Computação Paralela e Distribuída

L.EIC028-2022/2023-2S

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

PERFORM EVALUATION OF A SINGLE CORE

Licenciatura em Engenharia Informática e Computação

3LEIC28 - Grupo 16

Afonso Pinto
up202008014@fe.up.pt
Patrícia Miranda
up202007675@fe.up.pt

Índice

1. Introdução

A performance de um dado processador depende tanto de fatores controlados pelo programa, como operações I/O e gestão da hierarquia de memória, como das características físicas do dispositivo (número de *cores*, velocidade do processador e da memória, etc.).
Este relatório tem como objetivo estudar a relação entre o desempenho do *CPU* e diferentes tipos de algoritmos com variados tamanhos de dados. Para tal, utilizámos a *API PAPI* e as linguagens C++ e JAVA.

2. Explicação dos algoritmos

2.1. Multiplicação pelo método algébrico simples

Dada duas matrizes, $\mathbf{A = [a_{ij}]}$, e $\mathbf{B = [b_{ij}]}$, de dimensões $\boldsymbol{m}$ x $\boldsymbol{n}$ e $\boldsymbol{n}$ x $\boldsymbol{p}$, respetivamente, a matriz resultado $\mathbf{AB = [c_{ij}]}$ é calculada pela fórmula $\mathbf{a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in}b_{nj}}$, para cada $\mathbf{1 \leq i \leq m}$ e $\mathbf{1 \leq j \leq p}$, sendo o número de colunas de A obrigatoriamente igual ao número de linhas de B.

```
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;
    }
}
```

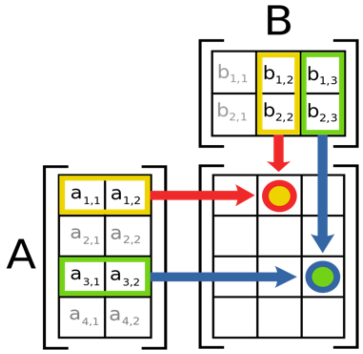

Figura 1 *- multiplicação algébrica simples.*

2.2. Multiplicação por linhas

Com este algoritmo, usando o exemplo anterior, um dado elemento de A é multiplicado por todos os elementos da linha correspondente em B.

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

2.3. Multiplicação por blocos

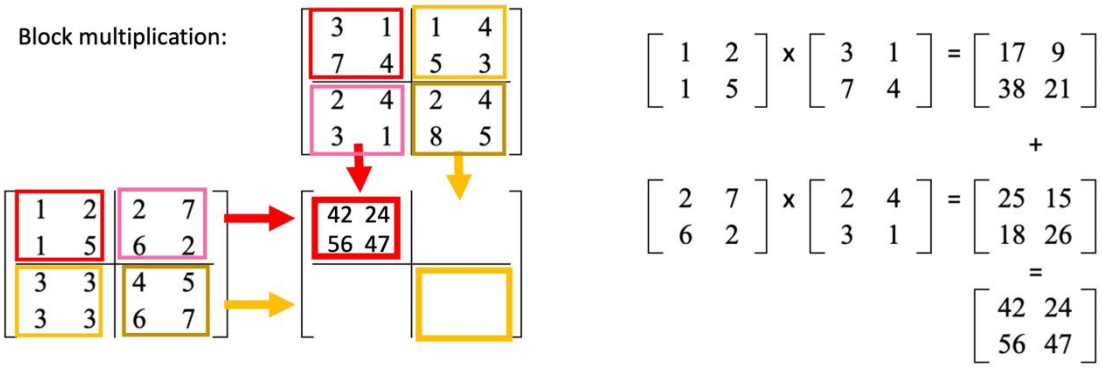

***Figura 2** - Multiplicação por blocos.*

Para implementar este algoritmo, dividimos a matriz em várias submatrizes, de tamanho *bkSize*, dado pelo utilizador, e calculamos recursivamente o produto de cada par de submatrizes através do método de multiplicação por linhas.

```
for (i = 0; i < m_ar; i += bkSize) {
    for (j = 0; j < m_br; j += bkSize) {
        for (k = 0; k < m_ar; k+=bkSize) {
            for (l = i; l < min(i + bkSize, m_ar); l++) {
                for (m = k; m < min(k + bkSize, m_br); m++) {
                    for(n = j; n < min(j + bkSize, m_ar); n++) {
                        phc[l * m_ar + n] += pha[l * m_ar + m] * phb[m * m_ar + n];
                    }
                }
            }
        }
    }
}
```

