Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
COMPUTAÇÃO PARALELA 2024/2

RESOLVENDO O PROBLEMA DE VIZINHOS COMUNS EM UM GRAFO COM OPENMP

Gustavo Henriques da Cunha

São João del-Rei
2024

Lista de Figuras

1 Gráfico de comparação entre as soluções em um grafo esparso. 4
2 Gráfico de comparação entre o crescimento das soluções em grafo esparso. . 4
3 Gráfico de comparação entre as soluções em um grafo denso. 5
4 Gráfico de comparação entre as soluções em um grafo denso. 6
5 Gráfico de comparação entre o crescimento das soluções em grafo denso. . . 6
6 Tabela de comparação entre o aumento de threads em um grafo esparso. . 7
7 Tabela de comparação entre o aumento de threads em um grafo denso. . . 7

Sumário

1 INTRODUÇÃO **1**

1.1 Objetivo 1

1.2 Motivação 1

1.3 Descrição do Problema 1

2 Solução Sequencial **1**

3 Solução OpenMP **2**

4 Testes **3**

5 Conclusão **7**

REFERÊNCIAS **8**

1 INTRODUÇÃO

Este é um trabalho prático da disciplina de Computação Paralela no curso de Ciência da Computação na UFSJ, tendo como docente o professor Rafael Sachetto Oliveira.

1.1 Objetivo

Este trabalho tem como objetivo resolver o problema de encontrar vizinhos comuns em um grafo utilizando a especificação Open Multi-Processing (OpenMP), mais especificamente a implementação para a linguagem C.

1.2 Motivação

A importância do trabalho é praticar os conceitos introduzidos em sala de aula, buscando uma melhor compreensão do funcionamento dos princípios de programação paralela em memória compartilhada.

1.3 Descrição do Problema

Dado um grafo não-direcionado $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, o problema consiste em determinar, para cada par de vértices u e v, o número de vizinhos comuns, ou seja, os vértices que são adjacentes tanto a u quanto a v. Em termos matemáticos, isso significa encontrar a intersecção entre os conjuntos de vizinhos de u e v, denotado como $N(u) \cap N(v)$.

Assim, deverá ser implementado duas soluções ao problema, uma versão sequencial e uma versão paralela com OpenMP, ambas utilizando a linguagem C. Posteriormente, deve ser feito a comparação de desempenho entre as duas.

2 Solução Sequencial

Para a construção da solução sequencial do problema, foi implementado toda a lógica para ler o arquivo com as arestas, as armazenando em uma estrutura de grafo no formato de lista de vértices, que armazena todos os vértices do grafo, e cada vértice possui uma lista com os vértices que possui ligação.

Seguindo esta etapa, para a solução do problema, foi utilizada da estrutura escolhida para percorrer o grafo vértice por vértice, onde para cada vértice seguinte é formado um par, que é então feito o calculo de quantos vizinhos em comum o par possui. Para fazer este calculo, foi utilizado de um vetor de booleanos, onde foi percorrida a lista de arestas do primeiro par e depois passando pela lista do segundo, conferindo se o valor da lista de booleanos é verdadeiro para cada uma de suas arestas, conferindo assim um vizinho em comum.

Como para formarmos todos os pares utilizamos de dois '*for*' encadeados, de forma que o segundo começa com um valor maior que o primeiro, temos um total de $\frac{N \times (N-1)}{2}$ pares, onde N é o número de vértices.

3 Solução OpenMP

A solução paralela utilizando OpenMP foi construida em cima da solução sequencial, fazendo ajustes para a lógica paralela, permitindo análise mais justa entre a performace das soluções. Desta forma, toda lógica para leitura e armazenamento do grafo, incluindo a estrutura de lista de arestas permanece.

A paralelização utilizando o OpenMP é extremamente simples, sendo necessário apenas utilizar a diretiva "pragma omp for schedule(dynamic)"dentro de uma seção paralela, no momento de percorrer os vértices do grafo, fazendo o cálculo de vizinhos para cada par. Assim, a carga de trabalho é dividida entre as threads, e como o for é desbalanciado, a utilização do "schedule(dynamic)"permite a divisão da carga de maneira dinâmica na execução, de maneira que a partir do momento que uma thread termina seu trabalho ela é alocada para o próximo.

