UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
TÓPICOS ESPECIAIS EM INTERNET DAS COISAS “B” - **IMD291**

# ANÁLISE DE ALGORITMOS PARALELOS E SERIAIS
## MULTIPLICAÇÃO DE MATRIZES QUADRADAS

Prof. Dr. Kayo Gonçalves e Silva
**Discente**: Oziel Alves do Nascimento Júnior
**Matrícula**: 20170065711

Natal, 2020

README.md

# Análise de Algoritmos Paralelos e Seriais

## Multiplicação de Matrizes

Universidade Federal do Rio Grande do Norte (UFRN), 2020.

Análise por:

Oziel Alves

Esta análise se encontra disponível em:

`https://github.com/ozielalves/prog-paralela/tree/master/Multz`

## Sumário

# Introdução

## Objetivos

Esta analise tem como propósito a avaliação do comportamento de códigos seriais e de um código paralelo referentes a implementação do algoritmo de **Multiplicação de matrizes**. Será destacado o speedud e a eficiência do código paralelo em relação a 2 códigos seriais, levando em consideração os tempos de execução e tamanhos de problema para composição dos resultados finais da análise. Os cenários irão simular a execução dos programas para 1 (serial), 4, 8, 16 e 32 threads, com 4 tamanhos de problema, definidos empiricamente com o objetivo de atingir o tempo mínimo de execução determinado pela referência desta análise para os limites do intervalo de tamanhos.

## Dependência

### G++ Compiler

É necessário para a compilação dos programam, visto que são feitos em c++.

```
# Instalação no Ubuntu 20.04 LTS:
sudo apt-get install g++
```

## Compilação e Execução

Instalada a dependência, basta executar o shellscript determinado para a devida bateria de execuções na raiz do repositório:
Serão realizadas **10 execuções** com **4 tamanhos de problema** , em **5 quantidades de threads** (1, 4, 8, 16 e 32).

```
# Para o algoritmo serial de multiplicação de matrizes usando o princípio da localidade
./multz1_serial_start.sh
```

```
# Para o algoritmo serial de multiplicação de matrizes com acesso de memória aleatório
./multz2_serial_start.sh
```

```
# Para o algoritmo paralelo de multiplicação de matrizes
./multz_paralelo_start.sh
```

**Obs.:** Caso seja necessário conceder permissão máxima para os scripts, execute `chmod 777 [NOME DO SCRIPT].sh`.

## Arquivo com Resultados

Após o termino das execuções do script é possível ter acesso aos arquivos de tempo `.txt` na pasta `serial` ou `paralelo`, de acordo com o script executado. Os resultantes da execução desses scripts foram utilizados para realização desta análise.

## Condições de Testes

### Informações sobre a máquina utilizada

- **Supercomputador (UFRN) - Nó computacional em lâmina**
- **Processador**: 2 x CPU Intel Xeon Sixteen-Core E5-2698v3 de 2.3 GHz/40M cache/ 9.6 GT/s
- **Número de Cores/Threads**: 32/32
- **Memória**: 128 GB tipo DDR4 – 2133MHz RDIMM (8 x 16GB)
- **Sistema**: Centos 6.5 x86_64

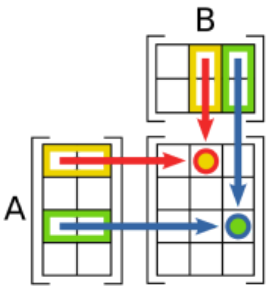
## Apresentação do Algoritmo

### Multiplicação de Matrizes Quadradas

O produto de duas matrizes é definido somente quando o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz. Se a matriz `A` é uma matriz **m**×**n** e `B` é uma matriz **n**×**p**, então a matriz `produto` resultante da multiplicação de `A` por `B` é uma matriz **m**×**p**. O elemento de cada entrada da matriz `produto` é dado pelo produto da **i**-ésima linha de `A` com a **j**-ésima coluna de `B`, ou seja:

$$(produto)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

O algoritmo desenvolvido para esta análise realiza a multiplicação de matrizes **m**×**m**. A imagem abaixo representa de maneira intuitiva o procedimento de multiplicação de duas matrizes.




