Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
REDES DE COMPUTADORES 2024/1

RESOLVENDO O PROBLEMA DE VIZINHOS COMUNS EM UM GRAFO COM OPEN MPI

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 o crescimento das soluções em grafo denso. . . 6
5 Gráfico de comparação entre as soluções em um grafo esparso. 7
6 Gráfico de comparação entre o crescimento das soluções em grafo esparso. . 7
7 Gráfico de comparação entre as soluções em um grafo denso. 8
8 Gráfico de comparação entre o crescimento das soluções em grafo denso. . . 8

Sumário

1 INTRODUÇÃO **1**

1.1 Objetivo 1

1.2 Motivação 1

1.3 Descrição do Problema 1

2 Solução Sequencial **1**

3 Solução MPI **2**

4 Testes **3**

5 Conclusão **9**

REFERÊNCIAS **9**

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 Message-Passing Interface (MPI), mais especificamente a implementação Open MPI na 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 distribuida.

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 Open MPI, 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 MPI

A solução paralela utilizando 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.

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.

Por fim, o processo de *rank* 0 irá escrever todos os dados no arquivo de saida, concluindo o programa.

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 de se o grafo é esparso ou denso. A medição do tempo foi feita utilizando a função *MPI_Wtime* no MPI, que retorna o tempo de relógio em segundos, e a função *clock* no programa sequencial, que faz o mesmo que a função do MPI.

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 MPI com 4 processos.

Vértices	**Sequencial**	**MPI**
10	0.000322	0.000182
50	0.001381	0.000255
100	0.004095	0.000388
500	0.074238	0.006274
1000	0.302496	0.027614
1500	0.567766	0.060609
2000	1.087811	0.100393
2500	3.398323	0.129211

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

Podemos observar com clareza a forma com que o programa sequencial perde em performace conforme as entradas começam a crescer.

Para mais fácil observação desta tendência, foi criado os gfaficos nas figuras 2 e 1 com a comparação entre o crescimento do tempo conforme a entrada entre a solução sequencial e a solução MPI com 4 processos para um grafo esparso.

