

Programação de Alto Desempenho

Atividade 5 - Programação Paralela em Sistemas de Mem. Compartilhada

Lucas Santana Lellis - 69618
PPGCC - Instituto de Ciência e Tecnologia
Universidade Federal de São Paulo

I. INTRODUÇÃO

Nesta atividade foram realizados experimentos relacionados com a implementação de técnicas de programação paralela em sistemas de memória compartilhada utilizando OpenMP. Cada experimento foi realizado 5 vezes, e os resultados apresentados são a média dos resultados obtidos em cada um deles, sendo calculado o speedup pela fórmula

$$\text{speedup}(P) = \frac{\text{Tempo para 1 thread}}{\text{Tempo para P threads}}$$

e a eficiência pela fórmula

$$\text{eficiencia}(P) = \frac{\text{speedup}(P)}{P}$$

Todos os programas foram feitos em C, com otimização -O3, utilizando a biblioteca PAPI para estimar o tempo total de processamento, e o número de cache misses.

As especificações da máquina utilizada estão disponíveis na Tabela I.

Tabela I: Especificações da Máquina

CPU	Intel i7 990X
Cores	6
Threads	12
Clock	3.47 GHz
Cache	12 MB Smartcache
RAM	20 GB
Hardware Counters	7
SO	Ubuntu 14.04
Kernel	3.13.0
GCC	6.2.0

II. EXPERIMENTO I - MULTIPLICAÇÃO DE MATRIZES

Neste experimento foi feita a implementação da multiplicação de matrizes com vetorização e blocagem para utilização efetiva de cache, utilizando OpenMP. Foi utilizada a vetorização automática do compilador, como confirmado utilizando a flag "info-vec-all" do compilador tendo como resultado a saída disponível na Figura 1.

A. Avaliando resultado da multiplicação

Utilizando o software R, foi possível conferir que a multiplicação das matrizes de tamanho 8x8 e com blocagem de tamanhos 2x2, 4x4 e 8x8, foi realizada com sucesso para até 4 threads.

Nas Figuras 2 e 3 estão as matrizes de entrada, foi feita a multiplicação dessas matrizes em um outro software confiável (R), e o resultado está na Figura 4, enfim, para todos os testes

```
gcc-6 src/ex01.c obj/* -o bin/ex01
-Iinc -lpapi -march=native -fopenmp-ffast-math
-ftree-vectorize -O3 -fopt-info-vec-all 2>&1 |
grep -e "LOOP VECTORIZED" -e "error" || true
```

```
src/ex01.c:64:13: note: LOOP VECTORIZED
```

Figura 1: Saída do compilador gcc, apontando vetorização automática do laço.

executados, obtivemos a mesma saída, representada na Figura 5.f Assim, comparando os resultados é fácil observar que a multiplicação está sendo feita corretamente.

```
77.26 54.95 97.36 30.08 77.49 52.89 61.91 09.30
93.01 98.07 26.35 34.81 01.79 02.58 16.81 02.21
55.81 06.73 33.68 29.09 51.98 84.37 58.02 92.30
39.49 46.66 96.38 25.62 37.06 43.71 30.64 11.47
30.14 03.15 18.54 38.51 77.74 11.46 43.59 77.36
94.13 75.96 32.54 15.72 29.80 50.51 68.89 10.42
07.24 26.37 09.34 86.87 51.28 13.41 02.09 41.48
78.32 77.30 82.55 07.50 15.98 99.84 47.04 10.94
```

Figura 2: Matriz de entrada A.

```
53.38 49.13 12.99 75.91 51.64 73.20 12.52 14.88
18.77 32.11 42.48 32.74 92.06 82.02 02.43 94.08
20.69 53.17 36.81 11.17 45.50 13.89 57.19 14.55
90.37 29.36 44.99 61.69 28.21 17.39 45.93 86.46
81.73 63.82 32.24 34.04 22.88 99.28 03.76 83.08
24.02 54.03 01.59 33.12 59.44 23.96 25.51 99.03
13.58 89.30 64.88 50.61 98.33 94.87 90.77 59.74
80.96 10.86 93.02 75.87 58.01 48.39 10.26 54.28
```