**Referência**: José Ruy, Giovanni (2002). Matemática fundamental: uma nova abordagem. São Paulo: FTD.

### Código serial usando o princípio da localidade (01)

Dado um número `size` de tamanho de matriz para criação das matrizes quadradas, a seguinte sub-rotina é implementada:

1. A matriz `fator_a` é alocada com `size` linhas e colunas e inicializada com inteiros semi-randômicos.
2. Em seguida, a matriz `fator_b` é também inicializada a partir de uma cópia da matriz `fator_a` utilizando `memcpy`.
3. Feito isto, a matriz `produto` é também alocada com `size` linhas e colunas.
4. Após a fase de inicialização das matrizes é então chamada a função que realiza a multiplicação da matriz `fator_a` pela matriz `fator_b`.

A implementação da função `MULTZ` é apresentada abaixo:

```
# Multiplica duas matrizes contemplando o princípio da localidade
void MULTZ(int size, int **fator_a, int **fator_b, int **produto)
{
    int i, j, k;            # Variáveis auxiliares
    int row_start, row_end; # Inicio e fim da linha da matriz
    int sum;                # Armazena o produto da multiplicação

    row_start = 0;
    row_end = size;

    # Para cada linha na matriz "fator_a"
    for (i = row_start; i < row_end; i++)
    {
        # Para cada coluna na matriz "fator_b"
        for (j = 0; j < size; j++)
        {
```

```
            sum = 0;
            for (k = 0; k < size; k++)
            {
                sum += fator_a[i][k] * fator_b[k][j];
            }
            produto[i][j] = sum;
        }
    }
}
```

#### Código serial com acesso de memória aleatório (02)

Dado um número `size` de tamanho de matriz para criação das matrizes quadradas, uma rotina idêntica a anterior é implementada, a alteração está somente na implementação da função `MULTZ`:

```
# Multiplica duas matrizes não contemplando o princípio da
void MULTZ(int size, int **fator_a, int **fator_b, int **produto)
{
    int i, j, k;                            # Variáveis auxiliares
    vector<int> rows(size), columns(size); # Vetor de Linhas e de Colunas da matriz
    int sum = 0;                            # Armazena o produto da multiplicação

    # Populando vectors de linhas e colunas
    for (i = 0; i < size; i++)
    {
        rows[i] = i;
        columns[i] = i;
    }

    # Aleatorizando as linhas e colunas
    random_shuffle(rows.begin(), rows.end());
    random_shuffle(columns.begin(), columns.end());

    # Para cada linha na matriz "fator_a"
    for (i = 0; i < size; i++)
    {
        # Para cada coluna na matriz "fator_b"
        for (j = 0; j < size; j++)
        {
            sum = 0;
            for (k = 0; k < size; k++)
            {
                sum += fator_a[rows[i]][k] * fator_b[k][columns[j]];
            }
            produto[rows[i]][columns[j]] = sum;
        }
    }
}
```

#### Código paralelo

Ainda sendo `size` o tamanho de matriz para criação das matrizes quadradas e `num_threads` o número de threads que serão utilizadas na execução do programa, a seguinte sub-rotina é implementada:

1. O vetor de threads é alocado de acordo com o número de threads `num_threads` fornecido.
2. As matrizes `fator_a`, `fator_b` e `produto` são alocadas com `size` linhas e colunas.
3. Em seguida, as matrizes `fator_a` e `fator_b` são inicializadas com inteiros semi-randômicos.
4. Após a fase de inicialização das matrizes é então iniciado o processo de multiplicação usando multithreading, a função chamada para cada thread é a `PTH_MULTZ` que irá realizar a multiplicação de uma fatia da matriz `fator_a` pela matriz `fator_b` em cada thread.

A implementação da função `PTH_MULTZ` é apresentada abaixo:

```
# Rotina da Thread.
# Cada thread realiza a multiplicação de uma fatia da matriz "fator_a" pela matriz "fator_b".
void *PTH_MULTZ(void *arg)
{
  int i, j, k;            # Variáveis auxiliáres
  int thread_id;          # Thread ID
  int slice;              # Fatia de multiplicação de cada thread
  int row_start, row_end; # Inicio e fim da fatia
  long sum;               # Armazena o produto da multiplicação

  thread_id = *(int *)(arg); # Recebe o ID da thread alocada sequencialmente.
  slice = size / num_threads;
  row_start = thread_id * slice;
  row_end = (thread_id + 1) * slice;

  # Para cada linha na matriz "fator_a"
  for (i = row_start; i < row_end; ++i)
  {
    # Para cada coluna na matriz "fator_b"
    for (j = 0; j < size; ++j)
    {
      sum = 0;
      for (k = 0; k < size; ++k)
      {
        sum += fator_a[i][k] * fator_b[k][j];
      }
      produto[i][j] = sum;
    }
  }
}
```

