Computação Paralela e Distribuída

Performance evaluation of a single/multi core

FEUP

Licenciatura em Engenharia Informática e Computação

Turma 11 – Grupo 14

Henrique Gonçalves Graveto Futuro da Silva - up202105647
João Tomás Matos Fernandes Garcia Padrão - up202108766
Tiago da Silva Azevedo - up202108840

Porto, 17 de Março de 2024

Índice

1. Descrição do problema

No âmbito deste primeiro projeto vamos estudar a performance do processador no acesso à memória quando acedemos/processamos grandes quantidades de dados. Para tal, vamos implementar vários algoritmos de multiplicação de matrizes e analisar os resultados obtidos. Estes algoritmos vão ter duas versões, uma onde o algoritmo utiliza apenas um *core*, e outra onde o algoritmo utiliza vários *cores*. Para obter os resultados da *performance* dos diferentes algoritmos, a biblioteca PAPI (Performance API) será utilizada.

2. Explicação dos algoritmos

2.1. Simple Matrix Multiplication

Este algoritmo foi-nos fornecido pelos professores e realiza a multiplicação de matrizes multiplicando a primeira linha da primeira matriz por cada coluna da segunda matriz.

```
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;
    }
}
```

2.2. Line Matrix Multiplication

Este algoritmo realiza a multiplicação de matrizes multiplicando a primeira linha da primeira matriz pela linha correspondente da segunda matriz.

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

2.3. Block Matrix Multiplication

Este algoritmo consiste em dividir as matriz principais em vários blocos, como se tratassem de "sub-matrizes". De seguida, aplicamos o algoritmo descrito na secção **2.2 Line Matrix Multiplication** a cada "sub-matriz".

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

3. Métricas de desempenho

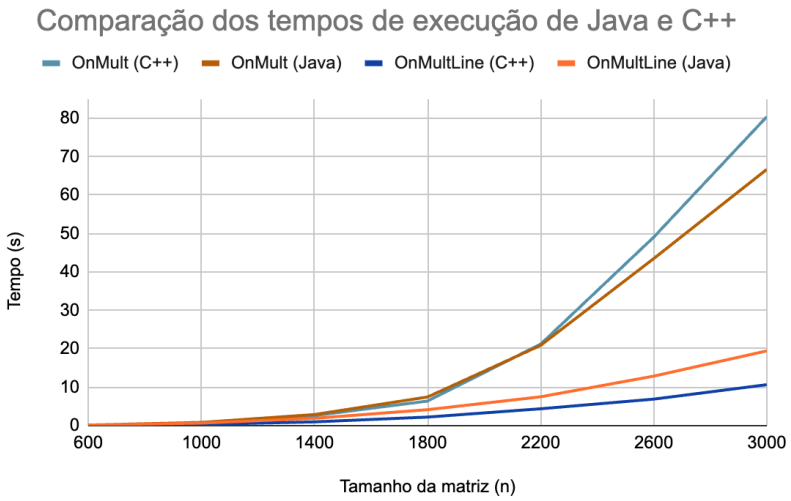
A análise da *performance* dos diferentes algoritmos em C++ foi feita usando a biblioteca PAPI (Performance API) onde se retirou os valores *Level 1 cache misses* e *Level 2 cache misses* do CPU. Estes dados em conjunto com o tempo de execução permitiu comparar assertivamente os diferentes algoritmos, já que um maior número de *cache misses* implica aceder à memória principal mais frequentemente resultando posteriormente no aumento do tempo de execução. Além disso, foi considerado o número de Operações de Ponto Flutuante por segundo, em milhões, (*MFLOP*), o *SPEEDUP* que é uma medida que indica o quão mais rápido um algoritmo pode ser executado em paralelo em comparação com sua execução sequencial e finalmente a *EFFICIENCY* que é uma medida que indica o quão eficiente os recursos do processamento em paralelo estão a ser utilizados em relação ao máximo potencial do desempenho do sistema, dividindo o SPEEDUP pelo número de *cores* do processador.

