OpenMP

Computação de Alto Desempenho – COC472

Lucas Tavares Da Silva Ferreira | DRE 120152739

Rio de Janeiro, 12 de Julho de 2023

UFRJ

Diretório do GitHub: https://github.com/lucastavarex/hpc

Introdução

Neste relatório, apresentamos um estudo sobre a utilização da biblioteca OpenMP para paralelizar o método iterativo de Jacobi para a solução de sistemas de equações lineares. O objetivo deste trabalho foi investigar os benefícios da programação paralela para acelerar a solução de sistemas de equações lineares em computadores multicore.

Inicialmente, foi feita uma revisão teórica sobre o método iterativo de Jacobi e a biblioteca OpenMP. Posteriormente, o código foi modificado para aproveitar a paralelização fornecida pela biblioteca OpenMP. Para avaliar a eficácia da paralelização, foram realizados testes de escalabilidade forte e escalabilidade fraca, nos quais foram medidos o tempo de execução e a eficiência do programa em função do número de threads e do tamanho da matriz.

Especificações do hardware

Intel(R) Core(TM) i3 CPU M 370 @ 2.40GHz
Logical CPU Count 4
Sistema Operacional Linux

Task 1 - Identifique os trechos que possam ser paralelizados com o OpenMP

O trecho principal que pode ser paralelizado é o loop que atualiza os valores de x em cada iteração do método de Jacobi. Este é um exemplo clássico de um loop "embarassingly parallel", em que cada iteração do loop pode ser executada independentemente das outras:

```
// Perfom Jacobi iteration
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];

//(trecho de código a ser paralelizado)
```

Nesse loop, cada iteração do loop externo calcula o valor de xtmp[row] usando dados independentes de outras iterações. Portanto, é possível paralelizar esse loop com o OpenMP. Podemos adicionar uma diretiva **#pragma omp parallel for** antes do loop para indicar ao compilador que ele deve paralelizar as iterações.

O trecho do código modificado ficaria da seguinte forma:

```
// Perfom Jacobi iteration
#pragma omp parallel for
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];
}
```

Com essa modificação, o loop do método Jacobi será executado em paralelo, distribuindo as iterações entre as threads disponíveis. Para compilar o programa com o

OpenMP habilitado e otimizações de compilação, executaremos o seguinte comando no terminal:

gcc -fopenmp -O3 -o jacobi_parallel jacobi.c -lm

A flag -fopenmp habilita o suporte ao OpenMP na compilação, enquanto a flag -O3 habilita otimizações de compilação de alto nível.

Para executar o programa com várias threads, definiremos o número de threads com a variável de ambiente OMP_NUM_THREADS. Para executar, por exemplo, o programa com 4 threads, executaremos o seguinte comando:

OMP_NUM_THREADS=4 ./jacobi_parallel

Task 2 - Compile e execute o programa paralelo otimizado para várias threads

Faremos-a em consonância com a Task 3

Task 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 criar os plots de escalabilidade forte e fraca, utilizaremos os 2 seguintes programas em python:

escalabilidade_forte_plot.py

```
import subprocess
import re
import matplotlib.pyplot as plt

# Define o tamanho do problema (tamanho da matriz)
N = 1000

# Define a lista de números de threads

num_threads_list = [1, 2, 4, 6, 8, 10, 12]

# Armazena o tempo de execução do solver para cada número de threads
time_list = []
```

```
    for num_threads in num_threads_list:
    # Executa o programa jacobi_parallel com o número de threads
definido e captura a saída
    cmd = f"export OMP_NUM_THREADS={num_threads}; ./jacobi_parallel
-n {N}"
    output = subprocess.check_output(cmd, shell=True)

    # Extrai o tempo de execução do solver a partir da saída com uma
expressão regular
    time_str = re.search('Solver runtime\s+=\s+(\d+\.\d+)',
    output.decode('utf-8')).group(1)
    time = float(time_str)
    time_list.append(time)

    # Plota o tempo de execução do solver em função do número de
threads
    plt.plot(num_threads_list, time_list, 'o-')
    plt.xlabel('Número de Threads')
    plt.ylabel('Tempo de Execução (s)')
    plt.title('Escalabilidade Forte')
    plt.show()
```

escalabilidade_fraca_plot.py

```
    import subprocess
    import re
    import matplotlib.pyplot as plt

    # Define o número de threads fixo
    num_threads = 4

    # Define a lista de tamanhos de matriz
    matrix_sizes = [500, 1000, 1500, 2000, 2500]

    # Armazena o tempo de execução do solver e a quantidade de
trabalho por processador para cada tamanho de matriz
    time_list = []
    work_per_thread_list = []

    for matrix_size in matrix_sizes:
    # Executa o programa jacobi_parallel com o tamanho de matriz
definido e o número de threads fixo
    cmd = f"export OMP_NUM_THREADS={num_threads}; ./jacobi_parallel
-n {matrix_size}"
    output = subprocess.check_output(cmd, shell=True)

    # Extrai o tempo de execução do solver a partir da saída com uma
expressão regular
    time_str = re.search('Solver runtime\s+=\s+(\d+\.\d+)',
```