3. Métricas e Desempenho

Como referido anteriormente, ao longo do trabalho desenvolvemos os programas em C++ e Java. Na primeira linguagem, utilizámos a Performance API **(PAPI),** de forma a obter o número de *Data Cache Misses* (DCM) nos níveis L1 e L2.
Realizámos testes com vários tamanhos de matrizes, para verificar como o comportamento dos diferentes algoritmos se altera.

4. Resultados e Análise

4.1. Multiplicação pelo método algébrico simples versus por linhas

Verificámos que o método da multiplicação por linhas, quando testado em C++, é mais eficiente que o tradicional. Isto acontece devido ao facto desta linguagem guardar os *arrays* 2D de uma

forma linear em memória, ou seja, da forma como as matrizes são construídas no programa (neste caso, são armazenadas em *row-order*).

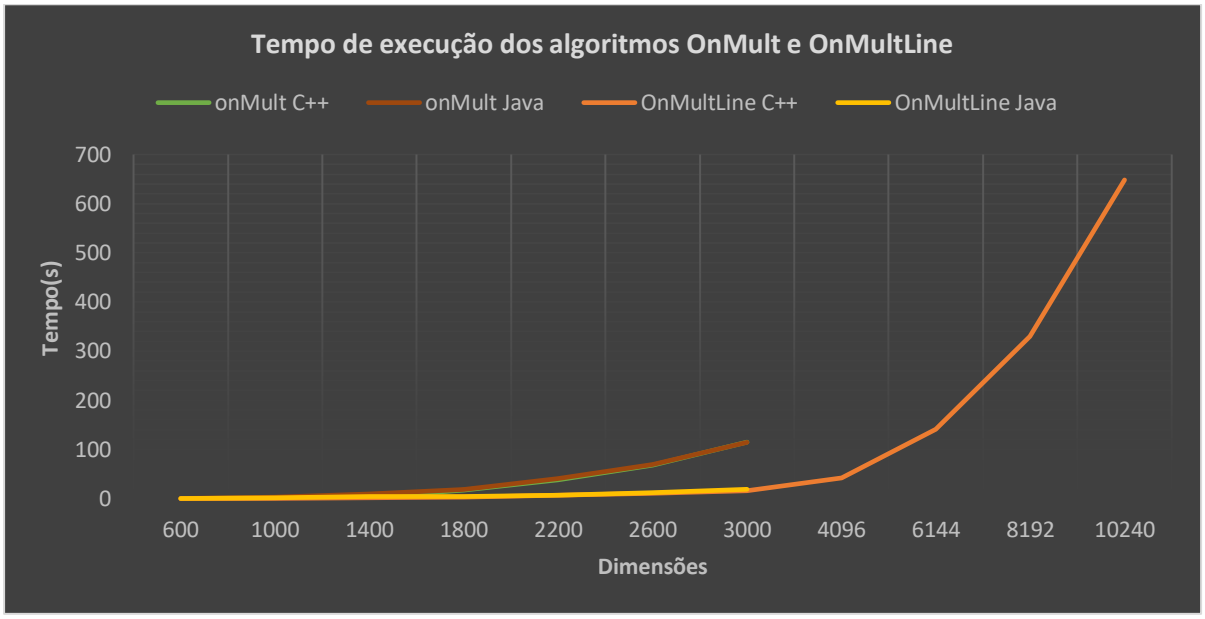

***Gráfico 1** - Tempos de execução nos algoritmos OnMult e OnMultLine em Java e C++.*

Assim, ao multiplicar por linhas, os elementos da matriz são percorridos na mesma sequência em que estão guardados, resultando em **menos *misses* e um melhor desempenho**, como é visível nos **Gráfico 2** e **3**.