**Obs.:** Note que o algoritmo paralelo utiliza o **princípio da localidade** também implementado no código serial 01.

## Desenvolvimento

Para esta análise, serão realizadas **10 execuções** com tamanhos de problema **1.408**, **1.664**, **1.920** e **2.176**, em **5 quantidades de threads** (1, 4, 8, 16 e 32). Se espera que o código paralelo consiga valores de speedup relevantes em relação ao tempo de execução para os códigos serias. Além disso, também é esperado que a eficiência do algoritmo paralelo, quanto à multiplicação das matrizes, apresente valores parecidos para as demais quantidades de threads utilizadas quando é aumentado somente o tamanho do problema. Uma descrição completa da máquina de testes pode ser encontrada no tópico Condições de Testes.

### Corretude

Para validar a corretude dos algoritmos implementados foi realizado um teste utilizando **6** como tamanho de problema para para ambos os códigos seriais e para o código paralelo:

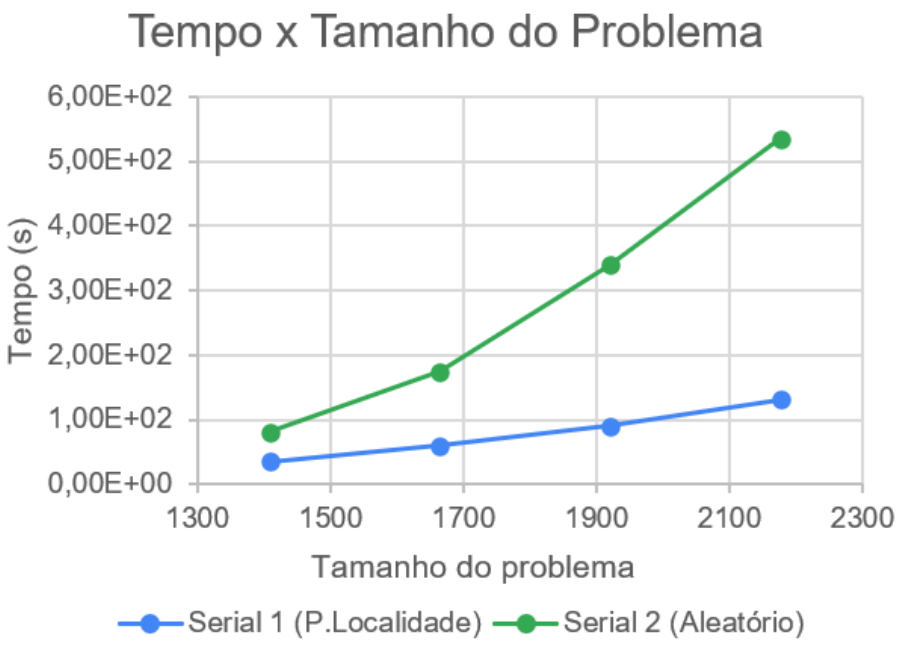
```
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ g++ -g -o ./serial/multz1 ./serial/multz1.cpp
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ ./serial/multz1 6
Matrix 1:

        |    6   0   1   1   2   8    |
        |    1   0   5   3   4   3    |
        |    7   4   6   2   2   8    |
        |    8   9   7   2   9   3    |
        |    7   9   9   4   1   6    |
        |    3   7   6   4   0   0    |

Matrix 2:

        |    6   0   1   1   2   8    |
        |    1   0   5   3   4   3    |
        |    7   4   6   2   2   8    |
        |    8   9   7   2   9   3    |
        |    7   9   9   4   1   6    |
        |    3   7   6   4   0   0    |

Resultado:

        |   89  87  85  50  25  71    |
        |  102 104 106  45  43  81    |
        |  142 116 143  75  62 134    |
        |  194 148 208 101  93 207    |
        |  171 123 179  88 105 173    |
        |   99  60 102  44  82 105    |
```

```
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ g++ -g -o ./serial/multz2 ./serial/multz2.cpp
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ ./serial/multz2 6
Matrix 1:

        |    6   0   1   1   2   8    |
        |    1   0   5   3   4   3    |
        |    7   4   6   2   2   8    |
        |    8   9   7   2   9   3    |
        |    7   9   9   4   1   6    |
        |    3   7   6   4   0   0    |

Matrix 2:

        |    6   0   1   1   2   8    |
        |    1   0   5   3   4   3    |
        |    7   4   6   2   2   8    |
        |    8   9   7   2   9   3    |
        |    7   9   9   4   1   6    |
        |    3   7   6   4   0   0    |

Resultado:

        |   89  87  85  50  25  71    |
        |  102 104 106  45  43  81    |
        |  142 116 143  75  62 134    |
        |  194 148 208 101  93 207    |
        |  171 123 179  88 105 173    |
        |   99  60 102  44  82 105    |
```

```
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ g++ -pthread -w -o ./paralelo/pth_multz ./paralelo/pth_multz.cpp
oziel@DESKTOP-CM0S03F:/mnt/c/Users/oziel/OneDrive/Documents/projects/prog-paralela/Multz$ ./paralelo/pth_multz 2 6
Matrix 1:

        |    7   9   4   2   0   4    |
        |    6   9   8   3   6   9    |
        |    5   4   5   8   7   6    |
        |    4   8   9   8   8   4    |
        |    5   7   5   8   0   9    |
        |    4   9   0   8   1   2    |

Matrix 2:

        |    7   9   4   2   0   4    |
        |    6   9   8   3   6   9    |
        |    5   4   5   8   7   6    |
        |    4   8   9   8   8   4    |
        |    5   7   5   8   0   9    |
        |    4   9   0   8   1   2    |

Resultado:

        |  147 212 138 121 102 149    |
        |  214 314 193 247 143 237    |
        |  175 268 184 230 129 193    |
        |  209 300 237 264 179 254    |
        |  170 273 173 207 150 163    |
        |  127 206 165 123 120 142    |
```