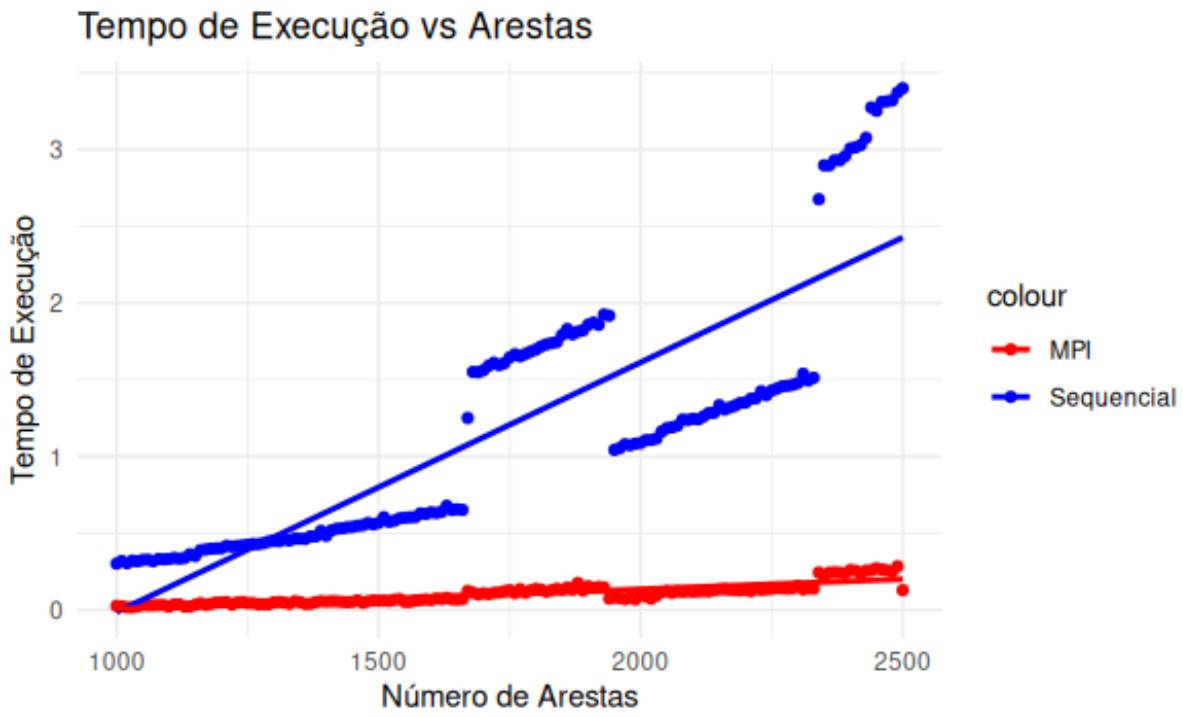

Figura 1: Gráfico de comparação entre as soluções em um grafo esparso.

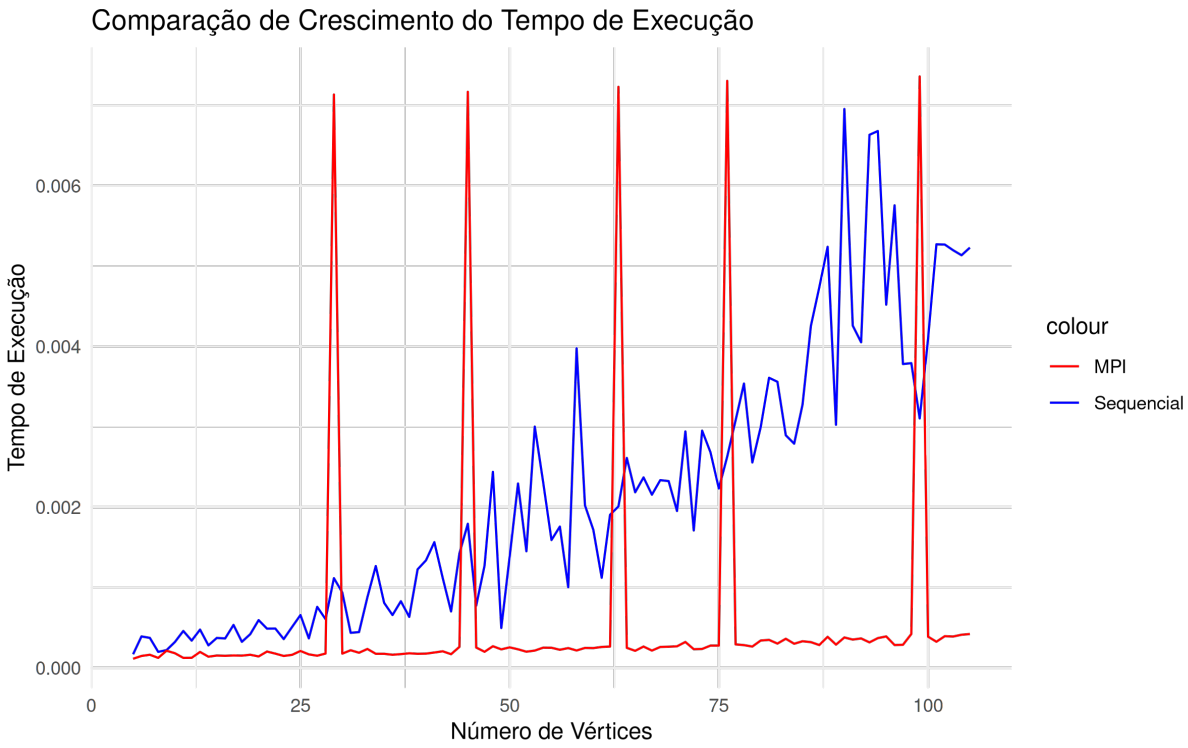

Figura 2: Gráfico de comparação entre o crescimento das soluções em grafo esparso.