Figura 3: Matriz de entrada B.

```
> mat <- scan('tests/matA.txt')
Read 64 items
> matA <- matrix(mat, ncol=8, byrow=TRUE)
> mat <- scan('tests/matB.txt')
Read 64 items
> matB <- matrix(mat, ncol=8, byrow=TRUE)
> matA %*% matB
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] 19085.60 25052.627 15739.183 18835.35 25870.69 27321.73 15405.991 26215.50
[2,] 11112.08 11920.428 9268.277 13877.77 17987.71 17764.61 6129.639 15531.88
[3,] 20966.57 19662.331 17719.448 21130.57 23118.34 22763.81 12538.554 25618.59
[4,] 12716.70 16902.660 11514.659 12312.51 18567.07 16690.43 11449.760 18455.39
[5,] 19015.72 14011.617 15489.058 16075.05 15010.09 19263.26 8549.099 18740.86
[6,] 13972.23 20151.302 12834.484 17929.95 24840.34 25054.49 11706.685 22538.45
[7,] 16824.95 8883.894 11134.989 12318.85 10259.21 11951.11 5829.013 18200.02
[8,] 13246.08 21673.026 12420.705 16922.20 26688.47 22320.94 13222.982 24906.10
```

Figura 4: Matriz de saída C - obtida no R.

B. Comparando speedup e eficiência

Na Tabela II estão disponíveis os resultados dos testes, para 1, 2, 4, 8 e 12 threads, da multiplicação de matrizes

19085.60 25052.62 15739.18 18835.35 25870.69 27321.73 15405.99 26215.50
11112.08 11920.43 09268.28 13877.76 17987.71 17764.61 06129.64 15531.88
20966.57 19662.33 17719.45 21130.57 23118.35 22763.81 12538.55 25618.59
12716.70 16902.66 11514.66 12312.51 18567.07 16690.43 11449.76 18455.39
19015.71 14011.62 15489.06 16075.05 15010.09 19263.26 08549.10 18740.86
13972.23 20151.30 12834.48 17929.95 24840.34 25054.49 11706.68 22538.45
16824.95 08883.89 11134.99 12318.85 10259.21 11951.11 05829.01 18200.02
13246.08 21673.03 12420.70 16922.20 26688.47 22320.95 13222.98 24906.10

Figura 5: Matriz de saída C - obtida em todos os testes.

com blocagem e vetorização automática, utilizando blocos de tamanho 128x128. Na Figura 6, está disponível a comparação do speedup da multiplicação de matrizes de largura 1024 e 4096.

Tam.	Threads	Tempo(s)	Cache Miss	Speedup	Eficiencia
1024	1	0.6230	82000544	1.0000	1.0000
1024	2	0.3221	41167963.8	1.9342	0.9671
1024	4	0.2041	20549918.4	3.0532	0.7633
1024	8	0.1412	7361344.4	4.4131	0.5516
1024	12	0.1483	10081804.4	4.2023	0.3502
4096	1	38.0568	5193524062.4	1.0000	1.0000
4096	2	19.1645	2599822977	1.9858	0.9929
4096	4	9.8656	1300069832.8	3.8575	0.9644
4096	8	7.6746	655620520.2	4.9588	0.6198
4096	12	6.3199	485937763.4	6.0218	0.5018

Tabela II: Avaliação do desempenho do algoritmo de multiplicação com blocagem e vetorização com 1, 2, 4, 8 e 12 threads.