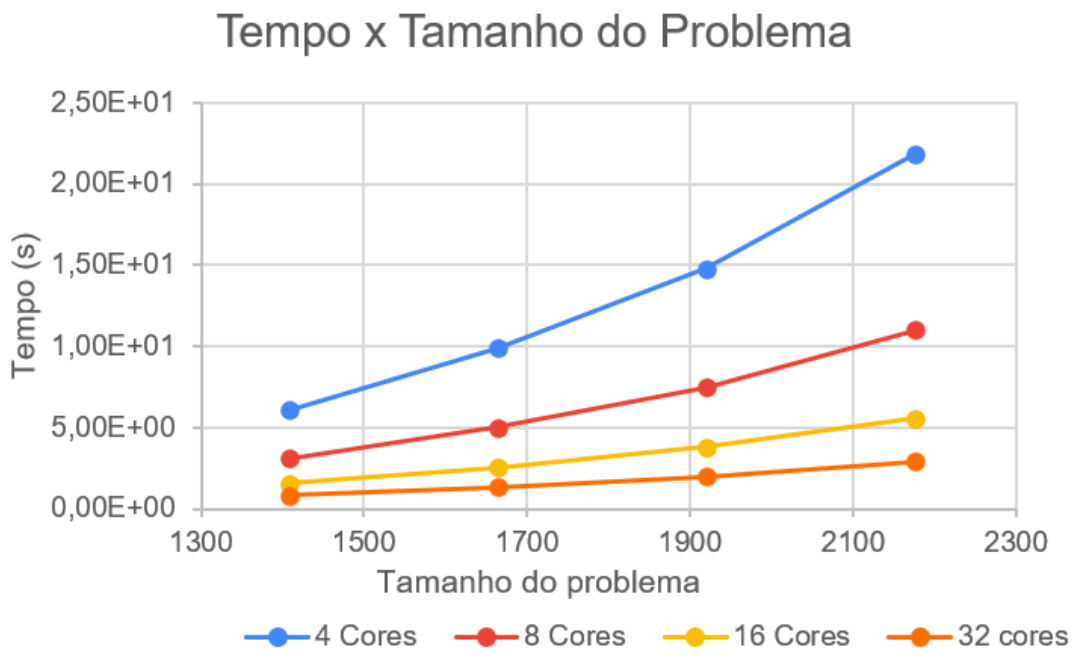
Como é possível perceber através dos prints de execução em terminal, todos os códigos conseguem realizar corretamente a multiplicação das matrizes geradas.

### Gráficos




Através do gráfico de tempos por problema para os códigos seriais, é possível observar que o código serial 1 consegue resolver o problema em um menor tempo para todos os tamanhos de problema quando comparado ao código serial 2. Isso acontece porque o Sistema de Memória tende a manter dados e instruções próximos aos que estão sendo executados no topo da Hierarquia de Memória, dessa forma, vetores e matrizes são armazenados em sequência de acordo com seus índices. Por isso, ao utilizar o **Principio da Localidade Espacial**, o código serial 1 ganha vantagem em cima do código serial 2 - que acessa os índices das matrizes de forma aleatória - quando gasta um tempo bem menor para acessar os índices consecutivos das matrizes.

