Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Performance Evaluation

Grupo:
Henrique Ferrolho - 201202772 - ei12079@fe.up.pt
João Pereira - 201104203 - ei12023@fe.up.pt
Leonel Peixoto - 201204919 - ei12178@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

29 de Março de 2016

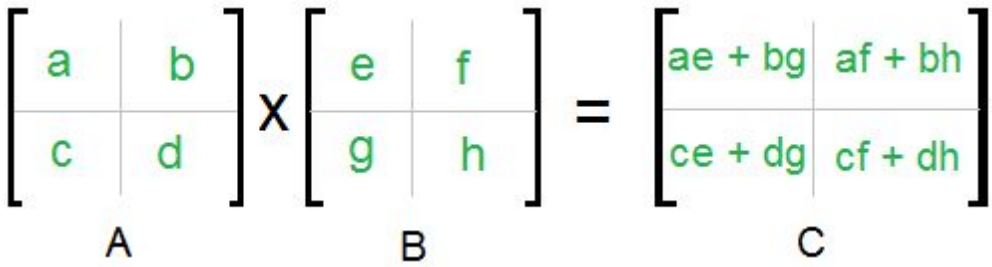
0. Índice

1. Descrição do problema e explicação do algoritmo

No âmbito da unidade curricular de Computação Paralela, foi proposto ao grupo estudar o desempenho de processadores relativamente à hierarquia de memória aquando do acesso a grandes volumes de dados.

Para analisar este problema foi sugerido o uso de um produto de matrizes com dois algoritmos distintos. Um dos algoritmos consiste em multiplicar uma linha da matriz A por cada coluna da matriz B, já o outro, consiste em multiplicar um elemento da matriz A pela linha correspondente da matriz B.

Algoritmo *OnMult*



Este algoritmo baseia-se no método tradicional de multiplicação de matrizes:

- $c_{i,j} = \sum_{k=1}^{m} a_{i,k} * b_{k,j}$, em que $c_{i,j}$ é o resultado do produto matricial entre A e B, m é o tamanho da matriz, $a_{i,k}$ é o elemento da matriz A na linha i e coluna k e $b_{k,j}$ é o elemento da matriz B na linha k e coluna j.
- A sua complexidade temporal é $O(n^3)$.
- O número de instruções aritméticas de vírgula flutuante é $2n^3$, devido às duas instruções - uma soma e uma multiplicação - efetuadas dentro dos ciclos.

Algoritmo *OnMultLine*

O segundo algoritmo a testar implica multiplicar um elemento da matriz A pela linha correspondente na matriz B, isto é, se o elemento for o primeiro elemento da linha na matriz A, multiplicará pela primeira linha na matriz B, se for o segundo elemento da linha na matriz A, então a multiplicação dá-se com a segunda linha da matriz B, e assim adiante.

- A fórmula utilizada neste algoritmo é a seguinte: $c_{i,k} += a_{i,j} * b_{j,k}$, considerando que a ordem dos ciclos é i, j, k, sucessivamente.
- A sua complexidade temporal é $O(n^3)$, semelhante à versão original do algoritmo.

2. Métricas de desempenho e metodologia de avaliação

A lista de métricas de desempenho disponíveis na biblioteca PAPI, para a execução dos algoritmos, era vasta. Assim, o grupo decidiu apenas apresentar resultados com duas delas - ***L1_DCM*** e ***L2_DCM*** - métricas que representam o nível da hierarquia da memória e que fornecem o número de vezes que foi necessário ir ao nível seguinte aceder aos valores em falta. Além destes valores, fez-se também o registo de outras métricas, e que é disponibilizado no ficheiro *Excel* submetido juntamente com este relatório e código fonte.

Outras métricas utilizadas para analisar os resultados foram o **tempo de execução** de cada algoritmo, o **tamanho das matrizes** a multiplicar, o **número de *threads*** utilizadas, e o **número total de instruções** executadas.

O grupo decidiu, ainda, calcular algumas métricas derivadas, entre as quais o cálculo das ***FLOPS*** (*FLoating-point Operations Per Second*), os **rácios** de DCM (*Data Cache Missing*) sobre o número de total de instruções realizadas, possibilitando assim, saber quantas instruções foram usadas para aceder a outro nível da memória para obter valores em falta.

