Universidade do Porto

FEUP Faculdade de Engenharia

Mestrado Integrado em

Engenharia Informática e Computação

Computação Paralela (CPAR)

Multiplicação de Matrizes

Relatório

Daniel Reis - up201308586 (up201308586@fe.up.pt)

José Mendes - up201304828 (up201304828@fe.up.pt)

Faculdade de Engenharia da Universidade do Porto Rua Roberto Frias, s/n,

4200-465 Porto, Portugal

13 de Março de 2017

Índice

1 Descrição do Problema

No decorrer da Unidade Curricular de Computação Paralela, foi proposto o estudo do desempenho do processador e a gestão de memória deste.

Foi pedido para avaliar de que forma diferentes tipos de acesso a memória podem afetar o desempenho de um algoritmo e de que forma se pode aumentar a *performance* através de programação paralela.

De forma a realizar este estudo, recorremos ao PAPI (***P**erformance **A**pplication **P**rogramming **I**nterface*), uma biblioteca capaz de aceder aos contadores de desempenho dos microprocessadores para obter o número de falhas de acesso a cache de nível 1 e de nível 2. Além essa API (***A**pplication **P**rogramming **I**nterface*), foi utilizado também o Tiptop para medir os recursos que estavam a ser utilizados aquando a execução do programa e outros detalhes relevantes, como o número de instruções por ciclo e a percentagem de falhas de acesso à cache. Para terminar, recorremos também ao OpenMP (***Open M**ulti-**P**rocessing*), uma API que permite programação multi-processo de *shared memory* e simultaneidade ao programa.

Para este estudo foi inicialmente facultado pelo professor da unidade curricular um algoritmo em C/C++ capaz de realizar a multiplicação de duas matrizes quadradas.

Na primeira fase do projeto e seguindo a especificação do enunciado, foi pedido para elaborar o mesmo algoritmo providenciado pelo professor numa linguagem à nossa escolha. Foram calculados os tempos de execução do mesmo algoritmo (C/C++ e Java) com matrizes de diferentes tamanhos sendo estes tempos devidamente registados assim como também os indicadores de performance mais relevantes.

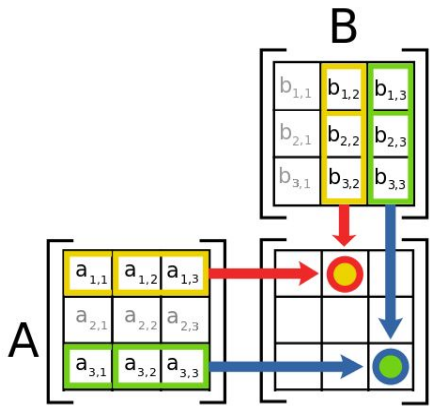
Numa segunda fase, foi pedido para implementar uma versão do algoritmo capaz de multiplicar um elemento da primeira matriz pela linha correspondente da segunda matriz. Ou seja, alterar a implementação do primeiro algoritmo para realizar multiplicação em linha em vez por coluna. Repetindo o procedimento da primeira fase, foram recolhidos tempos de processamento e indicadores de performance dos dois algoritmos (C/C++ e Java) novamente com matrizes de diferentes tamanhos.

Já na terceira e última fase do projeto, foi pedido para implementar as duas versões dos algoritmos (multiplicação da matriz em coluna e em linha) recorrendo a programação paralela com matrizes de diferentes tamanhos assim como diferentes números de *threads* registando todos os dados de performance e tempos relevantes como realizado na primeira e segunda fase do estudo. Para conseguir alcançar este parelismo foi usado o OpenMP.

2 Explicação dos Algoritmos

No desenvolvimento deste projeto foram usados vários algoritmos, tal como a multiplicação de matrizes em coluna e a multiplicação de matrizes em linha.