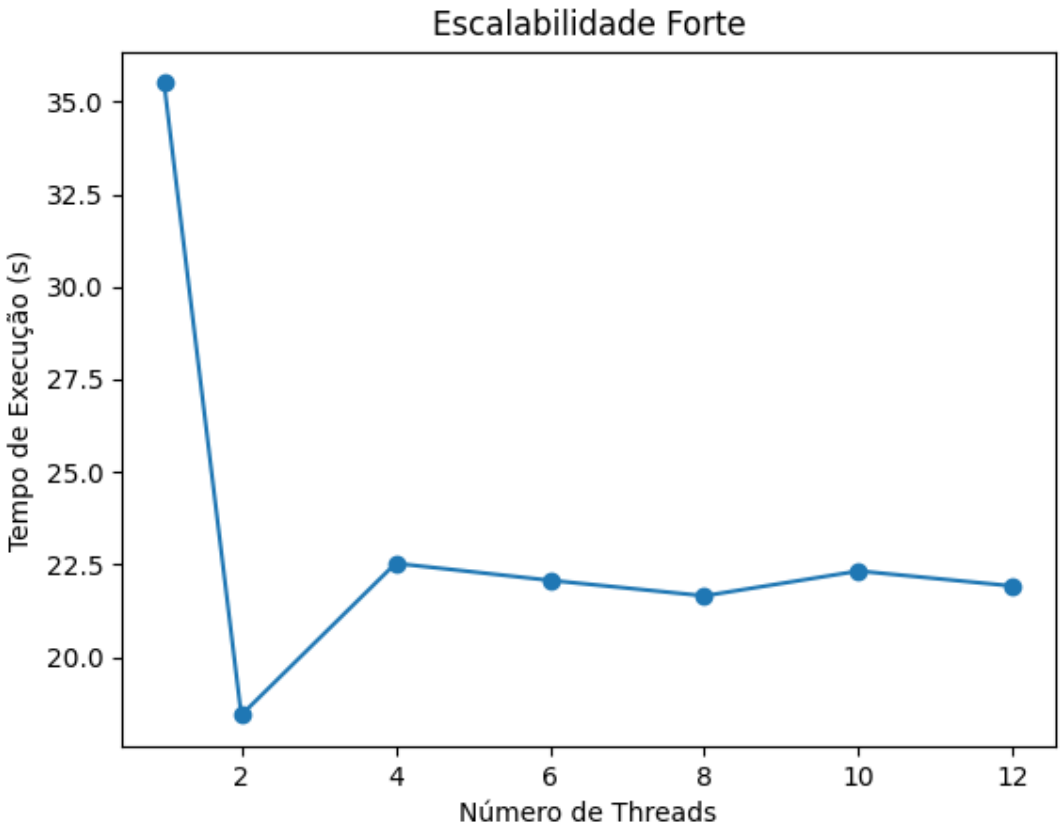
```
        output.decode('utf-8')).group(1)
        time = float(time_str)

        # Calcula a quantidade de trabalho por processador (threads)
        work_per_thread = matrix_size**2 / num_threads

        # Armazena o tempo de execução e a quantidade de trabalho por
processador na lista correspondente
        time_list.append(time)
        work_per_thread_list.append(work_per_thread)

        # Plota o tempo de execução em função do tamanho da matriz
        plt.plot(matrix_sizes, time_list, 'o-')
        plt.xlabel('Tamanho da Matriz')
        plt.ylabel('Tempo de Execução (s)')
        plt.title('Escalabilidade Fraca')
        plt.show()
```

