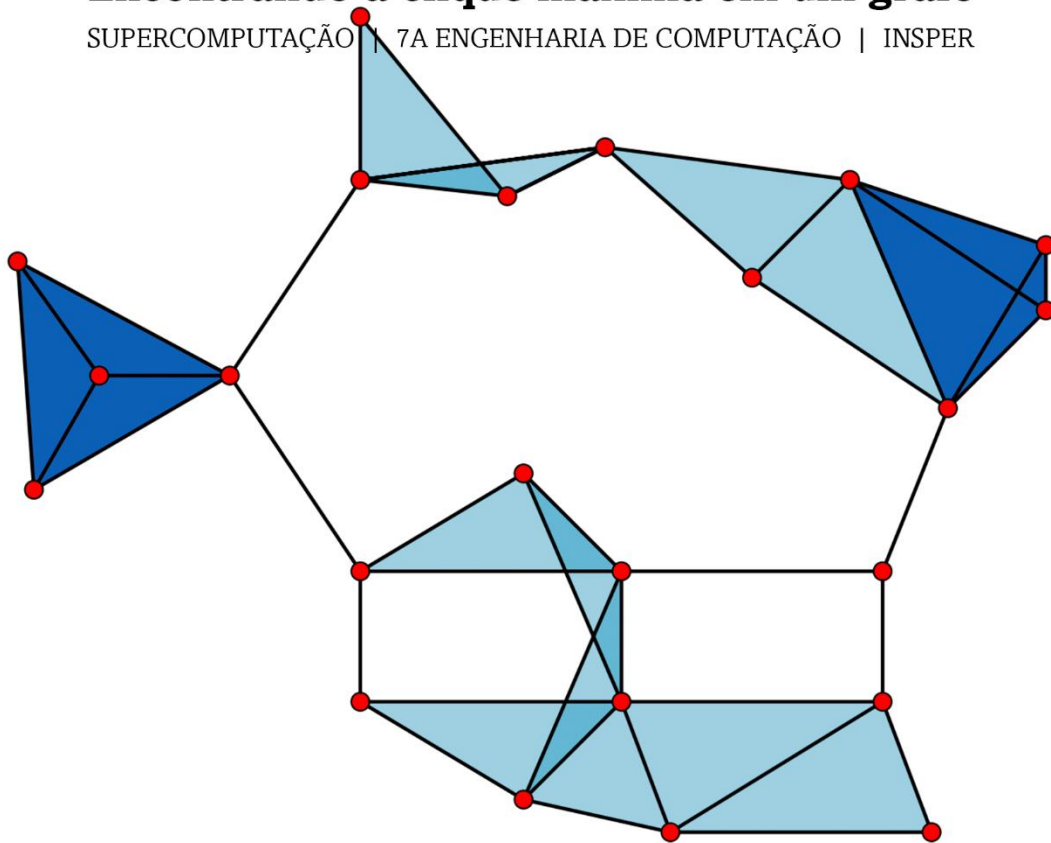




# Análise de Redes Sociais: Encontrando a clique máxima em um grafo

SUPERCOMPUTAÇÃO | 7A ENGENHARIA DE COMPUTAÇÃO | INSPer



**Aluno: Lister Ogusuku Ribeiro**  
**Professores: André Filipe e Michel Fornaciali**

**10 de dezembro de 2023**

**Insper**

## Sumário

<b>Introdução</b> .....	2
<b>Abordagem Exaustiva</b> .....	3
<b>Implementação com Threads OpenMP</b> .....	5
Speed Up .....	8
Algumas evidências das cliques .....	9
<b>Paralelização e Distribuição do Processamento com MPI</b> .....	12
Speed Up .....	15
Comparação entre OpenMP e MPI .....	16
Algumas evidências das cliques .....	16
<b>Conclusão</b> .....	20
<b>Referências</b> .....	21

## Introdução



Figura 1 - Redes sociais | Imagem: Gatton College

outros estudos pelo fato de que sua ênfase não é nos atributos (características) dos atores, mas nas ligações entre eles.

A ideia de uma clique em um grafo é relativamente simples. No nível mais geral, **uma clique é um subconjunto de uma rede no qual os atores são mais próximos entre si do que com outros membros da rede.** Em termo de laços de amizade, por exemplo, não é incomum encontrar **grupos humanos que formam cliques baseando-se em idade, gênero, raça, etnia, religião, ideologia, e muitas coisas coisas.** Uma clique é, portanto, um conjunto de vértices em um grafo em que cada par de vértices está diretamente conectada por uma aresta.

A **análise de redes sociais (ARS)** é uma abordagem oriunda de áreas tais como Sociologia, Psicologia Social e Antropologia. Tal abordagem estuda as **ligações relacionais (relational tie) entre atores sociais.** Os atores na ARS podem ser tanto pessoas e empresas, analisadas como unidades individuais, quanto unidades sociais coletivas como, por exemplo, departamentos dentro de uma organização, agências de serviço público em uma cidade, estados-nações de um continente, dentre outras. A ARS difere fundamentalmente de

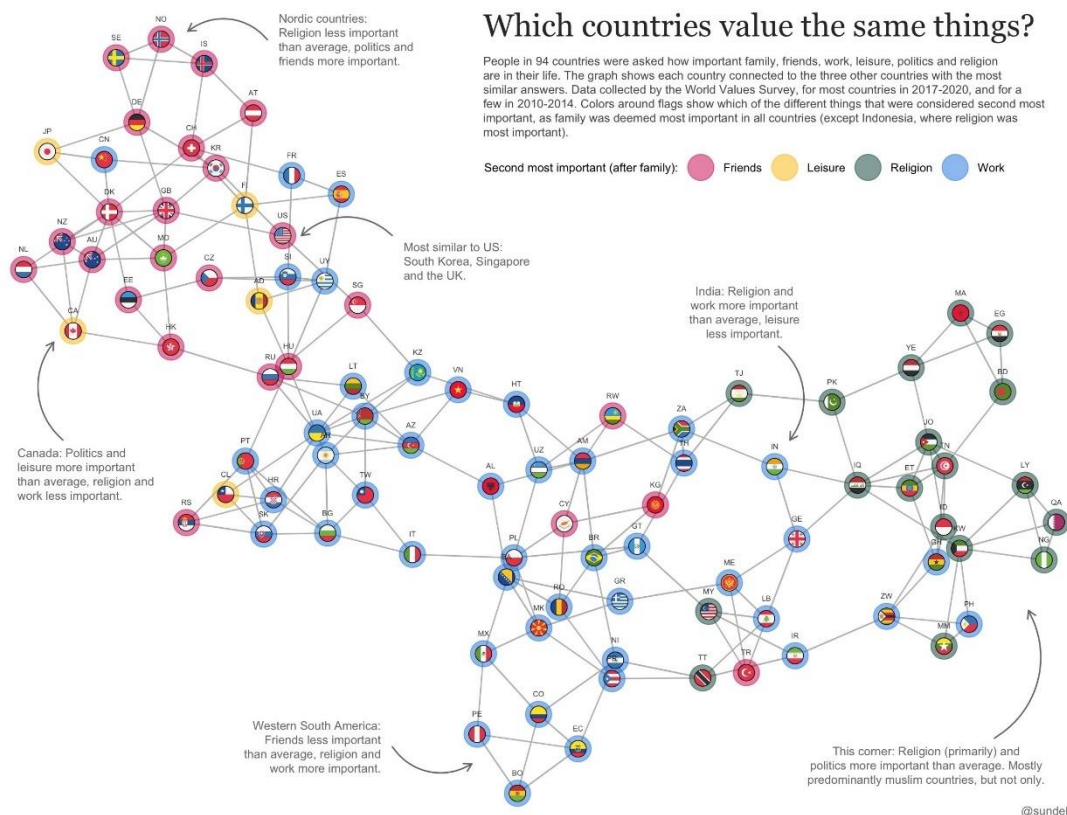
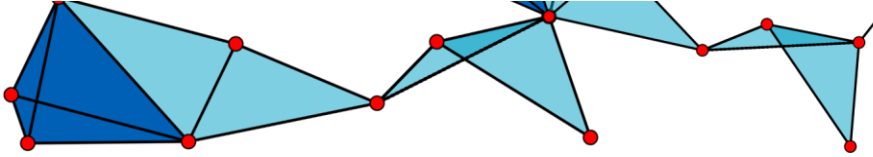


Figura 2 - "Quais países valorizam as mesmas coisas?" | Imagem: @sundellviz - X



Encontrar a clique máxima em um grafo é uma tarefa **computacionalmente desafiadora devido à natureza combinatória do problema**. A dificuldade computacional surge da necessidade de explorar todas as combinações possíveis de vértices para identificar a maior clique, o que se torna exponencial em relação ao número de vértices. Isso resulta em uma complexidade computacional alta, mesmo para grafos moderadamente grandes.

A importância de estudar cliques está notavelmente presente na análise de redes sociais, onde as cliques representam **grupos coesos de indivíduos que compartilham interesses, amizades ou conexões em comum**. A identificação de cliques ajuda a entender a estrutura de uma rede social, identificar influenciadores e grupos de afinidade, além de auxiliar na detecção de comunidades e na análise de dinâmicas sociais.