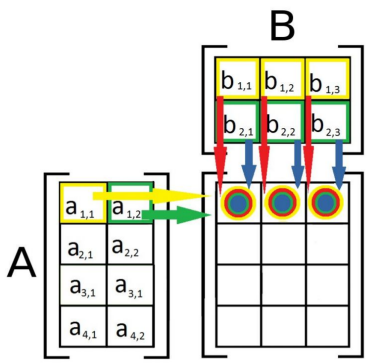
Na multiplicação de matrizes em coluna, multiplicamos a linha da matriz A pela coluna matriz B como é possível observar na figura em baixo:




```
For i:= 0 to N DO
    For j:= 0 TO N DO
        BEGIN
            SUM:=0;
            For k:= 0 TO N DO
            SUM:= SUM + A[i,k] * B[k,j];
            C[i,j]:= SUM;
        END;
```

$$C_{1,2} = a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2} + a_{1,3} * b_{3,2}$$
$$C_{3,3} = a_{3,1} * b_{1,3} + a_{3,2} * b_{2,3} + a_{3,3} * b_{3,3}$$
$$C_{i,k} = \Sigma_1^n (a_{i,n} * b_{n,k})$$

O outro método é ligeiramente diferente. A multiplicação de matrizes em linha multiplica um elemento de uma coluna da matriz A pela linha da matriz B.




```
For i:= 0 to N DO
    For k:= 0 TO N DO
        BEGIN
            SUM:=0;
            For j:= 0 TO N DO
            SUM:= SUM + A[i,k] * B[k,j];
            C[i,j]:= SUM;
        END;
```

$$C_{1,2} = a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2} + a_{1,3} * b_{3,2}$$
$$C_{3,3} = a_{3,1} * b_{1,3} + a_{3,2} * b_{2,3} + a_{3,3} * b_{3,3}$$
$$C_{i,j} = \Sigma_1^n (a_{i,k} * b_{k,j})$$

Analisando estes dois algoritmos, a implementação destes varia apenas no segundo ciclo *for* onde existe uma troca entre do terceiro ciclo com o segundo ciclo *for*.

A nível computacional esta troca tem um grande impacto na performance do algoritmo pois permite com que a multiplicação das matrizes seja feita em linha. Todos os

elementos da linha da matriz B são carregados para a memória cache e acedidos logo de seguida, evitando a re-escrita em cache.

Por outro lado, o primeiro método tem uma grande desvantagem, que é percorrer os elementos da matriz B em coluna, pois quando os elementos são carregados para a cache, toda a linha da matriz é carregada, ou seja, ao carregar cada elemento da coluna, toda a linha é carregada.

Além destes algoritmos, também aplicamos computação paralela a cada um deles.

3 Métricas de Desempenho

As métricas usadas para comparar o desempenho dos diferentes algoritmos foram as seguintes:

- Tempo de Execução
- Número de *Data Cache Misses* (DCM) da cache L1 e L2
- Número de Instruções por Ciclo (IPC)
- Percentagem de CPU utilizada
- Percentagem de falhas de cache

A forma de avaliação dos diferentes algoritmos consistiu, maioritariamente, na comparação entre as métricas dos mesmos.

Características do computador usado para a análise:

- Processador: Intel® Core™ i7-4790 Processor @ 3.60 GHz
- RAM: 16 GB
- Número de Cores: 4
- Número de Threads: 8
- 8 MB Cache

4 Resultados e Análises

De forma a medirmos todos os algoritmos usados, o programa foi corrido com *inputs* diferentes, isto é, com matrizes de tamanho diferentes. O programa foi testado com matrizes de tamanho 600x600 até 3000x3000, com um intervalo de 400. Além disso, no terceiro exercício, que envolve computação paralela, o programa foi corrido com 1, 2, 3 e 4 *threads*.

Por multiplicação normal, entende-se multiplicação das matrizes em coluna. Todos os valores anotados não presentes nos gráficos nesta secção encontram-se na secção dos Anexos.

