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 e MPI

Gustavo Henriques da Cunha

São João del-Rei
2025

Lista de Figuras

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

Sumário

1 INTRODUÇÃO **1**

1.1 Objetivo 1

1.2 Motivação 1

1.3 Descrição do Problema 1

2 Solução Sequencial **1**

3 Solução Paralela **2**

4 Testes **3**

5 Conclusão **6**

REFERÊNCIAS **6**

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) junto com a especificação Message-Passing Interface (MPI), 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 Paralela

A solução paralela utilizando OpenMP e MPI 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.

As mudanças para a solução sequencial começam depois que o grafo está devidamento armazenado. Assim, a partir do número de vértices e do número de processos executando, é feita uma distribuição de carga, que diz o número de pares que cada processo deve calcular o número de vizinhos comuns. Assim, a fórmula para o número de pares é usada e seu valor dividido para o número de processos. Depois disso, é feita a atribuição do intervalo de pares que o processo irá trabalhar através de seu *rank* no comunicador global. Também é feita uma distribuição justa dos pares restantes, caso existam, entre os processos.

Com a distribuição de carga feita, cada processo irá passar pela combinação de pares até chegarem em um par dentro de seu intervalo, onde deverão calcular o vizinho comum desse par. O resultado será armazendo em uma estrutura denominda '*pair*', que guarda os vértices u e v, além do número de vizinhos comuns entre eles. A utilizalçao do OpenMP é feita neste ponto, 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, cada processo irá criar suas

próprias threads para a solução do problema.

Finalizando o calculo dentro de seu intervalo, o processo irá então enviar para o processo de *rank* 0 o seu resultado, ou no caso do processo de *rank* 0, receber o resultado dos outros. Esta comunicação foi feita usando as funções *MPI_Send* e *MPI_Recv*, que são as funções mais simples para comunicação entre processos, e que funcionam bem mesmo ao enviar dados em estruturas. Existem outras maneiras de fazer a comunicação, como utilizando o *MPI_Gather*, mas as funções mais simples foram preferidas pelo fato do programa ter pouca troca de mensagens.

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 execução dos programas foi feita em um único computador com quatro processadores. Desta forma, as execuções do programa paralelo foram todas feitas com dois processos e duas threads, para povoar todos os processadores sem extender sua capacidade.

Para mais fácil observação dos resultados, foi criado os gráficos nas figuras 1 e **??** com a comparação entre o crescimento do tempo conforme a entrada entre a solução sequencial e a solução paralela para um grafo esparso, com aproximadamente o número de vértices igual ao número de arestas.

