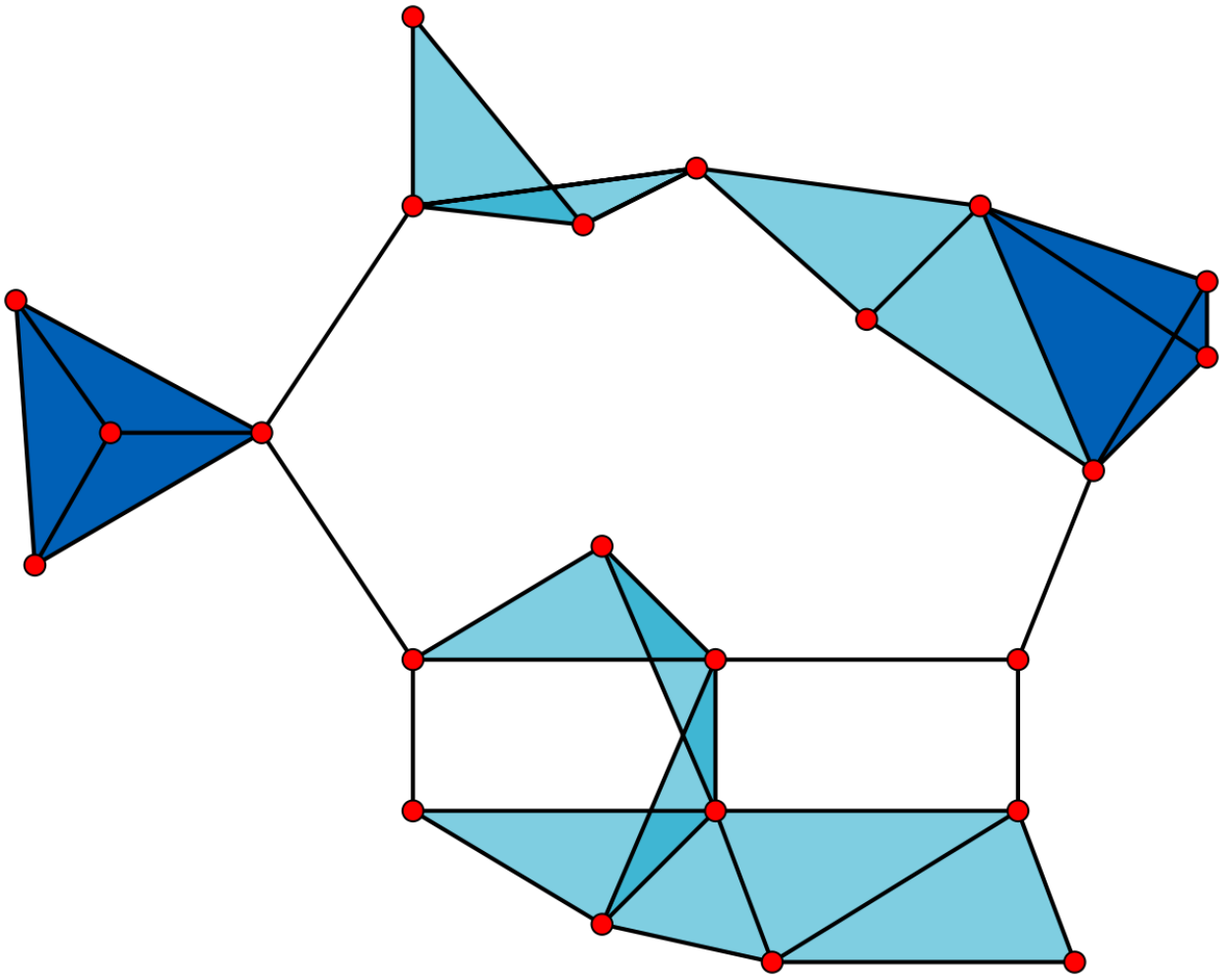
Projeto SuperComputação 2023.2

Análise de Redes Sociais: encontrando a clique máxima em um grafo.



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

A idéia 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 conectado por uma aresta.

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.

SUA TAREFA: Encontrar a clique máxima em um grafo.

Seu programa deve receber um grafo a partir de um input de texto (abaixo você vai encontrar o código gerador do input). A partir da leitura do arquivo, você deve armazenar o grafo computacionalmente (matriz de adjacência, por exemplo). E com isso, você deverá executar três implementações:

1. **Abordagem Exaustiva**

A exaustão é uma abordagem que seleciona iterativamente os vértices para formar um clique, geralmente começando com um vértice e adicionando outros que tenham o maior número de vizinhos já na clique. Aqui está um pseudo-código simplificado para detectar cliques em um grafo usando essa abordagem. ATENÇÃO: esse pseudo-codigo **não** é a solução completa dessa abordagem. Você pode se inspirar nele para compreender como resolver o problema, mas é parte de sua tarefa desenvolver a solução.

```
Função EncontrarCliqueMaxima(grafo, numVertices)
    cliqueMaxima = ListaVazia()
    candidatos = ListaDeNós()  # Inicialmente, todos os nós são candidatos

    Para cada i de 0 até numVertices - 1 Faça
        Adicione i à lista de candidatos

    Enquanto candidatos não estiver vazia Faça
        v = Último elemento de candidatos
        Remova o último elemento de candidatos

        podeAdicionar = Verdadeiro

        Para cada u em cliqueMaxima Faça
            Se grafo[u][v] == 0 Então
                podeAdicionar = Falso
                Pare o loop
            Fim Se
        Fim Para

        Se podeAdicionar for Verdadeiro Então
            Adicione v a cliqueMaxima
            novosCandidatos = ListaDeNós()

            Para cada u em candidatos Faça
                adjacenteATodos = Verdadeiro

                Para cada c em cliqueMaxima Faça
                    Se grafo[u][c] == 0 Então
                        adjacenteATodos = Falso
                        Pare o loop
```

```
                    Fim Se
                Fim Para

                Se adjacenteATodos for Verdadeiro Então
                    Adicione u a novosCandidatos
                Fim Se
            Fim Para

            candidatos = novosCandidatos
        Fim Se
    Fim Enquanto

    Retorne cliqueMaxima
Fim Função
```

Implemente o código em C++.

Aproveite para pensar se é possível adotar alguma heurística. Por exemplo, ordenar os nós em função do grau de adjacência, ajuda?

Sua segunda tarefa: Até qual tamanho de problema você conseguiu obter um resultado em tempo hábil (aprox. 15 min)? Você deve apresentar um pseudo-codigo de uma heurística para otimizar essa tarefa exaustiva. É possível implementar alguma poda? Algum critério que evite calcular um nó, dado que você já descobriu uma clique maior?

1. **Implementação com Threads OpenMP**

A implementação de uma solução com OpenMP para encontrar cliques em um grafo usando a heurística gulosa envolve a paralelização das iterações do algoritmo em diferentes threads. O OpenMP simplifica a criação de threads e a coordenação entre elas.

Nessa implementação você deve apresentar o código-fonte modificado, justificar a alteração feita, e mostrar também o speed up obtido. Faça uso de gráficos e tabelas para mostrar que sua implementação em openMP valeu a pena. Observe que mesmo assim você não vai conseguir resolver exaustivamente o problema, mas o tamanho do grafo será maior, e o tempo de processamento para um mesmo tamanho deve ser preferencialmente menor.

1. **Paralelização e Distribuição do Processamento com MPI**

Nesta implementação, você deve dividir o problema em várias partes e distribuí-las para diferentes processadores usando a biblioteca MPI (*Message Passing Interface*). Cada processador será responsável por encontrar cliques em uma parte do grafo, e os resultados serão combinados no final para encontrar todas as cliques no grafo.

Como apresentar seus resultados:

- Você deverá focar em comparar suas implementações em relação ao speedup. Aumente o tamanho do grafo e das arestas, e busque determinar a clique máxima.

Códigos-fonte de apoio

1. Código-fonte de geração do grafo (em Python)

- Codigo disponível em input.py

1. Trecho de código-fonte para leitura do grafo e armazenamento como matriz de adjacência

- Codigo disponível em grafo.cpp

1. Verificar (em Python) se seu programa encontrou a clique correta. Para isso, use a implementação abaixo, ela já está adaptada para ler nosso arquivo de input, ignorando a primeira linha.

- Codigo disponível em verificacao.py

Como Compilar

exaustiva.cpp (heuristica da busca_exasustiva)

- compilação

```
g++ -o exaustiva{id} exaustiva.cpp grafo.cpp -std=c++11
```

- gerando executavel

```
./exaustiva{id} grafo.txt
```

Se quiser pode colocar numeros ao lado do nome exaustiva pra indicar o numero de vertices do grafo
Essa aqui não precisa para as comparações faremos a comparação a partir da heuristica

heuristica.cpp (heuristica da busca_exasustiva)

- compilação

```
# no colab
g++ -o heuristica{id} heuristica.cpp grafo.cpp -std=c++11
```

```
# no cluster
g++ heuristica.cpp -o heuristica{id}
```

- gerando executavel

```
./heuristica{id} grafo.txt
```

Se quiser pode colocar numeros ao lado do nome heuristica pra indicar o numero de vertices (id) do grafo

threads.cpp

- compilação

```
g++ threads.cpp -fopenmp -o threads{id}
```

- gerando executavel

```
./threads{id} grafo.txt
```

- Onde o id é o número de vértices do grafo que você pode alterar no input.py

paralelizacao.cpp

- a compilação e execucao serão feitas no cluster de cada grupo da disciplina
- compilação

```
mpic++ paralelizacao.cpp -o mpi{id}
```

- submetendo job

```
sbatch paralel.slurm
```

- pegando output

```
cat slurm-{jobid}.out
```

Não esquecer de criar um arquivo.slurm e modificar o executavel como no exemplo a seguir:

```
#!/bin/bash
#SBATCH --job-name=projeto_gabriel
#SBATCH --nodes=1
#SBATCH --partition=express
#SBATCH --mem=500M
echo Output do Job $SLURM_JOB_ID
./executavel
```

Para os três casos serão discutidos os resultados a seguir na seção de resultados para diferentes de grafos.