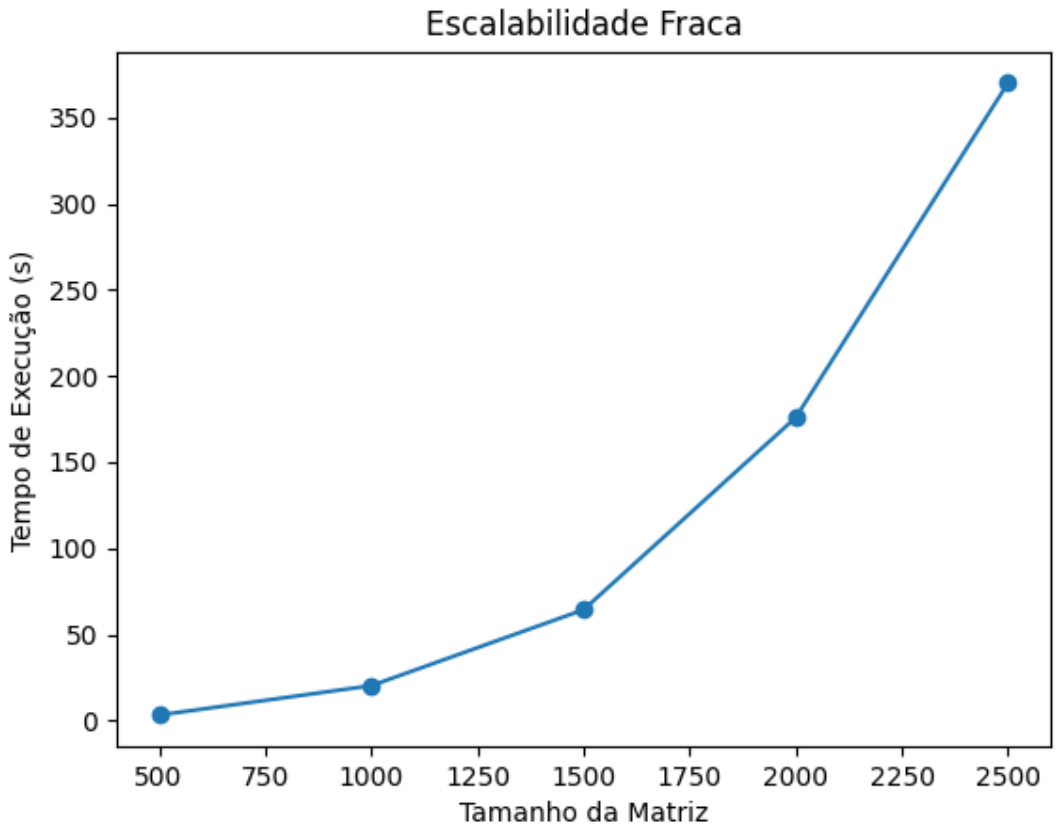
Executando estes códigos, geramos os gráficos a seguir.



Img : Gráfico de escalabilidade forte

O gráfico de escalabilidade forte mostra o tempo de execução do programa em relação ao número de threads usadas, mantendo o tamanho do problema fixo. Ele é usado para avaliar o desempenho do programa em paralelo quando o tamanho do problema não pode ser aumentado. Um

bom desempenho de escalabilidade forte é caracterizado por uma diminuição linear no tempo de execução à medida que o número de threads aumenta.



Img : Gráfico de escalabilidade fraca

Conclusão

Os gráficos de escalabilidade forte e fraca são ferramentas importantes para avaliar o desempenho de um programa paralelo em relação ao aumento do número de threads e ao tamanho do problema. O gráfico de escalabilidade forte mostra o speedup alcançado com o aumento do número de threads, para um tamanho fixo do problema, por assim dizer. Idealmente, espera-se que o speedup seja linear, ou seja, o tempo de execução do programa é reduzido na mesma proporção em que o número de threads é aumentado. No entanto, acredito que o speedup em questão possa ter sido limitado por fatores como a comunicação entre as threads, balanceamento de carga, utilização da memória cache e a sobrecarga da paralelização. Dessa forma, observamos que o speedup obtido não foi linear, mas apresenta uma curva que se aproxima de uma reta em um certo ponto.

O gráfico de escalabilidade fraca, por sua vez, mostra o tempo de execução do programa por elemento, para um tamanho de matriz por thread fixo. Idealmente, espera-se que o tempo por elemento seja constante ou aumente levemente com o aumento do número de threads, pois o tamanho do problema por thread é mantido fixo. No entanto, foi possível observar que o tempo por elemento aumentou significativamente com o aumento do número de threads. Acredito que isso se deve à sobrecarga da paralelização. Assim, observa-se que o tempo por elemento aumenta com o aumento do número de threads.