Como metodologias de avaliação, optou-se por usar um **método de comparação** entre os valores obtidos para a execução sequencial e para a execução *multi-threaded*. Estabeleceram-se algumas comparações preponderantes, tais como a comparação dos tempos de execução, das falhas de acesso a valores nos níveis de memória, das operações em vírgula flutuante por segundo, do melhoramento das execuções *multi-threaded* face à execução sequencial, e dos rácios de DCM pelo número de instruções executadas.

Os registos obtidos foram para um processador Intel(R) Quad Core(TM) i7-4700HQ a 2,4GHz, com cache L1 de 4x 32KBytes, L2 de 4x 256KB, L3 de 6MB e serão analisados no ponto seguinte, **Resultados e análise**.

3. Resultados e análise

3.1 Tempos de execução

Visualizando os gráficos em anexo, é possível verificar que qualquer que seja o número de *threads* a executar, a **versão optimizada** apresenta tempos de execução sempre menores que a **versão não optimizada**. Podemos dizer que, para ambos os algoritmos, a versão *C#* é sempre mais lenta que a versão *C++*.
Na versão não optimizada, os tempos para 3 e 4 threads são muito semelhantes; enquanto que na versão optimizada, verifica-se que a versão de 3 threads é mais rápida do que a de 4 threads.

3.2 *MFLOPS*

A **versão original** apresenta valores entre os 800 e os 1800 *MFLOPS*, enquanto que a **versão melhorada** apresenta valores na gama dos ~4k a 12k *MFLOPS*.
Acreditamos que a versão melhorada apresenta um valor muito superior de *FLOPS* devido ao número inferior de acessos aos níveis de memória - e consequentemente o número de operações por segundo é superior.

3.3 Melhoramento relativamente a execução sequencial (*MFLOPS*)

Para uma *thread* em *OpenMP*, não se nota melhoria significativa. Na **versão não optimizada**, para matrizes de dimensão 600, não se notam melhorias independentemente do número de *threads*. Para matrizes com dimensões superiores, nota-se uma clara melhoria para 2, 3, e 4 *threads*. Porém, a diferença entre 3 e 4 *threads* não é muito clara.
Na **versão optimizada** nota-se uma melhoria para qualquer número de *threads*. Contudo, a diferença entre 3 e 4 *threads* continua a ser pouco significativa.
De forma geral, verifica-se que a versão optimizada apresenta melhores resultados do que a versão não optimizada.

3.4 Frequência das *Data Cache Missing*

Neste ponto podemos verificar que existe uma grande diferença na taxa de falhas de acesso aos dados na *cache* entre os dois algoritmos. Isto deve-se à forma como ambos os algoritmos estão implementados.
Na **versão original** do algoritmo para uma *thread*, a frequência de L1 *DCM* é constante para qualquer tamanho das matrizes. Para um valor superior no número

de *threads*, inicialmente não existe L1 *DCM* consideráveis; mas à medida que o tamanho da matriz aumenta, a taxa de L1 *DCM* aumenta.
Quanto à **versão melhorada**, independentemente do número de *threads*, a variação da taxa de **L1 *DCM*** é semelhante entre si. Este fenómeno também é visível nos gráficos de **L2 *DCM*** para as duas versões do algoritmo.

4. Conclusões

Pode-se afirmar que o uso da linguagem *C#* não é vantajoso neste problema, porque corre numa máquina virtual e consequentemente introduz mais *overheads* do que com a linguagem *C++*.

Na versão optimizada, constata-se que não existe melhoria significativa no tempo de execução a partir de 3 *threads*. Esta observação é igualmente coerente com as diferenças pouco perceptíveis entre 3 e 4 *threads* nos resultados de *MFLOPS* e melhoramento.

A utilização da *cache* pelo algoritmo melhorado é mais eficiente do que a versão original. Este facto é corroborado pelos valores dos gráficos de *MFLOPS* e frequências de DCM. Como o algoritmo não necessita de fazer tantos acessos à memória, usa a maior parte do tempo para processar instruções aritméticas de vírgula flutuante - daí o valor mais elevado de *MFLOPS*.

