Performance Evaluation of a Single Core

Relatório

4º ano do Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Elementos do grupo:
José Pedro Pereira Amorim - 201206111 - ei12190@fe.up.pt
Miguel Oliveira Sandim - 201201770 - miguel.sandim@fe.up.pt

30 de março de 2016

1 Introdução

No âmbito da Unidade Curricular de Computação Paralela foi proposto um estudo do impacto da hierarquia de memória no desempenho do processador, quando o acesso a memória envolve uma grande quantidade de dados.

No decorrer deste relatório será analisado o problema do produto de matrizes, bem como dois algoritmos possíveis para a sua resolução. Serão também analisadas medições experimentais do desempenho de ambos em duas linguagens de programação diferentes segundo diversas métricas.

2 Descrição do Problema

O problema em análise neste relatório é o do produto de matrizes. Dada uma matriz A de dimensão $l \times m$ e uma matriz B de dimensão $m \times n$, o produto de A e B ($A \times B$) é uma matriz C de dimensão $l \times n$ em que cada elemento é dado pela Eq. 1 [1].

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \cdot B_{k,j} \tag{1}$$

Para simplificação do problema, só serão consideradas matrizes quadradas de dimensão $N \times N$ (i.e. neste caso $l = n = m$).

2.1 Algoritmos Analisados

No decorrer deste trabalho foram analisados dois algoritmos diferentes para este problema, que se encontram descritos de seguida. Ambos os algoritmos apresentam complexidade $\mathcal{O}(N^3)$ e realizam um total de $2N^3$ operações em vírgula flutuante (FLOP).

2.1.1 Algoritmo 1

O *Algoritmo 1* consiste na multiplicação sucessiva de linhas da matriz A por colunas da matriz B (ver excerto de código 1). Neste caso, numa iteração do ciclo indexado pela variável i, a linha i da matriz A e todos os elementos da matriz B são lidos (sendo que os acessos a B são feitos por coluna) e uma linha da matriz C é escrita [1].

```
for(i = 0; i < N; i++)
{
        for(j = 0; j < N; j++)
        {
                temp = 0;
                for(k = 0; k < N; k++)
                {
                        temp += A[i][k] * B[k][j];
                }
                C[i][j] = temp;
        }
}
```

Código-fonte 1: Excerto de código do *Algoritmo 1*.

