INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANDREAS MUNTE FOERSTER – andreas.foerster@usp.br – #7143997

COMPARAÇÃO DE TEMPO DE EXECUÇÃO SEQUENCIAL E CONCORRENTE DO MÉTODO DE JACOBI PARA SISTEMAS DE EQUAÇÕES LINEARES:

SÃO CARLOS – SP

SET-15

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1. INTRODUÇÃO

Segundo (Tanenbaum, 2006), apesar de a capacidade computacional de processadores estar constantemente evoluindo, para muitas aplicações, especialmente na área da computação científica, o processamento nunca é suficiente. Com os avanços tecnológicos, as CPUs tornam-se cada vez mais rápidas e seu tamanho se reduz de acordo com a diminuição das dimensões dos transistores, que por vezes não passam de alguns átomos. É evidente que a velocidade dos computadores está se aproximando do limite permitido pela física; no entanto, a demanda por mais processamento nunca cessa.

Portanto, com intuito de saciar esta necessidade, engenheiros elétricos perseguem alternativas que permitam que vários processos sejam executados ao mesmo tempo – e assim, também simples telefones celulares por vezes possuem CPUs com 4 ou mais núcleos. Apesar do grande poder computacional fornecido por estes processadores modernos, programas sequenciais convencionais ainda são limitados ao uso de apenas um destes núcleos. Para que haja o aproveitamento de todo o computador, o software deve ser quebrado em tarefas menores e independentes entre si.

2. OBJETIVO

Este trabalho tem como objetivo verificar a importância da divisão de tarefas em threads em programas com alto custo computacional, assim como medir e comparar o tempo de execução para diferentes quantidades de threads usadas.

3. MÉTODOS

3.1. Método Numérico de Jacobi

O método de Jacobi é um dos métodos numéricos iterativos mais simples para soluções de sistemas de equações lineares e consiste em uma série de iterações que atualizam valores a serem usados na iteração seguinte.

$$\begin{vmatrix} a_{1\,1} & \dots & a_{1\,n} \\ \dots & \dots & \\ a_{n\,1} & & a_{n\,n} \end{vmatrix} * \begin{vmatrix} x_1 \\ \dots \\ x_n \end{vmatrix} = \begin{vmatrix} b_1 \\ \dots \\ b_n \end{vmatrix}$$

$$A * X = B$$

Para garantir a convergência do método pode-se realizar um teste simples que consiste em verificar se, para cada linha, o elemento da diagonal tem valor superior à soma de todos os elementos restantes dessa linha:

$$\sum_{j=1,\ j\neq i}^{n} a_{i\,j} < a_{i\,i}\ (i = 1, 2, \dots, n)$$

Havendo garantia de convergência, diagonaliza-se a matriz e isola-se o termo desconhecido da diagonal:

$$x_i = b_i - \sum_{j=1,\ j\neq i}^{n} a_{i\,j} * x_j\ (i = 1, 2, \dots, n)$$

Para calcular a iteração $K + 1$ do método, assumem-se os valores de x calculados anteriormente para calcular o termo da diagonal:

$$x_i^{k+1} = b_i - \sum_{j=1,\ j\neq i}^{n} a_{i\,j} * x_j^k (i = 1, 2, \dots, n)$$

As iterações são calculadas consecutivamente até que o critério de parada seja atingido:

$$\left|x_i^{k+1} - x_i^k\right| < \varepsilon\ (i = 1, 2, \dots, n)$$

Sendo ε a precisão desejada.

3.2. Programação

A fim de testar o método e coletar dados foi escrito um programa e C utilizando POSIX threads. Uma estrutura de dados armazena todas as informações relevantes a serem passadas a cada thread assim como o conjunto de linhas que serão processadas pela mesma. Devido à maioria dos dados contidos na struct serem apenas usados para leitura não há necessidade de chaveamento de acesso.

Devido ao método requerer os valores da iteração anterior para realizar cada cálculo, cada thread é limitada a apenas uma iteração ao travar uma variável mutex no início de cada iteração. Tal variável é novamente destravada somente após todas as threads terem terminado e os novos valores terem sido avaliados pela thread principal.