Como foi referido, não houve melhorias significativas entre 3 e 4 *threads*, pelo que segundo a *Lei de Amdahl* pode-se concluir que foi encontrada a máxima melhoria esperada para o algoritmo.

5. Anexos

5.1 Excertos de código

Código utilizado para o algoritmo *OnMult* (versão original C++)

```
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;
        }
}
```

Código utilizado para o algoritmo *OnMult* (verão original C#)

```
for (uint i = 0; i < m_ar; i++)
{
   for (uint j = 0; j < m_br; j++)
   {
      double temp = 0;

      for (uint k = 0; k < m_ar; k++)
         temp += pha[i * m_ar + k] * phb[k * m_br + j];

      phc[i * m_ar + j] = temp;
   }
}
```

Código utilizado para o algoritmo *OnMultLine* (versão melhorada C++)

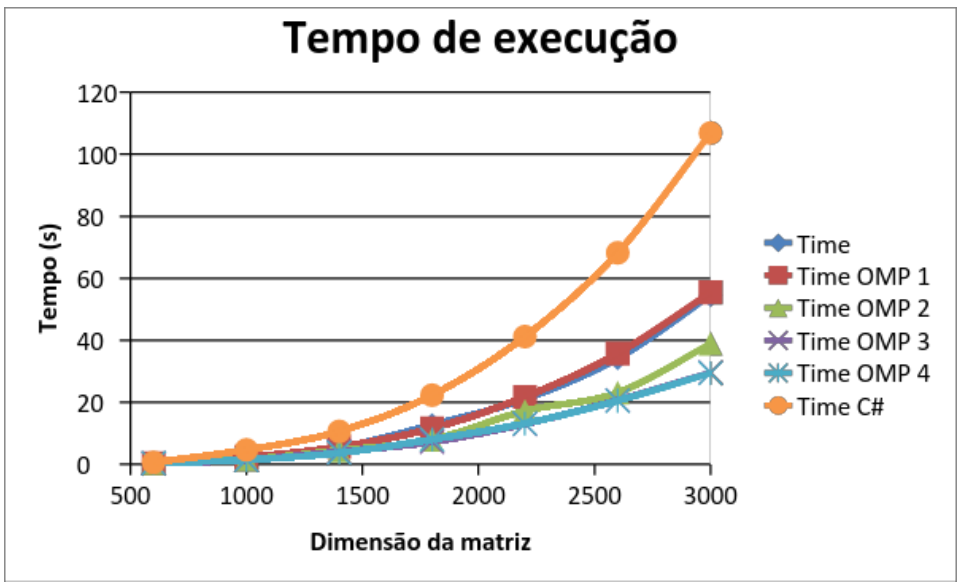
```
for(i=0; i<m_ar; i++) {
        for(j=0; j<m_br; j++) {
                for(k=0; k<m_ar; k++)
                {
                        phc[i*m_ar+k] += pha[i*m_ar+j] * phb[j*m_br+k];
                }
        }
}
```

Código utilizado para o algoritmo *OnMultLine* (versão melhorada C#)