2.1.2 Algoritmo 2

O *Algoritmo 2* assume que os elementos da matriz C estão inicializados a 0 e consiste na multiplicação sucessiva de elementos de uma linha da matriz A por todas as linhas da matriz B (ver excerto de código 2). Neste caso, numa iteração do ciclo indexado pela variável i, a linha i da matriz A e todos os elementos da matriz B são lidos (sendo que os acessos a B são feitos por linha) e uma linha da matriz C é escrita.

```
for(i = 0; i < N; i++)
{
        for(k = 0; k < N; k++)
        {
                for(j = 0; j < N; j++)
                {
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

Código-fonte 2: Excerto de código do *Algoritmo 2*.

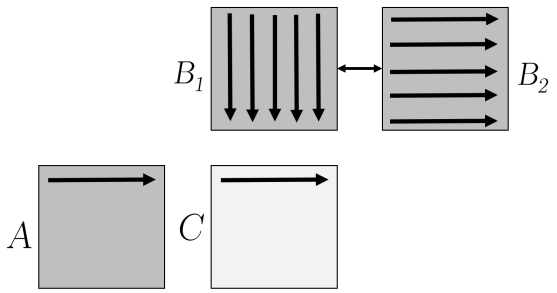

Figura 1: Ilustração representativa da leitura dos valores da matriz A e B (sendo que no *Algoritmo 1* ocorre como representado em B_1 e no *Algoritmo 2* ocorre como em B_2) e leitura/escrita na matriz C durante uma iteração do ciclo indexado pela variável i em ambos os algoritmos.

3 Experiências e Análise dos Resultados

3.1 Descrição das Experiências e Metodologia de Avaliação

Para avaliar cada um dos algoritmos foram realizadas várias experiências com as linguagens C++ e Java e com a biblioteca PAPI em C++, tendo sido medidos o tempo de execução e o número de *data cache misses* nas *caches* L1 e L2. A partir destas medidas foi calculado o número de *data cache misses* por FLOP para L1 e L2 (Eq. 2) e o número de FLOP/s (Eq. 3).

$$Cache\ Misses/FLOP = \frac{Cache\ Misses}{FLOP} = \frac{Cache\ Misses}{2N^3} \tag{2}$$

$$FLOP/s = \frac{FLOP}{Time\ Elapsed\ (s)} = \frac{2N^3}{Time\ Elapsed\ (s)} \tag{3}$$

As experiências foram realizadas num *Desktop* com um CPU Intel Core™ i7-4790 de frequência 3.60GHz (com "Turbo Boost" até 4.00GHz) com 4 *cores*. O CPU apresenta 4 *caches* de dados L1 de 32KB, 4 *caches* de dados/instruções L2 de 256KB (uma por cada *core*), bem como uma *cache* de dados/instruções L3 de 8MB.

3.2 Análise dos Resultados

3.2.1 *Peformance* dos algoritmos em C++ e Java

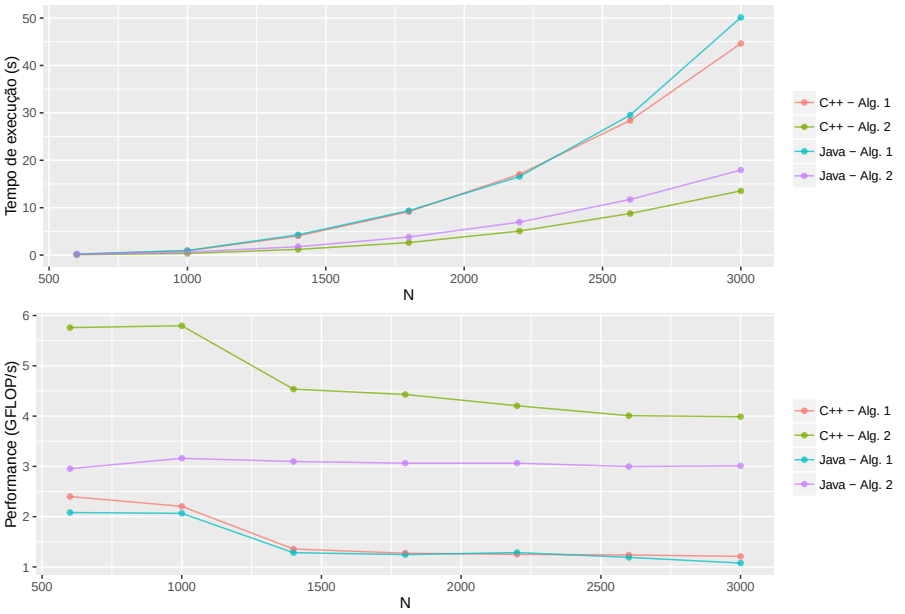

Figura 2: Tempo de execução (gráfico acima) e desempenho (gráfico abaixo) de cada um dos algoritmos, implementados em C++ e Java.

Para qualquer um dos algoritmos, a sua implementação em C++ obteve um desempenho superior à da linguagem Java (mais notória no *Algoritmo 2*). Esta observação vai também de acordo com o esperado, já que geralmente a linguagem Java é de mais lenta execução do que a linguagem C++.

3.2.2 Impacto do uso da *cache* na *performance* dos algoritmos

Na Fig. 2 é possível observar-se as diferenças de *performance* dos dois algoritmos em C++ e Java, permitindo concluir que para a mesma linguagem (seja ela C++ ou Java), o *Algoritmo 2* tem sempre um desempenho melhor do que o *Algoritmo 1*. Isto deve-se ao facto das implementações dos dois algoritmos nas linguagens C++ e Java seguirem uma política *row-major order* de armazenamento das matrizes em memória RAM (i.e. por linha).

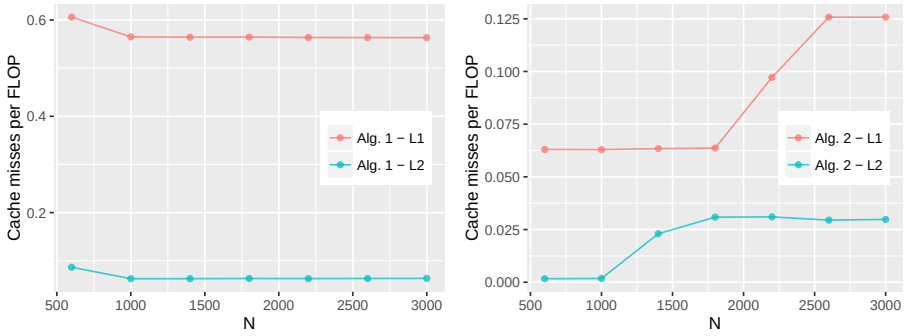

Figura 3: Número de *Data cache misses per FLOP* na *cache* L1 e L2 para os dois algoritmos.