4.1 Primeiro Teste

No primeiro teste, fizemos o cálculo do produto matricial em coluna de duas matrizes quadradas nas linguagens de C++ e java. Como se pode observar com a análise dos gráfico abaixo apresentado a execução do algoritmo em C++ foi mais rápida que a do algoritmo em Java. Isto deve-se ao facto das aplicações Java geralmente correrem em JVM

(*Java Virtual Machine*) enquanto que programas em C++ correm sem necessidade de recorrerem a aplicações externas. Além disso, a linguagem Java contém várias facilidades de programação, tal como o *Garbage Collector*, que pioram o desempenho do programa.

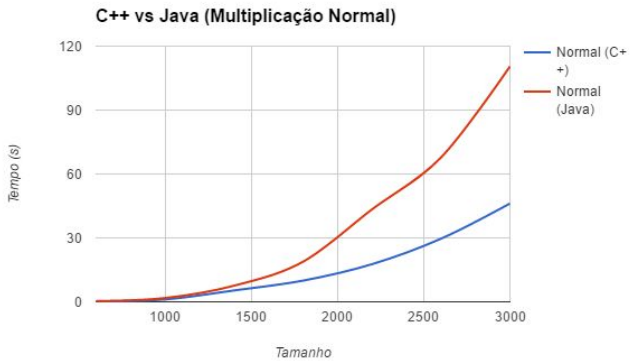


Figura 1: Produto matricial em coluna em C++ e Java

4.2 Segundo Teste

No segundo teste, foi implementado um algoritmo melhorado da primeira versão que calcula o produto matricial em linha. Como se pode observar com os gráficos em baixo, a multiplicação em linha é bastante mais rápida que a multiplicação em coluna. Isto deve-se ao facto de os algoritmos interagem de diferentes formas a memória cache. Como o processador tem de carregar para memória o valor da linha, se usarmos o algoritmo de multiplicação em coluna, ele carregará para memória apenas o primeiro valor e prossegue para a próxima iteração. Na multiplicação em linha isto já não acontece. Pois o algoritmo usa todos os valores que carregou previamente para memória e usa-os até passar para a próxima iteração. Este processo segue-se para as restantes colunas da matriz.

Foram também registados os tempos de processamento de matrizes de tamanhos entre 3000x3000 e 10000x10000 com incremento de 2000.

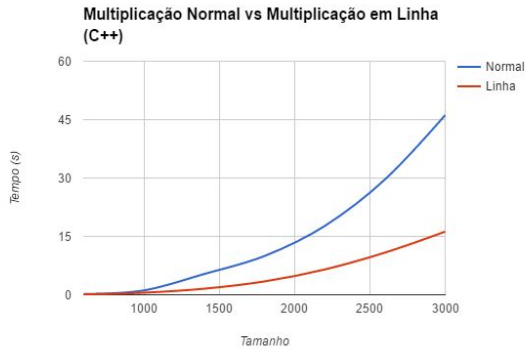


Figura 2: Produto matricial em coluna e linha em C++

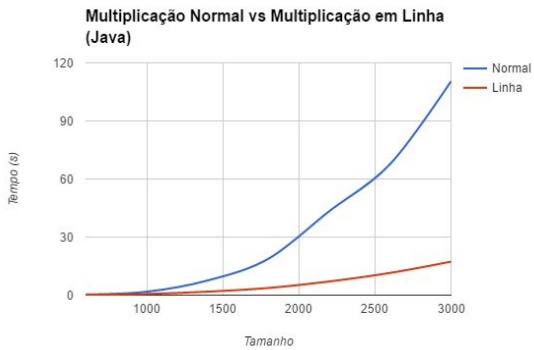


Figura 3: Produto matricial em coluna e linha em Java

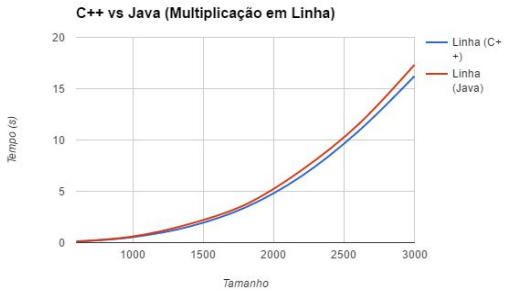


Figura 4: Produto matricial em linha em C++ e Java

4.3 Terceiro Teste

No terceiro e último teste, foi testada a computação paralela utilizando o algoritmo de multiplicação em coluna e em linha, em C++. Na multiplicação em coluna ocorreu o que seria esperado, o aumento do número de *threads* diminui o tempo de execução.

