

Relatório do projeto 1 - “Diagonais principais de uma matriz”

Aluno: André L. R. Estevam

R.A.: 166348

Limeira, 19 de junho de 2017

Sobre o problema

Foi requerido um programa que calcule a soma dos valores das diagonais principais de uma matriz M por N utilizando T threads. Cada diagonal deve ser processada por uma thread e ao fim do cálculo uma thread deve ter outra diagonal para calcular.

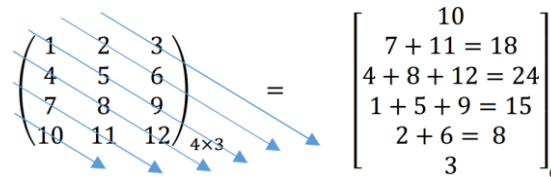


Figura 1 - Exemplo de entrada e saída disponibilizado pelo professor

Sobre a solução

Foram desenvolvidas funções tanto para resolução do problema em si quanto funções adicionais para realizar o gerenciamento de memória, arquivos e interface com o usuário.

Para a soma foram desenvolvidas duas lógicas diferentes e opostas:

A **lógica A** faz o possível para dividir o trabalho de acordo com o tempo de vida das threads, isto é, as que existirem por mais tempo devem processar mais.

A quantidade de threads T é interpretada como um limite. Apesar de T threads serem criadas as mais antigas podem em muitos casos terminar o trabalho antes das mais recentes.

Já de acordo com a **lógica B** o trabalho será dividido entre as threads da maneira mais igualitária quanto seja possível. O número de threads indica exatamente quantas devem ser criadas e envolvidas no processamento (desde que o tamanho da matriz permita).

Ambas as soluções utilizam a ideia de numerar as diagonais e utilizar este valor para encontrar as coordenadas de início diagonal em questão, fazendo somas sucessivas e testando a cada passo se os limites da matriz foram atingidos.

Entradas, saídas e padrões

O programa irá, por padrão, ler os valores da matriz a partir do arquivo *"in.txt"*. A resposta será escrita na tela e nos arquivos *"out_sol_A.txt"* e *"out_sol_B.txt"*, para a execução das soluções A e B respectivamente.

Nos três arquivos o padrão adotado para armazenar os valores de ponto flutuante é:

- Ponto "." para separar a parte inteira da real;
- Espaço " " para separar valores;
- Obs.: não deve haver marcação de quebra de linha em qualquer parte do arquivo.

Para a compilação em sistemas Linux pode ser feito o uso do arquivo *"Makefile"* incluso com o código do programa. O resultado da compilação deverá ser acessado através do arquivo *"projso.o"*

Exemplo:

Para um arquivo de entrada:

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0

Entrada de dados pelo terminal:

```
##### [ ENTRADA DE DADOS ] #####
Numero de threads: 2
Matriz (m linhas): 4
Matriz (n colunas): 3

----- [ Voce escolheu ] -----
Processar uma matriz: [4 x 3]
Usando: 2 threads
Pressione enter para continuar...
```

Figura 2- Exemplo de entrada de dados pela interface do programa.

Para matrizes pequenas será feita a impressão dos valores lidos na tela:

```
----- [ MATRIZ ] -----
  0    1    2
0 [ 1][ 2][ 3]
1 [ 4][ 5][ 6]
2 [ 7][ 8][ 9]
3 [10][11][12]
```

Figura 3 – Matriz de entrada

A saída do programa mais relevante é uma tabela contendo:

- Coordenadas do primeiro elemento da matriz;
- Número da diagonal;
- Resultado da soma;
- Quantas somas foram feitas;

```
----- [ IMPRIMINDO RESULTADOS ] -----
```

COORDENADAS DO PRIMEIRO ELEM.	NUMERO DA DIAGONAL	RESULTADO DA SOMA	SOMAS FEITAS NA DIAGONAL
(3, 0)	0	10	1
(2, 0)	1	18	2
(1, 0)	2	24	3
(0, 0)	3	15	3
(0, 1)	4	8	2
(0, 2)	5	3	1

Figura 4 – Saída principal (para a execução da Solução A).

Será gerado também um gráfico com informações sobre o processamento:

```

THREAD [ 0 ] | ***** [ 6 somas ] [ 3 diagonais ]
THREAD [ 1 ] | ***** [ 6 somas ] [ 3 diagonais ]

```

Figura 5 - Informações sobre o processamento.

Os arquivos de saída gerados para essa entrada contêm os valores:

```

10.000000 18.000000 24.000000 15.000000 8.000000 3.000000

```

Para facilitar a execução de testes o arquivo de entrada pode ser automaticamente preenchido com algum valor utilizando a função:

```

fillFileWithValue("<nome do arquivo>", <quantidade de vezes a repetir o
número>, <número em si>);

```

Para sair do programa insira valor zero para o número de threads e dimensões da matriz.

Visualização da matriz

Foi atribuído um número a cada diagonal, indo de 0 até $m + n - 2$, desta forma é possível visualizar a matriz como uma série de vetores de tamanho variável, como por exemplo:

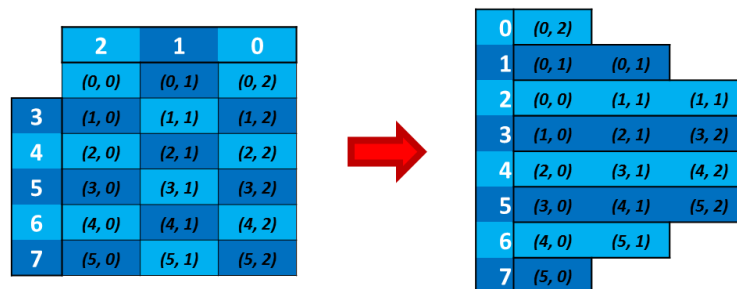


Figura 6 - Visualização da matriz como vetores de cada diagonal.

Considerações sobre os testes

Os testes foram realizados em um computador com o processador Intel® Core™ i5-5200U CPU @ 2.20GHz com 4 núcleos.

Tanto para a Solução A quanto para a Solução B, o processamento foi feito para 1, 2, 4, 8 e 16 *threads*. Para cada grupo foram utilizadas matrizes de dimensões 100 por 100, 300 por 300, 500 por 500, e assim por diante até o máximo de 1500 por 1500. Para cada dimensão de matriz o teste foi repetido 30 vezes de onde foi obtida a média do tempo de processamento.