Com o intuito de bloquear o processamento da thread principal até que todas as linhas sejam processadas, a thread principal bloqueia um semáforo que é desbloqueado apenas ao fim de cada thread filha.

3.3. Coleta de Dados

Para facilitar a coleta de dados o programa recebe até 3 argumentos opcionais, dentre eles: número de threads a utilizar; quantidade de vezes que deve realizar o teste para cálculo de desvio padrão do tempo de execução; e arquivo de saída para o resultado do método e tempos individuais de cada execução. As informações iniciais do problema são lidas através de um arquivo cujo nome é fornecido como primeiro argumento.

Para facilitar a coleta de dados um script shell foi escrito com instruções para executar o método para matrizes de tamanho 250, 500, 1000, 1500, 2000 e 3000, cada uma utilizando de 1 a 10 threads. Cada teste executa o método 10 vezes para cálculo da média e desvio padrão.

4. RESULTADOS

Para coleta dos dados abaixo foi utilizada uma máquina ARCH-Linux com as seguintes especificações coletadas pelo comando `slcpu`:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 30
Stepping:              5
CPU MHz:               933.000
BogoMIPS:              3459.73
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
```

Tabela 1 - Média de 10 resultados em matriz de 250 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	1.981 +/- 0.958	-
2	1.289 +/- 0.909	1.537 +/- 1.314
3	0.942 +/- 1.020	2.103 +/- 2.494
4	1.043 +/- 0.929	1.899 +/- 1.925
5	1.036 +/- 0.974	1.912 +/- 2.022
6	0.861 +/- 0.931	2.301 +/- 2.725
7	1.025 +/- 0.913	1.933 +/- 1.959
8	0.991 +/- 0.797	1.999 +/- 1.876
9	0.990 +/- 0.928	2.001 +/- 2.111
10	0.908 +/- 1.042	2.182 +/- 2.717

A matriz precisou de 1.327 iterações para atingir a precisão desejada de 0,001.

Tabela 2 - Média de 10 resultados em matriz de 500 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	6.730 +/- 6.548	-
2	4.644 +/- 2.578	1.449 +/- 1.623
3	3.251 +/- 2.377	2.070 +/- 2.519
4	3.237 +/- 2.576	2.079 +/- 2.613
5	2.930 +/- 1.481	2.297 +/- 2.518
6	2.077 +/- 0.972	3.240 +/- 3.498
7	1.871 +/- 0.906	3.597 +/- 3.909
8	1.966 +/- 1.398	3.423 +/- 4.125
9	3.132 +/- 2.088	2.149 +/- 2.534
10	2.506 +/- 1.996	2.686 +/- 3.377

A matriz precisou de 2.295 iterações para atingir a precisão desejada de 0,001.

Tabela 3 - Média de 10 resultados em matriz de 1000 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	31.470 +/- 14.608	-
2	14.941 +/- 4.168	2.106 +/- 1.141
3	12.335 +/- 3.507	2.551 +/- 1.389
4	11.201 +/- 3.105	2.810 +/- 1.519