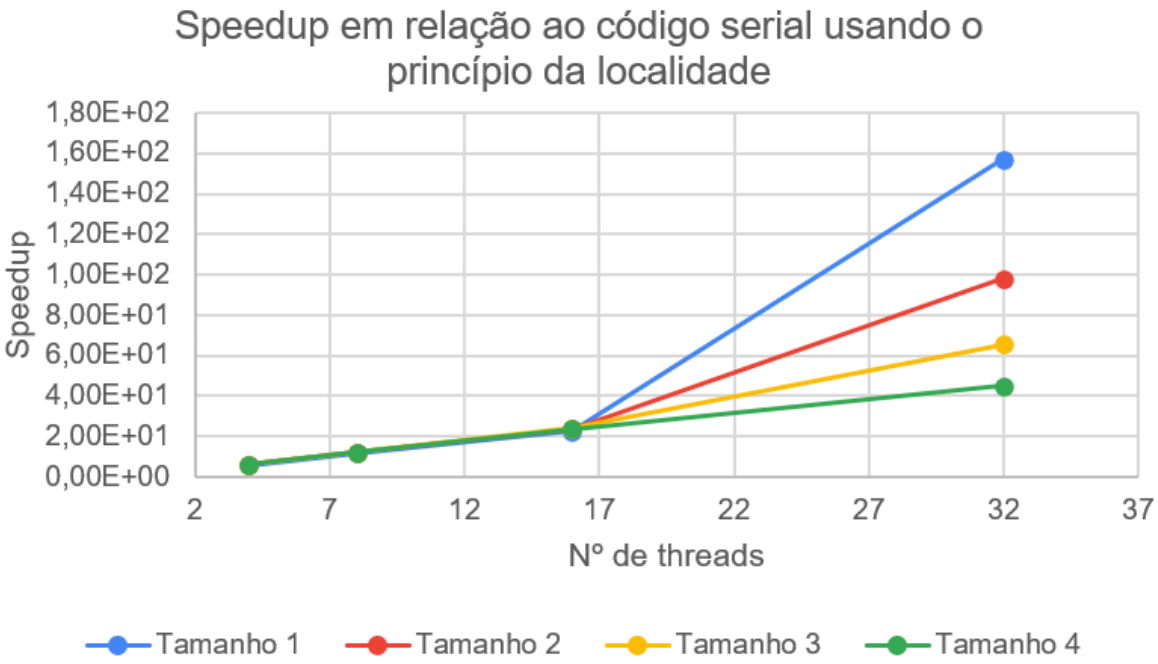
Observe agora o gráfico que compara o tempo de execução por problema para cada número de threads utilizadas.




De maneira perceptível o código paralelo consegue diminuir o seu tempo de execução para todos os tamanhos de problema quando aumentado o número de threads utilizadas. Se o gráfico for comparado com o gráfico anterior também é possível identificar uma redução dramática no tempo de execução para a relação codigo paralelo - código serial, mesmo quando observado apenas o menor número de threads utilizadas no código paralelo (4).