As cliques são importantes, pois além de desenvolver em seus membros comportamentos homogêneos, elas têm, por definição, grande proximidade, aumentando a velocidade das trocas. Assim, informações dirigidas a uma clique são rapidamente absorvidas pelos seus membros, que tendem a percebê-las de forma semelhante. Isso é importante, por exemplo, em estratégias de segmentação.

Portanto, a resolução eficiente do problema da clique máxima tem aplicações valiosas em áreas que vão desde a ciência da computação até a análise de dados em redes sociais.

## Abordagem Exaustiva

A abordagem exaustiva, no contexto da identificação da clique máxima em um grafo, é um método que envolve a **avaliação sistemática de todas as combinações possíveis de vértices para encontrar a maior clique possível**. Essa abordagem é direta e pode ser detalhada da seguinte maneira:

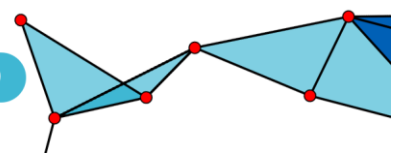
**Seleção Iterativa de Vértices:** O processo começa selecionando um vértice e, em seguida, adiciona outros vértices que estão conectados a ele. A ideia é formar um conjunto de vértices que estejam todos conectados entre si, ou seja, formando uma clique.

**Expansão da Clique:** A cada etapa, o algoritmo verifica se é possível adicionar um novo vértice ao conjunto atual, mantendo a propriedade de que todos os vértices dentro do conjunto estão conectados uns aos outros. Geralmente, dá-se preferência aos vértices que têm o maior número de conexões (vizinhos) já presentes na clique formada.

**Verificação de Todas as Combinações:** A busca exaustiva explora todas as combinações possíveis de vértices. Isso significa que, para cada vértice, o algoritmo considera todas as possibilidades de incluir ou não incluir outros vértices na clique.

**Identificação da Clique Máxima:** Durante a busca, o algoritmo mantém um registro da maior clique encontrada. Ao final do processo, quando todas as combinações possíveis foram exploradas, a maior clique identificada é considerada a clique máxima.

**Complexidade Computacional:** A abordagem exaustiva tem uma alta complexidade computacional, pois o número de combinações possíveis de vértices cresce exponencialmente com o tamanho do grafo. Isso significa que, mesmo para grafos de tamanho moderado, o tempo necessário para encontrar a clique máxima pode se tornar impraticável.





Apesar de sua simplicidade e garantia de encontrar a solução correta, a **abordagem exaustiva é frequentemente inviável para grafos grandes devido ao seu alto custo computacional**. Por essa razão, métodos mais eficientes e heurísticos são geralmente preferidos em aplicações práticas, especialmente em análises de redes sociais grandes e complexas. Esses métodos alternativos mais eficientes são justamente o que vamos explorar mais a fundo neste projeto.

```

1 // Projeto de Supercomputação | Engenharia de Computação - Insper
2 // Professores: André Filipe e Michel Fornaciali
3
4 // Inclui bibliotecas padrão necessárias para o funcionamento do programa.
5 #include <iostream> // Para operações de entrada e saída.
6 #include <vector> // Para usar o tipo de dados 'vector', uma coleção dinâmica.
7 #include <algorithm> // Para algoritmos como max_element.
8 #include <fstream> // Para manipulação de arquivos.
9 #include <chrono> // Para medir o tempo de execução do programa.
10
11 using namespace std;
12
13 // Função que lê um grafo de um arquivo e retorna sua matriz de adjacência.
14 vector<vector<int>> LerGrafo(const string& fileName, int& qntVertices) {
15     ifstream arquivo(fileName); // Abre o arquivo para leitura.
16     int qntArestas;
17     arquivo >> qntArestas; // Lê a quantidade de vértices e arestas do arquivo.
18     vector<vector<int>> grafo(qntVertices, vector<int>(qntVertices, 0)); // Cria uma matriz de adjacência.
19
20     // Lê as arestas do arquivo e atualiza a matriz de adjacência.
21     for (int i = 0; i < qntArestas; ++i) {
22         int u, v;
23         arquivo >> u >> v; // Lê um par de vértices que formam uma aresta.
24         grafo[u-1][v-1] = 1; // Atualiza a matriz para representar a aresta.
25         grafo[v-1][u-1] = 1; // Garante a bidirecionalidade da aresta.
26     }
27     arquivo.close(); // Fecha o arquivo.
28     return grafo; // Retorna a matriz de adjacência.
29 }
30
31 // Função que busca exaustivamente todas as cliques maximais no grafo.
32 void buscaExaustiva(const vector<vector<int>>& grafo, int qntVertices, vector<int>& cliqueAtual, vector<vector<int>>& cliquesMaximas) {
33     bool maxima = true; // Flag para verificar se a clique atual é máxima.
34
35     // Verifica se todos os vértices na cliqueAtual estão conectados entre si.
36     for (int i : cliqueAtual) {
37         for (int j : cliqueAtual) {
38             if (i != j && grafo[i][j] == 0) {
39                 maxima = false; // Se algum par de vértices não está conectado, não é uma clique máxima.
40                 break;
41             }
42         }
43         if (!maxima) break;
44     }
45
46     // Se a clique atual é uma clique máxima, adiciona-a ao vetor de cliques maximais.
47     if (maxima) {
48         cliquesMaximas.push_back(cliqueAtual);
49     }
50
51     // Tenta adicionar cada vértice à cliqueAtual e faz uma chamada recursiva.
52     for (int i = 0; i < qntVertices; ++i) {
53         bool podeAdicionar = true; // Flag para verificar se um vértice pode ser adicionado à clique.
54         for (int v : cliqueAtual) {
55             if (grafo[i][v] == 0) { // Se o vértice não está conectado a todos na cliqueAtual, não pode ser adicionado.
56                 podeAdicionar = false;
57                 break;
58             }
59         }
60
61         // Se o vértice pode ser adicionado, faz uma chamada recursiva.
62         if (podeAdicionar) {
63             cliqueAtual.push_back(i);
64             buscaExaustiva(grafo, qntVertices, cliqueAtual, cliquesMaximas);
65             cliqueAtual.pop_back(); // Remove o vértice adicionado para explorar outras possibilidades.
66         }
67     }
68 }

```

Figura 3 - Código da abordagem Exaustiva - Heurística (parte 1)

```

1 int main(int argc, char* argv[]) {
2     // Verifica se o nome do arquivo foi passado como argumento na linha de comando.
3     if (argc < 2) {
4         cerr << "Uso: " << argv[0] << " <input_file>" << endl;
5         return 1; // Encerra o programa se o nome do arquivo não foi fornecido.
6     }
7
8     string input_file = argv[1]; // Nome do arquivo passado como argumento.
9     int qntVertices;
10    vector<vector<int>> grafo = LerGrafo(input_file, qntVertices); // Lê o grafo do arquivo.
11    vector<int> cliqueAtual; // Armazena a clique que está sendo explorada no momento.
12    vector<vector<int>> cliquesMaximas; // Armazena todas as cliques maximais encontradas.
13
14    // Mede o tempo de execução da busca exaustiva.
15    auto start_time = chrono::high_resolution_clock::now();
16    buscaExaustiva(grafo, qntVertices, cliqueAtual, cliquesMaximas);
17    auto end_time = chrono::high_resolution_clock::now();
18    chrono::duration<double> elapsed_time = end_time - start_time; // Calcula o tempo decorrido.
19
20    // Encontra a maior clique máxima.
21    vector<int> clique_maxima = *max_element(cliquesMaximas.begin(), cliquesMaximas.end(), [](const vector<int>& a, const vector<int>& b) {
22        return a.size() < b.size(); // Função para comparar o tamanho das cliques.
23    });
24
25    // Imprime todas as cliques maximais encontradas.
26    cout << "Cliques máximas encontradas: " << endl;
27    for (const auto& clique : cliquesMaximas) {
28        cout << "[";
29        for (int i = 0; i < clique.size(); ++i) {
30            cout << clique[i]+1; // Imprime cada vértice da clique.
31            if (i < clique.size() - 1) {
32                cout << ", ";
33            }
34        }
35        cout << "]" << endl;
36    }
37
38    // Imprime a maior clique máxima encontrada.
39    cout << "Clique máxima encontrada: ";
40    cout << "[";
41    for (int i = 0; i < clique_maxima.size(); ++i) {
42        cout << clique_maxima[i] + 1; // Imprime cada vértice da clique.
43        if (i < clique_maxima.size() - 1) {
44            cout << ", ";
45        }
46    }
47    cout << "]" << endl;
48    cout << "Tempo de execução: " << elapsed_time.count() << "s" << endl; // Imprime o tempo de execução.
49    return 0;
50 }

```

Figura 4 - Código da abordagem Exaustiva - Heurística (parte 2)

## Implementação com Threads OpenMP

OpenMP é uma API que suporta a programação multithreaded em C, C++ e Fortran. Ele permite a **paralelização de loops e seções críticas do código de forma simples e eficaz**. A implementação de uma solução com OpenMP para encontrar cliques em um grafo envolve a paralelização das iterações do algoritmo em diferentes threads. O OpenMP simplifica a criação de threads e a coordenação entre elas. Quando falamos em implementação com OpenMP, alguns passos que podemos seguir são, por exemplo:

**Identificação de Regiões Paralelizáveis:** O primeiro passo é identificar as partes do algoritmo que podem ser executadas em paralelo. No caso da heurística gulosa para encontrar cliques, podemos paralelizar a etapa onde verificamos a possibilidade de adicionar novos vértices à clique.