No mais, a única diferença da versão paralela para a sequencial é a escrita no arquivo de saída. Para garantir que a escrita seja feita de maneira ordenada e por apenas uma thread, ela é feita fora da seção paralela. Assim, para armazenar os resultados calculados nas threads foi sacrificado memória para criar uma matrix com tamanho igual ao número de vértices do grafo, onde o índice (i, j) da matrix representa o número de vizinhos entre o vértice i e o vértice j, de forma que no momento de escrever os resultados, a matrix é percorrida.

4 Testes

Para testar a performance de cada solução, foi criado um *script* para a automação dos testes, onde foram criados arquivos no formato .edgelist com dados aleatórios e foram variados o número de vértices do grafo além da densidade do grafo. A medição do tempo foi feita utilizando a função "gettimeofday"para ambas as soluções.

A Tabela 1 mostra os resultados obtidos a partir de testes realizados com diferentes tamanhos no número de vértices do grafo, comparando o tempo em segundos para os programas solucionarem o problema, considerando a solução OpenMP com 4 threads.

Vértices	**Sequencial**	**OpenMP**
5000	0.210632	0.121655
7500	0.498322	0.256473
10000	0.919435	0.459933
12500	1.465178	0.704122
15000	2.179459	1.122000
17500	3.284183	1.659459
20000	5.127116	2.170805

Tabela 1: Comparação de desempenho entre as soluções em grafo esparso.

Podemos observar com clareza o aumento mais significativo do tempo de execução conforme a entrada aumenta na solução sequencial.

Para mais fácil observação desta tendência, foi criado os gráficos nas figuras 2 e 1 com a comparação entre o crescimento do tempo conforme a entrada entre a solução sequencial e a solução OpenMP com 2, 3 e 4 threads para um grafo esparso, com aproximadamento o número de vétices igual ao número de arestas.

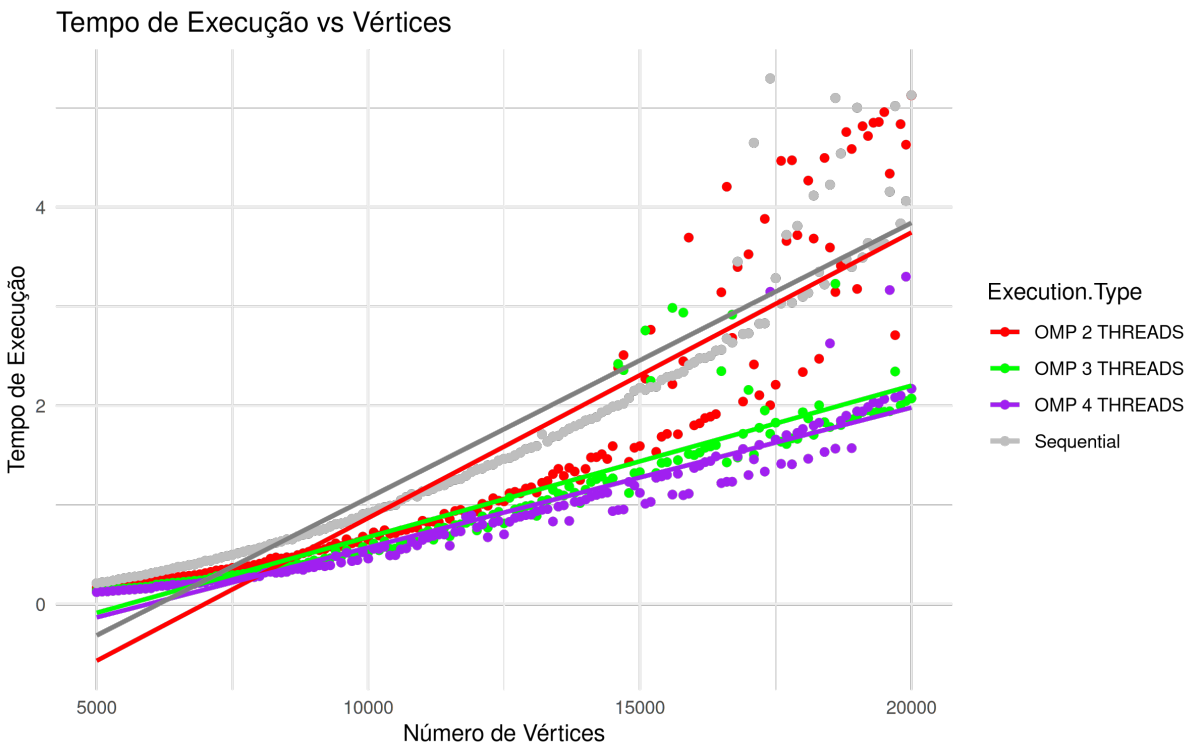

Figura 1: Gráfico de comparação entre as soluções em um grafo esparso.