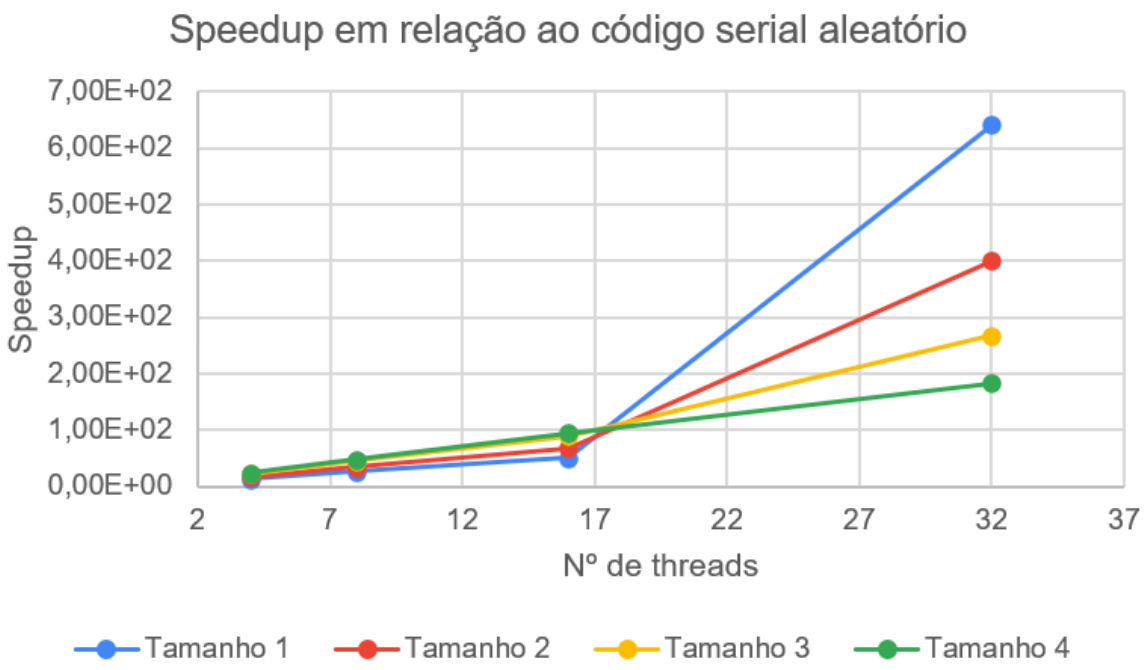
## Análise de Speedup

É possível definir o *speedup*, quando da utilização de `n` threads, como sendo o tempo médio de execução no código serial dividido pelo tempo médio de execução para `n` threads em um dado tamanho de problema. Dessa forma, o speedup representa um aumento médio de velocidade na resolução dos problemas. No gráfico abaixo é possível perceber de maneira mais clara o que acontece com o speedup do código paralelo quando aumentado o número de threads em utilização.



A forma como as linhas do gráfico assumem um comportamento diferente a partir do uso de 16 threads demonstra a relação inversamente proporcional existente entre o tamanho do problema e o speedup relativo por quantidade de threads em execução. Ou seja, quanto mais próximo do número de threads for o tamanho da fatia de matriz que cada thread irá multiplicar, mais rápida será a execução do programa.

Observe agora o speedup do código paralelo relativo ao código serial com acesso de memória aleatório:




O código paralelo consegue resultados ainda melhores em termos de performance do que os observados na comparação anterior. O motivo dessa melhora expressiva está relacionado não somente ao paralelização do algoritmo, como também ao uso da mesma estratégia de acesso a memória utilizado no código serial 01 (**Princípio da localidade espacial**), que por sí só obteve resultados superiores em performance quando comparado ao código serial 02. A combinação dessess fatores junto ao comportamento do código paralelo para tamanhos de fatia próximos ao número de threads em utilização - explicados anteriormente - traduz a grande disparidade que o gráfico apresenta.

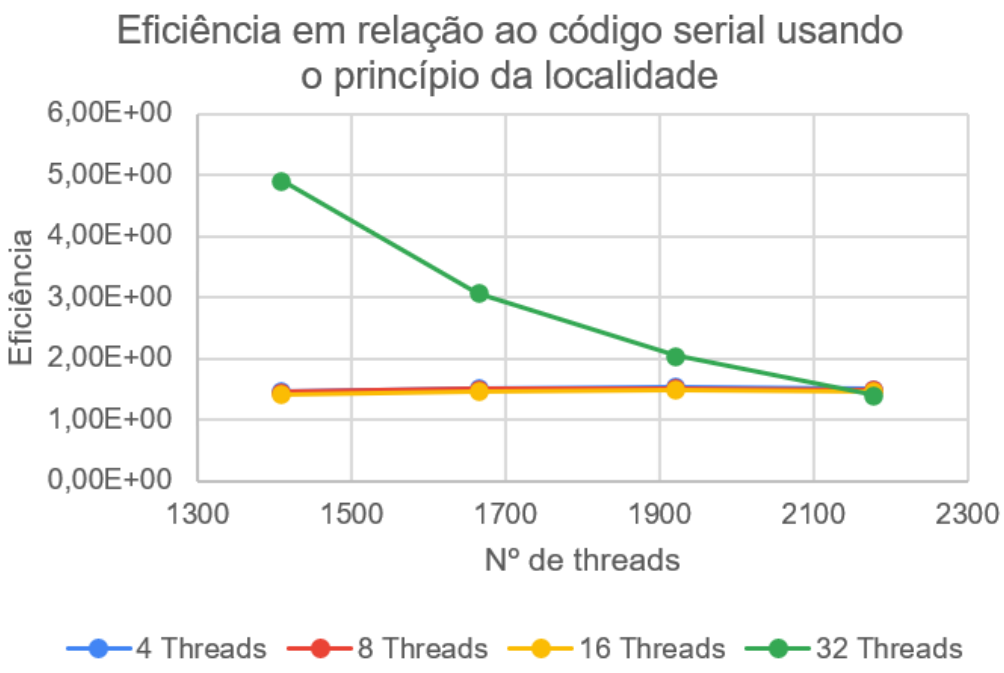
A tabela abaixo apresenta uma comparação mais detalhada do speedup relativo ao código serial utilizando o princípio da localidade espacial e ao código serial utilizando acesso de memória aleatório.