**Uso de Diretivas OpenMP:** Utilizamos diretivas OpenMP como `#pragma omp parallel for` para distribuir iterações de loops entre diferentes threads.

**Gerenciamento de Concorrência:** Como múltiplas threads podem modificar estruturas de dados compartilhadas, como a lista de cliques, é crucial gerenciar a concorrência. Isso pode ser feito através de seções críticas (`#pragma omp critical`) ou locks.

No gráfico representado pela figura 5 é possível analisar a diferença obtida entre as implementações. Note que a implementação utilizando OpenMP apresentou um desempenho melhor, rodando num tempo de execução menor para valores de N entre 5 até 25:

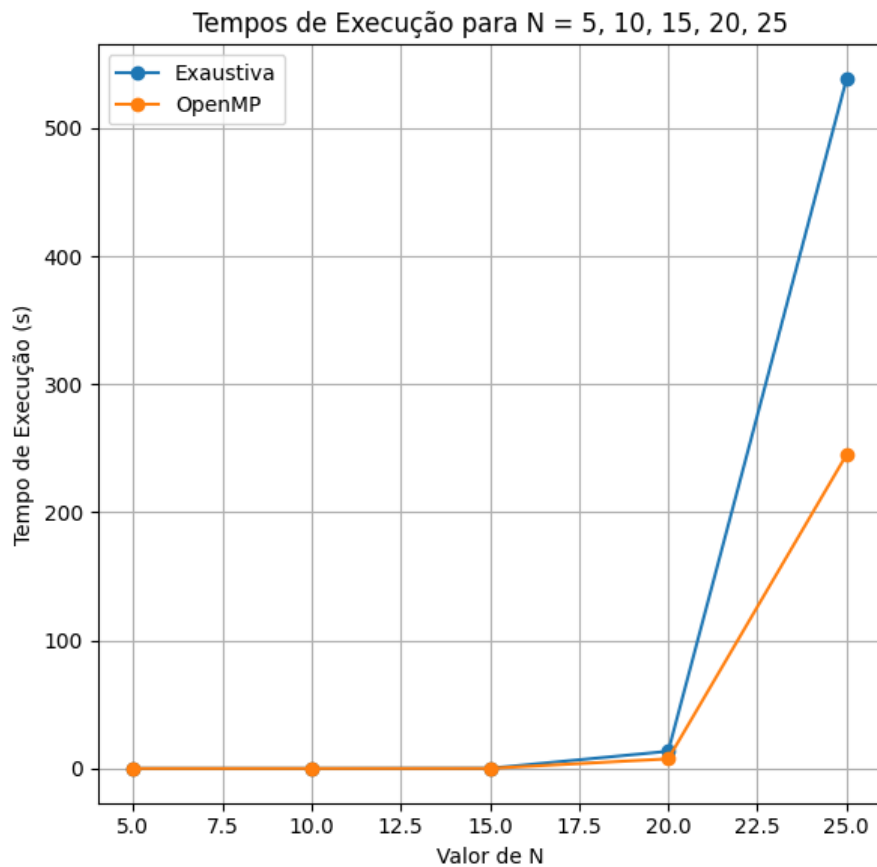


Figura 5 - Comparação dos tempos de execução entre a abordagem Exaustiva/Heurística e OpenMP de acordo com a variação da quantidade dos nós (N) entre 5 até 25

Já no gráfico da figura 6, foi variado o parâmetro de 'probabilidade de haver conexão' para 50%, dado que o valor anteriormente testado estava muito alto e demorando muito para rodar. Após a variação do parâmetro, foi possível analisar que a diferença obtida entre as implementações continuava a mesma, com a implementação em OpenMP ainda assim performando num tempo melhor.

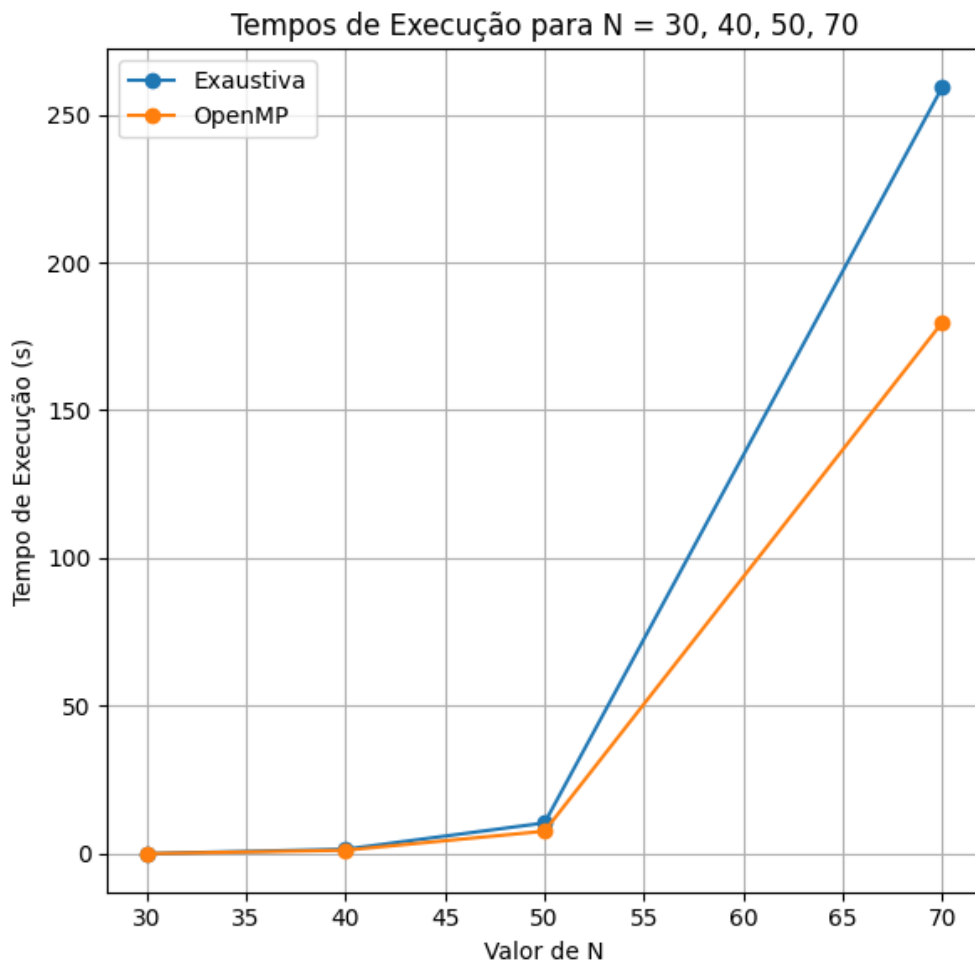


Figura 6 - Comparação dos tempos de execução entre a abordagem Exaustiva/Heurística e OpenMP de acordo com a variação da quantidade dos nós (N) entre 30 até 70



## Speed Up

Quando os tempos de execução têm uma grande diferença, o *speed up* tende a ser menos perceptível em um gráfico comum, pois as linhas de tempo de execução dominam a visualização. Para ter uma ideia mais clara do *speed up*, vamos plotar o gráfico do *speed up* em um eixo y secundário, com uma escala própria. Isso ajudará a visualizar melhor as suas variações, separando-o das linhas de tempo de execução.