Ao observar os resultados deste teste, pudemos observar que existe um ponto em que deixa de compensar o recurso a programação paralela. Tudo depende dos dados de *input* para o problema. No caso em questão e nos intervalos considerados, não compensa o recurso de 4 *threads.* A diferença entre 3 e 4 threads é mínima e isto pode dever-se principalmente ao facto do tamanho das matrizes testadas não ser suficientemente significativo, o que pode ser explicado pelo facto de existir sempre um custo para a criação de threads.

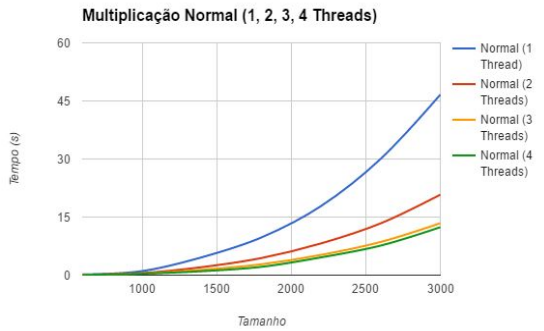


Figura 5: Produto matricial em coluna com 1, 2, 3 e 4 threads threads

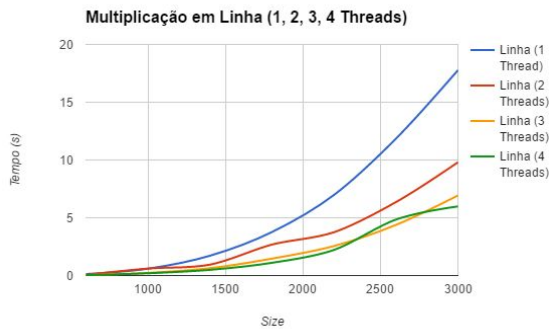


Figura 6: Produto matricial em linha com 1, 2, 3 e 4 threads

5 Conclusões

A realização deste estudo foi bastante importante para a compreender as vantagens e quando recorrer a programação paralela.

Para paralelizar algoritmos devemos partir sempre de uma solução sequencial ótima de forma a obter resultados significativos, mas é também necessário inferir primeiramente se é realmente vantajoso implementar programação paralela. O *overhead* na programação paralela pode mesmo tornar a execução do algoritmo mais lenta caso o tamanho de dados seja demasiado pequeno. Sendo assim, a quantidade de dados a processar e forma como estes se processam é determinante para saber de que forma se pode aumentar a performance.

Ao analisar todos os testes corridos, retiramos que o algoritmo de multiplicação em linha é bastante vantajoso, quer nas falhas no acesso a cache, quer no tempo de execução. Além disso, nota-se um aumento substancial no número de instruções por ciclo também neste algoritmo.

6 Referências

- Tiptop - http://tiptop.gforge.inria.fr/
- Documentação do Tiptop - https://hal.inria.fr/hal-00639173/document
- Documentação do OpenMP - http://www.openmp.org/
- Tutorial de OpenMP - http://bisqwit.iki.fi/story/howto/openmp/
- Noções sobre Matrizes e Sistemas de Equações Lineares, José Trigo Barbosa, ISBN: 9789727521371
- C vs Java - http://stackoverflow.com/questions/418914/why-is-c-so-fast-and-why-arent-other-languages-as-fast-or-faster

7 Anexos

Exercício 1

C++

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.184	244 575 885	27 402 373
1000x1000	1.125	1 128 891 591	126 660 322
1400x1400	5.338	3 095 257 735	370 080 965
1800x1800	9.951	6 580 842 354	739 360 644
2200x2200	17.669	12 000 252 694	1 342 455 140
2600x2600	29.625	19 801 456 753	2 229 455 927
3000x3000	46.148	30 412 254 313	3 690 152 512