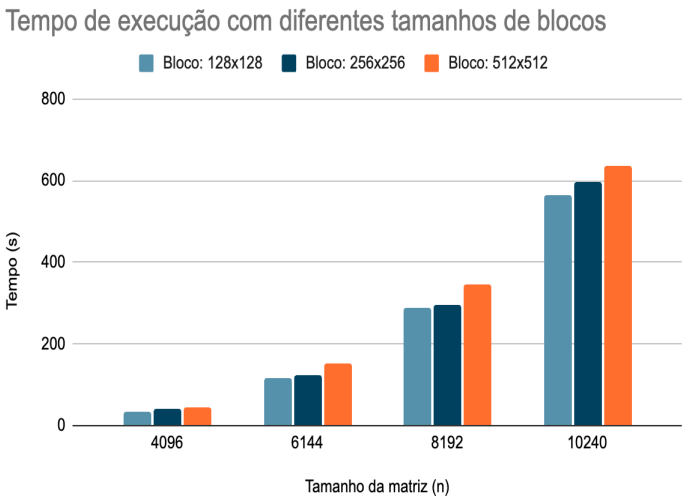
Com o objetivo de manter os dados recolhidos o mais reais possíveis, foi utilizado para todas as medições o mesmo computador. Para cada um dos algoritmos foram feitas trinta medições à exceção dos casos com *"Matrix size* = 8192/10240" onde foram feitas cinco medições permitindo assim fazer uma média ponderada do tempo de execução mais assertiva. O computador do qual foram recolhidos os dados possui o *Ubuntu 22.04.1 LTS* (usando o WSL versão 2), com um processador *AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz,* que possui uma *cache L1* com 512KB, uma cache L2 de 4MB e uma cache L3 compartilhada de 16MB. O processador possui também 8 *cores.*

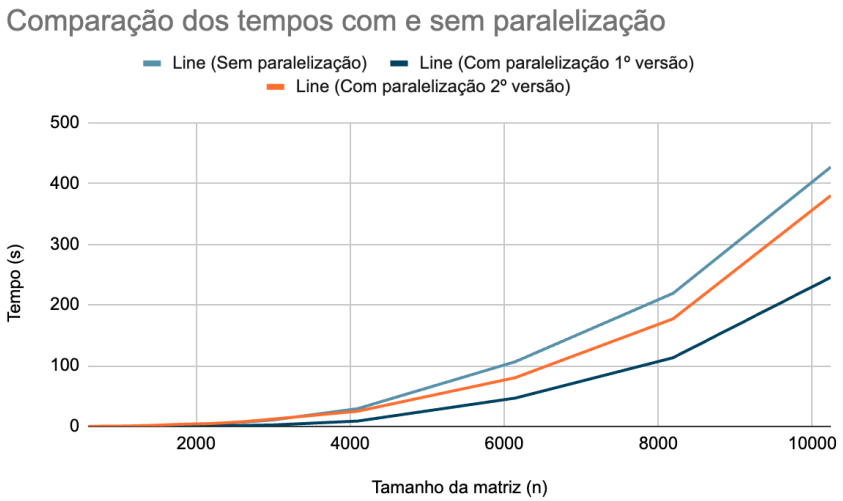
Os algoritmos na versão de C++ foram compilados usando a flag de otimização -O2. Como a biblioteca PAPI (Performance API) foi apenas utilizada na linguagem C++, foi apenas comparado o tempo de execução entre C++ e Java.

4. Resultados e análise

4.1. Tempo de execução dos algoritmos em C++ e Java







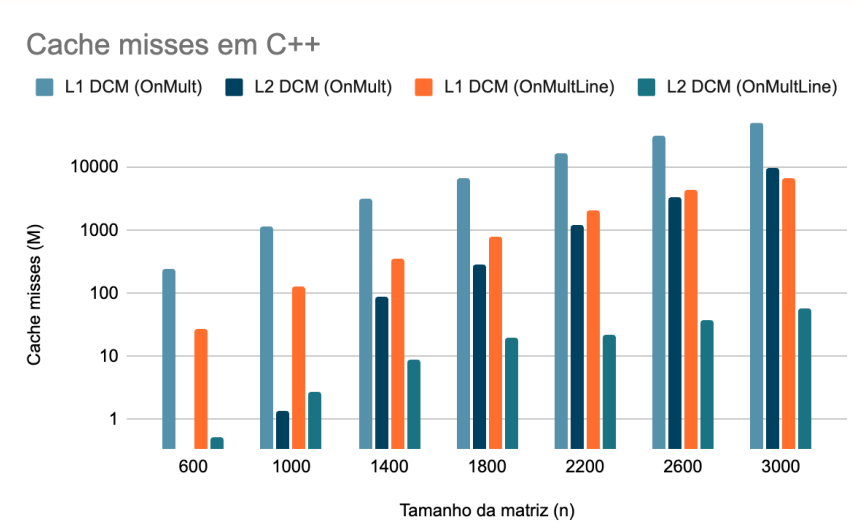
Com base no gráfico da esquerda, podemos observar que em matrizes mais pequenas o tempo de execução é muito semelhante. No entanto, à medida que o tamanho da matriz aumenta, podemos observar que o algoritmo apresentado na secção **2.2 Line Matrix Multiplication** possui um tempo de execução muito menor do que o algoritmo apresentado na secção **2.1 Simple Matrix Multiplication**. Esta diferença pode ser justificada pela proximidade dos valores que estão guardados na memória. Como o segundo algoritmo multiplica "linha por linha", estamos a aceder a dados sequenciais na memória, enquanto que, no primeiro algoritmo, os dados não são contínuos uma vez que este multiplica "linha por coluna", implicando um maior tempo de execução do algoritmo.