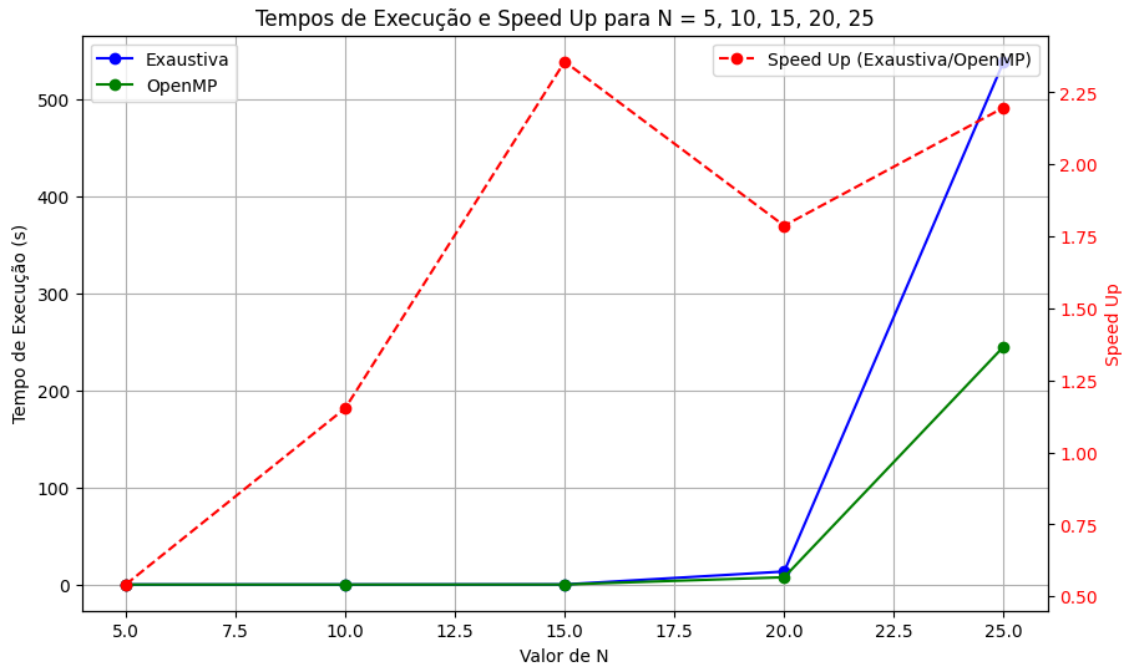


Figura 7 - Tempos de Execução e Speed Up das abordagens Exaustiva/Heurística e OpenMP com quantidade de nós (N) entre 5 até 25

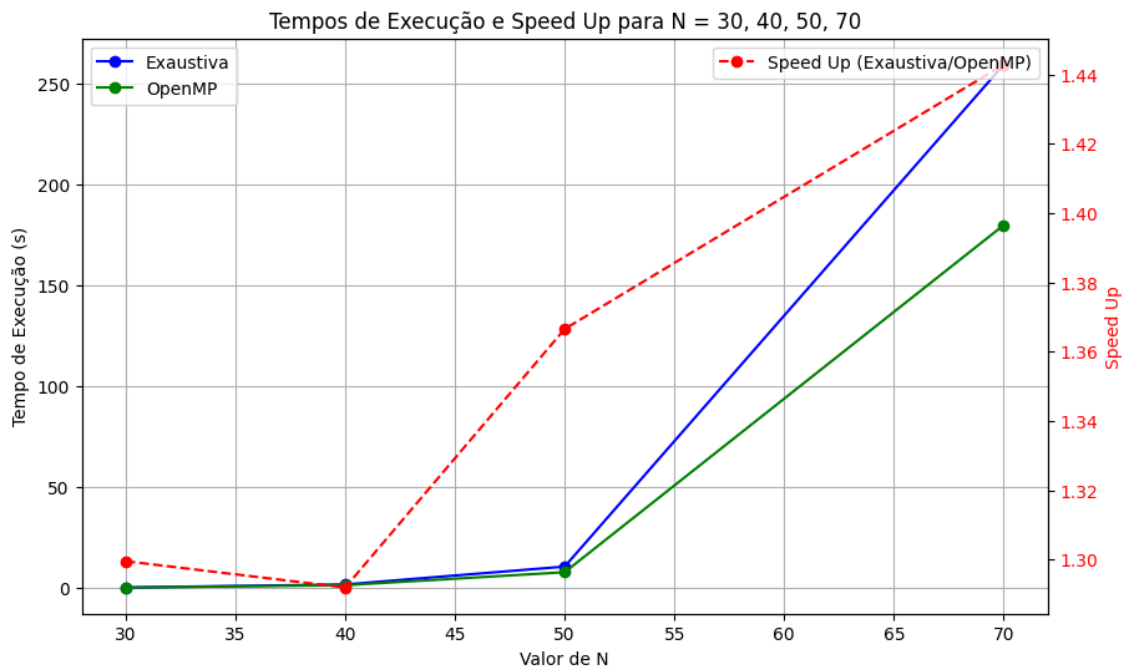


Figura 8 - Tempos de Execução e Speed Up das abordagens Exaustiva/Heurística e OpenMP com quantidade de nós (N) entre 30 até 70

Agora, atualizamos o gráfico com um eixo secundário para o *speed up*, onde as linhas azul e verde representam os tempos de execução das implementações Exaustiva e OpenMP, respectivamente. Já a linha pontilhada vermelha, no eixo à direita, mostra o *speed up* (Exaustiva/OpenMP) para cada valor de N.

A linha de *speed up* nos gráficos indica a razão do tempo de execução da implementação Exaustiva pelo tempo da implementação OpenMP. **Um valor de *speed up* maior do que 1 indica que a implementação OpenMP é mais rápida do que a Exaustiva.** Quanto maior for o valor do *speed up*, maior é a vantagem de desempenho da implementação OpenMP sobre a Exaustiva.

Um *speed up* consistente e significativo em valores mais altos de N pode ser um indicativo forte para preferir a implementação OpenMP sobre a Exaustiva em problemas de grande escala, enquanto em problemas menores, a diferença de desempenho pode não ser tão crítica. (É importante considerar que o *speed up* depende não apenas da implementação do algoritmo, mas também do hardware em que os testes são executados, bem como de outros fatores ambientais. Portanto, os resultados obtidos e aqui apresentados são específicos para o contexto em que os testes foram realizados (no cluster da disciplina, nesse caso).

## Algumas evidências das cliques

Algumas evidências dos outputs recebidos durante a execução das implementações (e utilizados para a confecção dos gráficos acima) podem ser vistas a seguir:

```
upercomputação\projeto-novo-repositorio\projeto-supercomp-novo-repo> python .\output.py
Cliques máximas encontradas:
['5', '10', '3', '8', '6']
['5', '10', '7']
['5', '4']
['8', '2', '3']
['1', '9', '3', '2']
['1', '9', '3', '6']
['1', '9', '7']
['1', '10', '3', '6']
['1', '10', '7']
['1', '4', '2']
Clique máxima encontrada: ['5', '10', '3', '8', '6']
```

Figura 9 - Clique máxima encontrada para n = 10

```
[10, 8, 6, 5]
[10, 8, 6, 5, 3]
Clique máxima encontrada:[3, 5, 6, 8, 10]
Tempo de execução: 0.00106659s
```

Figura 10 - Clique máxima encontrada e tempo de execução para n = 10 na abordagem Exaustiva/Heurística

```
[3, 5, 6, 8, 10]
Clique máxima encontrada (OpenMP): [6, 3, 5, 8, 10]
Tempo de execução: 0.000924805s
```

Figura 11 - Clique máxima encontrada e tempo de execução para n = 10 na abordagem OpenMP

```
['14', '11', '19', '4', '13', '10', '15']
Clique máxima encontrada: ['20', '11', '2', '18', '6', '17', '15', '8']
PS C:\Users\lister\OneDrive - Insper - Instituto de Ensino e Pesquisa\7 se
```

Figura 12 - Clique máxima encontrada para n = 20

```
[1, 2, 6, 10, 11, 16, 17, 20]
Clique máxima encontrada:[1, 2, 6, 10, 11, 16, 17, 20]
Tempo de execução: 13.3899s
```

Figura 13 - Clique máxima encontrada e tempo de execução para n = 20 na abordagem Exaustiva/Heurística

```
[12, 9, 20, 8]
Clique máxima encontrada (OpenMP): [11, 1, 2, 6, 10, 16, 17, 20]
Tempo de execução: 7.49242s
[11, 1, 2, 6, 10, 16, 17, 20]
```

Figura 14 - Clique máxima encontrada e tempo de execução para  $n = 20$  na abordagem OpenMP

```
[20, 18, 23, 23, 11]
Clique máxima encontrada: ['21', '19', '10', '22', '1', '15', '24', '12', '14', '4']
PS C:\Users\liste\OneDrive - Insper - Instituto de Ensino e Pesquisa\7 semestre Insper
```

Figura 15 - Clique máxima encontrada para  $n = 30$

```
[25, 23, 21, 19, 4]
[25, 23, 21, 19, 11]
Clique máxima encontrada: [1, 4, 10, 12, 14, 15, 19, 21, 22, 24]
Tempo de execução: 538.341s
```

Figura 16 - Clique máxima encontrada e tempo de execução para  $n = 30$  na abordagem Exaustiva/Heurística

```
[23, 11, 20, 7, 18]
Clique máxima encontrada (OpenMP): [14, 1, 4, 10, 12, 15, 19, 21, 22, 24]
Tempo de execução: 244.992s
[14, 1, 4, 10, 12, 15, 19, 21, 22, 24]
```