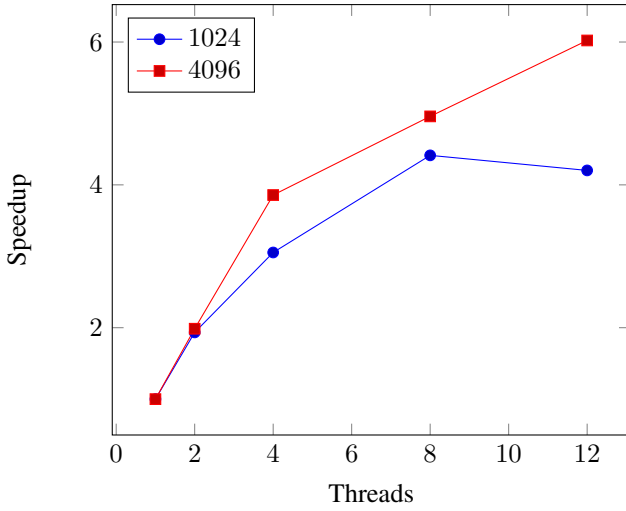


Figura 6: Comparação do speedup da multiplicação de matrizes para 1, 2, 4, 8 e 12 Threads.

III. EXPERIMENTO II - ODD-EVEN SORT

Nesse experimento foi feita a paralelização do algoritmo Odd-Even Sort, realizando a ordenação de valores pseudo aleatórios com 1, 2 e 4 threads obteve-se os mesmos resultados, como visto na Figura 7.

Foram então realizados testes para vetores de tamanho $4 * 10^4$, os resultados estão disponíveis na Tabela III, e o gráfico de speedup está na Figura 8.

IV. EXPERIMENTO III - CONTAGEM DE NÚMEROS

Nesse experimento foi implementada a paralelização de um algoritmo para contagem da ocorrência de números em um

Entrada : 6 8 5 3 6 9 8 1 9 4
1 Thread : 1 3 4 5 6 6 8 8 9 9
2 Threads: 1 3 4 5 6 6 8 8 9 9
4 Threads: 1 3 4 5 6 6 8 8 9 9
8 Threads: 1 3 4 5 6 6 8 8 9 9
12 Threads: 1 3 4 5 6 6 8 8 9 9

Figura 7: Resultados dos testes do algoritmo Odd-Even sort de uma lista de 10 elementos para 1, 2, 4, 8 e 12 threads.

Tam.	Threads	Tempo(s)	Cache Miss	Speedup	Eficiencia
40000	1	1.9728	1118460.4	1.0000	1.0000
40000	2	0.9978	261475.2	1.9771	0.9886
40000	4	0.6639	287415.6	2.9715	0.7429
40000	8	0.52666	204544.2	3.7459	0.4682
40000	12	0.76676	184864.2	2.5729	0.2144

Tabela III: Avaliação do desempenho do algoritmo Odd-Even sort para 1, 2, 4, 8 e 12 threads.

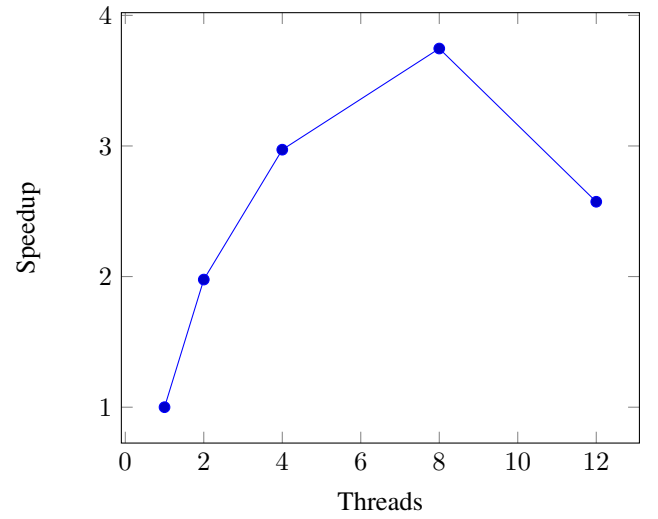


Figura 8: Comparação do speedup do Odd-Even sort para 1, 2, 4, 8 e 12 Threads.

vetor, para 1, 2 e 4 threads foi possível identificar que o valor total da soma de ocorrências de cada número é igual a 10^8 , que é igual ao tamanho do vetor de entrada, o que confirma a validade da solução utilizada para paralelização.

