UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação

Programação Paralela e Distribuída

Escalabilidade da Multiplicação de Matrizes

Prof. Helio Crestana Guardia

Integrantes do Grupo:

Caio Ueda Sampaio, 802215, BCC
Vinícius de Oliveira Guimarães, 802431, BCC

22 de Novembro de 2023

LINK PARA COLAB:

Link para o colab contendo os códigos que foram criados para realizar as paralelizações:
https://colab.research.google.com/drive/11zBBY3Tkz1tKExh6uBz2rqXytTY2B0Zg?usp=sharing

1. INTRODUÇÃO

Primeiramente, antes de realizar os testes de paralelização da multiplicação de matrizes, considerando um computador que possui 4 cores (8 processadores lógicos), paralelizamos a criação delas com números randômicos. Para isso, utilizamos o rand_r(), que é thread safe, já que utiliza como argumento o valor de um ponteiro para calcular o número pseudo-randômico. Assim, utilizamos omp_get_thread_num() na composição da seed de forma que cada thread possua uma seed diferente e, consequentemente, números gerados diferentes. Nota-se que cada thread necessita de sua própria cópia do iterador 'i', por isso a cláusula 'private()' é utilizada.

```
    // Paralelização da inicialização da matriz A
    #pragma omp parallel for private(i)
    for(i = 0; i < lin_a * col_a; i++) {
        unsigned int seed = (unsigned int)time(NULL) ^ (unsigned
int)omp_get_thread_num();
        A[i] = (float)rand_r(&seed) / (float)RAND_MAX;
    }

    // Paralelização da inicialização da matriz B
    #pragma omp parallel for private(i)
    for(i = 0; i < lin_b * col_b; i++) {
        unsigned int seed = (unsigned int)time(NULL) ^ (unsigned
int)omp_get_thread_num();
        B[i] = (float)rand_r(&seed) / (float)RAND_MAX;
    }
```

Código 1 - Código da paralelização da criação de matrizes

Antes de mostrar os resultados obtidos com a paralelização dos cálculos da multiplicação entre a matriz A e B, fizemos o cálculo sequencialmente Tratando-se das comparações das execuções com diferentes níveis de paralelização, estes são os resultados apenas com a paralelização da criação das matrizes, mantendo a multiplicação na forma sequencial (Fizemos isto para proporcionar meios de comparação):

Cálculo Sequencial usando threads apenas para inicialização das matrizes				
Número de Threads	**1º Run (s)**	**2º Run (s)**	**3º Run (s)**	**Média Elapsed (s)**
1	127,02	138,69	141,54	138,69
2	142,54	134,37	140,71	140,71
4	131,54	123,38	141,1	131,54
8	125,26	125,2	134,34	125,26
16	125,51	137,16	125,51	125,51
32	120,48	116,42	112,04	116,42
64	122,22	118,62	134,9	122,22
128	136,78	138,22	159,64	138,22

Imagem 1 - Resultados com cálculo sequencial e apenas inicialização paralela.

Gráfico mostrando a sequência de médias para cada número de threads utilizado:

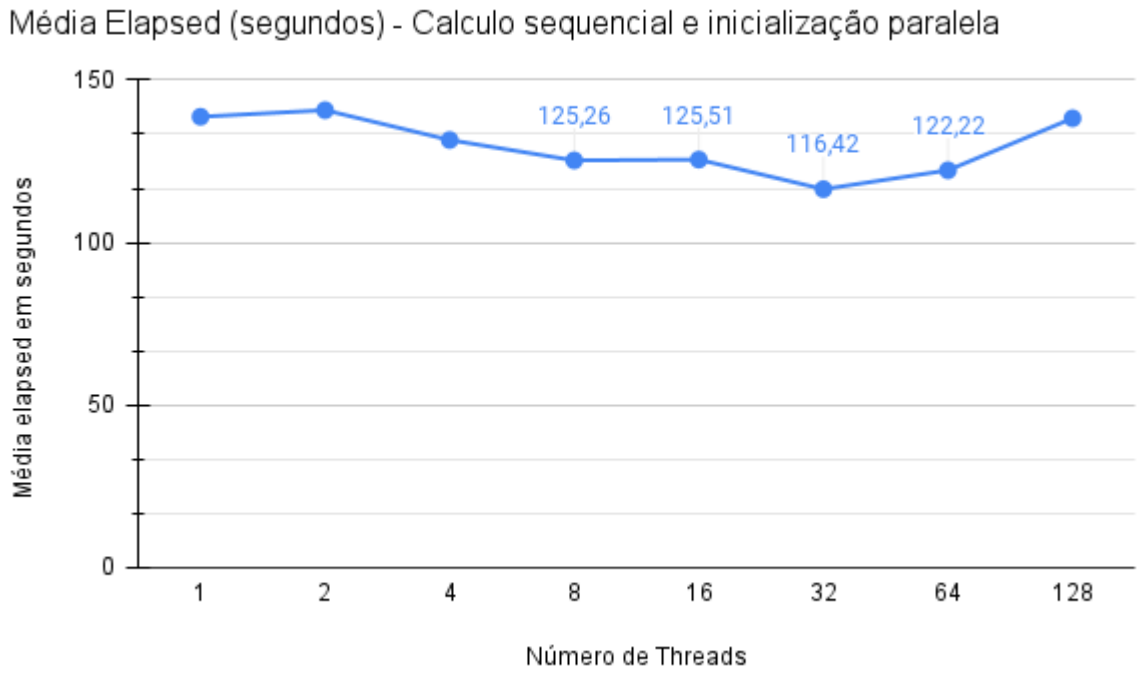

Imagem 2 - Gráfico de linhas da paralelização do cálculo sequencial e inicialização paralela

---

2. PARALELIZAÇÃO DOS CÁLCULOS

2.1 - PARALELIZAÇÃO - 1º FOR

```
// Paralelização considerando o primeiro for
#pragma omp parallel for private(i, j, k)
for(int i = 0; i < lin_c; i++) {
    for(j = 0; j < col_c; j++) {
```

```
            float auxC = 0;
            for(k = 0; k < col_a; k++) {
                auxC += A[i*col_a+k]* B[k*col_b+j];
            }
            C[i*col_c+j] = auxC;
        }
    }
```

Código 2 - Código da paralelização do primeiro 'for'

No primeiro teste de paralelização da multiplicação de matrizes, paralelizamos apenas o primeiro 'for'. Para isso, foi necessário adicionar a cláusula 'private' para tornar privados os iteradores dos laços para cada thread.

Assim, o que acontece é que, como o primeiro for itera o contador "i", percorrendo as linhas da matriz A quando "k" for incrementado, então estaremos dividindo a carga de trabalho de forma que cada thread executa um conjunto de linhas da matriz A (Ou seja, cada thread no final será responsável pelo cálculo de um conjunto de linhas na matriz resultante C). Como cada thread executa uma parte independente das outras, não há conflito de dependência e com isso não foi necessário utilizar mecanismos de sincronização.

Essa descrição feita está também representada na imagem abaixo, que mostra exatamente o que cada thread vai calcular. As linhas da matriz A com cores diferentes representam as diferentes threads, onde para cada linha de A, a thread específica irá calcular aplicando nos elementos de B, gerando assim os resultados nas linhas de C.

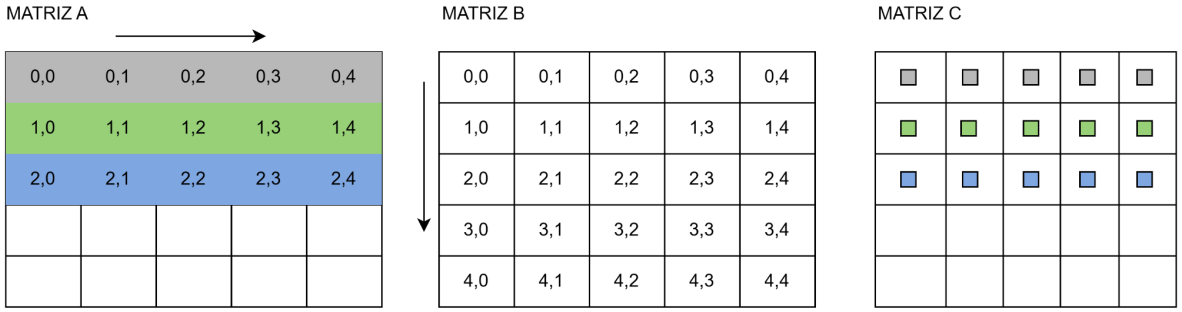

Imagem 3 - Representação da paralelização da multiplicação de matrizes considerando o 1º for.