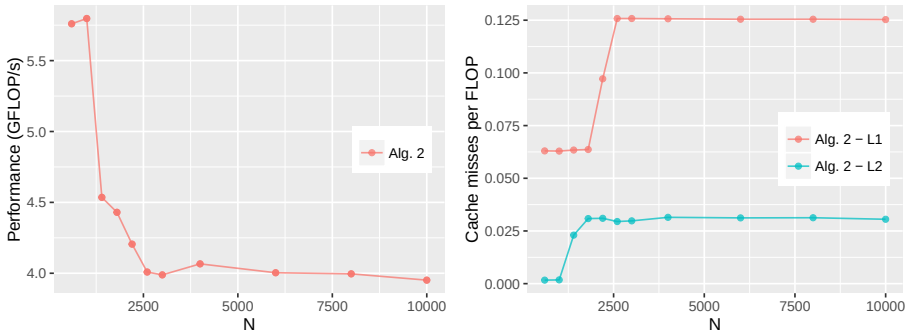

Figura 4: Desempenho (gráfico da esq.) e número de *Data cache misses per FLOP* na *cache* L1 e L2 (gráfico da dir.) para o *Algoritmo 2*.

Sempre que um valor em memória RAM é lido, um bloco de memória contendo não só o valor lido, como também valores armazenados próximos deste são copiados para a *cache* (princípio que é conhecido como o "Principle of Locality" [2]), o que neste caso corresponderão a elementos adjacentes na mesma linha e de linhas anteriores e seguintes. A cópia deste bloco de memória tem como objetivo assegurar um tempo de leitura menor, já que se espera que valores próximos dos lidos anteriormente sejam também lidos num futuro bastante próximo.

Assim, será de esperar que o *Algoritmo 2*, por efetuar a leitura das matrizes linha a linha, apresente uma *performance* mais elevada, o que de facto é comprovado na Fig. 2. A razão desta *performance* mais elevada é visível na Fig. 3, em que o *Algoritmo 2* apresenta um número de *Cache misses per FLOP* muito menor que o *Algoritmo 1*.

Também é de notar uma diminuição notória no desempenho de ambos os algoritmos na Fig. 2 quando o tamanho da matriz ultrapassa um certo valor dentro do intervalo [1000, 1400]. Este decréscimo no desempenho, documentado por J Quirm [1], deve-se ao facto da matriz B ser inteiramente percorrida para cada linha da matriz A, pelo que no final de uma iteração do ciclo indexado pela variável i toda a matriz B terá sido copiada para *cache* (ver Fig. 1). Quando é começada uma nova iteração deste ciclo, se a matriz B for de-

masiado grande, todos os valores desta terão de ser novamente acedidos em memória RAM e copiados para *cache*, o que diminuirá o desempenho do algoritmo. Assumindo que toda a *cache* seria utilizada para albergar a matriz B (o que é falso pois elementos das matrizes A e C também serão armazenados, bem como instruções e outros dados), existem duas capacidades de armazenamento possíveis que dependem da arquitetura do CPU (nos casos em que as *caches* L1, L2 e L3 possuem dados redundantes e outra em que os dados em cada *cache* são mutuamente exclusivos). Dado que apenas uma das *caches* L1 e e uma das *caches* L2 são utilizadas por um *core* do CPU, existe no primeiro caso uma capacidade de armazenamento de 8MB = 8388608 *bytes* e no segundo caso de 32KB + 256KB + 8MB = 8480KB = 8683520 *bytes*. Dado que cada elemento da matriz é um *double* (e ocupa 8 *bytes*), poderemos albergar matrizes B de $N = \sqrt{\frac{8388608}{8}} = 1024$ no primeiro caso e de $N = \sqrt{\frac{8683520}{8}} = 1041$ no segundo caso. Ambos os valores estão entre 1000 e 1400, pelo que a diminuição do desempenho apontada pode ser explicada por este fenómeno.

No caso do *Algoritmo 1*, a partir de $N = 1400$ os valores de *Cache misses per FLOP* L1 e L2 estabilizam nos valores 0.56 e 0.063 respetivamente (ver Fig. 3) e o desempenho estabiliza nos 1.2 GFLOP/s (ver Fig. 2). Na Fig. 4 é mostrado o desempenho (em GFLOP/s) e o número de *Cache misses per FLOP* do *Algoritmo 2* para valores de N entre 600 e 10000, permitindo concluir que a partir de $N = 2600$ os valores de *Cache misses per FLOP* L1 e L2 estabilizam nos valores 0.125 e 0.030 respetivamente e o desempenho estabiliza nos 4.0 GFLOP/s.

