Universidade Federal do Rio de Janeiro
Aluno: Gabriel Rodrigues da Silva
Disciplina: Computação de Alto Desempenho - 2024.1

Trabalho 5 - OpenMP

Neste trabalho, será utilizado novamente o programa jacobi.c , que implementa a solução de um sistema de equações lineares pelo método iterativo de Jacobi. Relembrando, esse método resolve sistemas de equações lineares da forma $Ax = b$, ao decompor a matriz A em uma matriz diagonal D e uma matriz R com os elementos fora da diagonal. A solução é atualizada iterativamente, recalculando cada variável x_i usando apenas os valores da iteração anterior e os elementos de D e R. Por fim, o processo continua até que a diferença entre iterações sucessivas seja suficientemente pequena, garantindo assim a convergência para a solução correta.

Também será utilizado o OpenMP (Open Multi-Processing), uma API (Application Programming Interface) criada para o desenvolvimento de programas paralelos em máquinas de memória compartilhada.

Fonte dos programas:
jacobi.c : https://github.com/UoB-HPC/intro-hpc-jacobi/blob/master/jacobi.c
OpenMP: https://www.openmp.org/

Exercício 1: **identifique os trechos que possam ser paralelizados com o OpenMP.**

Podemos paralelizar os seguintes trechos:

Trecho 1

```
// Perfom Jacobi iteration
      #pragma omp parallel for private(row, col, dot) shared(A, x, b,
xtmp)
      for (row = 0; row < N; row++)
      {
      dot = 0.0;
      for (col = 0; col < N; col++)
      {
      if (row != col)
      dot += A[row + col*N] * x[col];
      }
      xtmp[row] = (b[row] - dot) / A[row + row*N];
      }
```

Modificação após a linha 55

- **#pragma omp parallel for** inicia uma região paralela que distribui as iterações do loop entre diferentes threads;
- **private(row, col, dot)** declara que as variáveis row, col, e dot devem ser privadas para cada thread, evitando condições de corrida;
- **shared(A, x, b, xtmp)** declara que os arrays A, x, b, e xtmp são compartilhados entre todas as threads.

Trecho 2

```
      // Check for convergence
      sqdiff = 0.0;
      #pragma omp parallel for reduction(+:sqdiff) private(diff)
      for (row = 0; row < N; row++)
      {
      diff   = xtmp[row] - x[row];
      sqdiff += diff * diff;
      }
```

Modificação após a linha 74

- **#pragma omp parallel for reduction(+:sqdiff) private(diff)** : o cálculo do erro pode ser paralelizado, pois cada elemento do vetor x é independente dos outros.

Trecho 3

```
// Initialize data
srand(SEED);

#pragma omp parallel for shared(A, b, x)
for (int row = 0; row < N; row++)
{
     double rowsum = 0.0;
     for (int col = 0; col < N; col++)
     {
     double value = rand()/(double)RAND_MAX;
     A[row + col*N] = value;
     rowsum += value;
     }
     A[row + row*N] += rowsum;
     b[row] = rand()/(double)RAND_MAX;
     x[row] = 0.0;
}
```

Modificação após a linha 107

- **#pragma omp parallel for shared(A, b, x)** : a inicialização da matriz A e vetor b pode ser paralelizada, pois cada elemento é independente dos outros.

Link do programa modificado:
https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho5/jacobi_paralelizado.c

Exercício 2: **compile e execute o programa paralelo otimizado para várias threads.**

Vamos compilar o programa utilizando o compilador da Intel[1], com otimização automática e adicionando o parâmetro -qopenmp para obter suporte da API OpenMP.