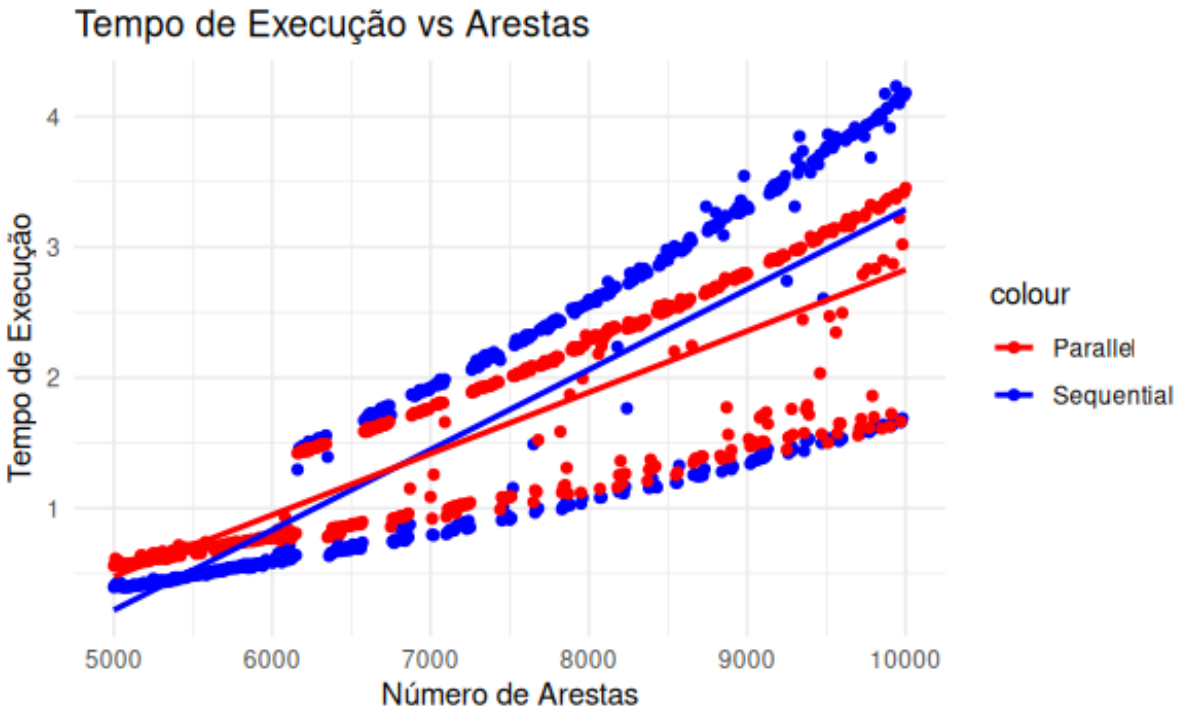

Figura 1: Gráfico de comparação entre as soluções em um grafo esparso.

A 1 mostra alguns picos de tempo em todas as curvas, que podem ter decorrido de momentos específicos no uso da CPU. Ela também mostra como para entradas muito pequenas a solução sequencial supera a solução paralela, o que é esperado já que com pouco trabalho a ser feito o overhead da criação de threads e a comunicação entre os processos 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 2 e 3, onde, de maneira até mais marcante, é possível confirmar a disparidade da solução paralela em relação a solução sequencial, já que o número de cálculos realizados nos grafos mais densos são maiores, o que faz com que a divisão de trabalho seja mais equilibrada, de forma que as threads sempre irão ter trabalho a fazer, o que beneficia a execução em paralelo.

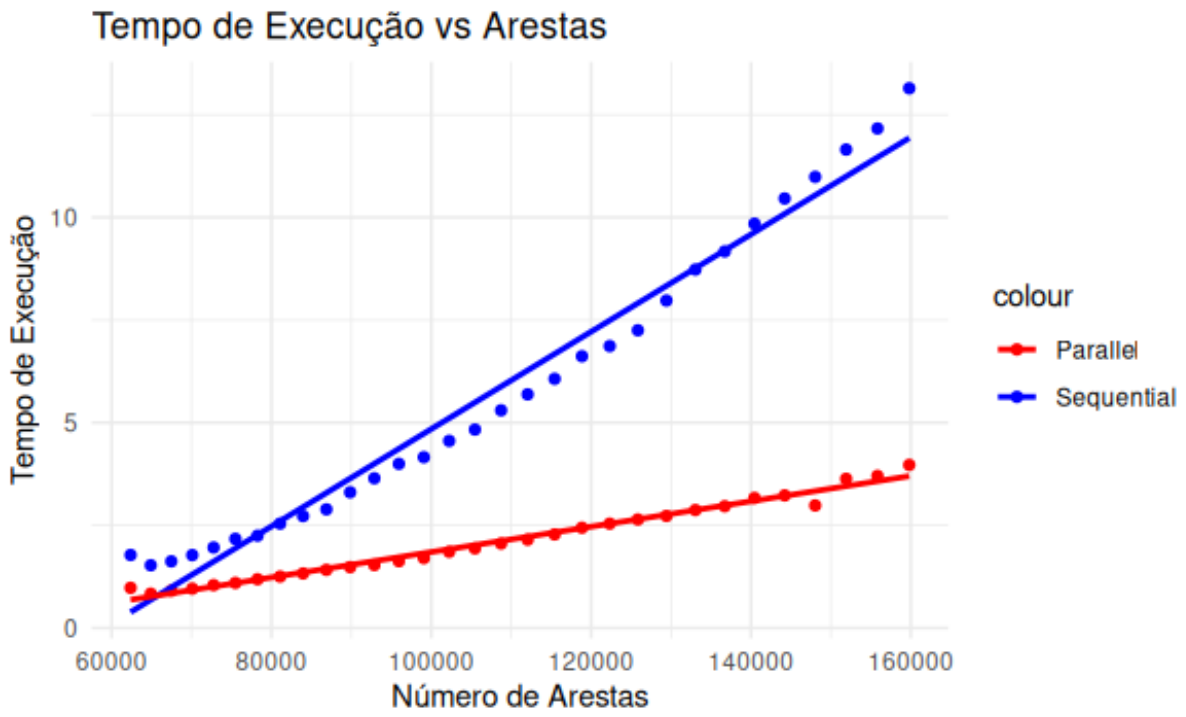

Figura 2: Gráfico de comparação entre as soluções em um grafo denso.