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

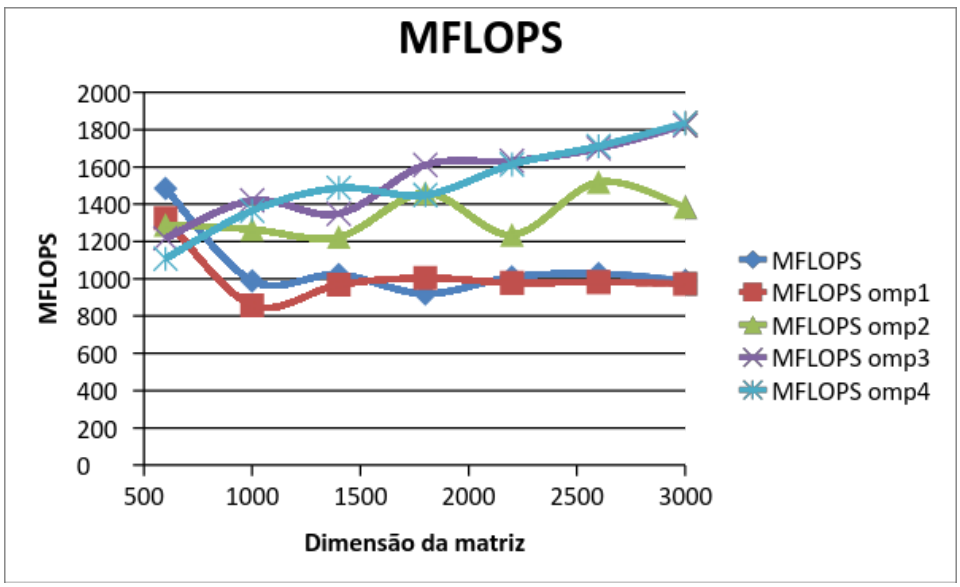
5.2 Gráficos

Tempos de execução do algoritmo *OnMult* (original)

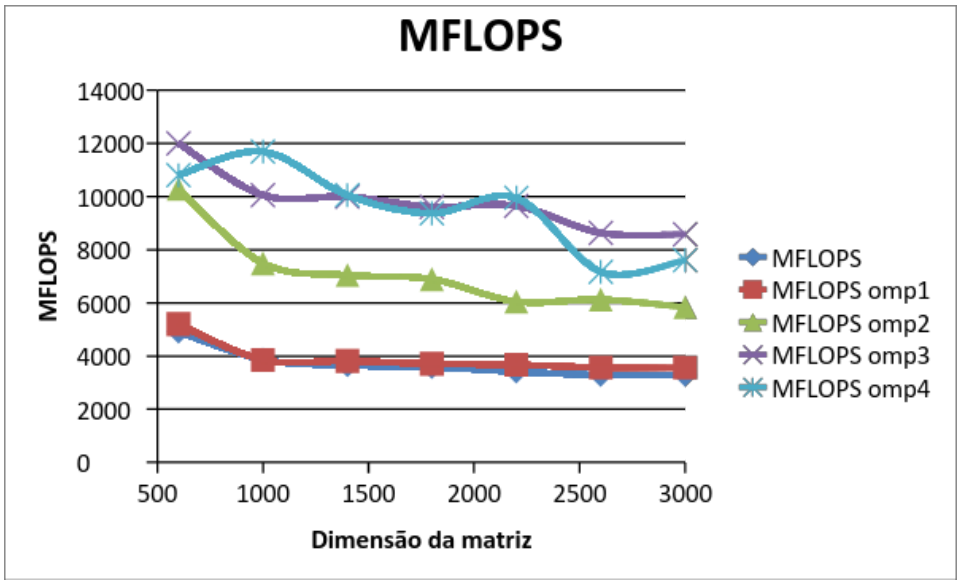


Tempos de execução do algoritmo *OnMultLine* (melhorado)

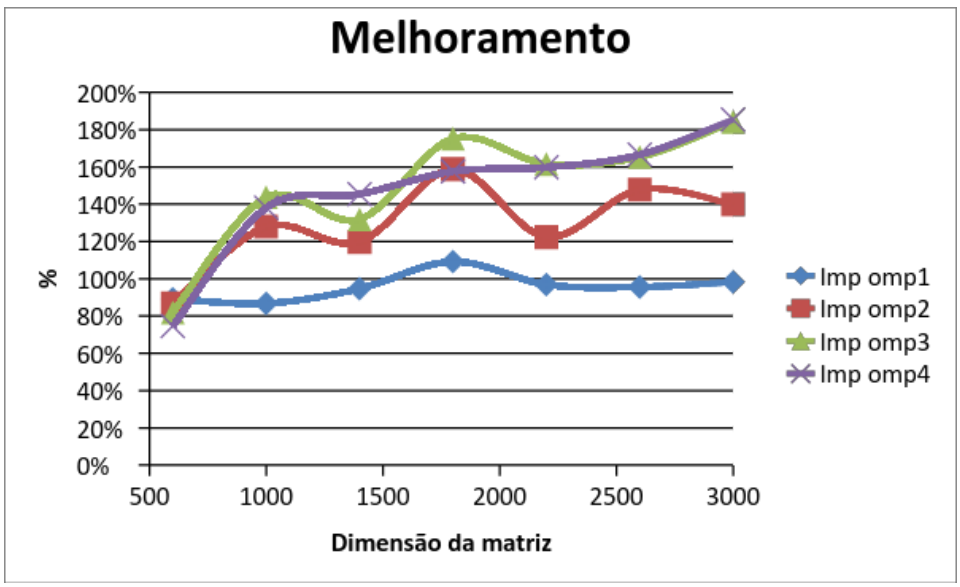
MFLOPS do algoritmo *OnMult* (original)