5	12.808 +/- 2.160	2.457 +/- 1.213
6	10.278 +/- 0.995	3.062 +/- 1.452
7	9.336 +/- 0.873	3.371 +/- 1.596
8	9.074 +/- 0.962	3.468 +/- 1.651
9	11.882 +/- 2.923	2.649 +/- 1.391
10	10.512 +/- 1.178	2.994 +/- 1.430

A matriz precisou de 3.955 iterações para atingir a precisão desejada de 0,001.

Tabela 4 - Média de 10 resultados em matriz de 1500 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	79.029 +/- 25.178	-
2	40.557 +/- 8.601	1.949 +/- 0.746
3	33.098 +/- 0.929	2.388 +/- 0.764
4	27.863 +/- 5.052	2.836 +/- 1.040
5	35.737 +/- 4.450	2.211 +/- 0.756
6	28.894 +/- 1.179	2.735 +/- 0.879
7	26.714 +/- 0.989	2.958 +/- 0.949
8	23.890 +/- 0.975	3.308 +/- 1.063
9	32.849 +/- 3.271	2.406 +/- 0.803
10	28.987 +/- 1.140	2.726 +/- 0.875

A matriz precisou de 5.274 iterações para atingir a precisão desejada de 0,001.

Tabela 5 - Média de 10 resultados em matriz de 2000 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	160.185 +/- 30.104	-
2	84.605 +/- 14.850	1.893 +/- 0.487
3	69.856 +/- 0.998	2.293 +/- 0.432
4	52.775 +/- 1.296	3.035 +/- 0.575
5	68.851 +/- 4.053	2.327 +/- 0.458
6	61.363 +/- 1.361	2.610 +/- 0.494
7	57.175 +/- 0.910	2.802 +/- 0.528
8	51.388 +/- 1.340	3.117 +/- 0.591
9	64.739 +/- 2.666	2.474 +/- 0.476
10	60.316 +/- 1.335	2.656 +/- 0.503