Solução A

Visão geral do algoritmo como uma série de passos

1. Ler a quantidade de threads T e as dimensões da matriz (M, N) a partir dos dados inseridos pelo usuário;
2. Criar e inicializar um *mutex*;
3. Criar e inicializar a região crítica;

4. Criar a matriz com base nos dados inseridos pelo usuário;
5. Criar um vetor para o armazenamento da resposta;
6. Criar um vetor para o armazenamento dos Ids das *threads*;
7. Ler o arquivo de entrada de forma sequencial e colocar os valores na matriz criada (a matriz será preenchida do topo para a base com valores sendo inseridos da direita para a esquerda);
8. Criar o tipo de dado que guarda argumentos para as *threads*. O argumento é formado por:
 - a. Descritor da matriz;
 - b. Vetor de resposta;
 - c. *Mutex*;
 - d. Região crítica;
9. Criar as T *Threads* enviando para todas o mesmo argumento;
10. Espere até que todas as *threads* tenham finalizado o processamento;
11. Destrua o *mutex* criado;
12. Grave o vetor de resposta de forma sequencial no arquivo de resposta;
13. Imprima os dados da resposta na tela;
14. Remova da memória todo vetor e matriz alocado.

Algoritmo da soma por meio de *threads*

É passado para cada *thread* um semáforo para controlar uma região crítica, que irá armazenar o número da próxima diagonal que não foi processada. Também são passados descritores para a matriz e o vetor de resposta.

Enquanto as somas não tiverem terminado cada *thread* irá realizar sucessivamente:

1. Obter acesso à região crítica *reg*;
2. Copiar o valor na região crítica para uma variável local *loc*;
3. Incrementar em um o valor na região crítica ($reg = reg + 1$);
4. Converter o número copiado (*loc*) para o endereço (m, n) da matriz onde essa diagonal começa;
5. Somar sucessivamente os valores da diagonal até que o fim da matriz tenha sido atingido;
6. Gravar o valor da soma no vetor de resposta na posição *loc*

Número da diagonal			
	2	1	0
3			
4			
5			
6			
7			
8			
9			
10			

IDENTIFICADOR	
0	
1	
2	
3	

Figura 7- Exemplo da divisão de trabalho entre as *threads* com a Solução A - as *threads* mais antigas processam mais enquanto as mais recentes não foram criadas.

Vantagens:

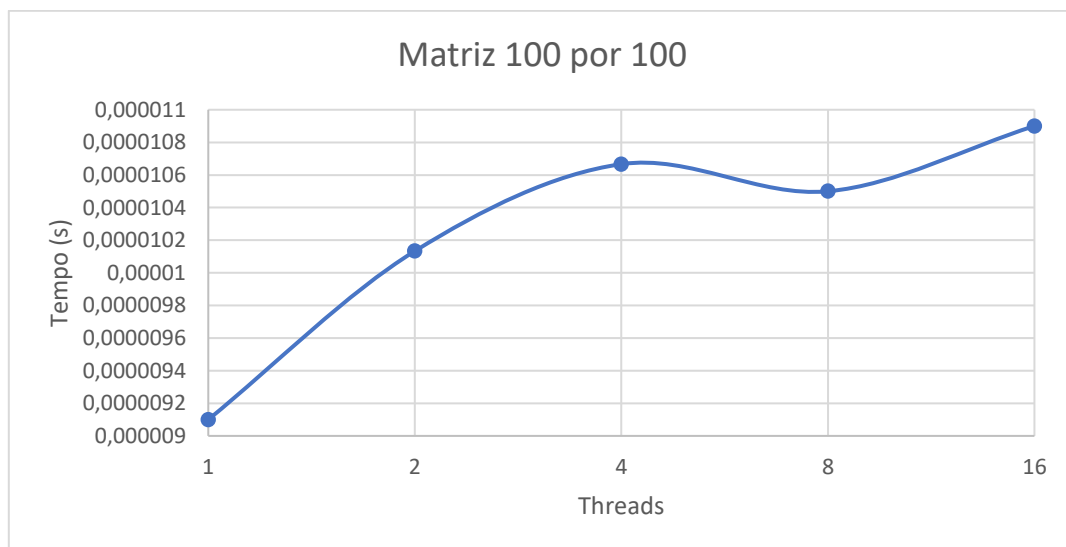
- As *threads* irão se auto organizar, o trabalho é proporcional ao tempo de vida de cada thread.
- Quando uma *thread* terminar o processamento ela irá buscar outra diagonal para processar.

Desvantagens:

- Para matrizes de tamanho pequeno e utilizando grande quantidade de threads ocorre que: as primeiras a serem criadas podem já ter terminado o processamento antes que as mais recentes tenham a chance de processar alguma diagonal.
- A necessidade do uso de um *mutex* cria um gargalo na execução que pode diminuir a velocidade de processamento.

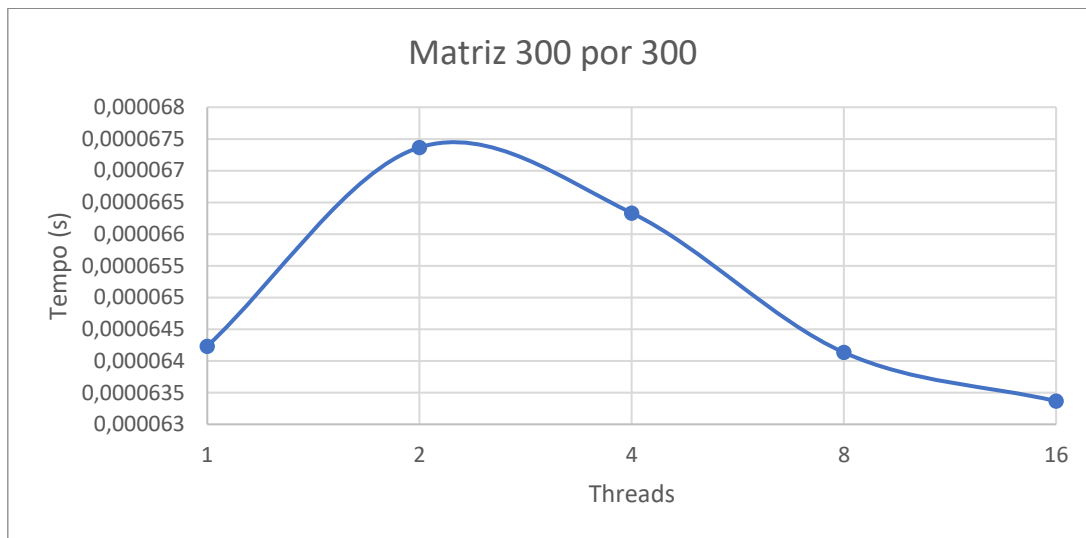
Eficiência do programa – visualização por meio de gráficos