Figura 17 - Clique máxima encontrada e tempo de execução para  $n = 30$  na abordagem OpenMP

```

1 // Projeto de Supercomputação | Engenharia de Computação - Insper
2 // Professores: André Filipe e Michel Fornaciari
3
4 #include <iostream> // Inclui a biblioteca padrão de entrada e saída.
5 #include <vector> // Inclui a biblioteca para usar vetores dinâmicos.
6 #include <algorithm> // Inclui a biblioteca para operações de algoritmo como sort.
7 #include <omp.h> // Inclui a biblioteca OpenMP para paralelização.
8 #include <fstream> // Inclui a biblioteca para manipulação de arquivos.
9 #include <chrono> // Inclui a biblioteca para medir o tempo.
10
11 using namespace std;
12
13 // Função para ler um grafo de um arquivo e retornar sua matriz de adjacência.
14 vector<vector<int>> LerGrafo(const string& fileName, int& qntVertices) {
15     ifstream arquivo(fileName); // Abre o arquivo para leitura.
16     int numArestas;
17     arquivo >> qntVertices >> numArestas; // Lê a quantidade de vértices e arestas.
18
19     vector<vector<int>> grafo(qntVertices, vector<int>(qntVertices, 0)); // Cria matriz de adjacência.
20
21     for (int i = 0; i < numArestas; ++i) {
22         int u, v;
23         arquivo >> u >> v; // Lê as arestas.
24         grafo[u-1][v-1] = 1; // Preenche a matriz de adjacência.
25         grafo[v-1][u-1] = 1;
26     }
27
28     arquivo.close(); // Fecha o arquivo.
29     return grafo; // Retorna a matriz de adjacência.
30 }
31
32 vector<vector<int>> cliquesMaximasGloab; // Vetor global para armazenar cliques máximas.
33 vector<int> cliqueMaximaGloab; // Vetor para armazenar a maior clique máxima.
34
35 // Função que utiliza OpenMP para encontrar cliques máximas.
36 void cliquesMaximasOpenMP(const vector<vector<int>>& grafo, int qntVertices) {
37     #pragma omp parallel // Inicia uma região paralela OpenMP.
38     {
39         vector<bool> visitados(qntVertices, false); // Vetor de visitados.
40         vector<int> cliqueAtual; // Vetor para armazenar a clique atual.
41
42         #pragma omp for nowait // Paraleliza o loop com OpenMP.
43         for (int i = 0; i < qntVertices; ++i) {
44             // Chama a função recursiva para cada vértice.
45             encontrarCliqueMaximoRecursivo(grafo, cliqueAtual, visitados, i);
46         }
47     }
48
49     // Ordena e remove duplicatas das cliques máximas.
50     sort(cliquesMaximasGloab.begin(), cliquesMaximasGloab.end(), [](const auto& a, const auto& b) {
51         return a.size() > b.size();
52     });
53
54     cliquesMaximasGloab.erase(unique(cliquesMaximasGloab.begin(), cliquesMaximasGloab.end()), cliquesMaximasGloab.end());
55 }
56
57 // Função para verificar se um nó pode ser adicionado a uma clique.
58 bool adicionarNoClique(const vector<vector<int>>& grafo, const vector<int>& cliqueAtual, int vizinho) {
59     for (auto j : cliqueAtual) {
60         if (grafo[vizinho][j] == 0) {
61             return false;
62         }
63     }
64     return true;
65 }
66
67 // Função para encontrar todos os possíveis vizinhos para a clique atual.
68 void possiveisVizinhos(const vector<vector<int>>& grafo, vector<int>& possiveisVertices, const vector<int>& cliqueAtual, const vector<bool>& visitados) {
69     for (int i = 0; i < grafo.size(); ++i) {
70         if (!visitados[i] && adicionarNoClique(grafo, cliqueAtual, i)) {
71             possiveisVertices.emplace_back(i);
72         }
73     }
74 }
75
76 // Função para atualizar o vetor global de cliques máximas.
77 void atualizarClique(const vector<int>& cliqueAtual) {
78     #pragma omp critical // Região crítica para garantir a integridade dos dados.
79     {
80         cliquesMaximasGloab.push_back(cliqueAtual);
81         if (cliqueAtual.size() > cliqueMaximaGloab.size()) {
82             cliqueMaximaGloab = cliqueAtual;
83         }
84     }
85 }

```

Figura 18 - Código da abordagem utilizando OpenMP (parte 1)

```

1 // Função recursiva para encontrar a clique máxima.
2 void encontrarCliqueMaximoRecursivo(const vector<vector<int>>& grafo, vector<int>& cliqueAtual, vector<bool>& visitados, int vertice) {
3     cliqueAtual.push_back(vertice); // Adiciona o vértice à clique atual.
4     visitados[vertice] = true; // Marca o vértice como visitado.
5
6     vector<int> possiveisVertices; // Vetor para armazenar os possíveis vizinhos.
7     possiveisVizinhos(grafo, possiveisVertices, cliqueAtual, visitados); // Encontra possíveis vizinhos.
8
9     for (auto vizinho : possiveisVertices) {
10        // Chamada recursiva para cada vizinho possível.
11        encontrarCliqueMaximoRecursivo(grafo, cliqueAtual, visitados, vizinho);
12    }
13
14    if (possiveisVertices.empty() && cliqueAtual.size() > 1) {
15        atualizarCliques(cliqueAtual); // Atualiza as cliques maximas.
16    }
17
18    cliqueAtual.pop_back(); // Remove o último vértice da clique atual.
19    visitados[vertice] = false; // Marca o vértice como não visitado.
20 }
21
22 int main(int argc, char* argv[]) {
23     // Verifica se o nome do arquivo foi passado como argumento.
24     if (argc < 2) {
25         cerr << "Uso: " << argv[0] << " <nome_do_arquivo_de_entrada>" << endl;
26         return 1;
27     }
28
29     string nome_arquivo_entrada = argv[1]; // Obtém o nome do arquivo de entrada.
30     int qntVertices;
31     vector<vector<int>> grafo = LerGrafo(nome_arquivo_entrada, qntVertices); // Lê o grafo do arquivo.
32
33     auto start_time = chrono::high_resolution_clock::now(); // Inicia a contagem do tempo.
34     cliquesMaximasOpenMP(grafo, qntVertices); // Encontra cliques maximas com OpenMP.
35
36     // Encerra a contagem do tempo.
37     auto end_time = chrono::high_resolution_clock::now();
38     chrono::duration<double> elapsed = end_time - start_time;
39
40     // Imprime as cliques maximas encontradas.
41     for (const auto& clique : cliquesMaximasGlobal) {
42         cout << "[";
43         for (int i = 0; i < clique.size(); ++i) {
44             cout << clique[i] + 1;
45             if (i < clique.size() - 1) cout << ", ";
46         }
47         cout << "]" << endl;
48     }
49
50     // Imprime a maior clique máxima encontrada.
51     cout << "Clique máxima encontrada (OpenMP): [";
52     for (int i = 0; i < cliqueMaximaGlobal.size(); ++i) {
53         cout << cliqueMaximaGlobal[i] + 1;
54         if (i < cliqueMaximaGlobal.size() - 1) cout << ", ";
55     }
56     cout << "]" << endl;
57     cout << "Tempo de execução: " << elapsed.count() << "s" << endl;
58
59     return 0;
60 }

```

Figura 19 - Código da abordagem utilizando OpenMP (parte 2)

## Paralelização e Distribuição do Processamento com MPI

A paralelização e distribuição do processamento de um problema de busca de cliques em um grafo usando a biblioteca MPI (Message Passing Interface) envolve dividir o grafo em subconjuntos menores, atribuindo cada subconjunto a diferentes processadores (ou nós de computação) para processamento paralelo.

MPI é um padrão de comunicação entre processos, usado principalmente em computação paralela em sistemas distribuídos. Ele permite a troca de mensagens (dados) entre processos, que podem estar executando no mesmo computador ou em computadores diferentes conectados por uma rede.

A implementação de uma solução de busca de cliques em grafos usando MPI permite explorar recursos de computação paralela e distribuída para processar grafos grandes de forma mais eficiente. O sucesso desta abordagem depende da eficácia na divisão do



grafo, no balanceamento de carga entre os processadores e na gestão eficiente da comunicação e sincronização entre os processos.

A implementação MPI foi integralmente realizada utilizando o cluster do laboratório de redes e supercomputação do Insper, tal como solicitado pela rubrica deixada pelos professores da disciplina.