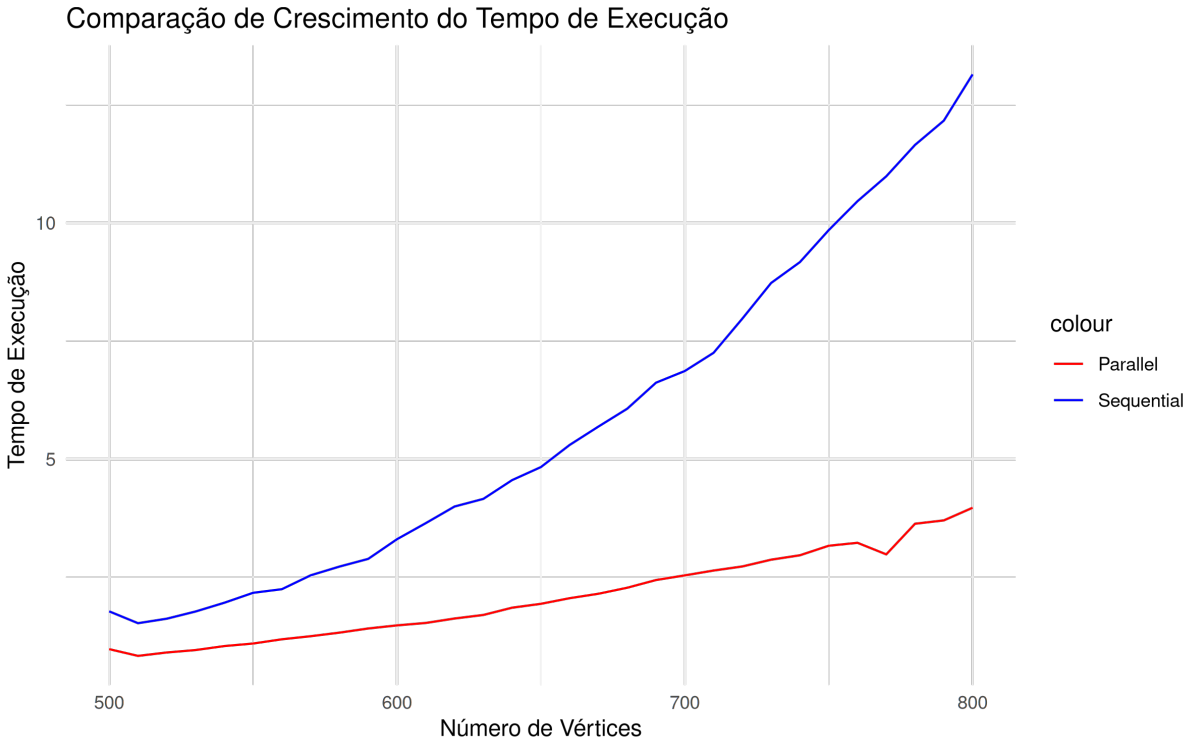

Figura 3: Gráfico de comparação entre o crescimento das soluções em grafo denso.

Outro aspecto importante possível de se observar principalmente na figura 3 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 a tabela na figuras 4, que mostra o speed up de alguns testes, onde é possível confirmar o grande ganho da solução paralela, principalmente quando é necessário mais trabalho de computação, com grafos mais densos e maiores.

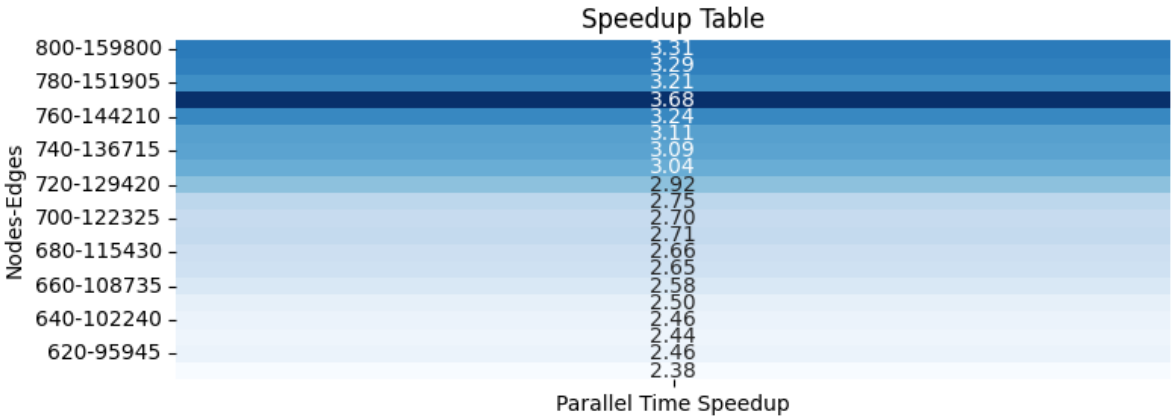

Figura 4: Tabela de comparação entre o aumento de threads em um grafo denso.

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 ferramenta como o OpenMP e o MPI permitem esse ganho, considerado suas complexidades de implementação.

Deve se acentuar 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.