Numa matriz de 100 por 100, o uso de threads se mostra ineficiente já que ao se aumentar a quantidade de threads o tempo gasto também aumenta. Isso ocorre, pois, o tempo para criar a thread não é compensado pela maior velocidade de processamento:

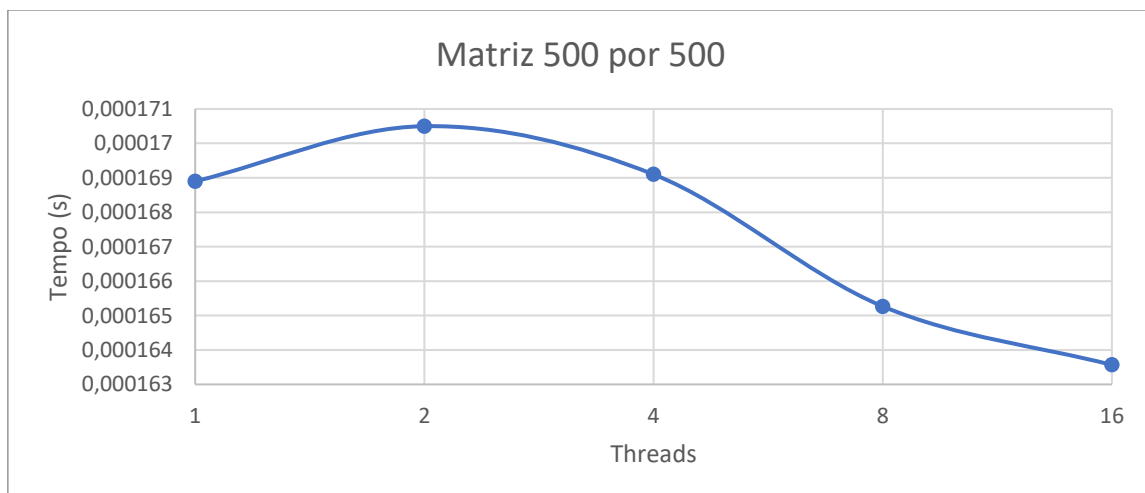


Com uma matriz de tamanho 300 por 300 o ganho no tempo de processamento passa a ficar mais aparente.

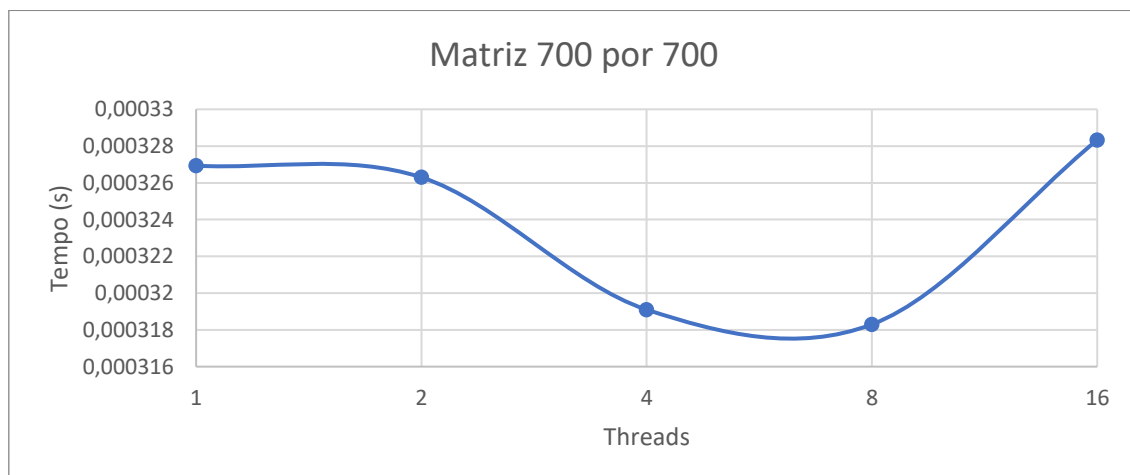
Apesar de inicialmente o tempo aumentar, ao chegar em 8 threads a maior velocidade ao processar vence o tempo de criação, reduzindo a duração do processamento:



Aumentando ainda mais o tamanho da matriz, o mesmo padrão da situação anterior se repete. A quantidade de threads que compensa o tempo de criação passa a acontecer quando há mais de quatro threads.



Numa matriz ainda maior o uso de mais que uma *thread* passa a compensar.

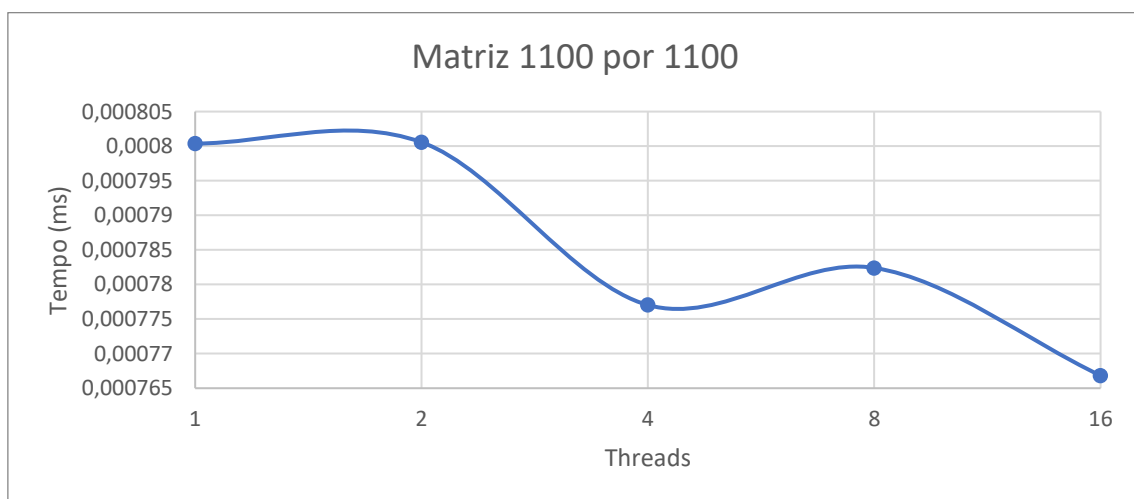
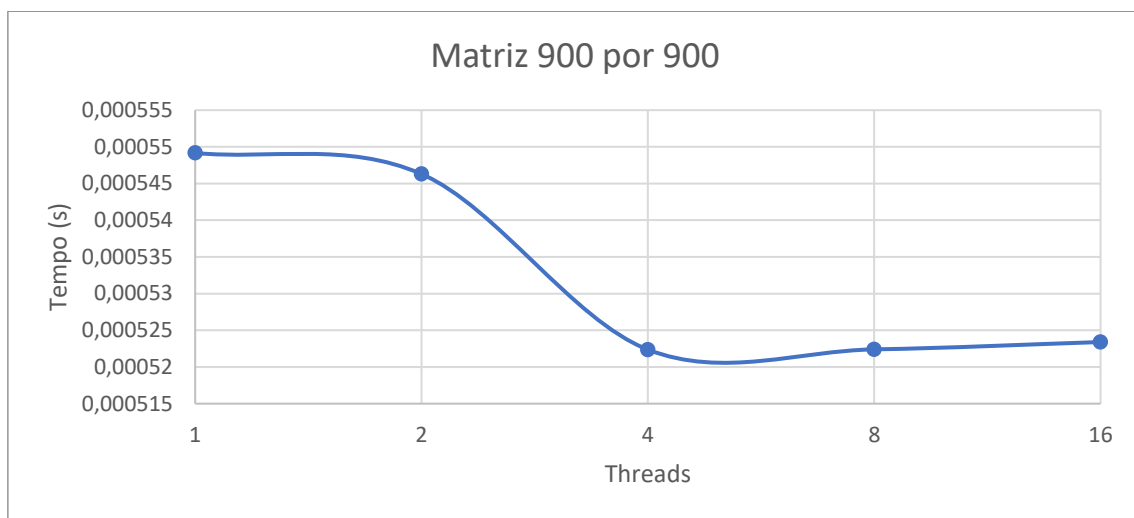