No contexto do projeto, foi possível obter uma diferença significativa em relação ao tempo ao rodar utilizando MPI, tal como pode ser conferido pelos gráficos a seguir:

Tempos de Execução para Diferentes Implementações (Primeiro Conjunto de Dados)

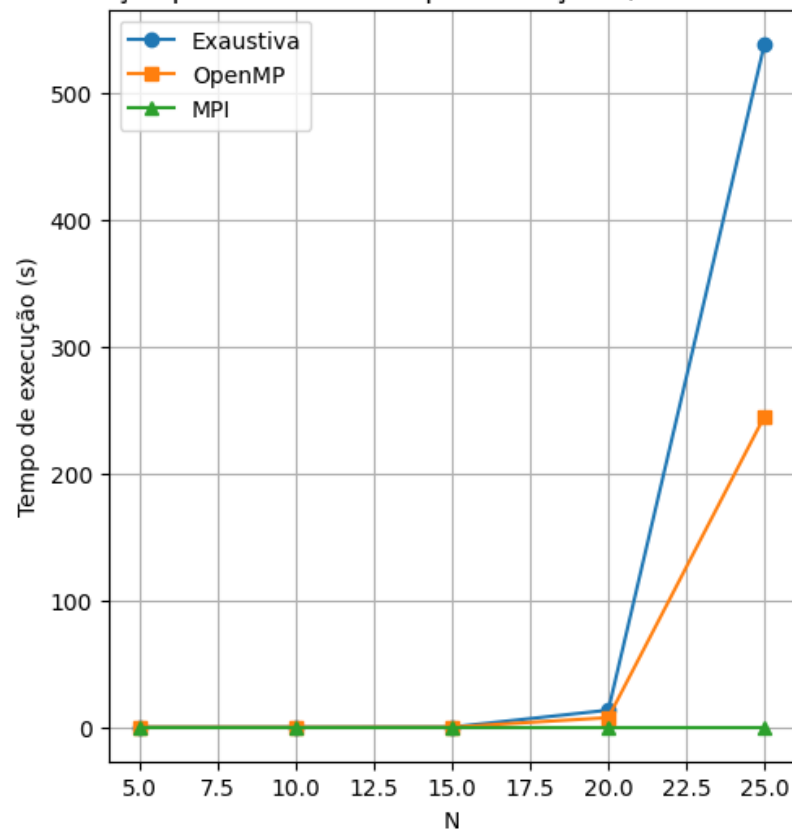


Figura 20 - Comparação dos tempos de execução entre as abordagens Exaustiva/Heurística, OpenMP e MPI de acordo com a variação da quantidade dos nós (N) entre 5 até 25

## Tempos de Execução para Diferentes Implementações (Segundo Conjunto de Dados)

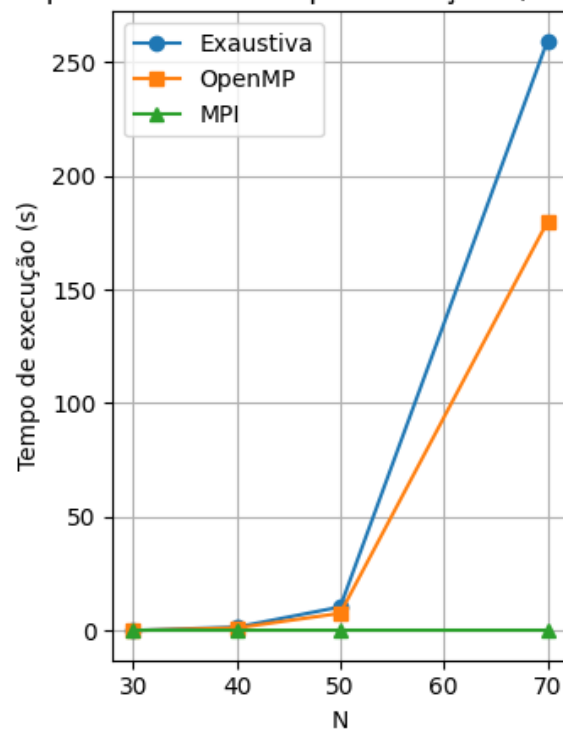


Figura 21 - Comparação dos tempos de execução entre as abordagens Exaustiva/Heurística, OpenMP e MPI de acordo com a variação da quantidade dos nós (N) entre 30 até 70

## Speed Up

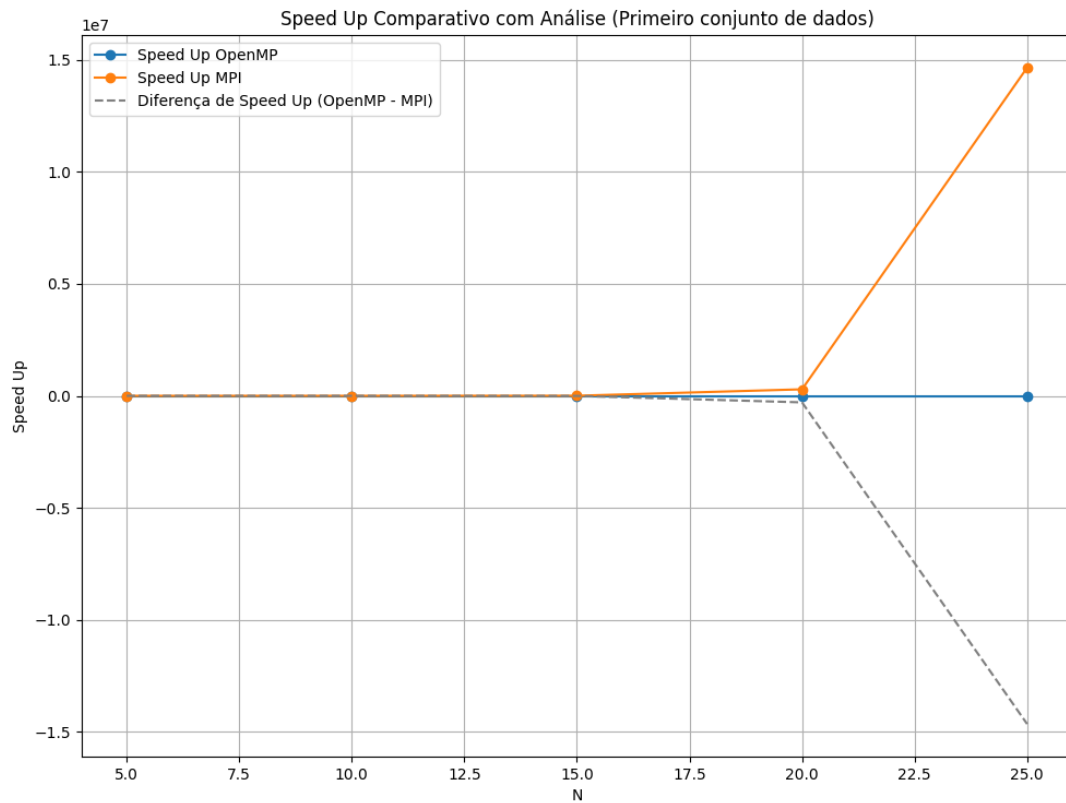


Figura 22 - Speed Up Comparativo das abordagens OpenMP e MPI com quantidade de nós (N) entre 5 até 25

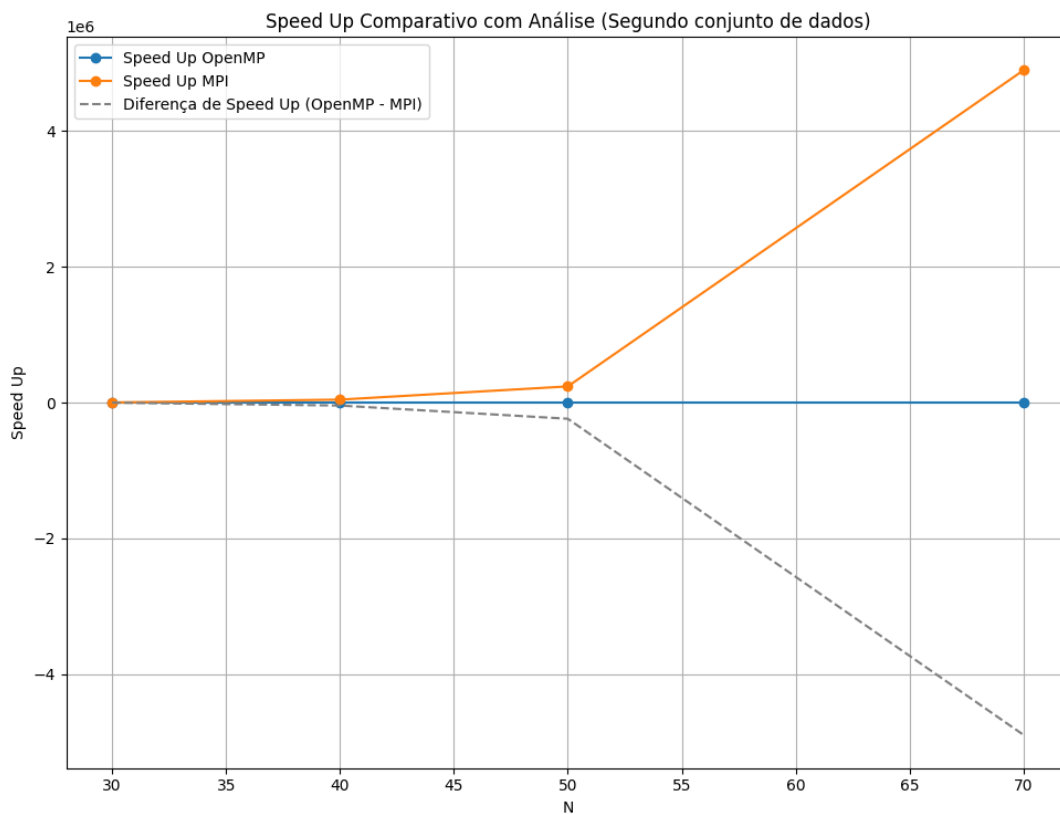


Figura 23 - Speed Up Comparativo das abordagens OpenMP e MPI com quantidade de nós (N) entre 30 até 70

## Comparação entre OpenMP e MPI

**Diferença de *Speed Up* (OpenMP - MPI):** A linha pontilhada nos gráficos mostra a diferença entre o *speed up* proporcionado pelo OpenMP e pelo MPI. **Se esta linha está acima de 0, significa que o OpenMP tem um *speed up* maior que o MPI; se está abaixo de 0, o MPI é mais eficiente.**

**Análise dos Conjuntos de Dados:** O *speed up* do MPI é significativamente maior do que o do OpenMP, especialmente para valores maiores de N. Isto sugere que, **para este conjunto de tarefas, a implementação MPI é mais eficiente na redução do tempo de execução.**

### Algumas conclusões:

**Eficiência do MPI:** Em ambos os conjuntos de dados, o MPI tende a ser mais eficiente do que o OpenMP, indicando que para estas tarefas específicas, a comunicação e a divisão de trabalho implementadas no MPI são mais adequadas para acelerar a execução.