3.2.3 Impacto do uso de paralelismo na *performance* dos algoritmos

Para se estudar o efeito da existência de paralelismo na *performance* de ambos os algoritmos, foi utilizada a biblioteca OpenMP para efetuar paralelismo na execução do ciclo *for* indexado pela variável i.

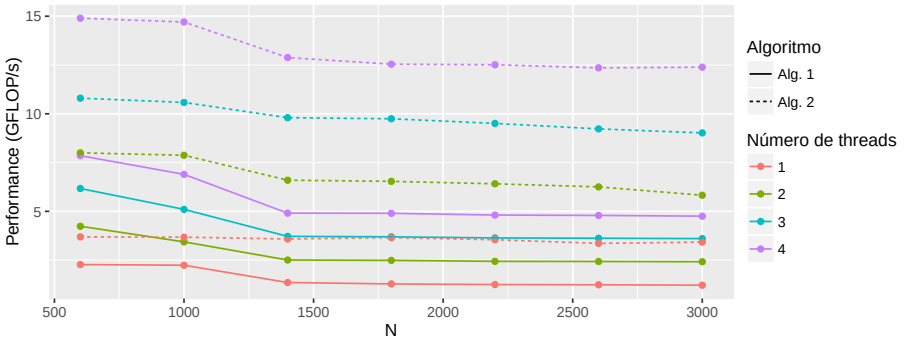

Figura 5: Desempenho de cada um dos algoritmos com um número de *threads* variável.

Como se pode concluir através Fig. 5, o desempenho de cada um dos algoritmos aumenta com o número de *threads*, sendo máximo para $N = 600$ com 7.5 GFLOP/s no *Algoritmo 1* e 15 GFLOP/s no *Algoritmo 2.*

Relativamente à melhoria do desempenho face à versão sequencial (Fig. 6), no caso do *Algoritmo 1* foi obtido um desempenho até 4x superior (obtendo

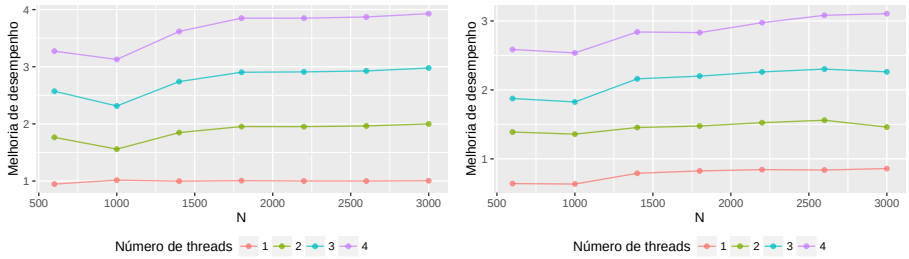

Figura 6: Melhoria de desempenho do *Algoritmo 1* (gráfico da esq.) e do *Algoritmo 2* (gráfico da dir.) com um número de *threads* variável (quociente entre o número de FLOP/s neste caso e no caso sequencial).

neste caso 5 GFLOP/s) e no *Algoritmo 2* até 3x superior (neste caso, de 12.5 GFLOP/s) face à versão sequencial, sendo que estas melhorias de desempenho foram obtidas utilizando 4 *threads* e para $N = 3000$. Em ambos os algoritmos e com um número de *threads* crescente constatou-se que se obtém uma melhoria de desempenho também crescente na versão paralela (sendo que esta também aumenta com o valor de N).

É de notar que, tal como na Fig. 2, existe uma diminuição significativa do desempenho quando o valor de N ultrapassa um certo valor entre 1000 e 1400. Esta diminuição advém do mesmo problema referido no subsecção 3.2.2: cada *thread* coloca a totalidade da matriz B na *cache* a que o *core* onde está a ser executada tem acesso (que pode ser o mesmo onde outra *thread* está a ser executada, dado que o CPU que correu as experiências suporta a execução simultânea de 2 *threads* no mesmo *core*) para cada linha da matriz A, pelo que se a matriz B for demasiado grande, todos os valores desta terão de ser novamente lidos de memória RAM e copiados para *cache*.

4 Conclusões

Este trabalho de análise de dois algoritmos (um *row-major* e outro *column-major*) permitiu aprofundar os conhecimentos da influência da diferente hierarquia dos tipos de memória em diferentes linguagens de programação, num problema aparentemente tão simples quanto o do produto de matrizes. Concluiu-se que na política *row-major order* de armazenamento de *arrays* bidimensionais em memória RAM, o *Algoritmo 2* é em geral o mais eficiente, beneficiando de uma melhoria de desempenho de até 3x superior utilizando paralelismo com 4 *threads* e um máximo de *performance* de 15 GFLOP/s.

Referências

[1] Michael J Quirm. *Parallel Programming in C with MPI and OpenMP*. 2003.

[2] David A Patterson e John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.