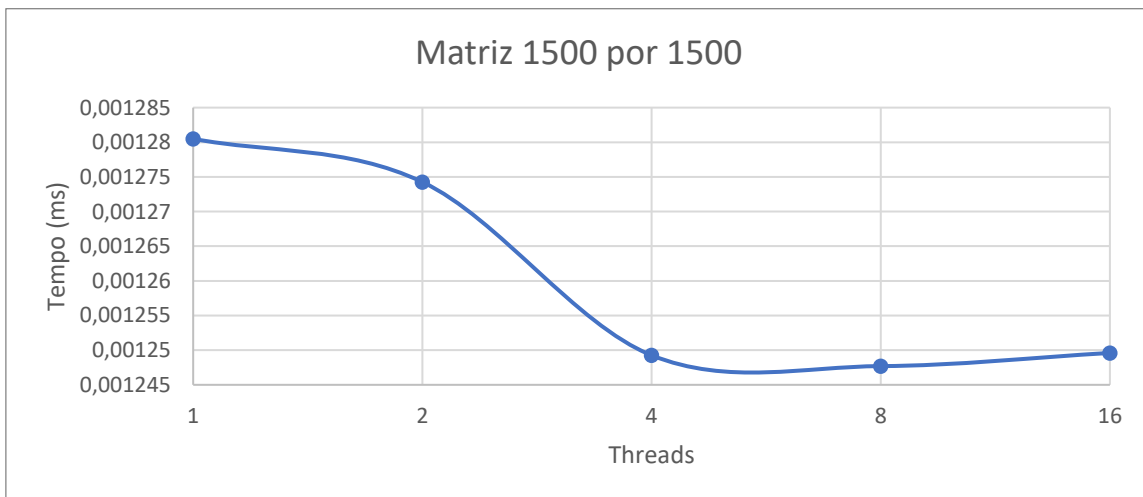
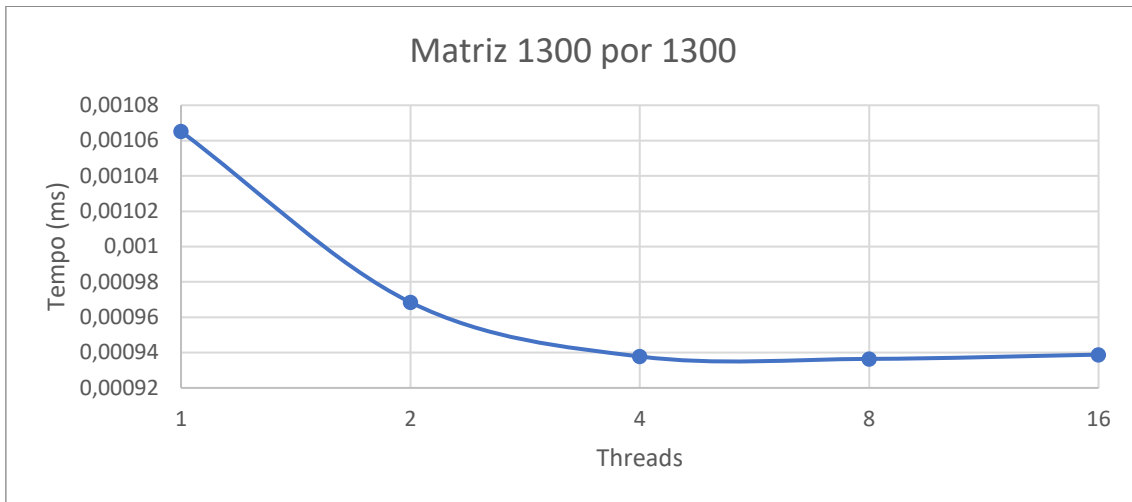
Para os demais resultados foi possível observar como a maior quantidade de *threads* aumenta a eficiência do programa.

Para valores mais altos um padrão evidente é o aumento no tempo de processamento com a criação de mais *threads*.

Como a relação entre tempo ganho ao criar mais uma *thread* e aumento na quantidade de trabalho não estão em proporção, eventualmente, a criação de mais *threads* levará num aumento da duração do processamento.

Isso se deve pela necessidade de trocar entre *threads* pausadas em execução já que no máximo quatro são executadas simultaneamente num computador com quatro núcleos, além disso pode-se somar o tempo para a criação de mais *threads*:





Solução B

Visão geral do algoritmo como uma série de passos

1. Ler a quantidade de threads T e as dimensões da matriz (M, N) a partir dos dados inseridos pelo usuário;
2. Criar a matriz com base nos dados inseridos pelo usuário;
3. Criar um vetor para o armazenamento da resposta;
4. Criar um vetor para o armazenamento dos Ids das *threads*;
5. Ler o arquivo de entrada de forma sequencial e colocar os valores na matriz criada (a matriz será preenchida do topo para a base com valores sendo inseridos da direita para a esquerda);
6. Crie um vetor de tamanho T para o armazenamento dos argumentos das *threads* que é formado por:
 - a. Quantidade de threads a criar T ;
 - b. A matriz base para a soma;
 - c. O vetor para armazenar a resposta;
 - d. Um número sequencial entre 0 e $(T - 1)$ que identifica a *thread*;
7. Para cada uma das T threads, faça:
 - a. Grave para o argumento da *thread* correspondente os valores fixos (itens a, b e c do tópico anterior);
 - b. No campo identificador grave o identificador correspondente a *thread* (ordem de criação da *thread* partindo de zero);
 - c. Crie uma nova *thread* enviando o argumento correspondente;
8. Espere o término do processamento;
9. Grave os valores do vetor de resposta no arquivo de resposta correspondente;
10. Imprima na tela os dados sobre a resposta (soma das diagonais)
11. Libere toda memória que foi alocada para vetores e para a matriz;