A 2 mostra alguns picos de tempo em ambas as soluções, que podem ter decorrido

de momentos específicos no uso da CPU. Mas, tirando isso, ainda 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 e 4. Nelas, também é possível confirmar a disparidade da solução paralela em relação a solução sequencial, de forma até mais marcante, 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, tornando o programa mais eficiente.

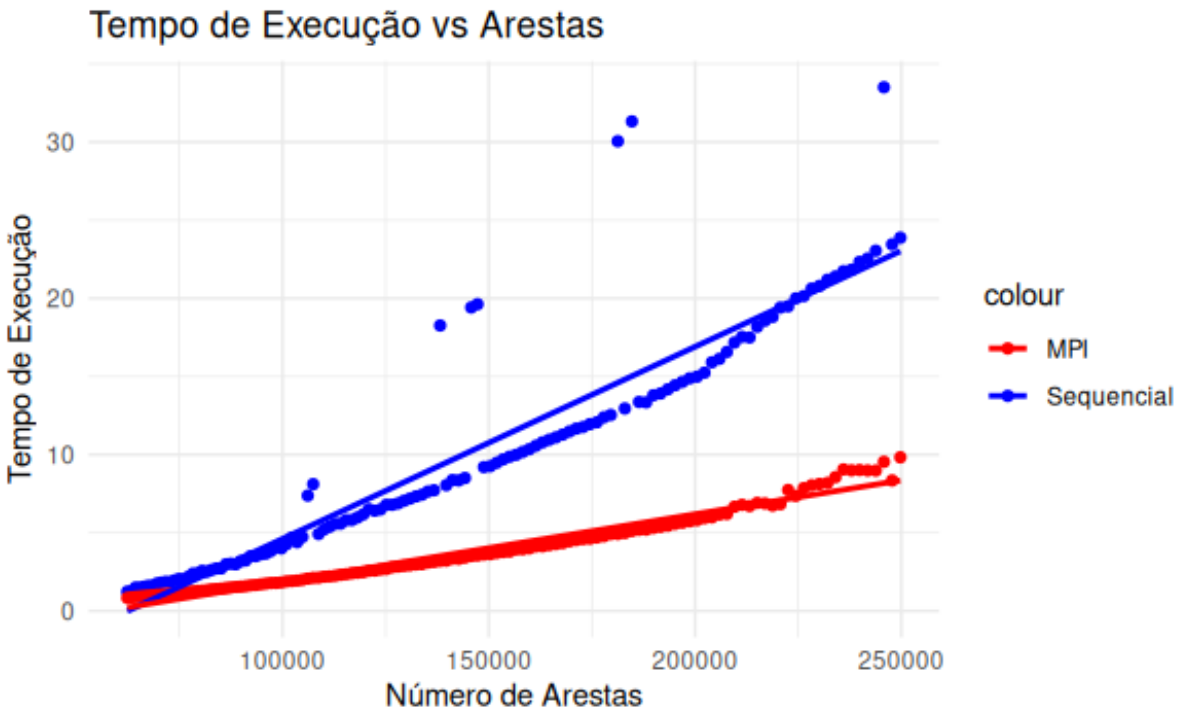

Figura 3: Gráfico de comparação entre as soluções em um grafo denso.