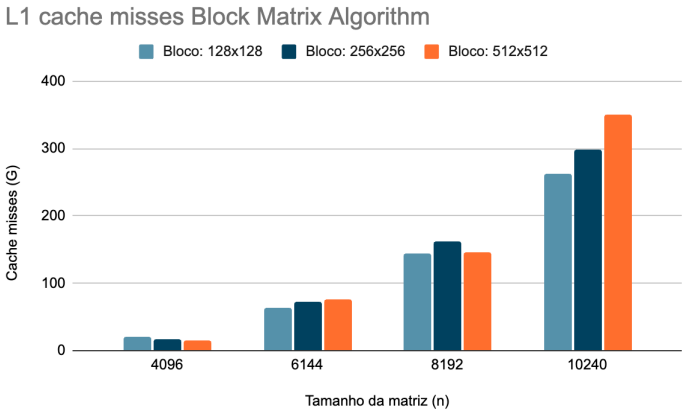
Em relação ao tempo de execução das diferentes linguagens, podemos observar que os tempos de execução são semelhantes, tanto para o primeiro algoritmo como para o segundo. A divergência que se verifica quando o tamanho da matriz é maior de 2200 pode ser causado pelo facto de Java ser uma linguagem compilada e interpretada, uma vez que o código é compilado para *byte-codes* e depois é interpretado pela JVM (Java *Virtual Machine)*.

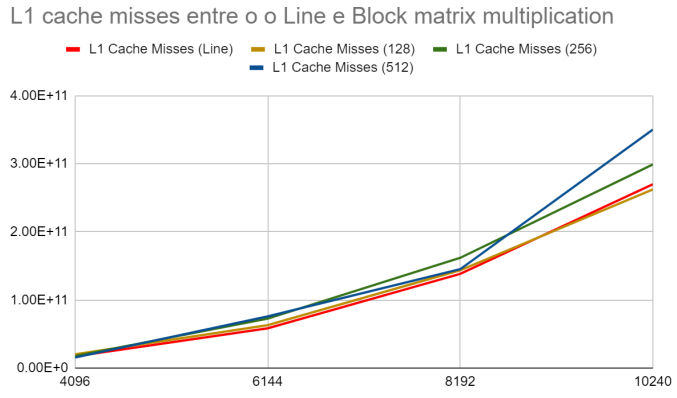
Ao observar o gráfico da direita, podemos verificar que os tempos de execução são muito semelhantes entre si. No entanto, à medida que o tamanho da matriz aumenta, verificamos que o bloco 128x128 começa a ser ligeiramente melhor em relação aos outros blocos.

Ao observar o último gráfico, podemos verificar que a primeira versão do algoritmo com paralelização foi o mais rápido. Isto deve-se ao facto de a primeira versão paralelizar um "*outer loop*", dividindo assim mais trabalho pelas diferentes *threads*. Cada *thread* vai ser responsável por calcular múltiplas linhas da matriz, resultando numa melhor utilização de cada *thread*. Na segunda versão estamos a criar vários *threads* "nested". Isto faz com que os *threads* do "*inner loop*" tenham de ser criados várias vezes, o que vai adicionar mais *overhead* devido à sincronização das *threads*.