A comparação de tempo de execução, speedup e eficiência para 1, 2 e 4 threads está disponível na Tabela IV.

Tam.	Threads	Tempo(s)	Cache Miss	Speedup	Eficiencia
100000000	1	0.10558	118691	1.0000	1.0000
100000000	2	0.05436	107471.6	1.9422	0.9711
100000000	4	0.03266	74594.4	3.2327	0.8082
100000000	8	0.03358	74620.4	3.1441	0.3930
100000000	12	0.03682	80558.4	2.8675	0.2390

Tabela IV: Avaliação do desempenho do algoritmo de contagem de números para 1, 2, 4, 8 e 12 threads.

V. EXPERIMENTO IV - CONJECTURA DE GOLDBACH

Nesse experimento foi feita a comparação entre diferentes modos de escalonamento para a paralelização do algoritmo da

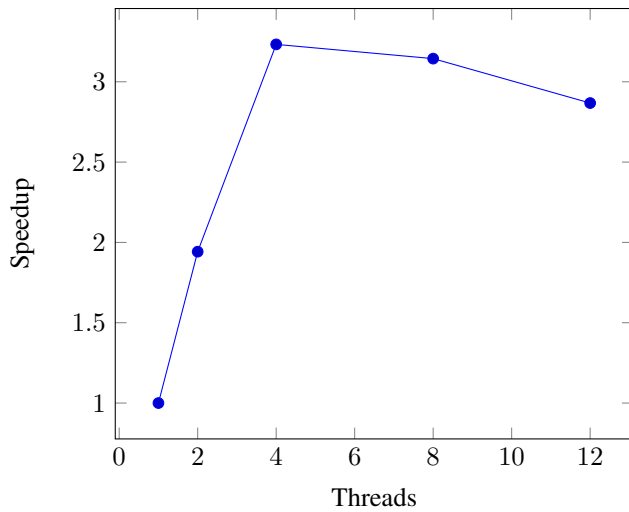


Figura 9: Comparação do speedup do algoritmo de contagem para 1, 2, 4, 8 e 12 Threads.

conjectura de goldbach.

A comparação de tempo de execução, speedup e eficiência para 1, 2 e 4 threads usando os diferentes tipos de escalonamento está disponível na Tabela V.

Escalonamento	Threads	Tempo(s)	Cache Miss	Speedup	Eficiencia
static.10	1	0.32988	91733.8	1.0000	1.0000
static.10	2	0.33176	74614	1.0057	0.5028
static.10	4	0.33134	88029	1.0044	0.2511
static.10	8	0.32966	75510.4	0.9993	0.1249
static.10	12	0.33428	71205	1.0133	0.0844
static.5	1	0.31546	109541.2	1.0000	1.0000
static.5	2	0.31296	84520	0.9921	0.4960
static.5	4	0.32362	75171.6	1.0259	0.2565
static.5	8	0.31866	98950.4	1.0101	0.1263
static.5	12	0.3142	91665.2	0.9960	0.0830
static.2	1	0.3227	90047.4	1.0000	1.0000
static.2	2	0.32234	71396	0.9989	0.4994
static.2	4	0.31538	76083.4	0.9773	0.2443
static.2	8	0.32002	81403	0.9917	0.1240
static.2	12	0.3272	72467.6	1.0139	0.0845
dynamic	1	0.28774	79895.4	1.0000	1.0000
dynamic	2	0.28714	97687	0.9979	0.4990
dynamic	4	0.28912	72925.4	1.0048	0.2512
dynamic	8	0.28558	81616.2	0.9925	0.1241
dynamic	12	0.28718	78508.2	0.9981	0.0832
guided	1	0.30054	113116.6	1.0000	1.0000
guided	2	0.29514	86411.8	0.9820	0.4910
guided	4	0.29678	72452.8	0.9875	0.2469
guided	8	0.29478	76879.2	0.9808	0.1226
guided	12	0.29384	101599	0.9777	0.0815

Tabela V: Avaliação do desempenho do algoritmo de contagem de números para 1, 2, 4, 8 e 12 threads.

