficiência desses algoritmos está na redução do número de alterações na cache, aproveitando melhor o princípio da localidade. Então, o que realmente faz a diferença é o método computacional usado para reduzir o acesso à memória. Portanto, sempre que possível, deve-se adotar a boa prática de escrever programas que utilizem algoritmos mais eficientes de modo a economizar recursos computacionais.