%CPU: 100%
%SYS: 0%
P: 6
Mcycle: 7900
Minstr: 8100
IPC (instructions per cycle): 1.03
%MISS: 1.8%
%BMIS: 0.00%
%BUS: 0.0%

Java

Size	Result (s)
600x600	0.358
1000x1000	1.779
1400x1400	7.572
1800x1800	18.814
2200x2200	43.394

2600x2600	67.723
3000x3000	110.485

%CPU: 100%
%SYS: 0%
P: 5
Mcycle: 7950
Minstr: 5425
IPC (instructions per cycle): 0.68
%MISS: 1.13%
%BMIS: 0.00%
%BUS: 0.0%

Exercício 2

C++

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.101	27 135 987	726 524
1000x1000	0.532	125 599 703	6 272 849
1400x1400	1.560	345 144 082	76 550 596
1800x1800	3.427	738 774 150	239 948 967
2200x2200	6.494	2 064 578 960	452 741 028
2600x2600	10.826	4 399 979 387	747 577 595
3000x3000	16.220	6 788 624 188	1 165 225 520

%CPU: 100%
%SYS: 0.0%
P: 2
Mcycle: 7952.05
Minstr: 22851.71
IPC (instructions per cycle): 2.87
%MISS: 0.21%
%BMIS: 0.00%
%BUS: 0.0%

Java

Size	Result (s)

600x600	0.122
1000x1000	0.599
1400x1400	1.807
1800x1800	3.704
2200x2200	7.071
2600x2600	11.490
3000x3000	17.325

Não foi possível usar o tiptop para recolher dados de desempenho devido ao código ser executado em JVM (Java Virtual Machine)

C++

Size	Result (s)	L1 DCM	L2 DCM
4000x4000	38.842	16090443297	2887666166
6000x6000	133.976	54194774619	9729601353
8000x8000	309.961	128595270005	23242019365
10000x10000	610.732	250605241623	45696875212

%CPU: 100%
%SYS: 0.0%
P: 6
Mcycle: 7 925
Minstr: 22 500
IPC (instructions per cycle): 2.82
%MISS: 0.21%
%BMIS: 0.00%
%BUS: 0.0%

Java

Size	Result (s)
4000x4000	40.394
6000x6000	131.021
8000x8000	331.489
10000x10000	644.571

Não foi possível usar o tiptop para recolher dados de desempenho devido ao código ser executado em JVM (Java Virtual Machine)

Exercício 3

Multiplicação Normal

1 Thread

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.184	255 334 906	33 447 250
1000x1000	1.051	1 129 451 535	125 328 805
1400x1400	4.616	3 095 889 585	345 055 460
1800x1800	9.740	6 582 154 933	737 332 594
2200x2200	17.848	12 001 609 569	1 342 329 761
2600x2600	30.042	19 805 841 045	2 234 238 577
3000x3000	46.621	30 418 056 865	3 617 269 202

%CPU: 100%
%SYS: 0.0%
P: 5
Mcycle: 7 880
Minstr: 8 158
IPC (instructions per cycle): 1.04
%MISS: 1.79%
%BMIS: 0%
%BUS: %

2 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.102	129 748 959	17 509 690
1000x1000	0.545	564 802 264	62 684 838
1400x1400	2.066	1 548 139 625	172 125 891
1800x1800	4.418	3 291 268 304	368 541 035
2200x2200	8.201	6 001 652 965	675 323 472

2600x2600	13.379	9 903 813 091	1 115 834 496
3000x3000	20.781	15 210 286 604	1 856 273 1346

%CPU: 200%
%SYS: 0.0%
P: 7
Mcycle: 15 744
Minstr: 18 348
IPC (instructions per cycle): 1.17
%MISS: 0.89%
%BMIS: 0%
%BUS: 0.0%