**Escalabilidade do Paralelismo:** A eficácia do paralelismo (tanto em OpenMP quanto em MPI) aumenta com o tamanho do problema, o que é um indicativo da importância de abordagens paralelas em problemas de grande escala.

Estes insights podem ser cruciais ao decidir qual abordagem paralela adotar em um determinado contexto de aplicação.

## Algumas evidências das cliques

O projeto foi integralmente rodado utilizando o cluster da disciplina, via SSH. As questões envolvendo MPI tiveram os jobs submetidos no cluster e rodaram sem problemas. Algumas evidências dos outputs recebidos (e utilizados para a confecção dos gráficos acima) podem ser vistas a seguir:

```
upercomputação\projeto-novo-repositorio\projeto-supercomp-novo-repo> python .\output.py
Cliques máximas encontradas:
['5', '10', '3', '8', '6']
['5', '10', '7']
['5', '4']
['8', '2', '3']
['1', '9', '3', '2']
['1', '9', '3', '6']
['1', '9', '7']
['1', '10', '3', '6']
['1', '10', '7']
['1', '4', '2']
Clique máxima encontrada: ['5', '10', '3', '8', '6']
```

Figura 24 - Clique máxima encontrada para  $n = 10$

```
[10, 8, 6, 5]
[10, 8, 6, 5, 3]
Clique máxima encontrada:[3, 5, 6, 8, 10]
Tempo de execução: 0.00106659s
```

Figura 25 - Clique máxima encontrada e tempo de execução para  $n = 10$  na abordagem Exaustiva/Heurística

```
Clique máxima encontrada (OpenMP): [6, 3, 5, 8, 10]
Tempo de execução: 0.000924805s
```

Figura 26 - Clique máxima encontrada e tempo de execução para  $n = 10$  na abordagem OpenMP

```
Clique máxima (MPI): 2 4 5 7 9
Tempo de execução: 2.646e-05s
```

Figura 27 - Clique máxima encontrada e tempo de execução para  $n = 10$  na abordagem MPI

```
['4', '7', '3', '11', '12', '2']
['4', '7', '3', '5', '14', '12']
['4', '7', '3', '5', '14', '6']
['4', '7', '3', '5', '2', '12']
['4', '7', '8', '5', '12', '2']
['4', '7', '8', '5', '6']
['4', '15', '2', '9', '10', '3']
['4', '15', '2', '8']
Clique máxima encontrada: ['4', '7', '3', '10', '9', '2']
```

Figura 28 - Clique máxima encontrada para  $n = 15$

```
[15, 10, 9, 4, 3, 2]
Clique máxima encontrada: [2, 3, 4, 5, 7, 12]
Tempo de execução: 0.117804s
```

Figura 29 - Clique máxima encontrada e tempo de execução para  $n = 15$  na abordagem Exaustiva/Heurística

```
Clique máxima encontrada (OpenMP): [9, 2, 3, 4, 7, 10]
Tempo de execução: 0.0499834s
```

Figura 30 - Clique máxima encontrada e tempo de execução para  $n = 15$  na abordagem OpenMP

```
Clique máxima (MPI): 1 2 3 8 9 14
Tempo de execução: 3.0694e-05s
```

Figura 31 - Clique máxima encontrada e tempo de execução para  $n = 15$  na abordagem MPI

```
['14', '11', '19', '4', '13', '10', '15']
Clique máxima encontrada: ['20', '11', '2', '18', '6', '17', '15', '8']
```

Figura 32 - Clique máxima encontrada para  $n = 20$

```
Clique máxima encontrada: [1, 2, 6, 10, 11, 16, 17, 20]
Tempo de execução: 13.3899s
```

Figura 33 - Clique máxima encontrada e tempo de execução para  $n = 20$  na abordagem Exaustiva/Heurística

```
Clique máxima encontrada (OpenMP): [11, 1, 2, 6, 10, 16, 17, 20]
Tempo de execução: 7.49242s
```

Figura 34 - Clique máxima encontrada e tempo de execução para  $n = 20$  na abordagem OpenMP

```
Clique máxima (MPI): 1 6 10 15 16 17 18 19
Tempo de execução: 4.5942e-05s
```

Figura 35 - Clique máxima encontrada e tempo de execução para  $n = 20$  na abordagem MPI



```
[15, 3, 28, 49]
Clique máxima encontrada: ['41', '30', '7', '44', '17', '21', '10']
PS C:\Users\liste\OneDrive - Insper - Instituto de Ensino e Pesquisa\7
```

Figura 36 - Clique máxima encontrada para  $n = 50$

```
[30, 49, 40, 37, 30, 4]
Clique máxima encontrada: [7, 10, 17, 21, 30, 41, 44]
Tempo de execução: 1.37692s
```

Figura 37 - Clique máxima encontrada e tempo de execução para  $n = 50$  na abordagem Exaustiva/Heurística

```
[11, 3, 43]
Clique máxima encontrada (OpenMP): [30, 7, 10, 17, 21, 41, 44]
Tempo de execução: 1.07188s
Clique máxima encontrada (OpenMP): [30, 7, 10, 17, 21, 41, 44]
```

Figura 38 - Clique máxima encontrada e tempo de execução para  $n = 50$  na abordagem OpenMP

```
Clique máxima (MPI): 3 8 29 36 47 48 49
Tempo de execução: 5.3106e-05s
```

Figura 39 - Clique máxima encontrada e tempo de execução para  $n = 50$  na abordagem MPI

```

1 // Projeto de Supercomputação | Engenharia de Computação - Insper
2 // Professores: André Filipe e Michel Fornaciali
3
4 #include <mpi.h>           // Inclui a biblioteca MPI para programação paralela em clusters.
5 #include <fstream>         // Inclui a biblioteca para manipulação de arquivos.
6 #include <iostream>        // Inclui a biblioteca padrão de entrada e saída.
7 #include <vector>          // Inclui a biblioteca para usar vetores dinâmicos.
8 #include <chrono>          // Inclui a biblioteca para medir o tempo.
9 #include <algorithm>       // Inclui a biblioteca para operações de algoritmo como sort.
10
11 using namespace std;
12
13 // Função para ler um grafo de um arquivo e retornar sua matriz de adjacência.
14 vector<vector<int>> LerGrafo(const string& fileName, int& qntVertices) {
15     ifstream entrada(fileName); // Abre o arquivo para leitura.
16     int totalArestas;
17     entrada >> qntVertices >> totalArestas; // Lê a quantidade de vértices e arestas.
18
19     vector<vector<int>> matrizAdjacencia(qntVertices, vector<int>(qntVertices, 0)); // Cria matriz de adjacência.
20
21     // Lê as arestas e preenche a matriz de adjacência.
22     for (int aresta = 0; aresta < totalArestas; ++aresta) {
23         int vertice1, vertice2;
24         entrada >> vertice1 >> vertice2; // Lê um par de vértices conectados.
25         matrizAdjacencia[vertice1-1][vertice2-1] = 1;
26         matrizAdjacencia[vertice2-1][vertice1-1] = 1;
27     }
28     entrada.close(); // Fecha o arquivo.
29     return matrizAdjacencia; // Retorna a matriz de adjacência.
30 }
31
32 // Função para buscar a clique máxima em uma parte do grafo.
33 vector<int> buscarCliqueMaxima(const vector<vector<int>>& matrizAdjacencia, int qntVertices, int inicio, int fim) {
34     vector<int> cliqueMax; // Armazena a maior clique encontrada.
35     vector<int> possiveisCandidatos; // Armazena os candidatos para serem adicionados à clique.
36
37     // Inicializa a lista de possíveis candidatos.
38     for(int i = inicio; i < fim; ++i) {
39         possiveisCandidatos.push_back(i);
40     }
41
42     // Busca pela clique máxima.
43     while(!possiveisCandidatos.empty()) {
44         int verticeAtual = possiveisCandidatos.back();
45         possiveisCandidatos.pop_back();
46
47         bool adicionar = true; // Flag para verificar se o vértice atual pode ser adicionado à clique.
48
49         // Verifica se o vértice atual é adjacente a todos na clique.
50         for(int adjacente : cliqueMax) {
51             if(matrizAdjacencia[adjacente][verticeAtual] == 0) {
52                 adicionar = false;
53                 break;
54             }
55         }
56
57         // Adiciona o vértice à clique e atualiza os candidatos.
58         if(adicionar) {
59             cliqueMax.push_back(verticeAtual);
60             vector<int> novosCandidatos;
61
62             // Atualiza a lista de candidatos para incluir apenas aqueles que são adjacentes a todos na clique.
63             for(int candidato : possiveisCandidatos) {
64                 bool adjacenteATodos = true;
65
66                 for(int membroClique : cliqueMax) {
67                     if(matrizAdjacencia[candidato][membroClique] == 0) {
68                         adjacenteATodos = false;
69                         break;
70                     }
71                 }
72
73                 if(adjacenteATodos) {
74                     novosCandidatos.push_back(candidato);
75                 }
76             }
77
78             possiveisCandidatos = novosCandidatos;
79         }
80     }
81
82     return cliqueMax; // Retorna a maior clique encontrada.
83 }