Com isso, após realizar a execução para várias quantidades de threads (1, 2, 4, 8, 16, 32, 64 e 128), obtivemos os seguintes resultados em segundos mostrados nas duas imagens:

Paralelização no 1º for				
Número de Threads	1º Run (s)	2º Run (s)	3º Run (s)	Média Elapsed (segundos)
1	120,84	120,85	118,31	120,84
2	75,39	72,8	71,97	72,8
4	58,22	62,87	60,47	60,47
8	59,16	70,29	60,39	60,39
16	62,76	58,19	57,7	58,19
32	57,05	57,34	57,35	57,34
64	57,88	56,7	58,64	57,88
128	58,44	59,13	57,4	58,44

Imagem 4 - Resultados da paralelização no primeiro 'for'

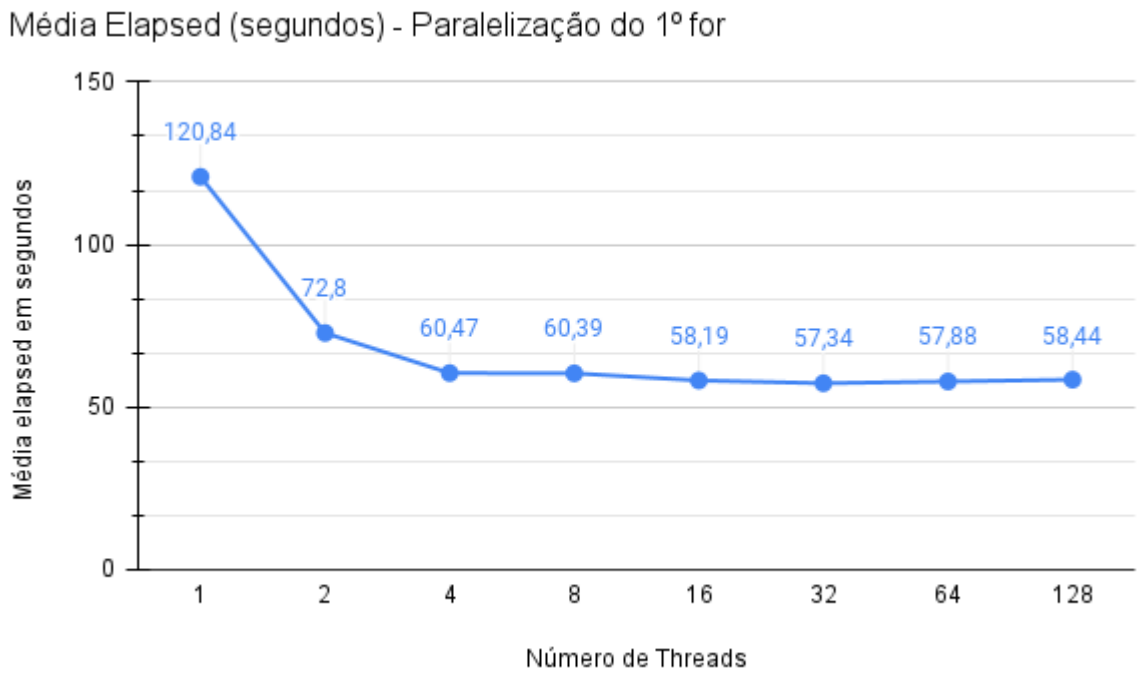

Imagem 5 - Gráfico de linhas da paralelização do 1º for

2.2 - PARALELIZAÇÃO - 2º FOR

```
// Paralelização considerando o segundo for
for(int i = 0; i < lin_c; i++) {
    #pragma omp parallel for private(j, k)
    for(j = 0; j < col_c; j++) {
        float auxC = 0;
        for(k = 0; k < col_a; k++) {
            auxC += A[i*col_a+k] * B[k*col_b+j];
        }
        C[i*col_c+j] = auxC;
    }
```

```
    }
```

Código 3 - Código da paralelização no segundo for

Na paralelização do segundo for, foi necessário a cláusula private para os iteradores ‘j’ e ‘k’, para que a divisão do segundo laço fosse feita corretamente pela biblioteca openmp.