- **/opt/intel/oneapi/2024.2/bin/icx -qopenmp -O3 -o jacobi_paralelizado jacobi_paralelizado.c -lm**

```
┌──(gbrods㉿gbrods)-[~/Documents/cad/trabalho5]
└─$ /opt/intel/oneapi/2024.2/bin/icx -qopenmp -O3 -o jacobi_paralelizado jacobi_paralelizado.c -lm
```

Na execução, podemos definir a variável de ambiente OMP_NUM_THREADS para controlar o número de threads que o programa irá utilizar. Por exemplo, se quisermos utilizar 8 threads, o resultado seria o seguinte:

- **export PATH=/opt/intel/oneapi/2024.2/bin:$PATH **
  **&& export LD_LIBRARY_PATH=/opt/intel/oneapi/2024.2/lib:$LD_LIBRARY_PATH **
  && OMP_NUM_THREADS=8 ./jacobi_paralelizado -n 4000

```
┌──(gbrods㉿gbrods)-[~/Documents/cad/trabalho5]
└─$ export PATH=/opt/intel/oneapi/2024.2/bin:$PATH && export LD_LIBRARY_PATH=/opt/intel/oneapi/2024.2/lib:$LD_LIBRARY_PATH && OMP_NUM_THREADS=8 ./jacobi_paralelizado -n 4000
```

```
------------------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
------------------------------------------------
Solution error = 0.199939
Iterations     = 10146
Total runtime  = 117.787158 seconds
Solver runtime = 115.865264 seconds
------------------------------------------------
```

OBS: o comando de execução está um pouco longo porque inclui a configuração necessária para localizar e carregar a biblioteca libiomp5.so . Foi necessário ajustar no terminal as variáveis de ambiente PATH e LD_LIBRARY_PATH para garantir que o sistema encontrasse a biblioteca.

[1] O compilador icc foi descontinuado. Veja:
https://community.intel.com/t5/oneAPI-Registration-Download/How-to-use-icc-if-i-only-installed-the-latest-Intel-HPC-toolkit/m-p/1606180?profile.language=pt

Então, irei utilizar o comando icx para compilar o programa, que está disponível em:
https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit-download.html

Exercício 3: **faça os gráficos de escalabilidade forte (tamanho do problema fixo, aumentando o número de threads) e escalabilidade fraca (aumentando o tamanho do problema e o número de threads)**

Para a escalabilidade forte, foi criado um shell script que executa o programa compilado anteriormente com tamanho fixo de uma matriz 4000x4000 com diferentes números de threads (1, 2, 4, 6, 8, 10 e 12). Veja:

```
#!/bin/bash

# Define o diretório onde a biblioteca icx está localizada
LIB_PATH="/opt/intel/oneapi/2024.2/lib/"

# Atualiza LD_LIBRARY_PATH
export LD_LIBRARY_PATH="$LIB_PATH:$LD_LIBRARY_PATH"

# Verifica se a variável de ambiente está configurada corretamente
echo "LD_LIBRARY_PATH: $LD_LIBRARY_PATH"

# Executa o programa com diferentes números de threads
strong_scaling() {
    local fixed_size=$1
    shift
    local threads_list=("$@")
    local results=()

    for threads in "${threads_list[@]}"; do
        echo "Executando com $threads threads"
        # Executa o programa com a variável de ambiente OMP_NUM_THREADS
        export OMP_NUM_THREADS=$threads
        output=$(./jacobi_paralelizado -n "$fixed_size" 2>&1)
        echo "$output"

        # Extrai o tempo total de execução
        total_runtime=$(echo "$output" | grep "Total runtime" | awk '{print $(NF-1)}')
        if [ -n "$total_runtime" ]; then
            results+=("$threads $total_runtime")
        fi
    done

    # Imprime e salva os resultados em um arquivo txt
    for result in "${results[@]}"; do
        echo "$result" >> resultados_escalabilidade_forte.txt
    done
}

# Parâmetros de entrada
FIXED_SIZE=4000
THREADS_LIST=(1 2 4 6 8 10 12)

# Executa a função
strong_scaling "$FIXED_SIZE" "${THREADS_LIST[@]}"
```

Fonte do programa:

https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho5/escalabilidade_forte.sh

