UNIVERSIDADE FEDERAL DE SÃO CARLOS
DEPARTAMENTO DE COMPUTAÇÃO

Programação Paralela e Distribuída

Exercício Programa - EP3

Professor: Hermes Senger

Aluno: Ivan Duarte Calvo, 790739

São Carlos, SP, 2023

1 Metodologia

Inicialmente, foi realizada a paralelização, com a utilização da biblioteca *openMP*, do código previamente fornecido (*laplace_seq_iteracoes.c*), compondo assim o código *laplace_openmp.c*.

Além da importação da biblioteca, foi necessário apenas a demarcação no código das áreas a serem paralelizadas. Tal demarcação foi feita da seguinte forma:

```
#pragma omp parallel for collapse(2)
for(int i = 1; i < size-1; i++) {
    for(int j = 1; j < size-1; j++) {
        new_grid[i][j] = 0.25 * (grid[i][j+1] + grid[i][j-1] +
                                 grid[i-1][j] + grid[i+1][j]);
    }
}

#pragma omp parallel for collapse(2)
for(int i = 1; i < size-1; i++) {
    for(int j = 1; j < size-1; j++) {
        grid[i][j] = new_grid[i][j];
    }
}
```

O código acima realiza duas etapas de iteração. Na primeira, a matriz da iteração atual é construída e na segunda etapa a matriz é atualizada com os valores da nova matriz. Ambas as etapas são paralelizadas individualmente. É utilizado também o complemento *collapse(2)* para que ambos os laços *for* sejam paralelizados.

As matrizes geradas pelo código paralelizado foram comparadas com a matriz gerada pelo código sequencial para garantir a corretude do cálculo, de maneira que o arquivo gerado por todos os códigos deve ser idêntico.

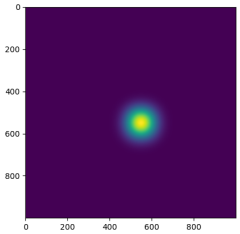

Figura 1: programa sequencial.

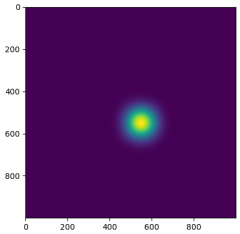

Figura 2: programa paralelizado.

As imagens acima são representações visuais das matrizes geradas pelos algoritmos. Tais imagens foram criadas através de um programa em *Python* previamente fornecido.

Para assegurar o resultado, os arquivos de saída com as matrizes podem ser comparados através do comando *diff* no terminal do linux.

2 Hardware

Após a adaptação do código para a programação paralela, o código foi submetido para a execução no Cluster da UFSCar, sendo executado com 1, 2, 4, 8, 16, 24, 32, 36 e 40 *threads*. Por fim, os respectivos tempos de execução são utilizados para a composição dos gráficos de *Speedup* e Eficiência, e para as comparações necessárias.

O arquivo *.sh* do *job* submetido ao o ao Cluster contém inicialmente o comando *lscpu* que retorna informações sobre a CPU utilizada no problema. Neste exercício, a CPU utilizada foi:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-95
Thread(s) per core:    2
Core(s) per socket:    24
Socket(s):             2
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            23
Model:                 49
Model name:            AMD EPYC 7402 24-Core Processor
Stepping:              0
CPU MHz:               2794.661
BogoMIPS:              5589.32
Virtualization:        AMD-V
L1d cache:             32K
L1i cache:             32K
L2 cache:              512K
L3 cache:              16384K
NUMA node0 CPU(s):     0-5,48-53
```