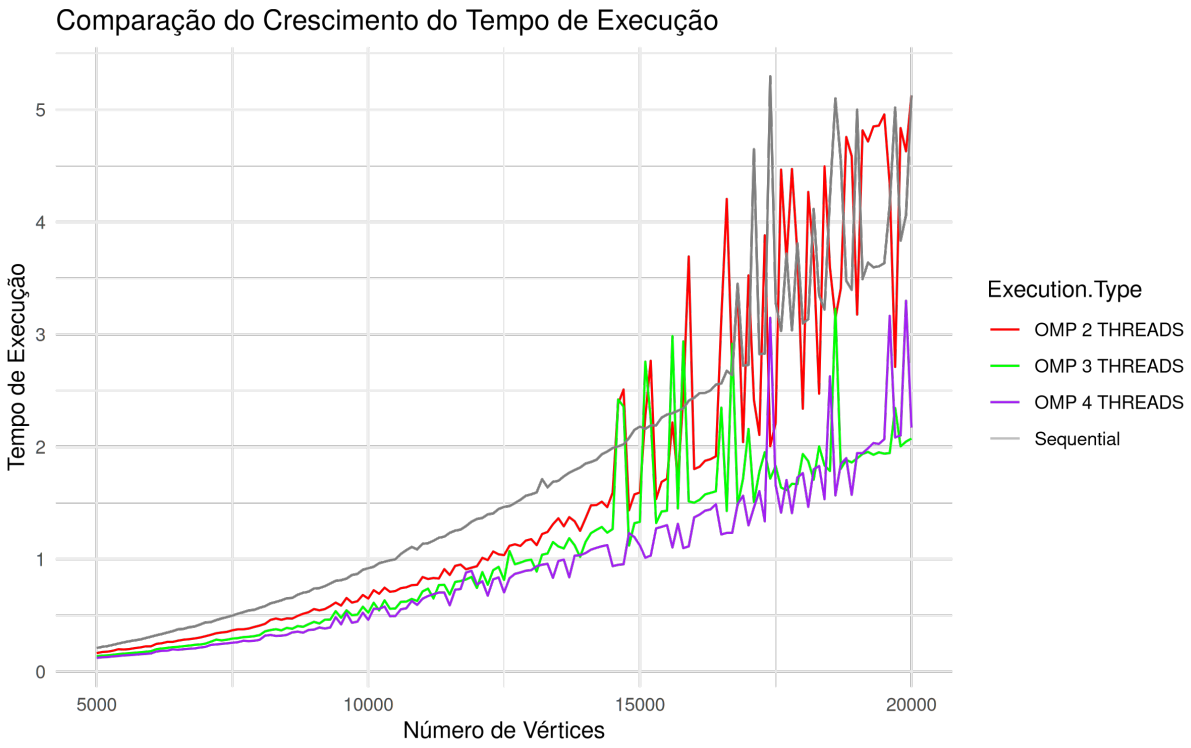

Figura 2: Gráfico de comparação entre o crescimento das soluções em grafo esparso.

A 2 mostra alguns picos de tempo em todas as curvas, que podem ter decorrido de

momentos específicos no uso da CPU. Já a 1 mostra como para entradas muito pequenas a solução sequencial supera a solução com threads, o que é esperado já que com pouco trabalho a ser feito o overhead da criação de threads se torna um gasto significativo. Mas, conforme a entrada aumenta, é claro o crescimento maior do tempo de execução do programa sequencial.

Os mesmos gráficos foram feitos para testes utilizando grafos mais densos, que são mostrados nas figuras 3, 4 e 5, onde, de maneira até mais marcante, é possível confirmar a disparidade da solução paralela em relação a solução sequencial, já que o número de calculos realizados nos grafos mais densos são maiores, o que faz com que a divisão de trabalho seja mais desejada, de forma que as threads sempre irão ter trabalho a fazer, o que beneficia a execução em paralelo. Isso já é possível ser observado com apenas 2 threads, com a tendencia de melhora de performace conforme esse número aumenta.