Algoritmo da soma por meio de *threads*

Nesse caso o trabalho será previamente dividido de forma lógica. Cada thread utilizará os valores do argumento para calcular quais diagonais ela deve processar:

A thread receberá um descritor da matriz a calcular, um descritor do vetor de resposta *vet*, um número identificador $id = (0, 1, 2, \dots, t)$ e a quantidade de threads t que serão criadas.

A thread irá, sucessivamente, enquanto o trabalho não estiver completo:

1. Calcular qual diagonal de número *diag* ela deve processar:
 - Na primeira execução: $diag = id$;
 - Nas próximas execuções do laço: $diag = diag + t$;

Exemplo: para $t = 4$, a thread de $id = 2$ (terceira a ser criada,) irá processar as diagonais de número: $(0 + 2 = 2)$; $(2 + 4 = 6)$; $(6 + 4 = 10)$; $(10 + 4 = 14)$; ...

2. Converter o número *diag* para as coordenadas (m, n) do primeiro elemento daquela diagonal;

3. Somar os elementos da diagonal até que o fim da matriz tenha sido atingido;
4. Armazenar a soma no vetor de resposta na posição *diag*.

Número da diagonal			
	2	1	0
3			
4			
5			
6			
7			
8			
9			
10			

4 THREADS		
IDENTIFICADOR	Diagonais processadas	Somas feitas
0		3
1		3
2		3
3		2

Figura 8 - Exemplo da divisão do trabalho entre threads com a Solução B

Vantagens

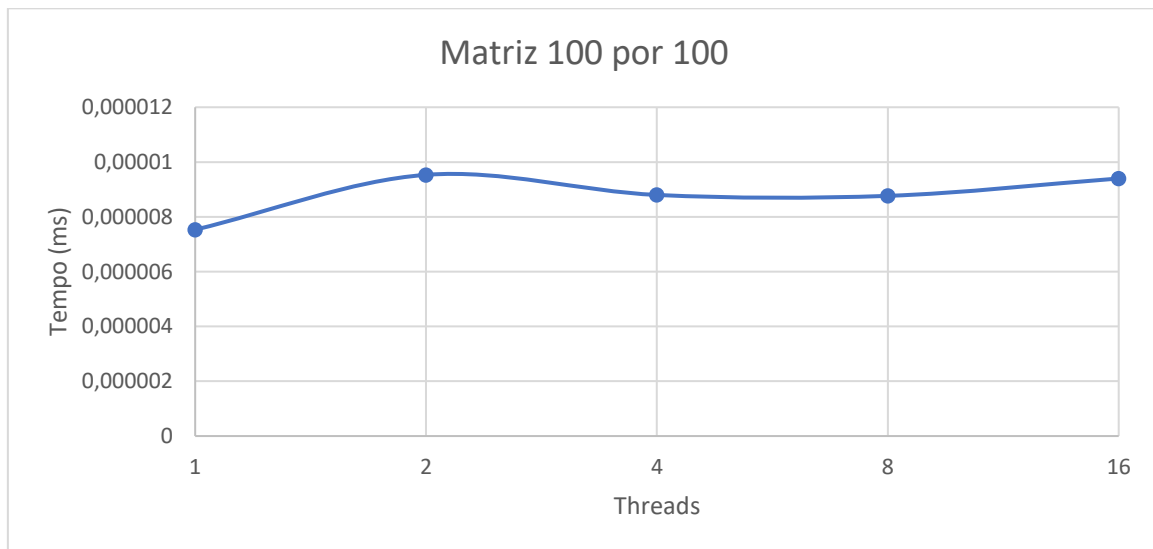
- Desde que o tamanho da matriz permita, toda *thread* criada efetuará algum processamento.
- A forma como é feita a divisão das diagonais maximiza a igualdade de trabalho entre as *threads* (efetuarão sempre uma quantidade total de operações de soma muito próximas).

Desvantagens

- Para matrizes muito pequenas uma *thread* pode acabar seu trabalho antes que a próxima seja criada. Provocando que para m e n baixos, seja mais eficiente usar menos threads do que mais.
- Threads mais antigas podem terminar o processamento antes das novas e quando uma *thread* terminar seu trabalho, não irá “ajudar” as demais.

Eficiência do programa - visualização por meio de gráficos

No caso da solução B, assim como na solução A, para uma matriz pequena não é vantajoso o uso de mais que uma *thread*.



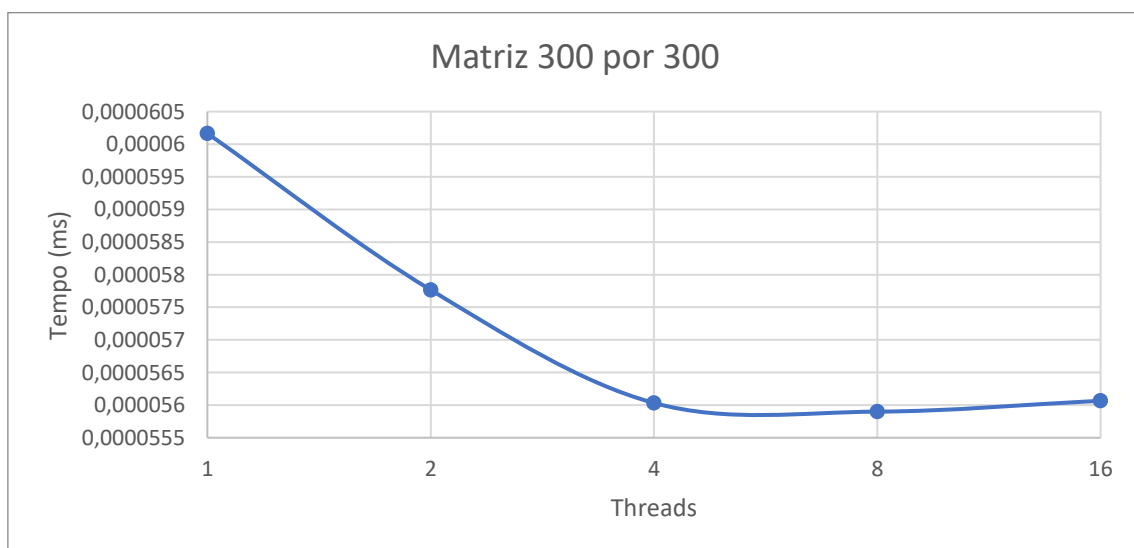
Apesar disso, ao se aumentar o tamanho da matriz a situação se inverte.