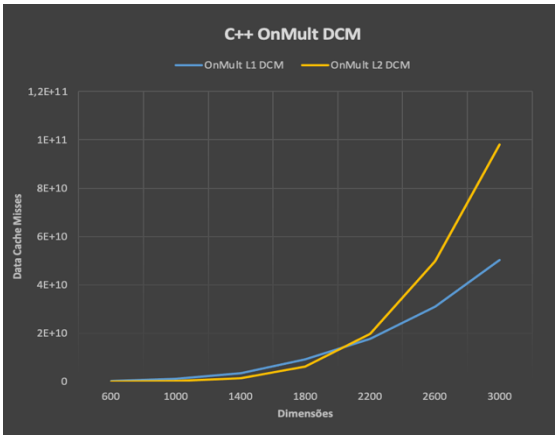

***Gráfico 2** - Data Cache Misses na multiplicação algébrica simples (OnMult) em C++.*

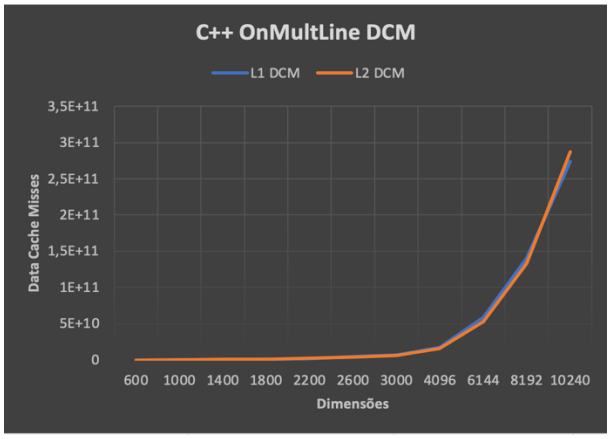

***Gráfico 3** - Data Cache Misses na multiplicação por linhas (OnMultLine) em C++.*

Por outro lado, em Java, os *arrays* 2D são armazenadas como um *array* de referências a outros *arrays*. Deste modo, apenas o *array* inicial é guardado linearmente, estando os outros em posições de memória pseudo-aleatórias. Por esta razão, o segundo algoritmo (OnMultLine) não resulta numa melhoria tão significativa nesta linguagem.
De notar que, de um modo geral, Java é mais lento que C++, visto que C++ é compilado para binário, correndo quase imediatamente, enquanto Java é interpretado durante o *runtime* (ver **Gráfico 1**).

4.2. Multiplicação por blocos

Chegámos à conclusão de que este era o algoritmo mais eficiente, tendo em conta que divide a matriz em blocos mais pequenos, o que faz com que o programa necessite de menos acessos a memória. Podemos verificar esta afirmação comparando os ***Data Cache Misses*** do ***OnMultLine*** e do ***OnMultBlock.*** Esta diferença aumenta drasticamente a partir das matrizes de tamanho **10240**, já que não há memória cache suficiente para armazenar cada linha da matriz.
Salientamos que, devido ao reduzido número de testes realizados, as discrepâncias entre diferentes tamanhos de bloco foram inconsistentes, não podendo assim tirar conclusões desses resultados. Em alternativa, deduzimos que, enquanto o tamanho de um bloco não for maior que o tamanho da cache, o programa beneficia em formar blocos maiores, notando uma maior eficiência com tamanhos maiores.

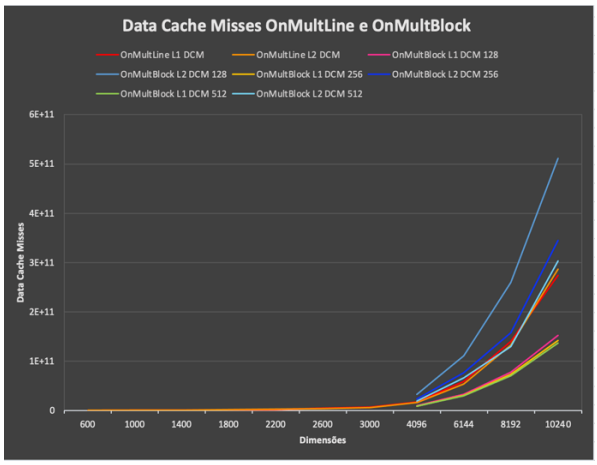

Gráfico 4 *– Data Cache Misses na multiplicação por linhas (OnMultLine) e na multiplicação por blocos (OnMultBlock) em C++.*