Output do programa:

```
┌──(gbrods㉿gbrods)-[~/Documents/cad/trabalho5]
└─$ ./escalabilidade_forte.sh
LD_LIBRARY_PATH: /opt/intel/oneapi/2024.2/lib/:
Executando com 1 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.199909
Iterations     = 10040
Total runtime  = 326.773599 seconds
Solver runtime = 326.523165 seconds
---------------------------------------
Executando com 2 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.199935
Iterations     = 10112
Total runtime  = 183.186029 seconds
Solver runtime = 179.722992 seconds
---------------------------------------
Executando com 4 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.199931
Iterations     = 10156
Total runtime  = 113.850415 seconds
Solver runtime = 112.536194 seconds
---------------------------------------
Executando com 6 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.199870
Iterations     = 10058
Total runtime  = 95.411055 seconds
Solver runtime = 93.379245 seconds
---------------------------------------
Executando com 8 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.200019
Iterations     = 10071
Total runtime  = 118.582870 seconds
Solver runtime = 116.330855 seconds
---------------------------------------
Executando com 10 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.200061
Iterations     = 10123
Total runtime  = 130.205241 seconds
Solver runtime = 127.828544 seconds
---------------------------------------
Executando com 12 threads
---------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------------
Solution error = 0.199872
Iterations     = 10128
Total runtime  = 123.262320 seconds
Solver runtime = 120.729334 seconds
---------------------------------------
```

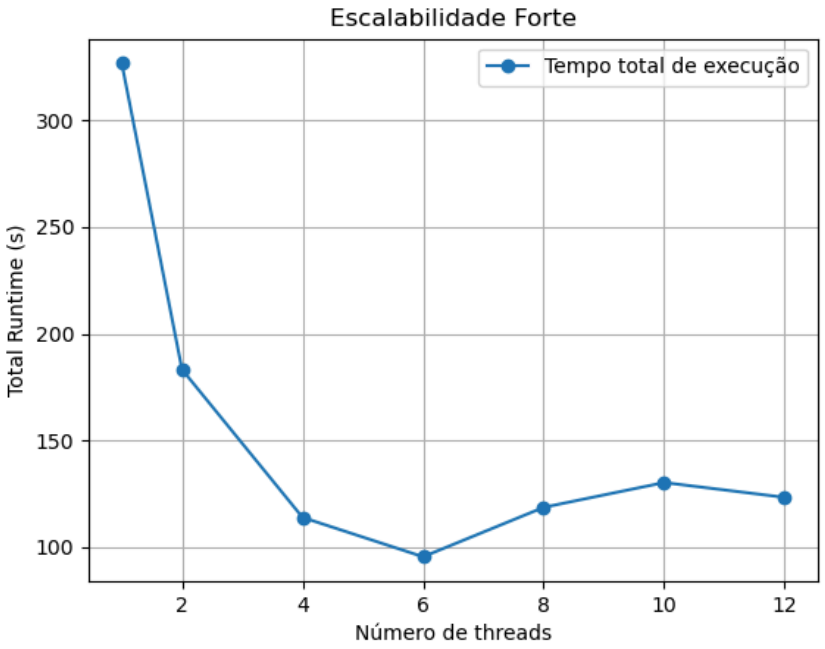
Para plotar o gráfico, foi utilizado o seguinte script python:

```python
import matplotlib.pyplot as plt

def read_results(filename):
    threads = []
    total_runtime = []

    with open(filename, 'r') as f:
        for line in f:
            thread, runtime = line.split()
            threads.append(int(thread))
            total_runtime.append(float(runtime))

    print(threads)
    print(total_runtime)
    return threads, total_runtime

def plot_results(threads, total_runtime, title, filename):
    plt.figure()
    plt.plot(threads, total_runtime, label='Tempo total de execução', marker='o')
    plt.xlabel('Número de threads')
    plt.ylabel('Total Runtime (s)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.xscale('linear')
    plt.yscale('linear')
    plt.savefig(filename)
    plt.show()

# Lê os resultados do arquivo
threads, total_runtime = read_results('resultados_escalabilidade_forte.txt')

# Chama a função de plotagem
plot_results(threads, total_runtime, 'Escalabilidade Forte', 'strong_scaling.png')
```

Fonte do programa:

https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho5/plot_escalabilidade_forte.py

O gráfico resultante é:



Foi possível observar que, à medida que o número de threads aumenta, o tempo de execução diminui. Isso ocorre porque mais threads podem processar diferentes partes do cálculo simultaneamente, reduzindo o tempo total necessário para a convergência.