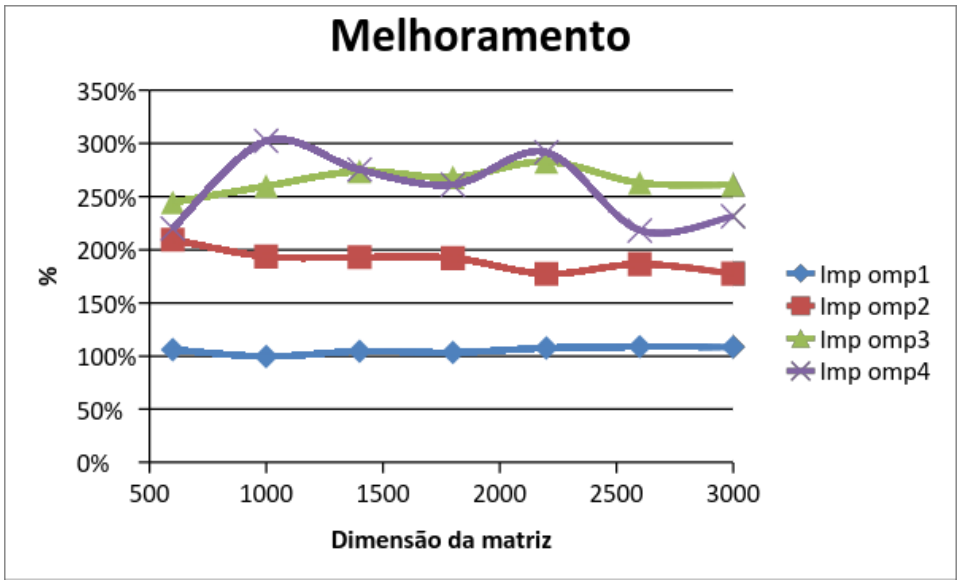
MFLOPS do algoritmo *OnMultLine* (melhorado)



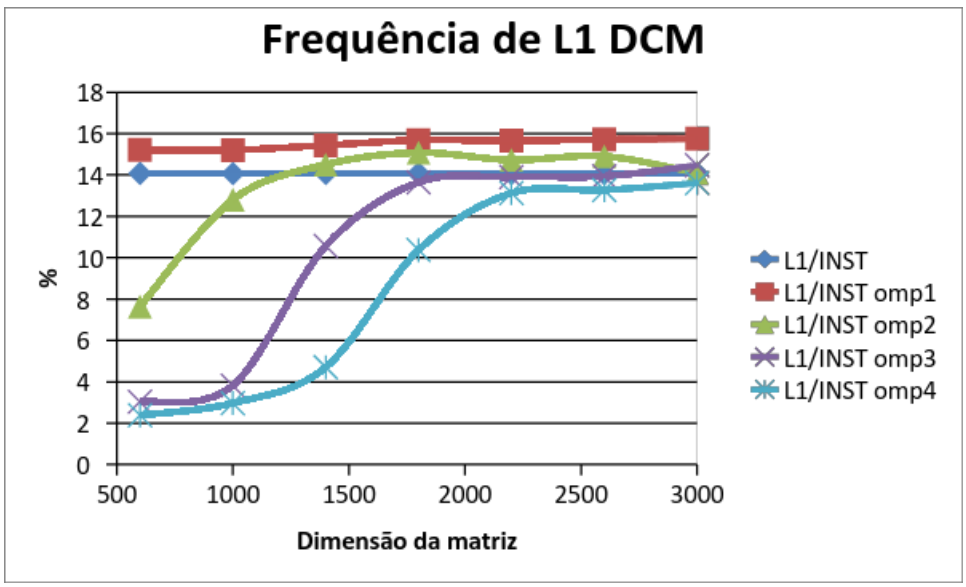
Melhoramento sobre a execução sequencial no algoritmo *OnMult* (original)