| Cores | Tamanho do Problema | Speedup (S01) | Speedup (S02) |
|---|---|---|---|
| 4 | 1408 | 5.87E+00 | 1.33E+01 |
| 4 | 1664 | 6.06E+00 | 1.76E+01 |
| 4 | 1920 | 6.16E+00 | 2.30E+01 |
| 4 | 2176 | 6.01E+00 | 2.45E+01 |
| 8 | 1408 | 1.16E+01 | 2.63E+01 |
| 8 | 1664 | 1.20E+01 | 3.48E+01 |
| 8 | 1920 | 1.21E+01 | 4.53E+01 |
| 8 | 2176 | 1.19E+01 | 4.85E+01 |
| 16 | 1408 | 2.25E+01 | 5.12E+01 |
| 16 | 1664 | 2.35E+01 | 6.83E+01 |

| Cores | Tamanho do Problema | Speedup (S01) | Speedup (S02) |
|---|---|---|---|
| 16 | 1920 | 2.39E+01 | 8.90E+01 |
| 16 | 2176 | 2.35E+01 | 9.56E+01 |
| 32 | 1408 | 1.57E+02 | 6.41E+02 |
| 32 | 1664 | 9.81E+01 | 4.00E+02 |
| 32 | 1920 | 6.55E+01 | 2.67E+02 |
| 32 | 2176 | 4.50E+01 | 1.83E+02 |

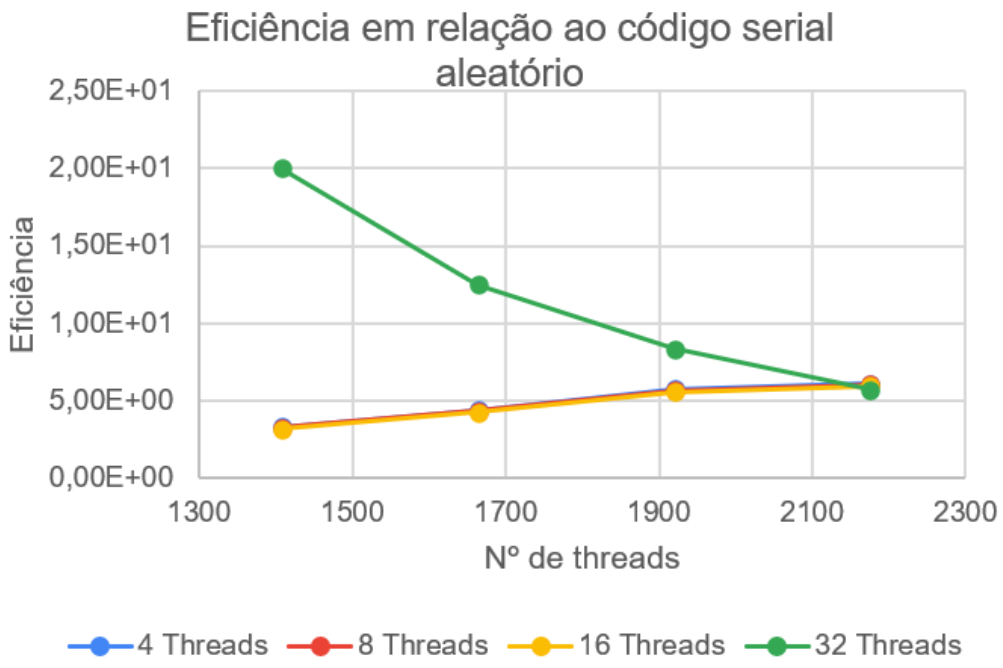
### Análise de Eficiência

Através do cálculo do speedup, é possível obter a eficiência do algoritmo quando submetido a execução com as diferentes quantidades de threads. Este cálculo pode ser realizado através da divisão do speedup do algoritmo utilizando `n` threads pelo número `n` de threads utilizados.