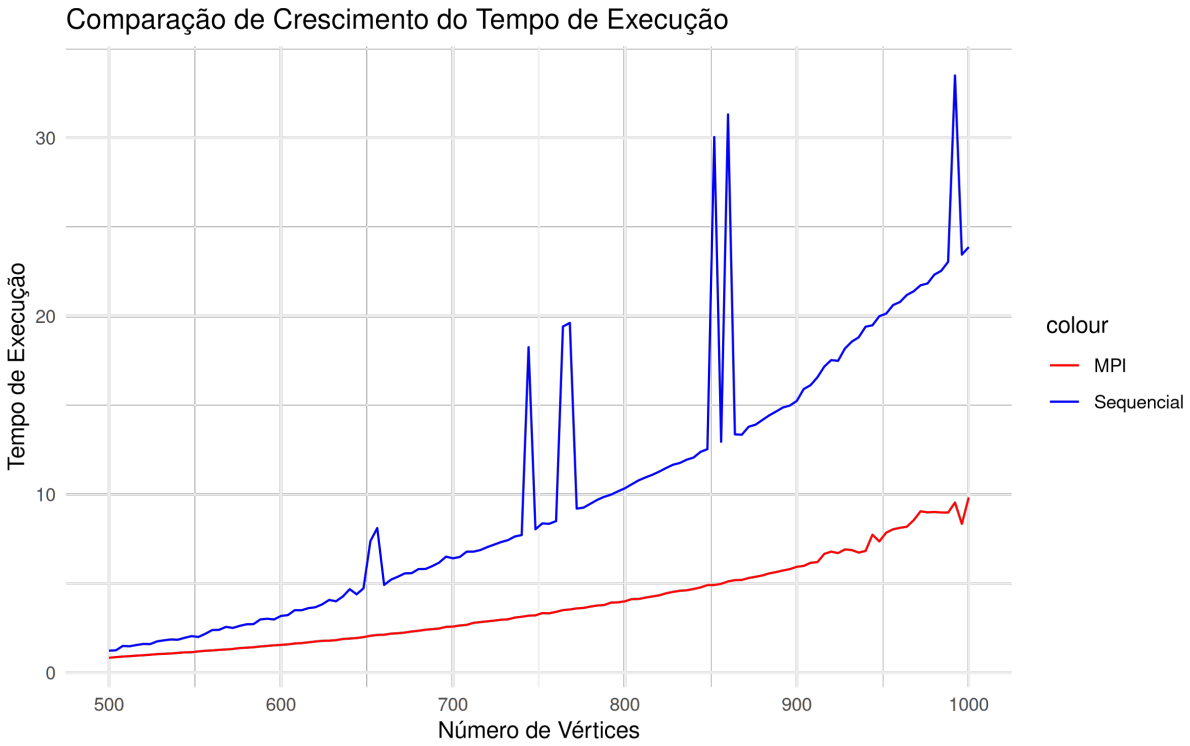

Figura 4: Gráfico de comparação entre o crescimento das soluções em grafo denso.

Por fim, as figuras 5 e 6 mostram uma comparação entre a solução sequencial e a solução MPI com 2, 3 e 4 processos para um grafo esparso, e as figuras 7 e 8 o mesmo, mas para um grafo denso.