```

Figura 40 - Código da abordagem utilizando MPI (parte 1)

```

1 int main(int argc, char* argv[]) {
2     MPI_Init(&argc, &argv); // Inicializa o ambiente MPI.
3
4     int idProcesso, qntProcessos;
5     MPI_Comm_rank(MPI_COMM_WORLD, &idProcesso); // Obtém o ID do processo atual.
6     MPI_Comm_size(MPI_COMM_WORLD, &qntProcessos); // Obtém o número total de processos.
7
8     int qntVertices;
9     vector<vector<int>> matrizAdjacencia = LerGrafo("grafo_output.txt", qntVertices); // Lê o grafo do arquivo.
10
11     // Divide os vértices entre os processos.
12     int verticesPorProcesso = qntVertices/qntProcessos;
13     int inicio = idProcesso * verticesPorProcesso;
14     int fim = (idProcesso == qntProcessos - 1) ? qntVertices : inicio + verticesPorProcesso;
15
16     // Cada processo busca pela clique máxima na sua parte do grafo.
17     chrono::high_resolution_clock::time_point tempoInicio;
18     if(idProcesso == 0) {
19         tempoInicio = chrono::high_resolution_clock::now(); // Inicia a contagem do tempo no processo principal.
20     }
21
22     vector<int> cliqueMaxLocal = buscarCliqueMaxima(matrizAdjacencia, qntVertices, inicio, fim);
23
24     // Coleta os tamanhos das cliques maximas de todos os processos.
25     int tamanhoCliqueMaxLocal = cliqueMaxLocal.size();
26     vector<int> tamanhosCliqueMax(qntProcessos);
27     MPI_Gather(&tamanhoCliqueMaxLocal, 1, MPI_INT, tamanhosCliqueMax.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);
28
29     // O processo principal determina a maior clique.
30     if(idProcesso == 0) {
31         int indiceMaximo = distance(tamanhosCliqueMax.begin(), max_element(tamanhosCliqueMax.begin(), tamanhosCliqueMax.end()));
32
33         if(indiceMaximo != 0) {
34             cliqueMaxLocal.resize(tamanhosCliqueMax[indiceMaximo]);
35             MPI_Recv(cliqueMaxLocal.data(), tamanhosCliqueMax[indiceMaximo], MPI_INT, indiceMaximo, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36         }
37     } else if(tamanhoCliqueMaxLocal == tamanhosCliqueMax[indiceMaximo]) {
38         MPI_Send(cliqueMaxLocal.data(), tamanhoCliqueMaxLocal, MPI_INT, 0, 0, MPI_COMM_WORLD);
39     }
40
41     // O processo principal imprime o resultado e o tempo de execução.
42     if(idProcesso == 0) {
43         auto tempoFim = chrono::high_resolution_clock::now();
44         chrono::duration<double> tempoDecorrido = tempoFim - tempoInicio;
45
46         sort(cliqueMaxLocal.begin(), cliqueMaxLocal.end());
47         cout << "Clique máxima (MPI): ";
48         for(int vertice : cliqueMaxLocal) {
49             cout << vertice << " ";
50         }
51         cout << endl;
52
53         cout << "Tempo de execução: " << tempoDecorrido.count() << "s" << endl;
54     }
55
56     MPI_Finalize(); // Finaliza o ambiente MPI.
57     return 0;
58 }

```

Figura 41 - Código da abordagem utilizando OpenMP (parte 2)

## Conclusão

Ao longo do desenvolvimento do projeto, ficou claro que a análise e a resolução de problemas complexos em redes sociais, **especialmente aqueles de natureza computacionalmente desafiadora**, vão muito além da simples aplicação de ferramentas e tecnologias avançadas.

Primeiramente, o projeto enfatizou a relevância de selecionar algoritmos e heurísticas otimizados. Ao explorar diferentes métodos, como a heurística gulosa e a busca exaustiva, **ficou claro que a escolha do algoritmo certo pode impactar significativamente a análise. A capacidade de otimizar constantemente essas soluções**, adaptando-as às necessidades específicas e à natureza dos dados de redes sociais, provou ser fundamental.

A paralelização e a distribuição do processamento surgiram como conceitos muito importantes ao longo da disciplina de Supercomputação e, conseqüentemente, ao longo do projeto. Ao implementar soluções que utilizam múltiplas threads e distribuição de processamento em vários núcleos ou máquinas (como demonstrado com OpenMP e

MPI), o projeto demonstrou como a divisão inteligente de tarefas e a execução paralela podem acelerar significativamente a análise de redes sociais.

A aplicação desses conceitos e estratégias em plataformas de supercomputação, como clusters e supercomputadores (os quais tivemos um contato aprofundado ao longo de todo o sétimo semestre no Insper), abriu novos horizontes para o processamento de dados em grande escala. **Essas plataformas ofereceram recursos computacionais avançados, permitindo uma exploração mais profunda, abrangente e rápida de redes sociais complexas.**

Em resumo, este projeto forneceu insights valiosos sobre a análise de redes sociais e destacou a **importância de combinar estratégias computacionais eficientes, seleção de algoritmos otimizados, alinhamento com plataformas de hardware adequadas e técnicas avançadas de paralelização e distribuição.** Essa abordagem integrada é essencial para enfrentar e resolver problemas computacionais "difíceis", permitindo uma compreensão mais profunda e um processamento mais eficiente dos dados em redes sociais.

## Referências:

A "Hands-on" Introduction to OpenMP (Part 1) | Tim Mattson, Intel. Disponível em: <[www.youtube.com/watch?v=pRtTIW9-Nr0](https://www.youtube.com/watch?v=pRtTIW9-Nr0)>. Acesso em 18 de setembro de 2023.

Aula 8 - Programação Paralela (OpenMP - Pt. 1). Disponível em: <<https://www.youtube.com/watch?v=nchK-7WfPQY>>. Acesso em: 18 de setembro de 2023.

Aula 9 - Programação Paralela (OpenMP - Pt. 2). Disponível em: <[https://www.youtube.com/watch?v=2F\\_GpRQ2mvl](https://www.youtube.com/watch?v=2F_GpRQ2mvl)>. Acesso em: 18 de setembro de 2023.

Aula 10 - Programação Paralela (OpenMP - Pt. 3). Disponível em: <<https://www.youtube.com/watch?v=PFW3MFiT9MA>>. Acesso em: 19 de setembro de 2023.

Graph Theory. Britannica. Disponível em: <<https://www.britannica.com/topic/graph-theory>>. Acesso em 8 de outubro de 2023.

Introduction to Graph Theory: A Computer Science Perspective. Disponível em: <<https://www.youtube.com/watch?v=LFKZLXVO-Dg>>. Acesso em 8 de outubro de 2023.

Social Network Analysis: From Graph Theory to Applications with Python. Disponível em: <<https://towardsdatascience.com/social-network-analysis-from-theory-to-applications-with-python-d12e9a34c2c7>>. Acesso em 7 de dezembro de 2023.

Supercomputação – Insper 2023/2. Disponível em: <<https://insper.github.io/supercomp/>>.

Open MPI Documentation. Disponível em: <<https://www.open-mpi.org/doc/>>.

OpenMP Reference Guides. Disponível em: <<https://www.openmp.org/resources/refguides/>>.

Supercomp. Disponível em: <<https://github.com/Insper/supercomp>>.

C++ reference. Disponível em: <<https://en.cppreference.com/w/>>.

É válido pontuar também como referência para a realização do projeto as aulas, conversas e tira-dúvidas com os professores da disciplina, além de também terem sido realizadas diversas discussões junto a outros colegas de sala durante a elaboração das abordagens aqui apresentadas. Além disso, como complementos à realização deste projeto, também foram utilizadas ferramentas como o ChatGPT (Oct 17 version, OpenAI 2023) e o Google Colaboratory.