A matriz precisou de 6.458 iterações para atingir a precisão desejada de 0,001.

Tabela 6 - Média de 10 resultados em matriz de 3000 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	415.816 +/- 28.429	-
2	235.865 +/- 6.546	1.763 +/- 0.130
3	204.489 +/- 1.608	2.033 +/- 0.140
4	152.834 +/- 0.850	2.721 +/- 0.187
5	186.405 +/- 2.478	2.231 +/- 0.155
6	175.092 +/- 1.050	2.375 +/- 0.163
7	166.494 +/- 1.663	2.497 +/- 0.173
8	147.288 +/- 1.052	2.823 +/- 0.194
9	168.255 +/- 2.037	2.471 +/- 0.172
10	164.711 +/- 2.554	2.525 +/- 0.177

A matriz precisou de 8.453 iterações para atingir a precisão desejada de 0,001.

Tabela 7 - Média de 10 resultados em matriz de 4000 com 1-10 threads

Threads	Tempo de Execução (s)	Speedup
1	876.336 +/- 47.966	-
2	498.585 +/- 17.491	1.758 +/- 0.114
3	430.831 +/- 9.322	2.034 +/- 0.120
4	322.028 +/- 1.084	2.721 +/- 0.149
5	387.438 +/- 2.413	2.262 +/- 0.125
6	364.564 +/- 1.151	2.404 +/- 0.132
7	344.277 +/- 1.775	2.545 +/- 0.140
8	308.433 +/- 1.078	2.841 +/- 0.156
9	339.728 +/- 3.570	2.580 +/- 0.144
10	343.126 +/- 0.747	2.554 +/- 0.140

A matriz precisou de 10.069 iterações para atingir a precisão desejada de 0,001.

5. DISCUSSÕES

Para melhor avaliação dos resultados, estes serão novamente apresentados em forma de gráficos:

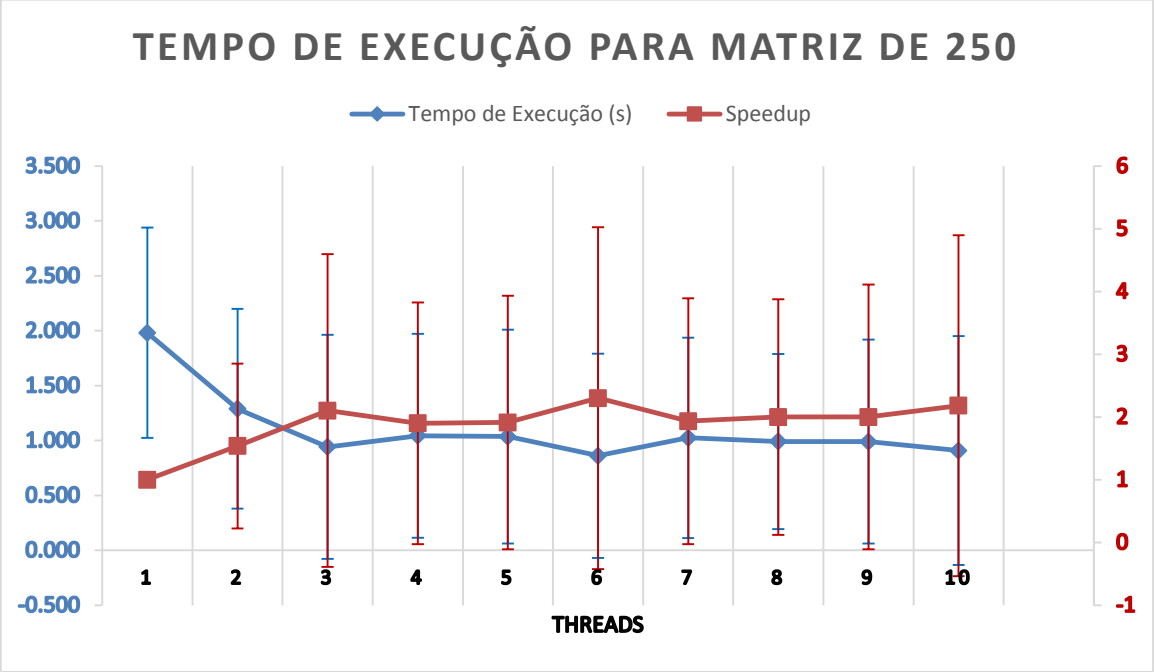

Figura 1 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 250

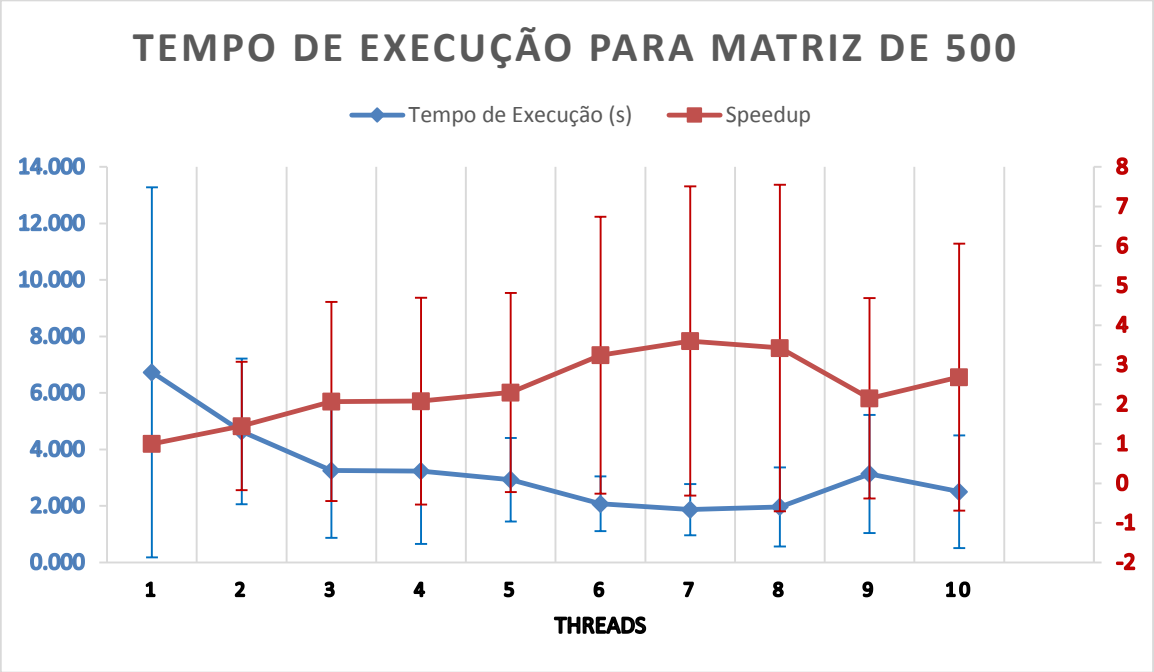

Figura 2 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 500

Para os testes realizados com as matrizes de 250 e 500 os resultados apresentaram uma variação de tempo muito grande, dificultando sua análise. No entanto, é notável que o tempo de execução decresce até o uso de 8 threads no segundo exemplo. Resultado coerente, considerando que a máquina utilizada possui 8 núcleos de processamento.

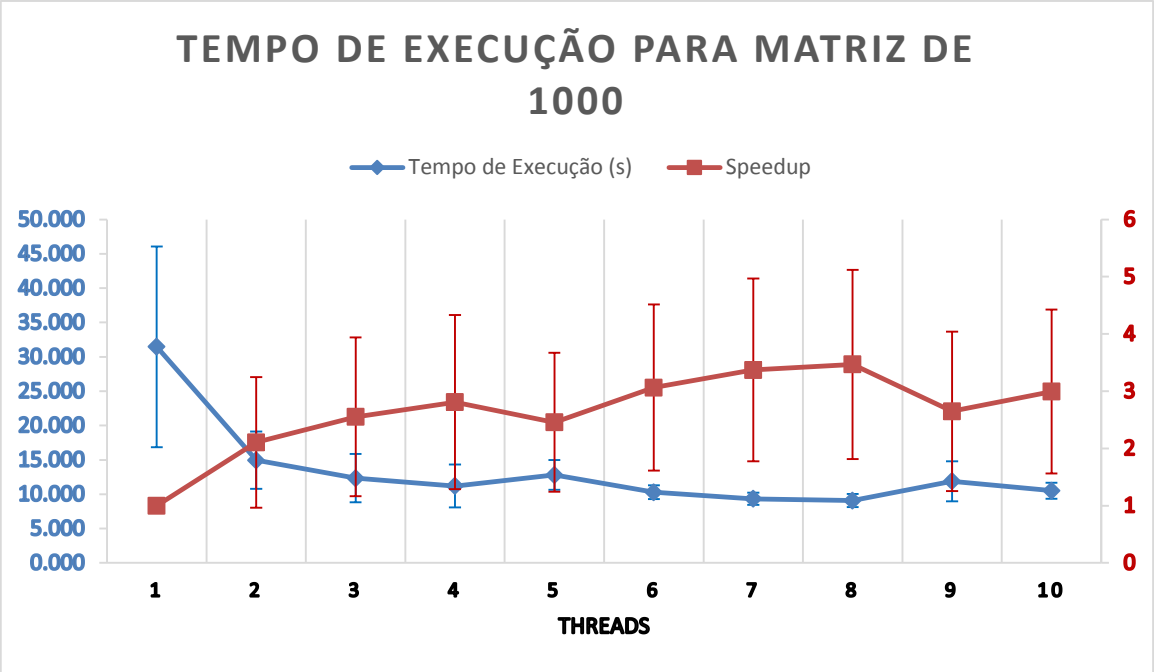

Figura 3 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 1000

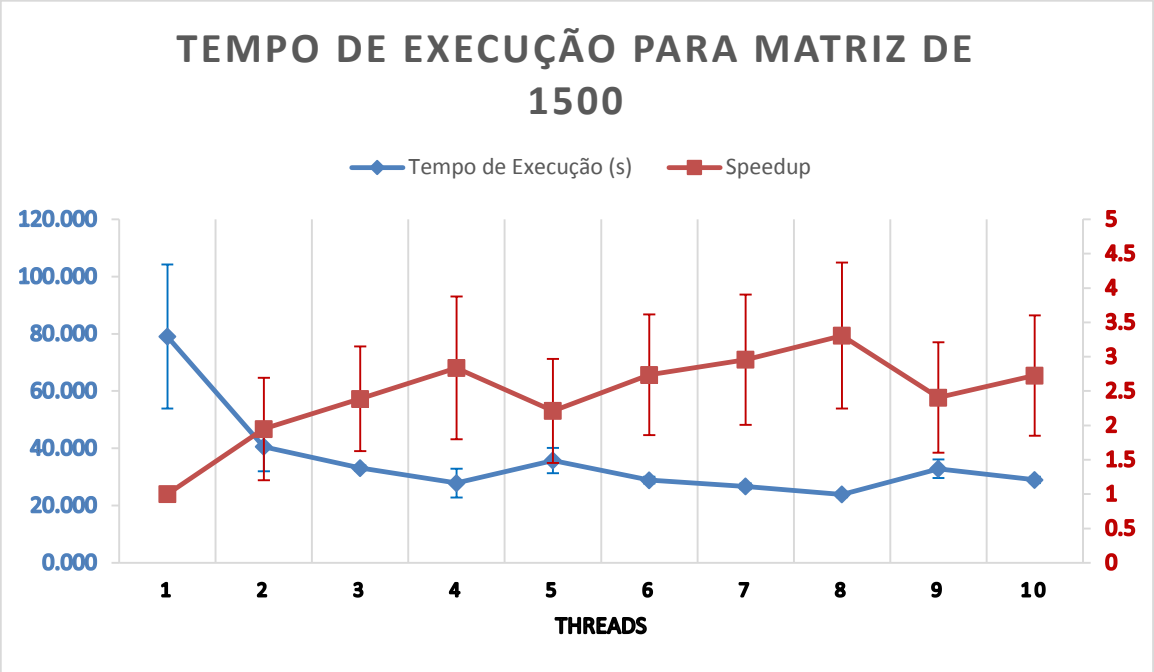

Figura 4 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 1500

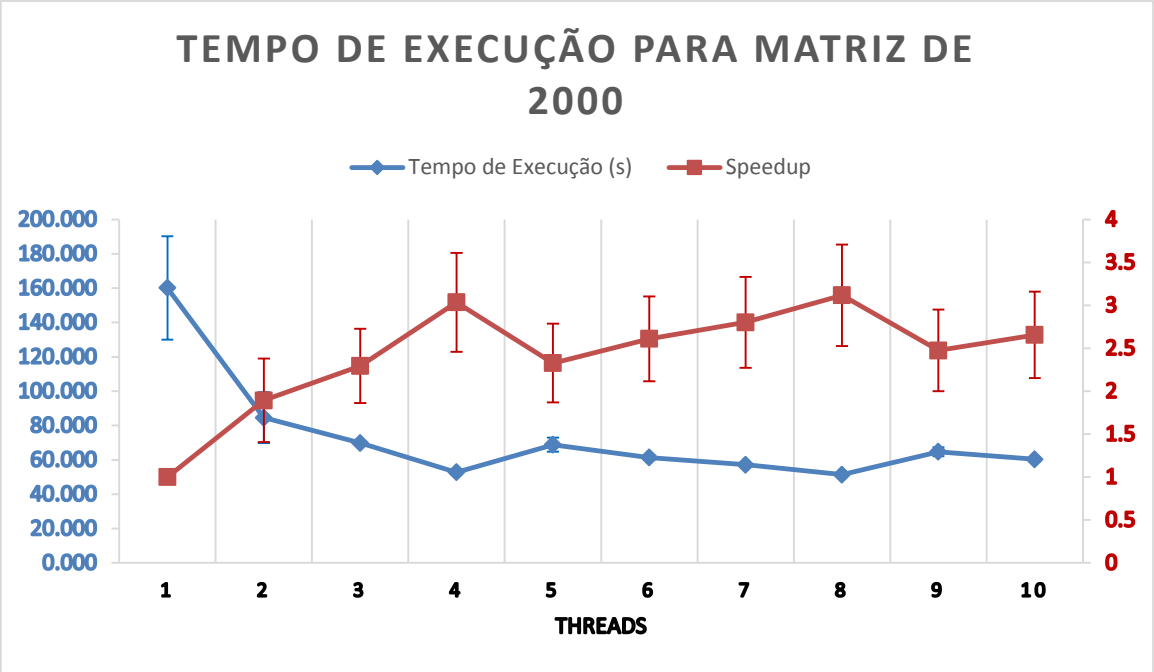

Figura 5 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 2000

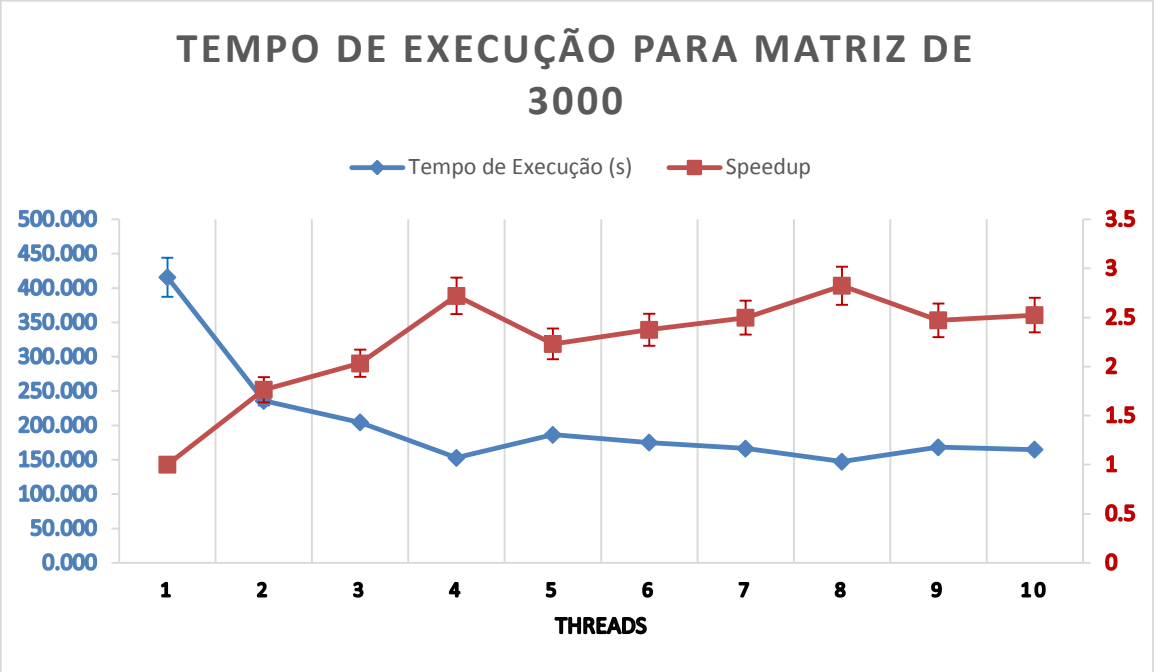

Figura 6 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 3000

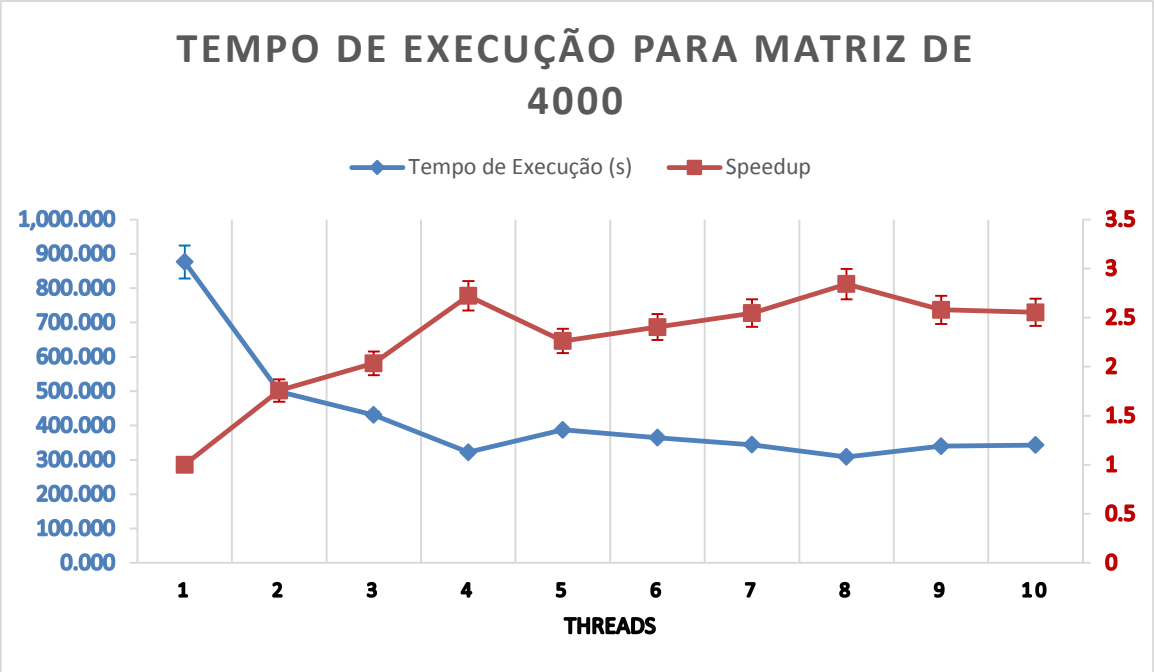

Figura 7 - Gráfico do tempo de execução e speedup relativo à quantidade de threads utilizadas na matriz de 4000