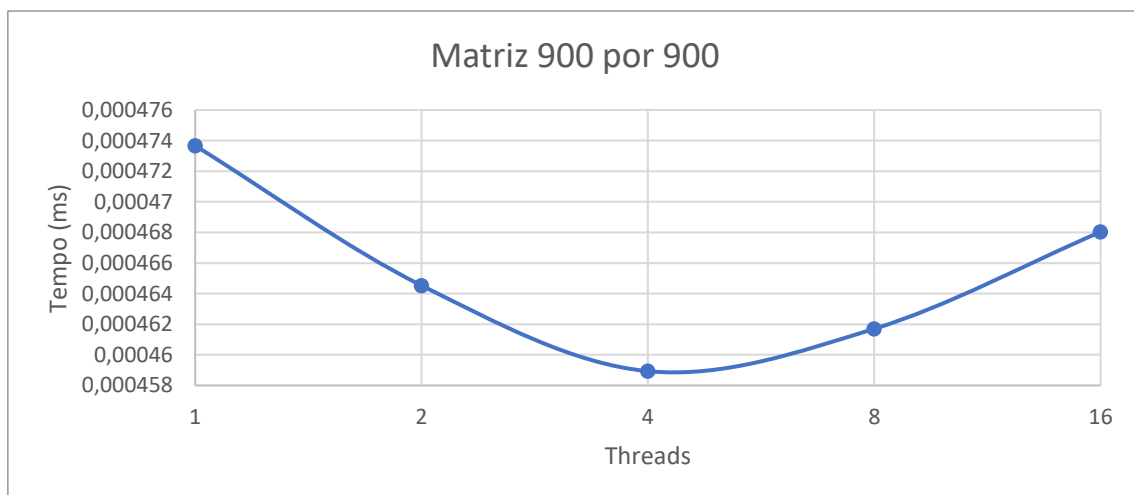
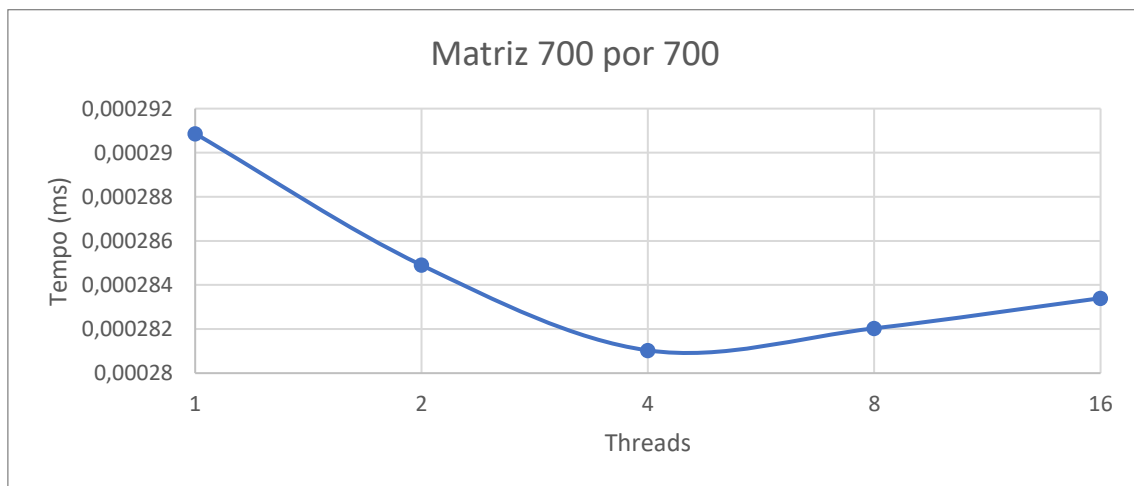
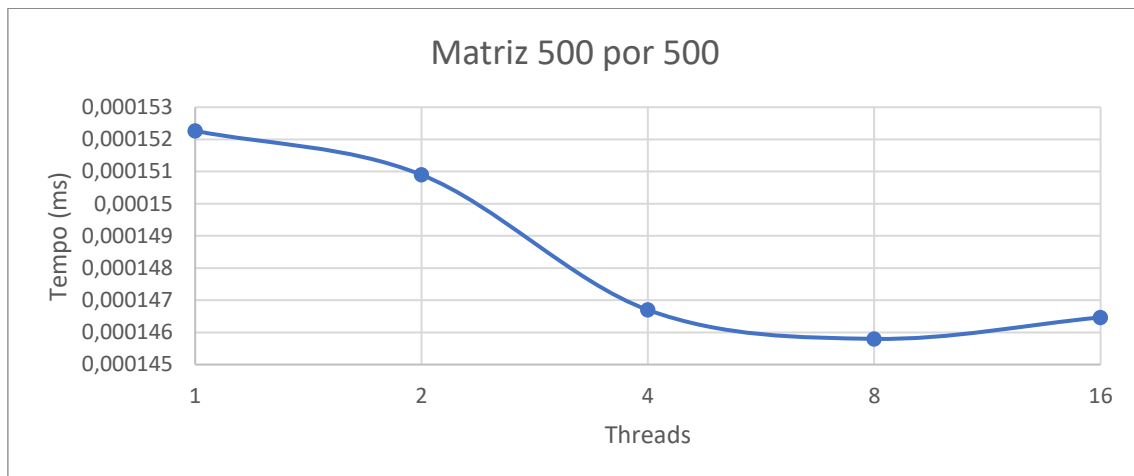
Como com quatro threads pode-se ocupar todos os núcleos a eficiência máxima é atingida. Com menos threads existiria capacidade de processamento desperdiçada e com mais threads no máximo quatro ficariam em execução simultaneamente (uma em cada núcleo).