4.2. *L1 e L2 cache misses nos algoritmos sem paralelização*

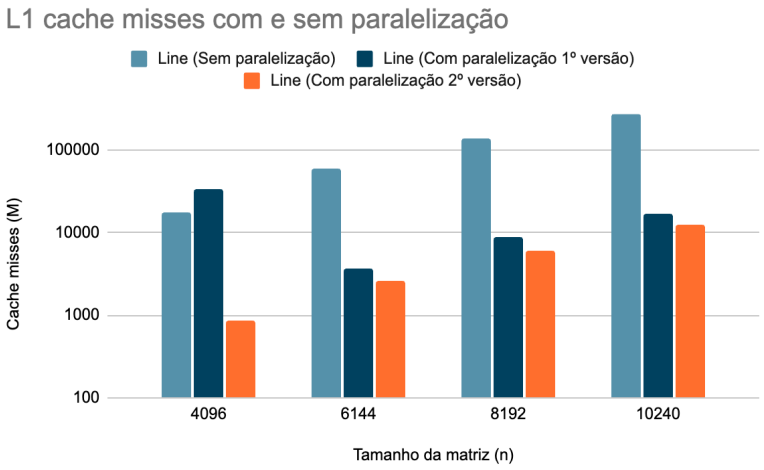


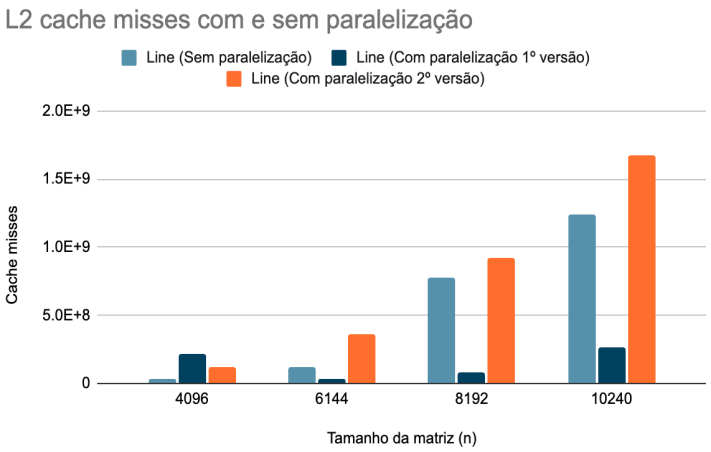




Ao observar o primeiro gráfico, podemos verificar que o segundo algoritmo (*Line Matrix Multiplication*) apresenta um número muito inferior de L1 e de L2 *cache misses*, reforçando a justificação apresentada no ponto 4.1. Em relação ao algoritmo de multiplicação por blocos, podemos verificar que o número de L1 cache misses segue uma estrutura bastante semelhante ao tempo de execução do algoritmo. Analisando o gráfico da direita, podemos observar que o número de *L1 cache misses* do segundo algoritmo é maioritariamente inferior ao último algoritmo e consequentemente tem um menor tempo de execução.

4.3. *Comparação das métricas de desempenho dos algoritmos com e sem paralelização*





Com base nos dois primeiros gráficos, podemos verificar que o algoritmo sem paralelização possui mais L1 *cache misses*. Ao analisar as versões com paralelização, podemos verificar que a primeira versão possui um número de L1 cache misses ligeiramente maior do que a segunda versão. Isto deveria fazer com que o tempo de execução fosse maior, no entanto, ao observar o número de L2 cache misses, podemos observar que a primeira versão possui um número de L2 *cache misses* muito inferior à segunda versão. Esta diferença de *L2 cache misses* pode explicar a diferença do tempo de execução, pois aceder à L3 cache demora mais tempo do que aceder à L2 cache, resultando numa penalização maior no tempo de execução.

Size (n)	Time (s)	MFlops
600	0,064	6707
1000	0,293	6820
1400	0,970	5654
1800	2,217	5259
2200	4,378	4863
2600	6,887	5103
3000	10,644	5072
4096	29,168	4711
6144	106,519	4354
8192	219,070	5018
10240	426,906	5030

Tabela 1 - Métricas do algoritmo Line Matrix Multiplication sem paralelização

Size (n)	Time (s)	MFlops	Speedup	Efficiency (%)
600	0,023	18572	2,76	34,61
1000	0,085	23433	3,43	42,94
1400	0,248	22041	3,89	48,72
1800	0,458	25430	4,83	60,44
2200	0,823	25857	5,31	66,45
2600	1,545	22739	4,45	55,69
3000	2,397	22522	4,43	55,49
4096	8,788	15638	3,31	41,48
6144	46,643	9944	2,28	28,54
8192	113,037	9727	1,93	24,22
10240	245,421	8750	1,73	21,74