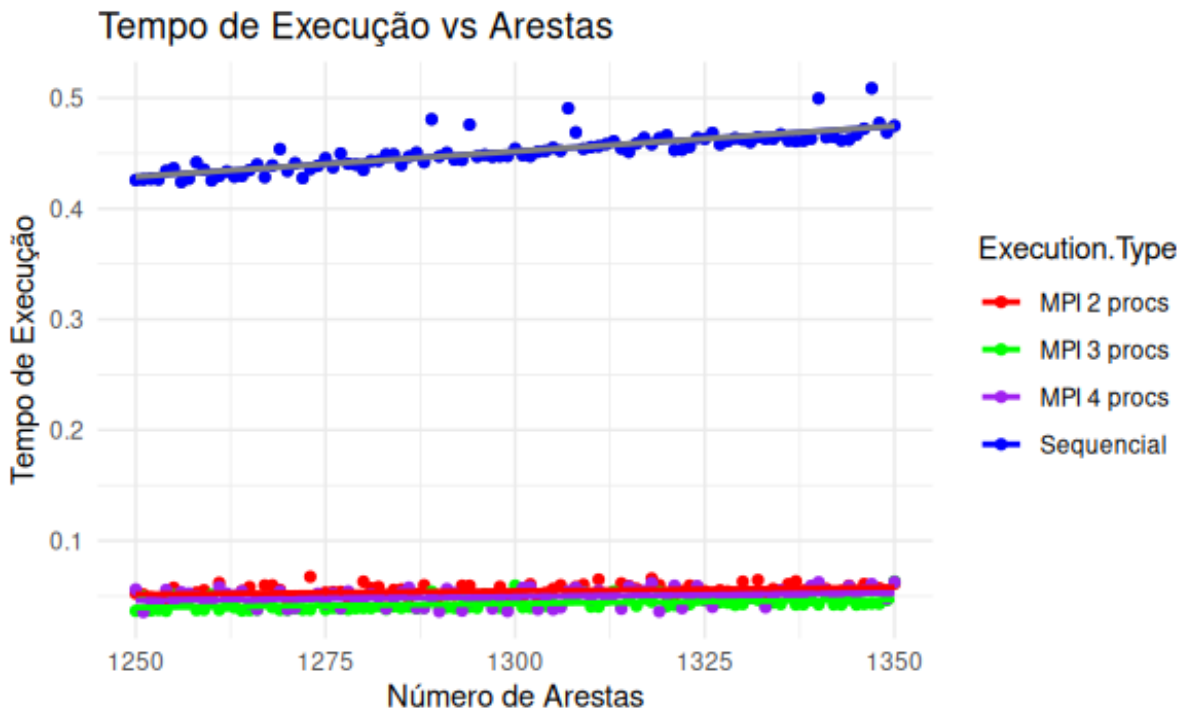

Figura 5: Gráfico de comparação entre as soluções em um grafo esparso.