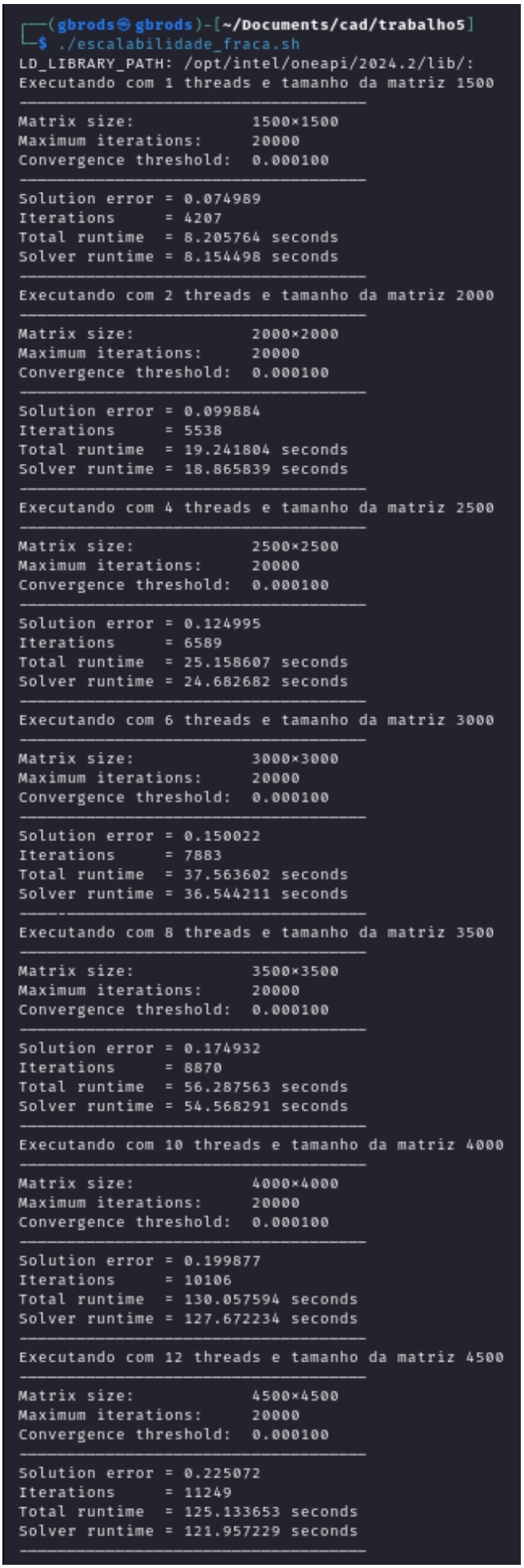
Já para a escalabilidade fraca, foi criado outro shell script que executa o programa compilado anteriormente com diferentes números de threads (1, 2, 4, 6, 8, 10 e 12), entretanto, o tamanho da matriz também aumenta (1500, 2000, 2500, 3000, 3500, 4000 e 4500). Veja:

```
#!/bin/bash

# Define o diretório onde a biblioteca icx está localizada
LIB_PATH="/opt/intel/oneapi/2024.2/lib/"

# Atualiza LD_LIBRARY_PATH
export LD_LIBRARY_PATH="$LIB_PATH:$LD_LIBRARY_PATH"

# Verifica se a variável de ambiente está configurada corretamente
echo "LD_LIBRARY_PATH: $LD_LIBRARY_PATH"

# Executa o programa com diferentes tamanhos da matriz e número de threads
weak_scaling() {
    local -n size_list=$1
    local -n threads_list=$2
    local results=()

    for i in "${!threads_list[@]}"; do
        local threads=${threads_list[i]}
        local size=${size_list[i]}
        echo "Executando com $threads threads e tamanho da matriz $size"

        # Executa o programa com a variável de ambiente OMP_NUM_THREADS
        export OMP_NUM_THREADS=$threads
        output=$(./jacobi_paralelizado -n "$size" 2>&1)
        echo "$output"

        # Extrai o tempo total de execução
        total_runtime=$(echo "$output" | grep "Total runtime" | awk '{print
$(NF-1)}')
        if [ -n "$total_runtime" ]; then
            results+=("$threads $size $total_runtime")
        fi
    done

    # Imprime e salva os resultados em um arquivo txt
    for result in "${results[@]}"; do
        echo "$result" >> resultados_escalabilidade_fraca.txt
    done
}

# Parâmetros de entrada
THREADS_LIST=(1 2 4 6 8 10 12)
SIZE_LIST=(1500 2000 2500 3000 3500 4000 4500)

# Executa a função
weak_scaling SIZE_LIST THREADS_LIST
```