Tabela 2 - Métricas do algoritmo com a primeira versão de paralelização

Size (n)	Time (s)	MFlops	Speedup	Efficiency (%)
600	0,211	2045	0,30	3,8
1000	0,534	3745	0,54	6,8
1400	1,542	3557	0,62	7,86
1800	3,498	3334	0,63	7,92
2200	4,737	4495	0,92	11,55
2600	7,613	4617	0,90	11,30
3000	12,133	4450	0,87	10,96
4096	24,961	5505	1,16	14,60
6144	80,253	5779	1,32	16,59
8192	177,085	6208	1,23	15,46
10240	380,039	5650	1,12	14,04

Tabela 3 - Métricas do algoritmo com a segunda versão de paralelização

Ao analisar as métricas obtidas, podemos verificar que a primeira versão do algoritmo com paralelização chega a ter uma eficiência de 66% e um speedup de 5,3x em relação ao algoritmo sem paralelização. Podemos ainda verificar que o número de MFlops da primeira versão é muito superior do que as outras versões.

Em relação à segunda versão, podemos observar que o algoritmo possui um desempenho pior em relação ao algoritmo sem paralelização. Isto deve-se ao *overhead* introduzido pela sincronização das threads do "*inner loop*". No entanto, para matrizes com maior tamanho, podemos verificar que esta versão consegue ter um desempenho ligeiramente superior, chegando a ter um *speedup* de 1.3x e uma eficiência de 16%.

5. Conclusões

Com a realização deste trabalho verificamos a importância da computação paralela no tratamento/processamento de grandes quantidades de dados. Foi ainda possível perceber a importância de como uma boa gestão no acesso à memória melhora a *performance* de um programa.

6. Anexos

6.1. Métricas do algoritmo Simple Matrix Multiplication

Size	Time	MFlops
600	0,149	2896
1000	0,794	2520
1400	2,610	2103
1800	6,385	1827
2200	21,185	1005
2600	49,039	717
3000	80,390	672

6.2. Métricas do algoritmo Line Matrix Multiplication sem paralelização

Size	Time	MFlops
600	0,064	6708
1000	0,293	6821
1400	0,971	5655
1800	2,218	5259
2200	4,379	4864
2600	6,887	5104
3000	10,645	5073
4096	29,169	4712
6144	106,519	4355
8192	219,070	5019
10240	426,906	5030

6.3. Métricas da primeira versão do algoritmo Line Matrix Multiplication com paralelização

Size	Time	MFlops	Speedup	Efficiency
600	0,0232601	18573	2,77	34,61%
1000	0,0853488	23433	3,44	42,95%
1400	0,248986	22041	3,90	48,72%
1800	0,45867	25430	4,84	60,44%
2200	0,823575	25858	5,32	66,46%

2600	1,54583	22740	4,46	55,69%
3000	2,39757	22523	4,44	55,50%
4096	8,78878	15638	3,32	41,49%
6144	46,6436	9945	2,28	28,55%
8192	113,037	9727	1,94	24,23%
10240	245,421	8750	1,74	21,74%

6.4. Métricas da segunda versão do algoritmo Line Matrix Multiplication com paralelização

Size	Time	MFlops	Speedup	Efficiency
600	0,211	2045	0,30	3,81%
1000	0,534	3745	0,55	6,86%
1400	1,543	3557	0,63	7,86%
1800	3,498	3334	0,63	7,92%
2200	4,737	4496	0,92	11,55%
2600	7,613	4617	0,90	11,31%
3000	12,133	4451	0,88	10,97%
4096	24,962	5506	1,17	14,61%
6144	80,254	5780	1,33	16,59%
8192	177,085	6209	1,24	15,46%
10240	380,039	5651	1,12	14,04%

6.5. Métricas do algoritmo Block Matrix Multiplication com block size 128

Size	Time	MFlops
4096	35,522	3869
6144	115,691	4009
8192	288,961	3805
10240	565,904	3795

6.6. Métricas do algoritmo Block Matrix Multiplication com block size 256

Size	Time	MFlops

4096	41,879	3282
6144	123,381	3760
8192	297,094	3701
10240	596,480	3600

6.7. Métricas do algoritmo Block Matrix Multiplication com block size 512

Size	Time	MFlops
4096	45,354	3030
6144	152,208	3048
8192	347,162	3167
10240	635,541	3379