O funcionamento da paralelização do segundo for é diferente da paralelização do primeiro for, já que, ao invés de ficarem responsáveis pelo cálculo do resultado de linhas da matriz C, as threads ficarão responsáveis pelo cálculo de colunas da matriz C. Como o segundo for divide as colunas da Matriz B entre as threads existentes, então ao multiplicar as linhas de A com as colunas de B, o resultado será aplicado na matriz C de forma correta, uma vez que também não há dependência entre os dados e nem necessidade de mecanismos de sincronização.

Além disso, é necessário lembrar que todas as matrizes estão sequencialmente alocadas na memória, de forma que o cache favoreça os cálculos da multiplicação entre as matrizes.

Essa explicação também está demonstrada através da imagem abaixo:

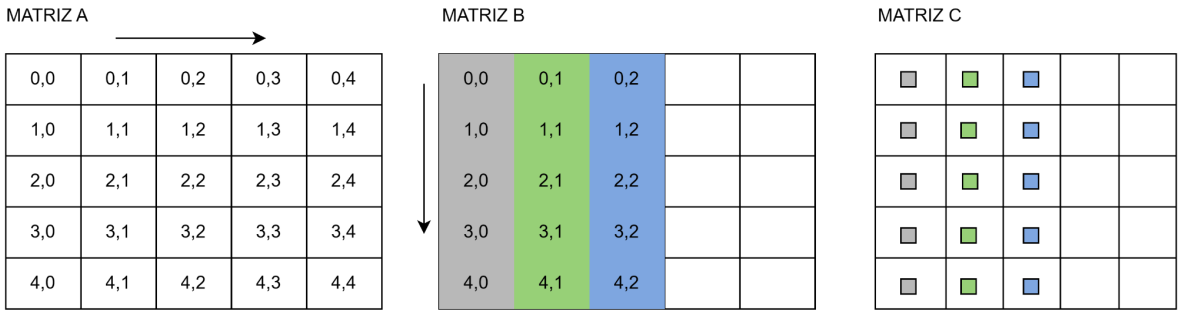

Imagem 6 - Representação da paralelização da multiplicação de matrizes considerando o 2º for.

Com isso, obtivemos os resultados a seguir para diferentes quantidades de threads:

Paralelização no 2º for				
Número de Threads	**1º Run (s)**	**2º Run (s)**	**3º Run (s)**	**Média Elapsed (s)**
1	124,54	123,7	118,44	123,7
2	82,78	75,65	76,68	76,68
4	70,24	65,85	68,4	68,4
8	63,73	62,47	62,09	62,47
16	60,64	62,6	60,98	60,98
32	63,82	62,46	60,99	62,46
64	62,61	63,21	67,02	63,21
128	67,24	68,94	67,27	67,27