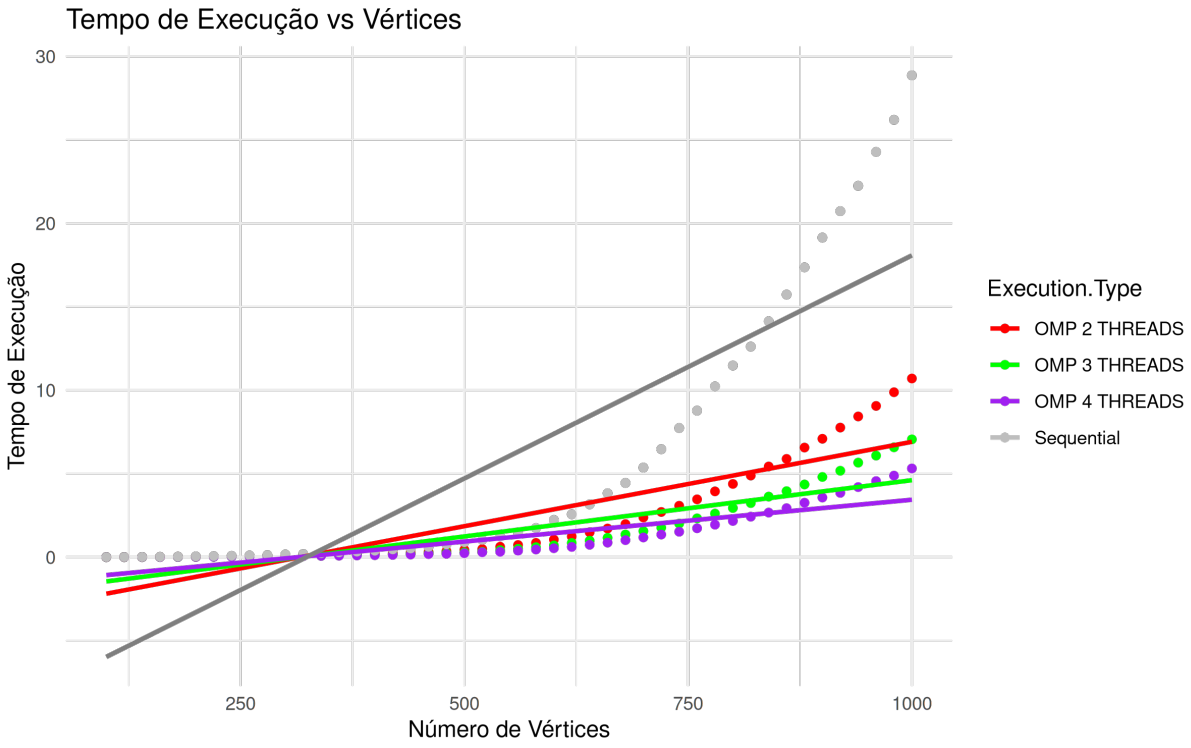

Figura 3: Gráfico de comparação entre as soluções em um grafo denso.