Fonte do programa:

https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho5/escalabilidade_fraca.sh

Output do programa:



```
┌──(gbrods㉿gbrods)-[~/Documents/cad/trabalho5]
└─$ ./escalabilidade_fraca.sh
LD_LIBRARY_PATH: /opt/intel/oneapi/2024.2/lib/:
Executando com 1 threads e tamanho da matriz 1500
----------------------------------------
Matrix size:            1500×1500
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.074989
Iterations     = 4207
Total runtime  = 8.205764 seconds
Solver runtime = 8.154498 seconds
----------------------------------------
Executando com 2 threads e tamanho da matriz 2000
----------------------------------------
Matrix size:            2000×2000
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.099884
Iterations     = 5538
Total runtime  = 19.241804 seconds
Solver runtime = 18.865839 seconds
----------------------------------------
Executando com 4 threads e tamanho da matriz 2500
----------------------------------------
Matrix size:            2500×2500
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.124995
Iterations     = 6589
Total runtime  = 25.158607 seconds
Solver runtime = 24.682682 seconds
----------------------------------------
Executando com 6 threads e tamanho da matriz 3000
----------------------------------------
Matrix size:            3000×3000
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.150022
Iterations     = 7883
Total runtime  = 37.563602 seconds
Solver runtime = 36.544211 seconds
----------------------------------------
Executando com 8 threads e tamanho da matriz 3500
----------------------------------------
Matrix size:            3500×3500
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.174932
Iterations     = 8870
Total runtime  = 56.287563 seconds
Solver runtime = 54.568291 seconds
----------------------------------------
Executando com 10 threads e tamanho da matriz 4000
----------------------------------------
Matrix size:            4000×4000
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.199877
Iterations     = 10106
Total runtime  = 130.057594 seconds
Solver runtime = 127.672234 seconds
----------------------------------------
Executando com 12 threads e tamanho da matriz 4500
----------------------------------------
Matrix size:            4500×4500
Maximum iterations:     20000
Convergence threshold:  0.000100
----------------------------------------
Solution error = 0.225072
Iterations     = 11249
Total runtime  = 125.133653 seconds
Solver runtime = 121.957229 seconds
----------------------------------------
```