VI. EXPERIMENTO V - JOGO DA VIDA

Nesse experimento foi implementada a paralelização de um algoritmo do jogo da vida. Para 1, 2 e 4 threads foi possível identificar que a solução permanece a mesma, em um tabuleiro de tamanho 1000x1000.

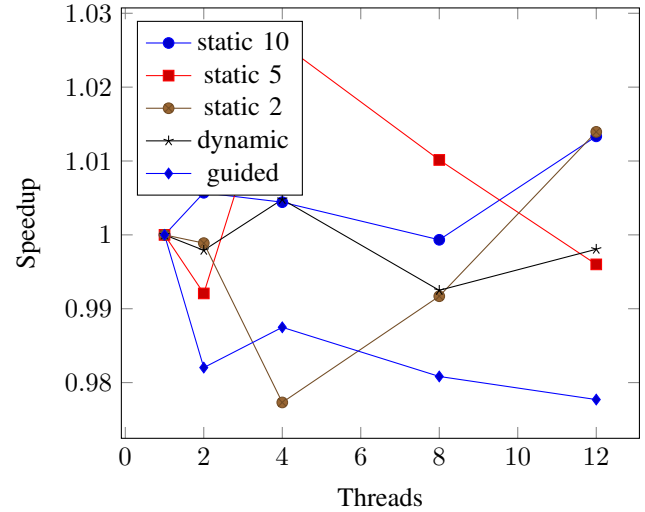


Figura 10: Comparação do speedup do algoritmo de goldbach, para diferentes tipos de escalonamento e para 1, 2, 4, 8 e 12 Threads.

Tamanho	Threads	Tempo(s)	Cache Miss	Speedup	Eficiencia
500	1	1.46442	1767006.2	1.0000	1.0000
500	2	0.73564	991115.4	1.9907	0.9953
500	4	0.43962	770585.4	3.3311	0.8328
500	8	0.35674	565758.4	4.1050	0.5131
500	12	0.4553	630884.2	3.2164	0.2680

Tabela VI: Avaliação do desempenho do algoritmo de contagem de números para 1, 2, 4, 8 e 12 threads.

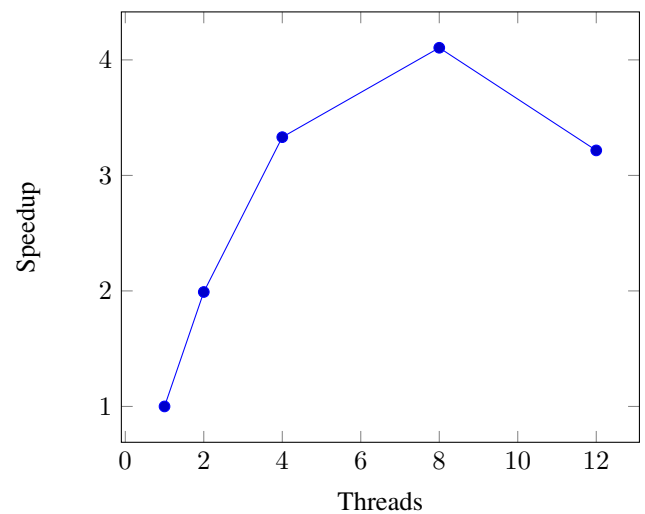


Figura 11: Comparação do speedup do algoritmo do jogo da vida para 1, 2, 4, 8 e 12 Threads.