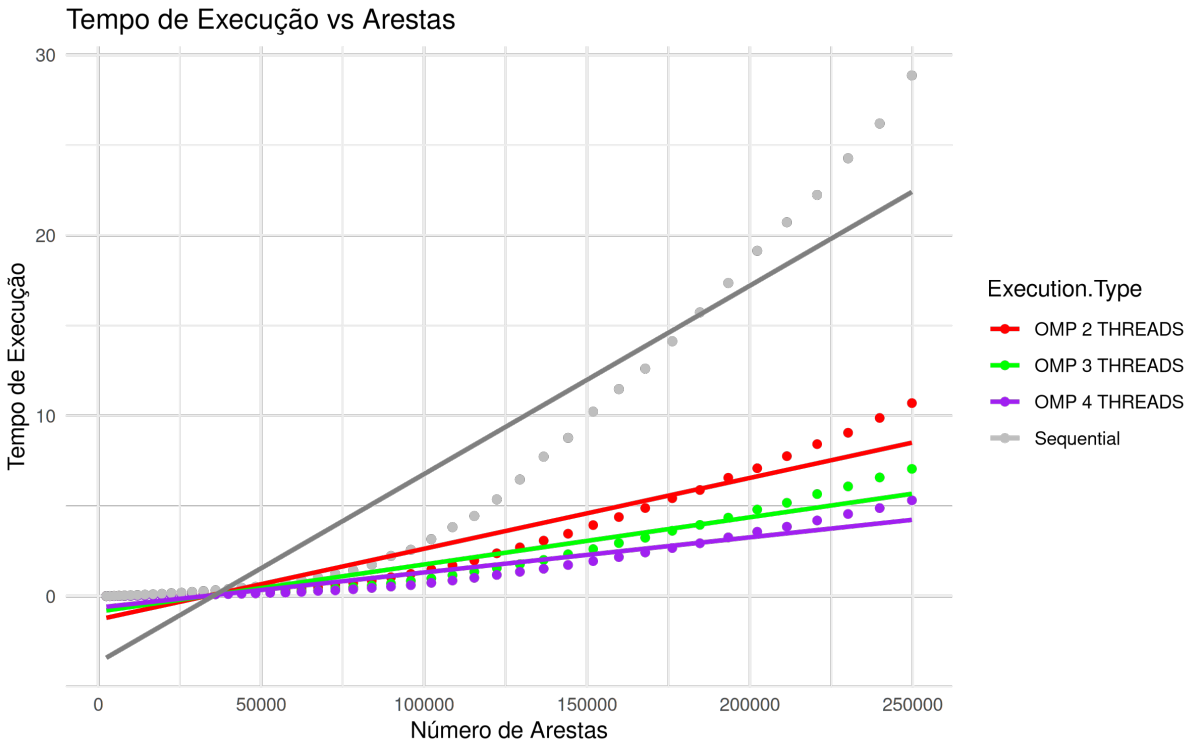

Figura 4: Gráfico de comparação entre as soluções em um grafo denso.

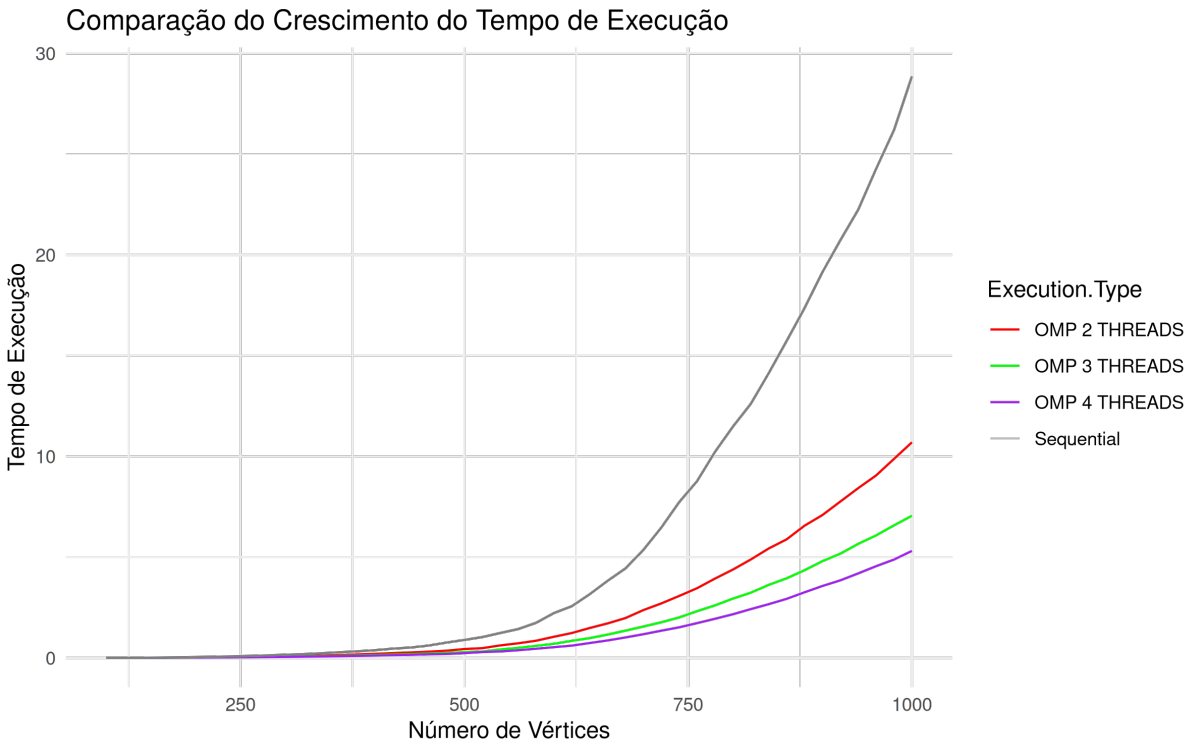

Figura 5: Gráfico de comparação entre o crescimento das soluções em grafo denso.

Outro aspecto importante possível de se observar principalmente na figura 5 é como

o aumento no número de threads resultou em uma melhora na performace. Para ser possível observar qual é esse aumento de forma numérica em relação a solução sequencial, foi montado as tabelas nas figuras 6 e 7, que mostra o speed up de alguns testes, onde é possível confirmar o grande ganho da solução paralela com 4 threads, principalmente quando é necessário mais trabalho de computação, com grafos mais densos e maiores, onde, apesar de variação, chegou a ser cerca de 5 vezes mais rápido que a solução sequencial, um pouco melhor que com 3 threads, que ficou entre 3 a 4 vezes melhor.