Imagem 7 - Resultado da paralelização considerando o segundo for

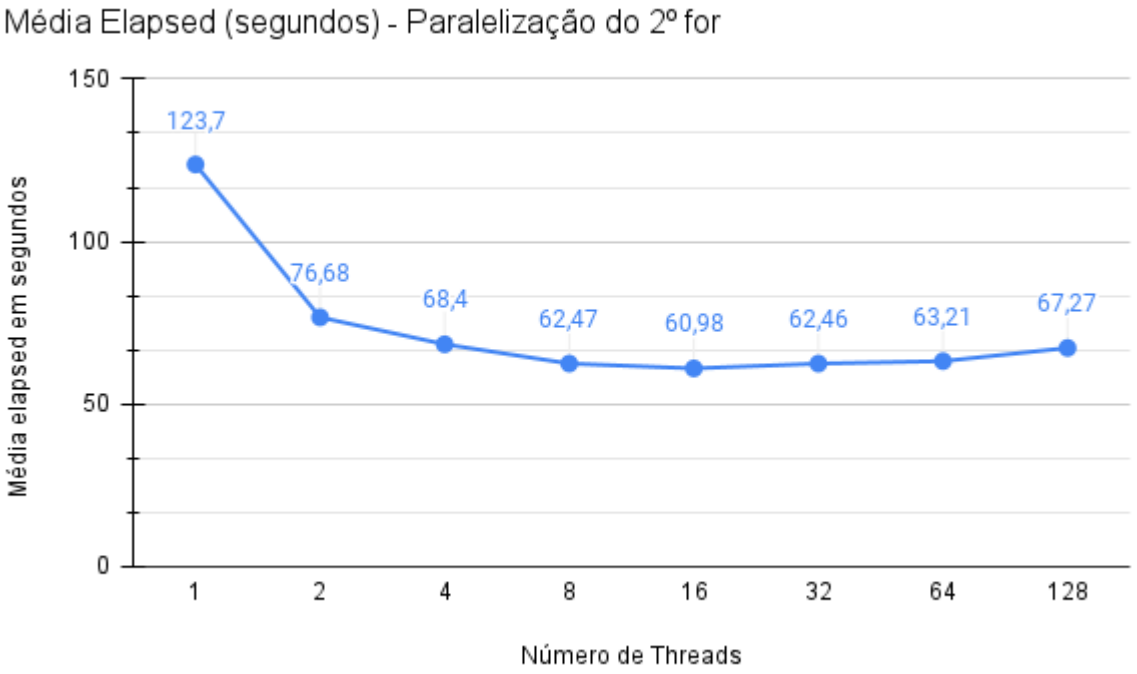

Imagem 8 - Gráfico de linhas da paralelização do 2º for

2.2 - PARALELIZAÇÃO - 3º FOR

```
// Paralelização considerando o terceiro for
for(int i = 0; i < lin_c; i++) {
    for(int j = 0; j < col_c; j++) {
        float auxC = 0;
        #pragma omp parallel for private(k) reduction(+:auxC)
        for(k = 0; k < col_a; k++) {
            auxC += A[i*col_a+k] * B[k*col_b+j];
        }
        C[i*col_c+j] = auxC;
    }
}
```

Código 4 - Código da paralelização no terceiro 'for'

A paralelização do terceiro for é muito diferente das duas paralelizações feitas anteriormente. Na forma atual cada thread ficará responsável por parte do cálculo de cada elemento da matriz C; ou seja, iremos compor o cálculo parcial de cada thread para cada elemento e só então conseguir obter o resultado correto dos elementos da matriz C somando os resultados parciais.

Para paralelizar o terceiro for foi preciso definir a variável "k" como sendo private e também inserir a expressão *reduction(+:auxC),* que permite cada thread calcular seu

resultado parcial (de acordo com o ‘k’ específico) e no final das execuções das threads, somar os resultados parciais das mesmas para dentro da variável *auxC* de maneira controlada e sincronizada.

Na imagem abaixo temos a representação de 3 threads, uma de cada cor (cinza, verde e azul), onde seus cálculos parciais (Representados por mini quadrados de cor variada) serão juntados para obter o resultado final de cada elemento na matriz C:

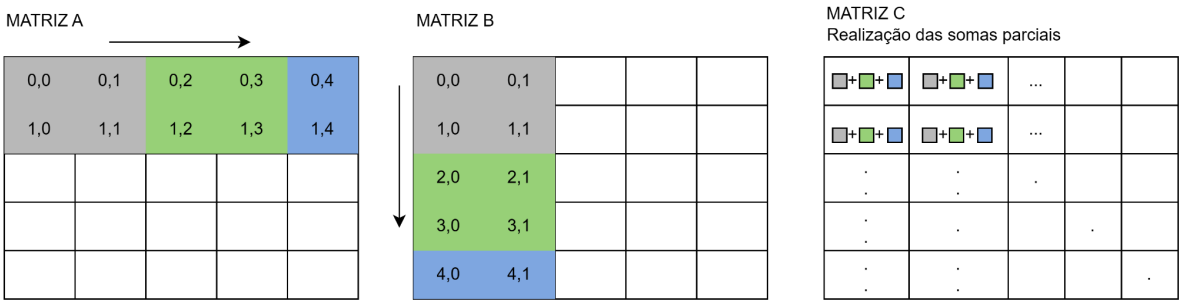

Imagem 9 - Representação da paralelização da multiplicação de matrizes considerando o 3º for (Somas parciais).