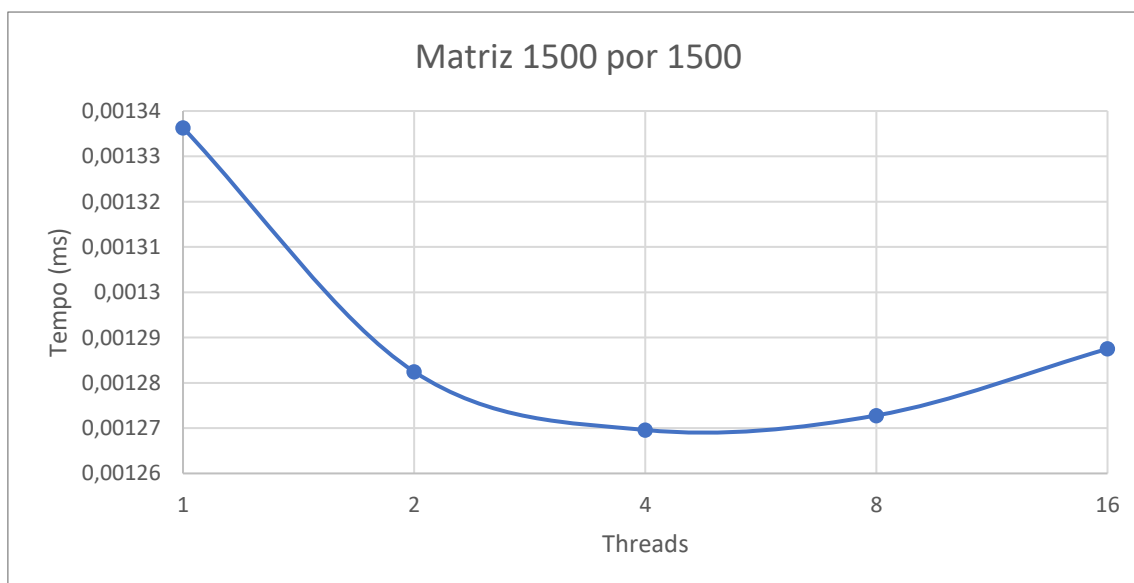
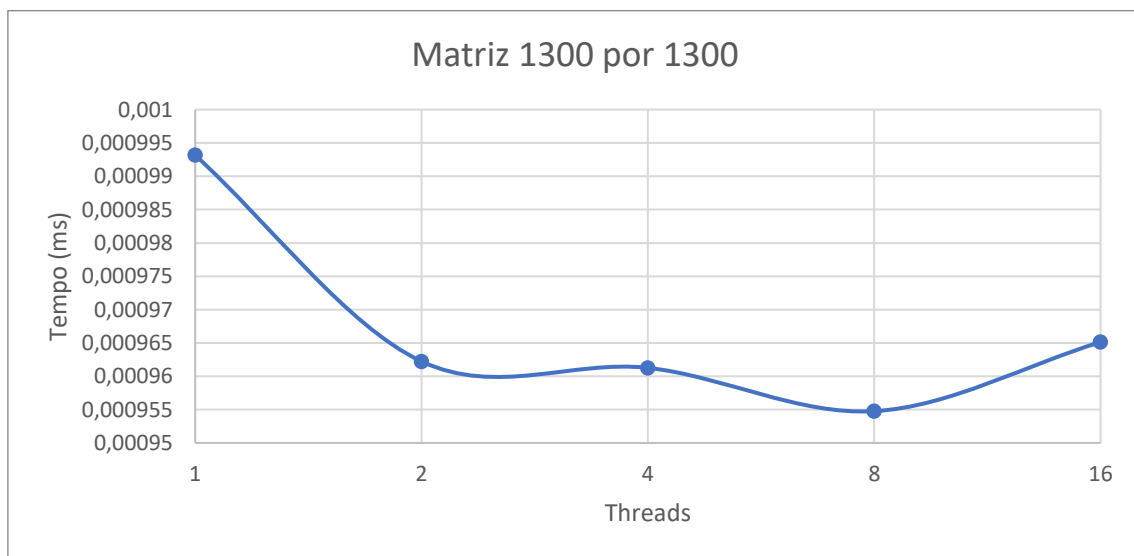
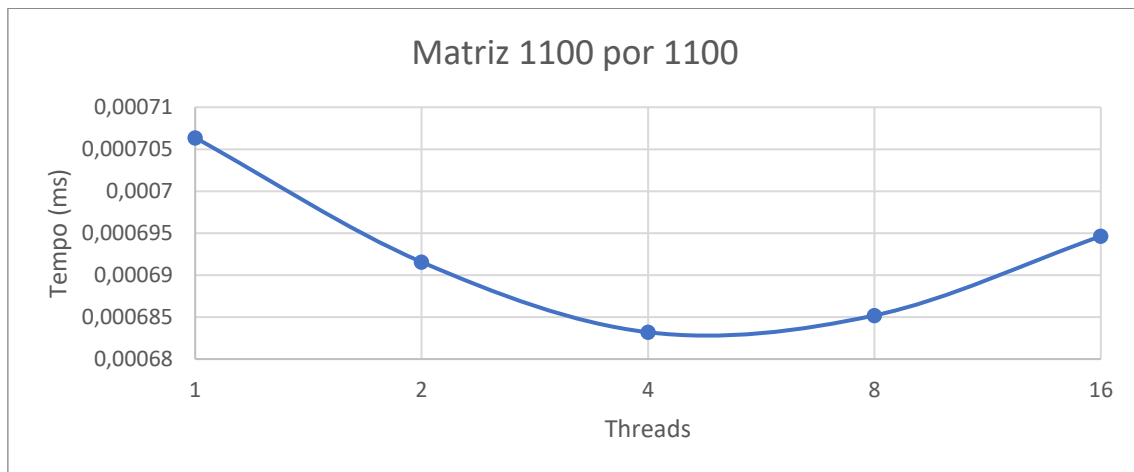
Com mais threads é necessário ainda que haja mais trocas entre as que estão em execução e as pausadas o que de forma acumulada faz o tempo de processamento aumentar.

Apesar do aumento, nos casos testados o tempo com mais threads mesmo que não na quantidade de máxima eficiência não atingem aquele quando se usa apenas uma *thread*.