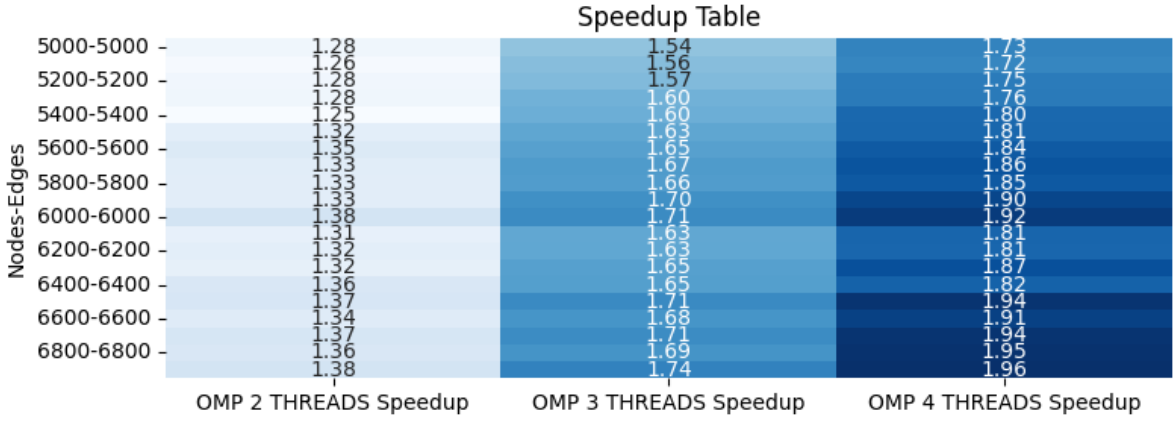

Figura 6: Tabela de comparação entre o aumento de threads em um grafo esparso.

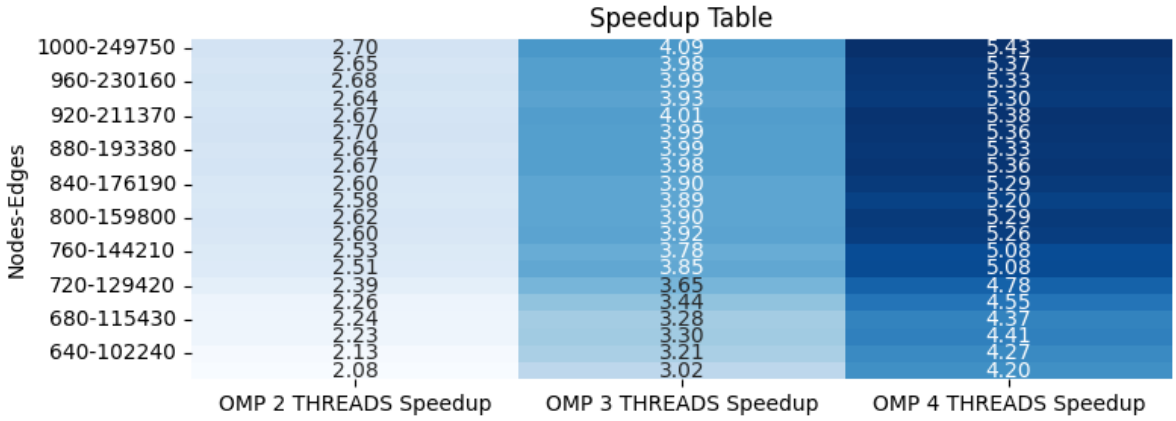

Figura 7: Tabela de comparação entre o aumento de threads em um grafo denso.

5 Conclusão

Ao observar os resultados, foi possível ter noção de como a programação paralela pode economizar muito tempo na solução de tarefas custosas, e como o OpenMP permite esse

ganho de maneira simples, aumentando poucas linhas no código sequencial.

Deve se entender, também, que a computação paralela é uma ferramenta poderosa mas também complexa, onde qualquer mudança e implementação deve ser meticulosamente testada e medida, pois é a melhor maneira de garantir que o código tem o comportamento e ganho experado.

Referências

[Chandra et al., 2001] Chandra, R., Menon, L., Dagum, D., Kohr, R., Maydan, D., and McDonald, J. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA.

[Chapman et al., 2007] Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge, MA.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall, 2nd edition.