Durante o processo de execução do código com o terceiro laço paralelizado, observamos uma demora muito grande para realizar o cálculo. Após análises percebemos que o aumento no tempo de processamento foi causado principalmente por invalidação de cache e overhead de gerenciamento das threads, onde o sistema operacional teve grande dificuldade para conseguir variar as execuções entre as threads num ambiente onde o computador disponibilizava apenas 4 núcleos (8 processadores lógicos) para utilização.

Média do tempo Sys - Paralelização do 3º for		
Número de threads	**Em segundos**	**Em horas:minutos:segundos**
1	1,77	00:00:01
2	1,81	00:00:01.81
4	2,23	00:00:02.23
8	33,12	00:00:33.12
16	3127,29	00:52:07.29
32	4769,44	01:19:29.44
64	9552,2	02:39:12.20
128	19892,07	05:31:32.07

Imagem 10 - Tabela do tempo médio Sys para a paralelização do 3º for

Paralelização no 3º for				
Número de Threads	1º Run (s)	2º Run (s)	3º Run (s)	Média Elapsed (segundos)
1	123,1	122,5	121,7	122,5
2	84,4	77,8	75,2	77,8
4	61,5	97,1	60,8	61,5
8	257,0	266,5	357,0	266,5
16	2798,6	2738,6	1834,4	2738,6
32	3622,0	3617,0	3614,0	3617,0
64	7134,0	7115,0	7113,0	7115,0
128	1855,2	19892,1	14762,0	14762,0

Imagem 11 - Tabela do tempo médio elapsed para a paralelização do 3º for

Podemos observar nas duas imagens acima que os processadores passaram a maior parte do tempo executando operações do sistema operacional para as threads, ou seja, fazendo o gerenciamento das threads e alternando suas execuções. Outro ponto a ser lembrado é que como as threads calculam o resultado parcial dos elementos, ainda teve o tempo necessário para pegar o resultado parcial de cada thread e somar para obter o valor resultante de cada posição da matriz C.

Com isso, a paralelização do terceiro laço obteve o pior desempenho dentre as 3 paralelizações diferentes realizadas.

Na imagem abaixo ainda temos o gráfico de linhas que relaciona os valores médios do tempo elapsed para a paralelização do 3º for:

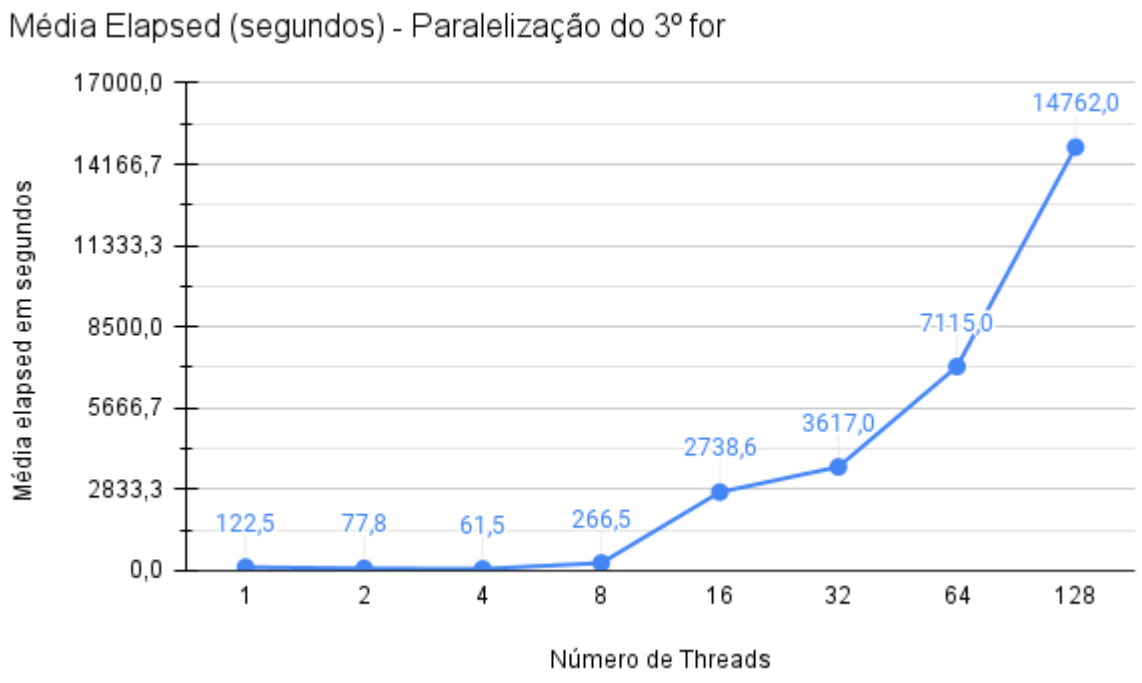

Imagem 11 - Gráfico de linhas da paralelização do 3º for (Elapsed time)

Também temos a seguir o gráfico de linha que demonstra o grande tempo gasto executando operações do sistema operacional para as threads criadas. É possível observar que quanto maior a quantidade de threads criadas superior a 8 (Número de threads que conseguem ser executadas por padrão em paralelo no computador utilizado), maior é o tempo gasto gerenciando e alternando a execução entre elas, já que apenas 8 conseguem ser executadas ao mesmo tempo, ocasionando então um overhead de processamento. (Isso é mais acentuado na paralelização do 3º for pois é necessário várias threads calcularem valores parciais para só depois chegar no resultado de cada elemento da matriz C)

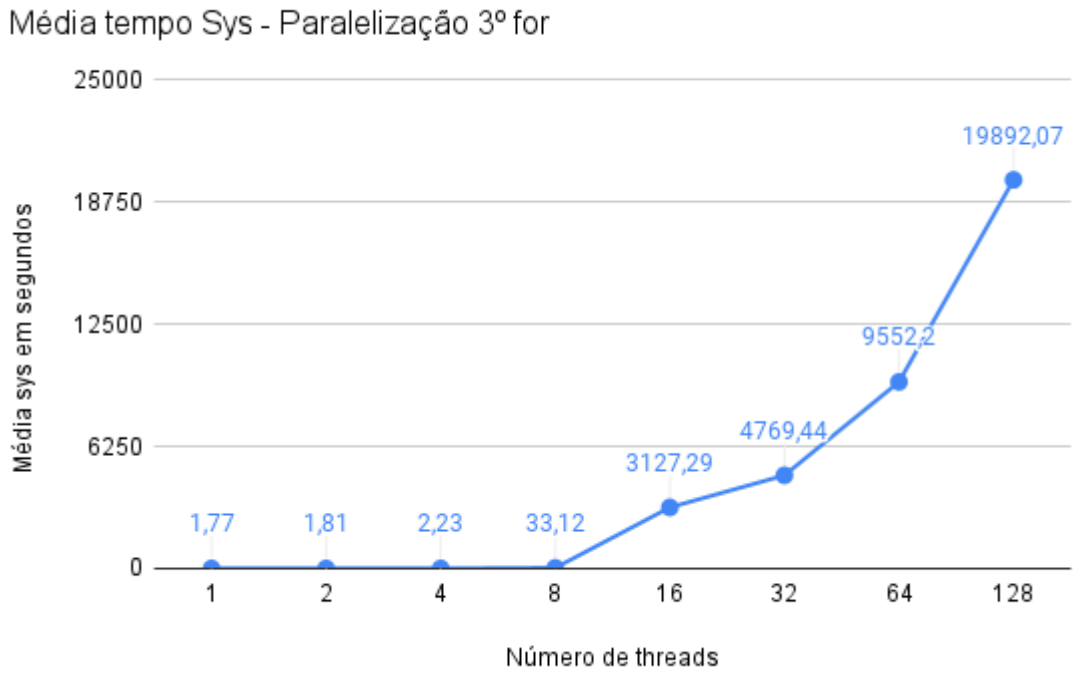

Imagem 12 - Gráfico de linhas da paralelização do 3º for (Sys time)