Observando as linhas que representam a eficiência para todas as quantidades de threads, é possível identificar uma manutenção da eficiência para um mesmo tamanho de problema conforme aumentamos somente o número de threads. Para **32 threads**, em especial, é possível identificar uma queda na eficiência conforme aumentado o tamanho do problema. Todavia, vale salientar o grande aumento da eficiência de maneira inversamente proporcional ao crescimento do problema para esta quantidade, para os 3 primeiros tamanhos de problema, a eficiência em 32 threads ficou acima da média se comparada às demais quantidades de threads. Apesar da redução da eficiência para 32 threads conforme aumentado o tamanho do problema, a linha que representa esta quantidade de threads no gráfico tende a se estabilzar em valores bem próximos aos observados para as outras quantidades de threads. Em consequência disto, é possível definir o algoritmo paralelo como **fortemente escalável** quando comparado ao algoritmo serial que reproduz a mesma estratégia de acesso a memória na multiplicação.

Observe agora o gráfico de eficiência do código paralelo em relação ao código serial que utiliza acesso de memória aleatório.




De maneira ainda mais surpreendente, na maioria dos casos, as linhas no gráfico são corrigidas positivamente em valor de eficiência à medida que aumenta o número de threads para um mesmo tamanho de problema. Este crescimento na eficiência pode ser observado até o uso de 16 threads, quando as linhas aparentam tender a um valor constante de eficiência. Assim como no gráfico anterior, a linha de eficiência para 32 threads também apresenta um comportamento peculiar. De todo modo, o encontro desta linha com as demais acontece na execução do problema de maior tamanho e tende a permanecer em um valor corrigido junto as demais. Por tanto, o código paralelo, quando baseado no algoritmo serial com utlização do acesso de memória de forma aleatória, pode ser classficiado como **fortemente escalável**.

A tabela abaixo apresenta a eficiência calculada através dos valores de speedup anteriormente fornecidos.

| Cores | Tamanho do Problema | Eficiência (S01) | Eficiência (S02) |
|---|---|---|---|
| 4 | 1408 | 1.47E+00 | 3.33E+00 |
| 4 | 1664 | 1.52E+00 | 4.40E+00 |
| 4 | 1920 | 1.54E+00 | 5.74E+00 |
| 4 | 2176 | 1.50E+00 | 6.12E+00 |
| 8 | 1408 | 1.45E+00 | 3.28E+00 |
| 8 | 1664 | 1.50E+00 | 4.35E+00 |
| 8 | 1920 | 1.52E+00 | 5.66E+00 |
| 8 | 2176 | 1.49E+00 | 6.06E+00 |
| 16 | 1408 | 1.41E+00 | 3.20E+00 |
| 16 | 1664 | 1.47E+00 | 4.27E+00 |
| 16 | 1920 | 1.49E+00 | 5.56E+00 |
| 16 | 2176 | 1.47E+00 | 5.98E+00 |
| 32 | 1408 | 4.91E+00 | 2.00E+01 |
| 32 | 1664 | 3.06E+00 | 1.25E+01 |

| Cores | Tamanho do Problema | Eficiência (S01) | Eficiência (S02) |
|---|---|---|---|
| 32 | 1920 | 2.05E+00 | 8.35E+00 |
| 32 | 2176 | 1.41E+00 | 5.73E+00 |

**Obs.:** Os valores superlineares encontrados para eficiência do algoritmo paralelo podem ser explicados pela utilização de algoritmos seriais não ótimos.

# Conclusão

## Considerações Finais

Por meio dos resultados coletados após a execução de todos os códigos, foi possível obter resultados bastante satisfatórios quanto a comparação do algoritmo paralelo em relação aos dois algoritmos seriais. O código paralelo se mostrou ser mais do que eficiente quando comparado em performance geral junto aos dois seriais, o que já era esperado. No entanto, foram encontrados valores de eficiência superlineares, mesmo na comparação com o código serial que implementa a mesma estrátegia de acesso a memória. Isto denuncia a não otimização dos códigos seriais utilizados para esta análise. Apesar disso, foi interessante perceber a diferença no desempenho dos algoritmos seriais utilizando os diferentes tipos de acesso a memória, a exploração do **Princípio da localidade espacial** foi ponto chave na conquista de performance para o código serial 01. Além disso, o comportamento singular da linha de eficiência para 32 threads obsevado nos gráficos foi importante para destacar a relação inversamente proporcional que existe entre o tamanho do problema e o speedup para este caso específico. Por fim, diante do comportamento da eficiência do código paralelo quando aumentado o número de threads para um mesmo tamanho de problema, foi possível classificar o algoritmo paralelo como **fortemente escalável** em ambos os casos de comparação.

## Softwares utilizados

```
~$: g++ --version
g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
~$: python3 --version
Python 3.8.5
```

```
~$: grip --version
Grip 4.5.2
```