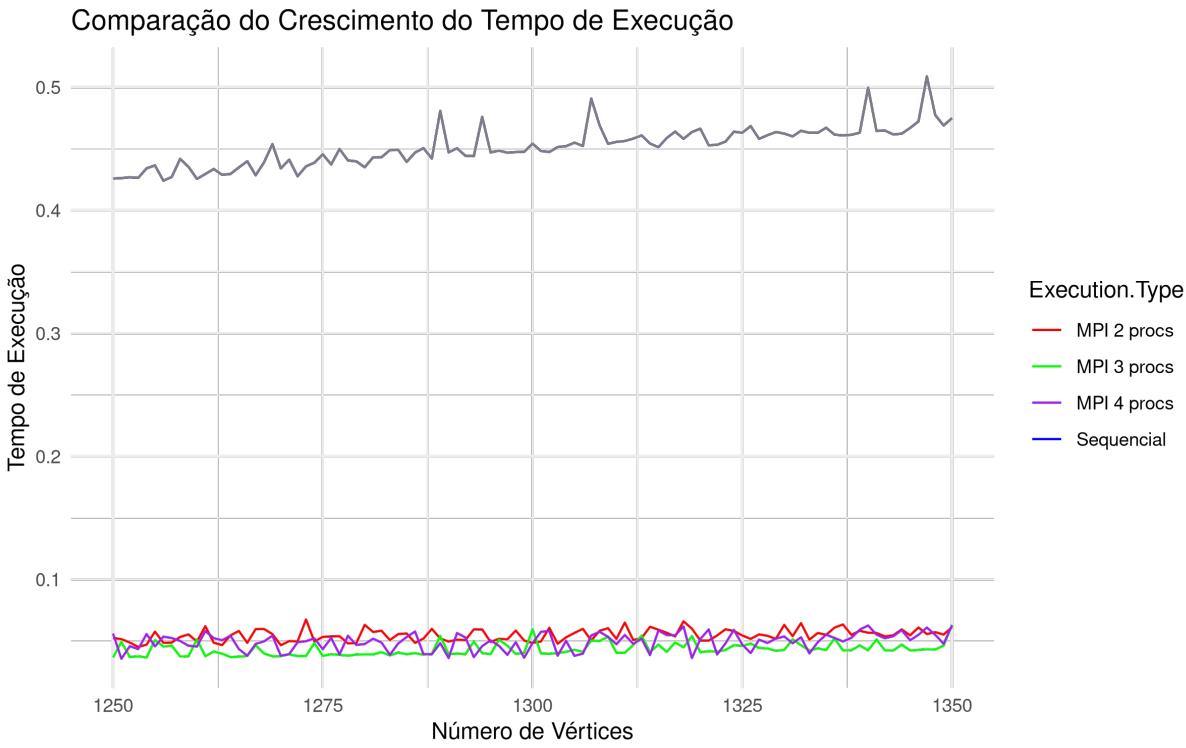

Figura 6: Gráfico de comparação entre o crescimento das soluções em grafo esparso.

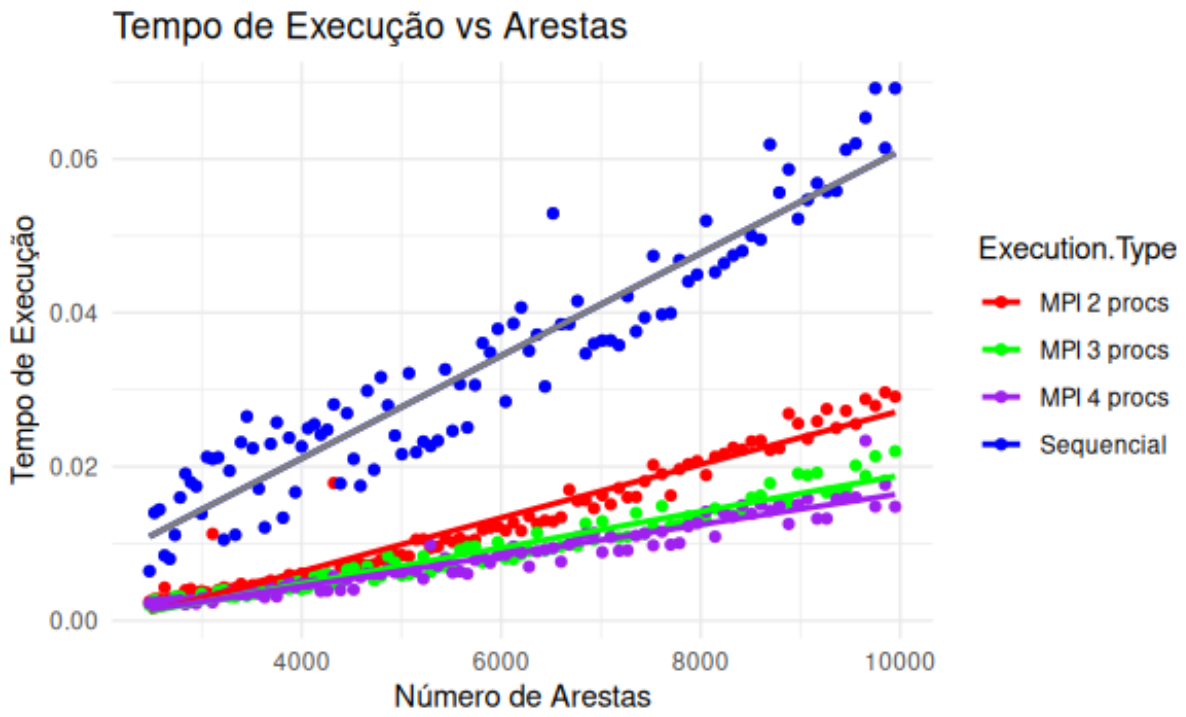

Figura 7: Gráfico de comparação entre as soluções em um grafo denso.