Nos 5 testes restantes, nota-se um comportamento muito semelhante na curva de speedup. O tempo de execução decresce quase linearmente até o uso de 4 threads e permanece quase constante tendo picos de eficiência em 4 e 8 threads.

Apesar de o computador utilizado possuir 8 núcleos de processamento, os resultados obtidos indicam que o uso de mais de 4 threads não produz resultados significativamente melhores, podendo até prejudicar o tempo de execução em alguns casos – um resultado bastante contra intuitivo. Este fato possivelmente se deve à quantidade excessiva de dados que são acessados pelas threads e a distância relativa entre eles na memória, o que levaria a muitas falhas na cache do processador. No entanto, tal teoria não foi verificada devido ao alto tempo de espera para a realização dos testes com matrizes grandes.

Outra observação importante é a alta variação no tempo de execução para testes com poucos dados devido à grande influência do momento e da quantidade de chaveamentos de cada thread para o tempo final de execução.

6. CONCLUSÃO

O uso de threads em um método iterativo causa um alto impacto no tempo de execução, tendendo a melhorar até uma certa quantidade de tarefas em paralelo, ponto no qual as limitações de hardware e software impedem que o problema seja resolvido mais rapidamente. Especificamente para o experimento realizado, observou-se que o uso de 8 threads em um processador de 8 núcleos produz um resultado no tempo ideal.

Além disso, no presente trabalho o uso de variáveis de chaveamento de threads se provou um recurso vital para a funcionalidade do software permitindo a sincronização das tarefas a cada iteração do método.

7. REFERÊNCIAS

Franco, N. B. (n.d.). *Cálculo Numérico.* Pearson Education.

Tanenbaum, A. S. (2006). *Structured Computer Organization* (5ft ed.). Upper Saddle River, New Jersey: Pearson Education.

8. APÊNDICE – A: SOBRE O PROGRAMA

O código escrito para este projeto pode ser obtido através de um repositório online do GitHub, assim como as matrizes utilizadas como exemplo e instruções de compilação e uso.

Projeto GitHub:

https://github.com/maglethong/USP-2015-Concurrent-Programming/tree/master/project1

Para compilar o programa execute:

```
mkdir bin
gcc -o ./bin/prog -I ./lib/ -lm –pthread -lrt ./src/*.c
```

Para executar o programa execute:

```
./bin/prog arquivo-de-entrada [ número-de-threads [ número-de-vezes-a-
      executar [ arquivo-de-saída ] ] ]
```

Valores padrão são:

```
número-de-threads            1
número-de-vezes-a-executar   1
arquivo-de-saída             não escreve a saída
```