3 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.066	81 518 361	9 113 968
1000x1000	0.419	363 546 658	40 871 108
1400x1400	1.366	1 033 063 316	115 698 074
1800x1800	2.808	2 196 256 218	249 133 541
2200x2200	5.231	4 006 016 392	447 646 882
2600x2600	8.571	6 605 412 310	751 317 610
3000x3000	13.418	9 781 994 902	1 224 563 983

%CPU: 300%
%SYS: 0.0%
P: 6
Mcycle: 7 771
Minstr: 9 571
IPC (instructions per cycle): 1.23
%MISS: 0.60%
%BMIS: 0.00%
%BUS: 0.0%

4 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.056	61 161 299	6 855 791
1000x1000	0.375	238 301 283	25 580 572

1400x1400	1.017	754 636 859	84 537 437
1800x1800	2.147	1 645 883 969	184 493 136
2200x2200	4.593	2 957 408 101	416 219 931
2600x2600	7.637	4 952 811 512	806 465 857
3000x3000	12.385	7 575 149 782	1 323 077 449

%CPU: 400%
%SYS: 0.0%
P: 6
Mcycle: 15 255
Minstr: 17 848
IPC (instructions per cycle): 1.17
%MISS: 0.57%
%BMIS: 0.00%
%BUS: 0.0%

Multiplicação Em Linha

1 Thread

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.131	27 125 940	734 472
1000x1000	0.593	125 706 541	6 163 504
1400x1400	1.716	346 609 224	60 466 754
1800x1800	3.765	751 611 480	204 265 454
2200x2200	6.975	2 083 062 669	402 584 851
2600x2600	11.848	4 359 766 863	691 383 274
3000x3000	17.774	6 782 583 059	1 082 453 881

%CPU: 100%
%SYS: 0.0%
P: 6
Mcycle: 7 943
Minstr: 21 318
IPC (instructions per cycle): 2.68
%MISS: 0.00%
%BMIS: 0.00%
%BUS: 0.0%

2 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.060	13 557 446	4 173 74
1000x1000	0.604	34 796 499	12 112 51
1400x1400	0.942	173 611 427	51 977 760
1800x1800	2.681	425 179 594	93 721 603
2200x2200	3.748	1 049 789 823	209 894 466
2600x2600	6.356	2 196 382 315	350 305 839
3000x3000	9.798	3 387 516 666	574 762 627

%CPU: 200%
%SYS: 0.0%
P: 6
Mcycle: 15 769
Minstr: 37 741
IPC (instructions per cycle): 2.38
%MISS: 0.13%
%BMIS: 0.00%
%BUS: 0.0%

3 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.042	9 066 190	3 218 34
1000x1000	0.229	42 654 237	4 634 224
1400x1400	0.636	123 634 766	33 657 802
1800x1800	1.464	251 484 908	78 447 282
2200x2200	2.552	691 380 175	145 770 513
2600x2600	4.382	1 452 435 192	242 280 540
3000x3000	6.930	2 261 899 069	414 128 006

%CPU: 300%
%SYS: 0.0%
P: 4
Mcycle: 7 770

Minstr: 19 188
IPC (instructions per cycle): 2.47
%MISS: 0.08%
%BMIS: 0.00%
%BUS: 0.0%

4 Threads

Size	Result (s)	L1 DCM	L2 DCM
600x600	0.036	6 797 016	234 296
1000x1000	0.198	31 536 064	2 638 861
1400x1400	0.493	87 058 429	26 560 470
1800x1800	1.103	191 466 282	63 490 514
2200x2200	2.215	521 643 768	105 209 951
2600x2600	4.844	1 101 995 965	214 107 439
3000x3000	5.983	1 687 993 150	298 259 809

%CPU: 400%
%SYS: 0.0%
P: 0
Mcycle: 15 241
Minstr: 36 594
IPC (instructions per cycle): 2.4
%MISS: 0.11%
%BMIS: 0.00%
%BUS: 0.0%