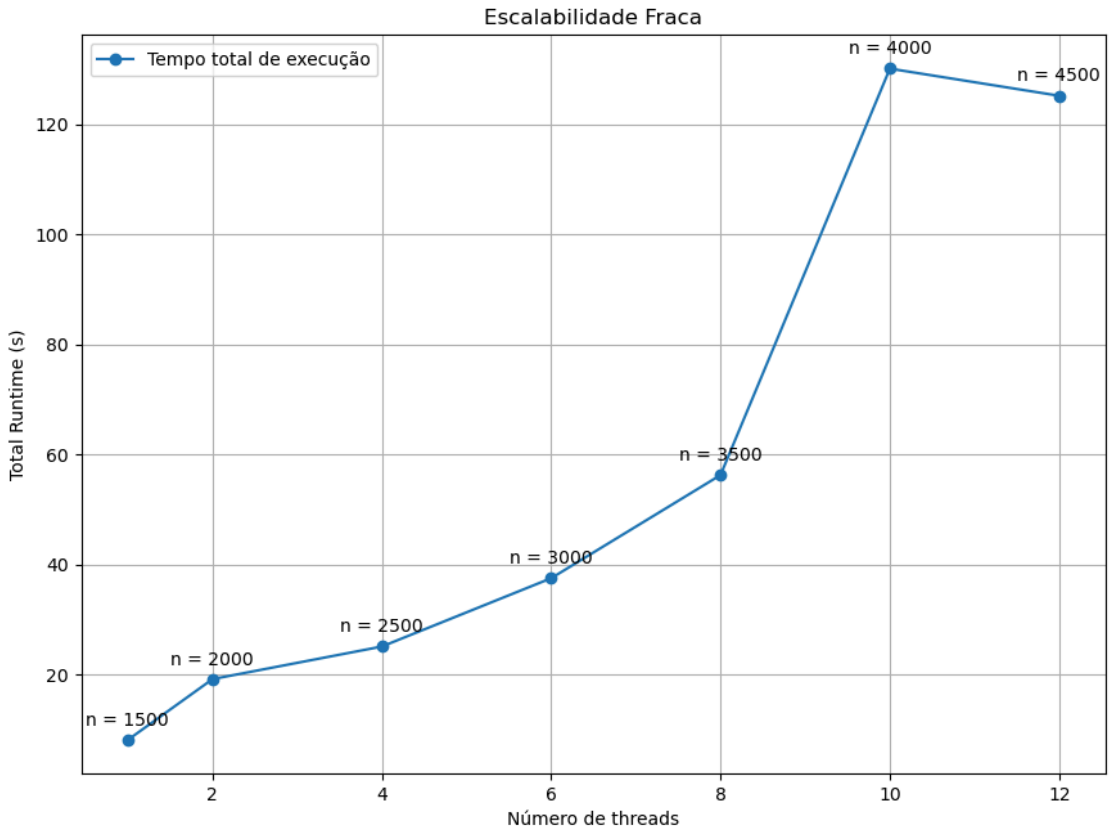
Para plotar o gráfico, foi utilizado o seguinte script python:

```python
import matplotlib.pyplot as plt

def read_results(filename):
    threads = []
    sizes = []
    total_runtime = []

    with open(filename, 'r') as f:
        for line in f:
            thread, size, runtime = line.split()
            threads.append(int(thread))
            sizes.append(int(size))
            total_runtime.append(float(runtime))

    print("Threads:", threads)
    print("Tamanhos da matriz:", sizes)
    print("Total Runtime:", total_runtime)
    return threads, sizes, total_runtime

def plot_results(threads, sizes, total_runtime, title, filename):
    plt.figure()
    plt.plot(threads, total_runtime, label='Tempo total de execução',
marker='o')
    for i, size in enumerate(sizes):
        plt.annotate(f'n = {size}', (threads[i], total_runtime[i]),
textcoords="offset points", xytext=(0, 8), ha='center')
    plt.xlabel('Número de threads')
    plt.ylabel('Total Runtime (s)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.xscale('linear')
    plt.yscale('linear')
    plt.savefig(filename)
    plt.show()

# Lê os resultados do arquivo
threads, sizes, total_runtime =
read_results('resultados_escalabilidade_fraca.txt')

# Chama a função de plotagem
plot_results(threads, sizes, total_runtime, 'Escalabilidade Fraca',
'weak_scaling.png')
```

Fonte do programa:

https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho5/plot_escalabilidade_fraca.py

O gráfico resultante é:



Em uma escalabilidade fraca ideal, o tempo de execução de um programa deve permanecer aproximadamente constante à medida que o tamanho do problema aumenta proporcionalmente ao número de threads. Entretanto, a partir dos dados obtidos para o meu processador, observa-se uma tendência de aumento não linear no tempo de execução com o aumento do tamanho da matriz e do número de threads.

Esse comportamento pode ser atribuído a alguns fatores, como:

- O aumento no número de threads pode estar resultando em maior sobrecarga de comunicação e sincronização, que não é compensada pelo aumento no número de threads;
- O algoritmo de Jacobi paralelizado pode não estar escalando de maneira ideal com o número crescente de threads;
- Caso haja alguma limitação de energia e gerenciamento térmico do meu processador (que é um notebook), isso pode afetar o desempenho ao aumentar o número de threads e o tamanho da matriz.

Portanto, é possível concluir que a escalabilidade fraca observada não atingiu o comportamento ideal esperado.