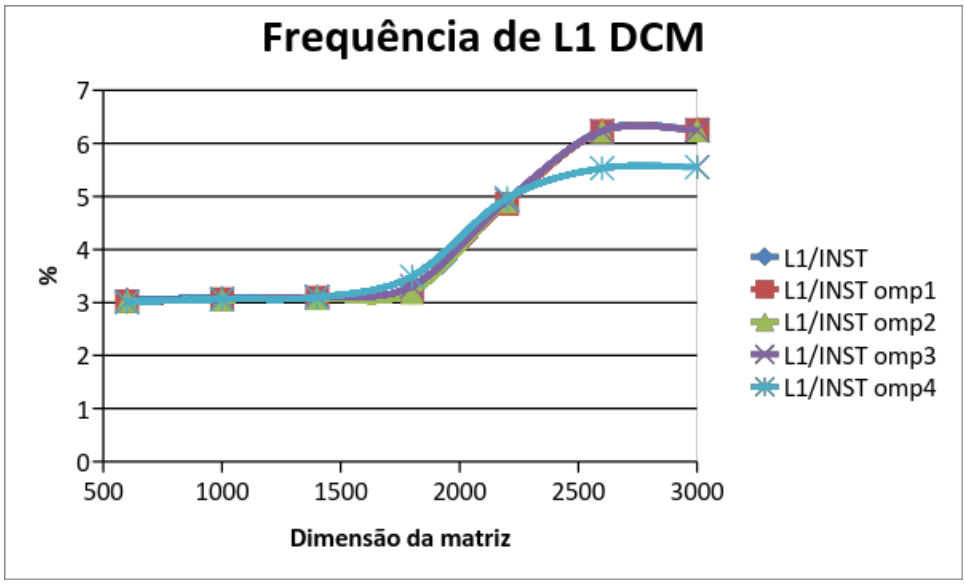
Melhoramento sobre a execução sequencial no algoritmo *OnMultLine* (melhorado)



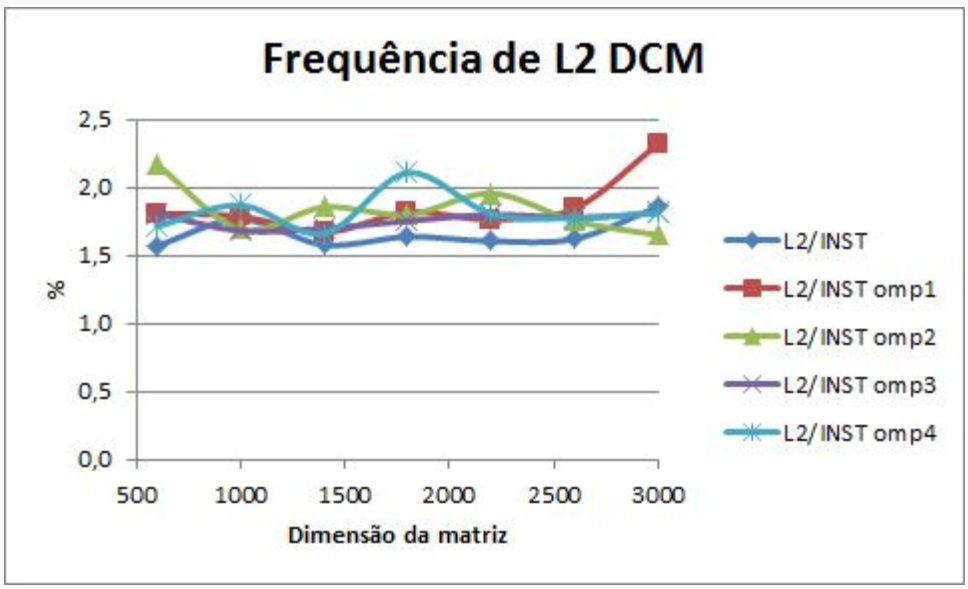
Frequência de L1 *cache misses* no algoritmo *OnMult* (original)



Frequência de L1 *cache misses* no algoritmo *OnMultLine* (melhorado)



Frequência de L2 *caches misses* no algoritmo *OnMult* (original)



Frequência de L2 *cache misses* no algoritmo *OnMultLine* (melhorado)