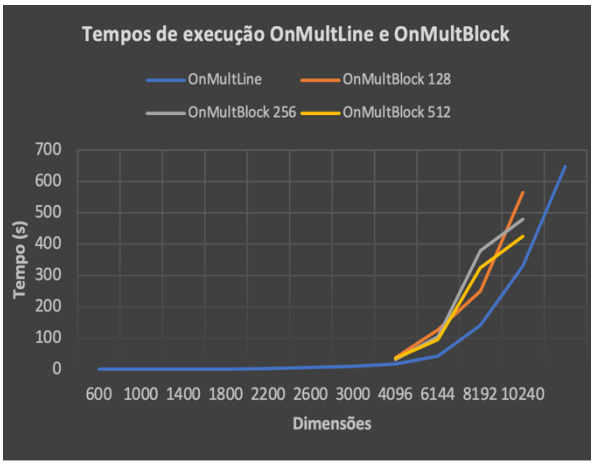

Gráfico 5 *– Tempos de execução na multiplicação por linhas (OnMultLine) e na multiplicação por blocos (OnMultBlock) em C++.*

4.3. Comparação de GFlops

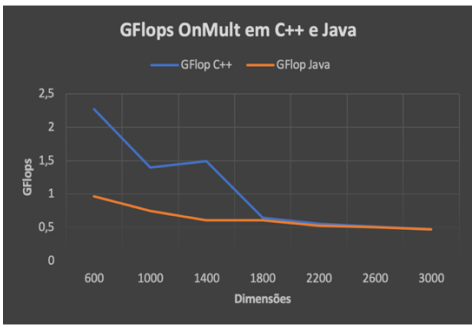

Gráfico 6 *– GFlops da multiplicação algébrica simples (OnMult) em C++ e Java.*

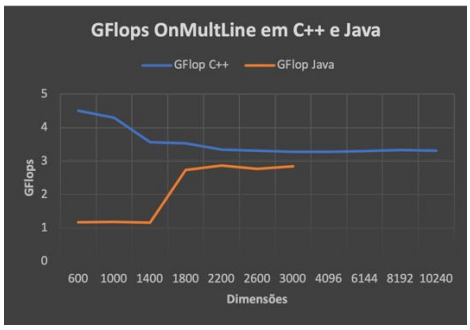

Gráfico 7 *– GFlops da multiplicação por linhas (OnMultLine) em C++ e Java.*

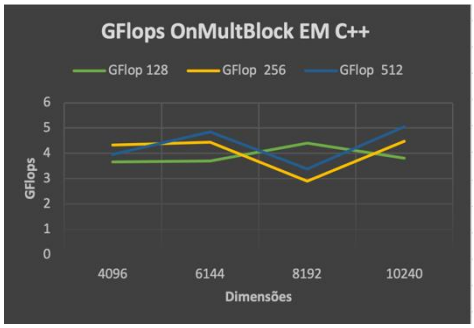

Gráfico 8 *– GFlops da multiplicação por blocos (OnMultBlock) em C++ e Java.*

De modo a confirmar as nossas conclusões, medimos os GFlops (*Floating-point operations per second*) de cada implementação. Assim, comprovámos os resultados anteriores, tanto na comparação dos diferentes algoritmos como entre as duas linguagens utilizadas.

5. Conclusão

Com este trabalho, percebemos a importância de adaptar diferentes algoritmos à arquitetura do processador e respetiva cache, tendo também em conta os acessos a memória necessários.
Concluímos que a melhor opção é a multiplicação por blocos, já que divide os blocos e aplica recursivamente a multiplicação por linhas (mais eficiente que a tradicional) culminando assim num algoritmo mais eficaz.