```
NUMA node1 CPU(s):   6-11,54-59
NUMA node2 CPU(s):   12-17,60-65
NUMA node3 CPU(s):   18-23,66-71
NUMA node4 CPU(s):   24-29,72-77
NUMA node5 CPU(s):   30-35,78-83
NUMA node6 CPU(s):   36-41,84-89
NUMA node7 CPU(s):   42-47,90-95
```

3 Resultados

O algoritmo foi submetido para execução no Cluster com instruções para ser calculado com 1, 2, 4, 8, 16, 24, 32, 36 e 40 *threads*. Os resultados obtidos foram os seguintes:

Tabela 1: Cluster da UFSCar

nº de *threads*	Tempo de Execução	*Speedup*	Eficiência
1	85.773590	1	1
2	42.648678	2.011166	1.005583
4	21.036237	4.077421	1.019355
8	11.731745	7.311239	0.913905
16	10.155965	8.445637	0.527852
24	6.918987	12.396842	0.516535
32	6.675573	12.848873	0.401527
36	5.191443	16.522110	0.458947
40	4.275872	20.059906	0.501498

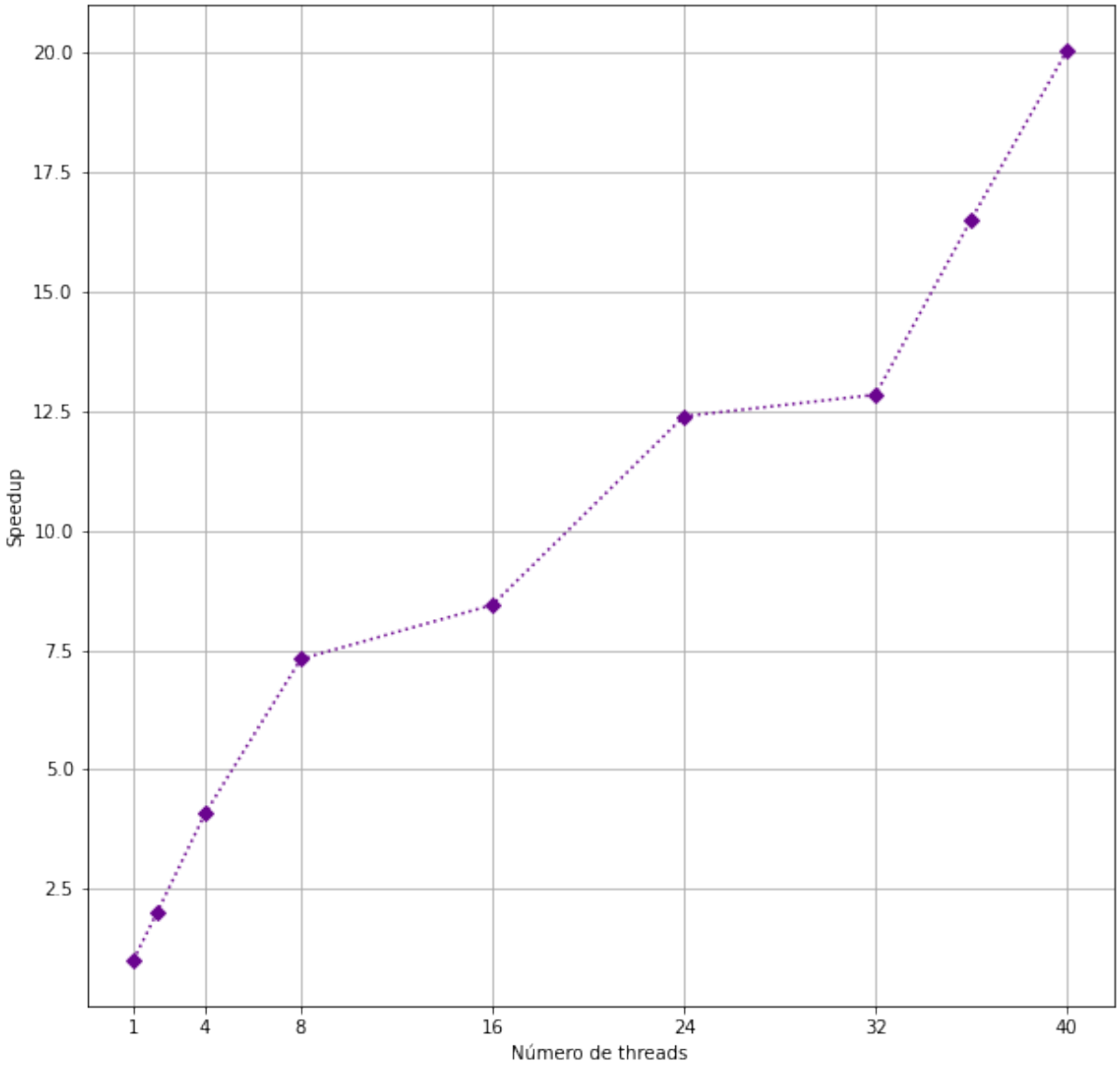

Figura 3: Gráfico de *Speedup* no Cluster.

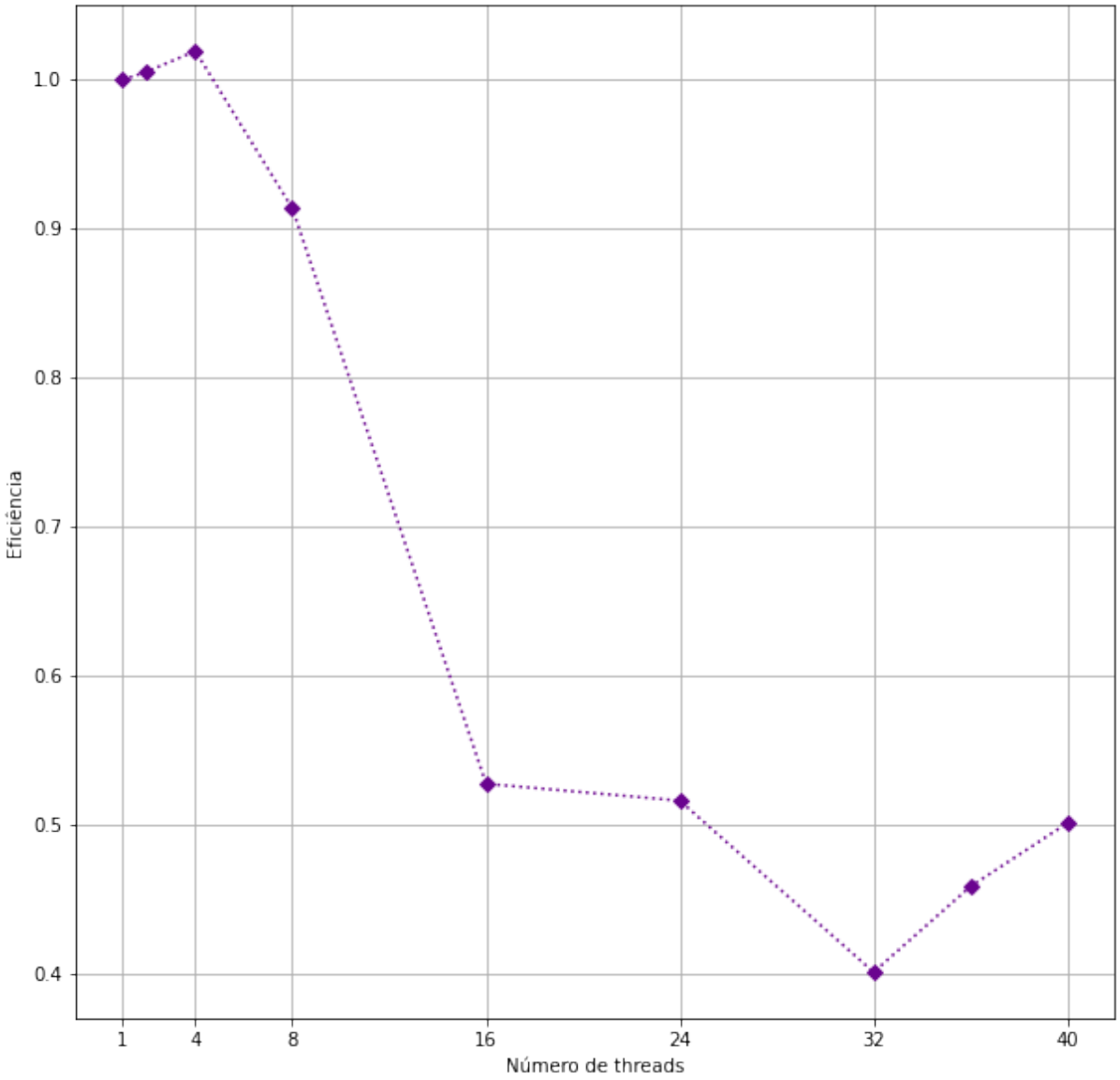

Figura 4: Gráfico de Eficiência no Cluster.

4 Conclusão

Apesar de não alcançar o maior *speedup* possível teoricamente, ainda assim o experimento obteve um resultado que pode ser considerado satisfatório, retornando, com a utilização de 40 *threads*, um *speedup* pouco maior que 20.

É interessante observar também os valores atingidos em eficiência. Com a utilização de 1 a 8 *threads* a eficiência fica superior à 90%, no entanto, a partir de 16 *threads* a eficiência cai para cerca de 50%. A queda dos valores da eficiência já é um comportamento esperado na implementação de programação paralela, pois com o aumento do número de *threads* a

serem utilizadas há também um pequeno aumento no gasto de tempo com processamentos alheios aos cálculos principais do algoritmo, como a comunicação e sincronização entre *threads*, por exemplo. Contudo, como ainda se obtém um valor de eficiência na casa dos 50% ao se utilizar todas as 40 *threads* disponibilizadas pelo Cluster é possível considerar que foi obtido um resultado satisfatório na implementação.

Atenuando a conclusão positiva, é de grande importância também pontuar que a implementação da paralelização através da biblioteca *openMP* é significativamente mais simples quando comparada à implementação através de *pthreads*, que para se atingir resultados semelhantes demandaria uma quantidade consideravelmente maior de tempo investido na construção do programa. Desta maneira, é possível concluir que a implementação da paralelização através do *openMP* pode ser muito vantajosa.