Concluindo, mesmo com uma quantidade de threads não ideal, o uso de mais threads demonstra como o programa pode ser executado de forma mais eficiente que o quando se usa apenas uma:





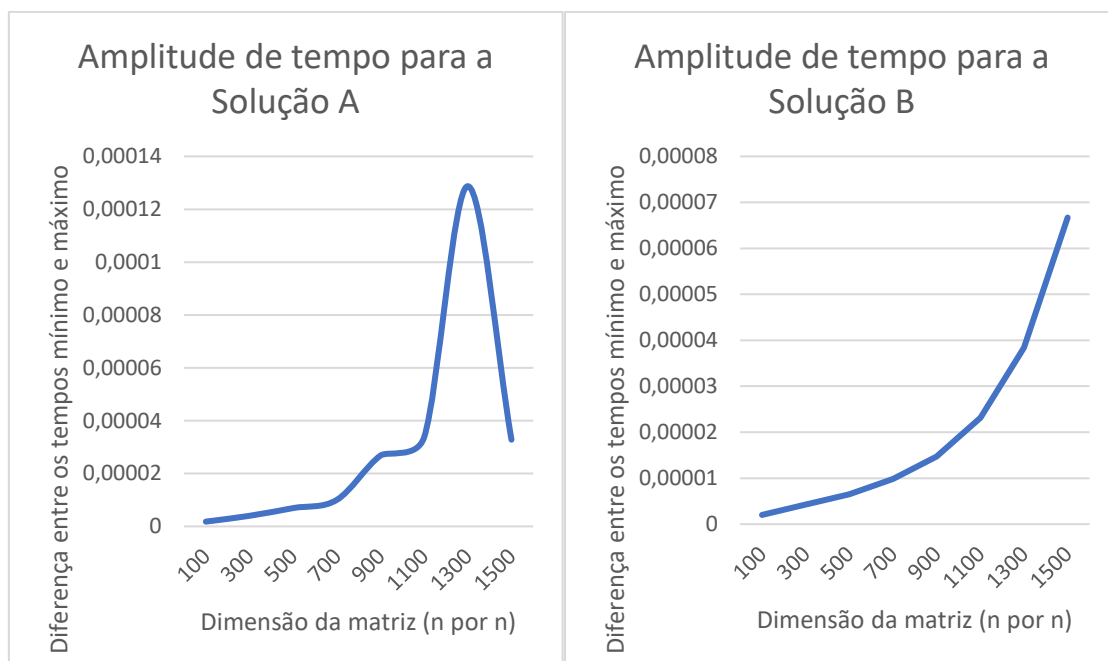


Conclusão

Independentemente da Solução utilizado fica claro que quando aplicado a certos problemas o uso das *threads* mostra que um ganho substancial no desempenho de determinado programa pode ser atingido.

Esse resultado pode ser observado à medida que se aumenta o trabalho por meio da ampliação das dimensões da matriz.

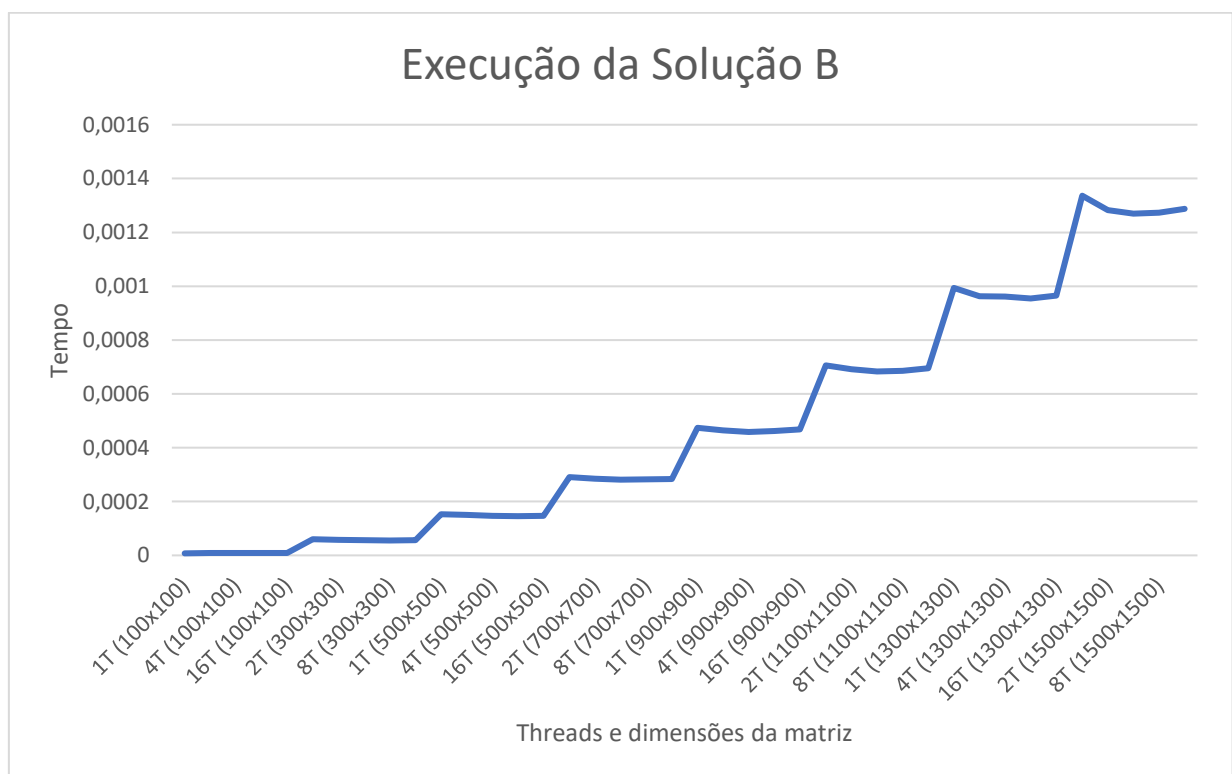
A diferença no tempo mínimo e máximo de execução cresce à medida que o trabalho cresce, denotando o aumento da eficiência ao se usar o *multithreading* a medida que a matriz aumenta:



Algo que também ficou evidente foi a relação entre a quantidade ideal de threads para o processamento e o número de núcleos da máquina.

Como a criação de threads consome tempo, o ideal é encontrar o mínimo número de threads que maximize o tempo de processamento. Mas mesmo que esse número não seja encontrado, no geral, o uso de mais que uma *thread* faz o processamento ocorrer de forma progressivamente mais rápida.

Colocando os testes realizados de forma conjunta é possível obter uma visão mais geral sobre o funcionamento:



O aumento do tamanho da matriz se reflete como um “degrau” no gráfico. A variação na quantidade de threads como as linhas horizontais em cada degrau.

Nos dois casos pode-se observar como ao aumentar o tamanho da matriz usar o *multithreading* reduz o tempo de processamento cada vez mais, pois, quanto mais altos os degraus, ou seja, quanto maior são as dimensões da matriz, mais inclinados esses degraus ficam, uma indicação de que quanto maior for o trabalho mais o uso de várias *threads* influencia na redução do tempo de execução, com exceção às quantidades muito elevadas de *threads*.

Material consultado

[POSIX Threads Programming](#)

[DAINF UFPR – Posix Threads - Fundamentos](#)

[Thegeekstuff – Mutex examples](#)

[Stackexchange – Why using more threads makes it slower than using less threads](#)

[Youtube – DbFaster: Tutorial PosixThreads](#)