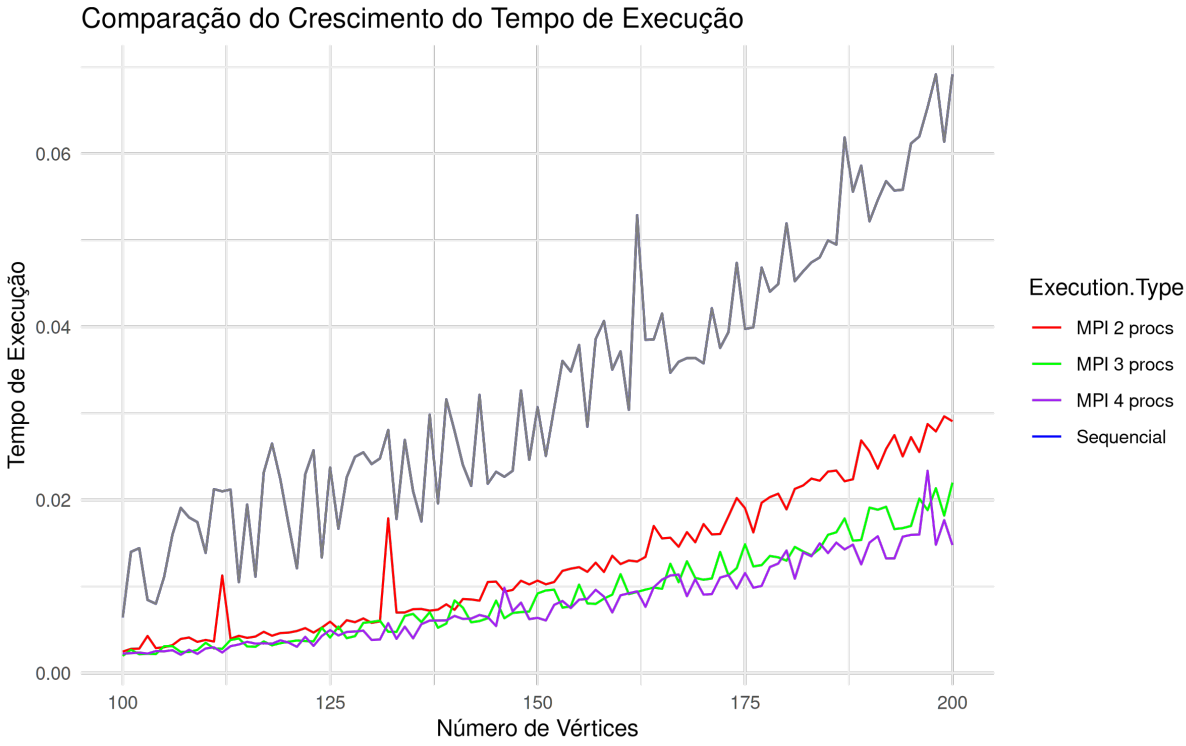

Figura 8: Gráfico de comparação entre o crescimento das soluções em grafo denso.

Fica claro pelas imagens como a soluçãp MPI melhora o desempenho mesmo com apenas 2 processos. Também é possível notar certa melhora conforme o número de processos

aumenta, apesar de ser pouca a melhora entre 3 e 4 processos, sendo necessário um número alto de arestas para ser possível notar. Testes mais robustos, incluindo mais máquinas, seriam necessários para confirmar um número mais aproximado da quantidade ideal de processos para resolver o problema, levando em conta o tempo gasto na comunicação entre os processos.

5 Conclusão

Ao observar os resultados, foi possível ter noção de como a programação paralela pode ser uma ferramenta poderosa para a melhora de desempenho na solução de problemas complexos. Este trabalho permitiu o aprendizado e experimento de técnicas que permitem esse aumento de performace.

Também é importante notar o desafio de se programar paralelamente, onde fatores como o pensamento da resolução do problema ou como debugar o programa são diferentes do que se normalmente usa, o que traz desafios ao tentar utilizar essa ferramenta.

Referências

[Gropp et al., 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2nd edition.

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