O texto acima mostra como gerar o executavel MPI e todos foram feitos dentro do cluster, infelizmente não peguei os executaveis via batch para deixar como arquivos no github do projeto.

Resultados

heuristica (heuristica da busca_exaustiva)

- Grafo de 5 vértices:

```
Clique máxima encontrada: ['1', '5', '2', '3']

Clique máxima encontrada:[1, 2, 3, 5]
```

- Grafo de 10 vértices:

```
Clique máxima encontrada: ['4', '9', '7', '10', '3']

Clique máxima encontrada:[2, 3, 4, 7, 10]
```

- Grafo de 15 vértices:

```
Clique máxima encontrada: ['9', '10', '1', '14', '3', '11', '5']

Clique máxima encontrada:[1, 2, 5, 10, 11, 12, 14]
```

- Grafo de 20 vértices

```
Clique máxima encontrada: ['13', '8', '1', '14', '15', '18']

Clique máxima encontrada:[1, 2, 4, 8, 14, 18]
```

- Grafo de 25 vértices

```
Clique máxima encontrada: ['25', '6', '19', '23', '9', '7', '10', '1']

Clique máxima encontrada:[1, 6, 7, 9, 10, 19, 23, 25]
```

- grafo de 30 vértices (houve mudança nas arestas a partir daqui)

```
Clique máxima encontrada: ['10', '19', '22', '20', '7', '13']
```

```
Clique máxima encontrada:[7, 10, 13, 19, 20, 22]
Tempo de execução: 0.0216389s
```

- grafo de 35 vértices

```
Clique máxima encontrada: ['31', '12', '20', '7', '25', '2', '1']
```

```
Clique máxima encontrada:[1, 2, 7, 12, 20, 25, 31]
Tempo de execução: 1.06683s
```

- grafo 50 vértices

```
Clique máxima encontrada: ['10', '36', '16', '15', '38', '34', '42', '7']
```

```
Clique máxima encontrada:[1, 4, 16, 19, 31, 36, 41, 43]
Tempo de execução: 18.2468s
```

Para cada tamanho de grafo as primeiras imagens são o arquivo de verificação.py (arquivo dado no projeto que verifica o tamanho e as cliques máximas do grafo) e as imagens subquentes são o resultado do arquivo heuristica.cpp

THREADS (OPENMP)

- Grafo de 5 vértices:

```
Clique máxima encontrada: [4, 1, 2]
```

- Grafo de 10 vértices:

```
Clique máxima encontrada: [6, 1, 3, 5, 7, 8]
```

- Grafo de 15 vértices:

```
Clique máxima encontrada: [10, 3, 6, 7, 8, 14]
```

- Grafo de 20 vértices:

```
Clique máxima encontrada: [2, 4, 5, 6, 7, 8, 15, 18, 20]
```

- Grafo de 25 vértices:

```
Clique máxima encontrada: [1, 2, 4, 5, 8, 12, 23, 24]
```

- Grafo de 50 vértices:

```
Clique máxima encontrada: [1, 4, 16, 19, 31, 36, 41, 43]
Tempo de execução: 12.1909s
```

MPI-paralelizacao

Para cada tamanho de grafo com os vértices a primeira imagem e a clique máxima da heuristica e a segunda imagem é a paralelização MPI com o tempo da execução

- Grafo de 5 vértices:

```
Clique máxima encontrada:[1, 2, 3, 4]
Tempo de execução: 0.000183143s
```

```
Clique máxima: 0 2 3 4
Tempo de execução: 2.5065e-05s
```

- Grafo de 10 vértices:

```
Clique máxima encontrada:[1, 2, 3, 6, 9, 10]
Tempo de execução: 0.024659s
```

```
Clique máxima: 0 4 5 6 8 9
Tempo de execução: 2.7099e-05s
```

- Grafo de 15 vértices:

```
Clique máxima encontrada:[1, 3, 8, 9, 12, 13]
Tempo de execução: 0.0167494s
```

```
Clique máxima: 9 12 13 14
Tempo de execução: 2.3233e-05s
```

- Grafo de 20 vértices

```
Clique máxima encontrada:[1, 3, 6, 7, 10, 11, 14, 16, 19]
Tempo de execução: 52.9513s
```

```
Clique máxima: 0 6 13 15 16 18 19
Tempo de execução: 3.9117e-05s
```

- Grafo de 25 vértices

```
Clique máxima encontrada:[1, 3, 6, 9, 12, 13, 14, 21, 25]
Tempo de execução: 33.8068s
```

```
Clique máxima: 8 16 19 20 22 24
Tempo de execução: 3.846e-05s
```

Cliques máximas MPI a partir de grafos com 30 vertices

- grafo 30 vértices

```
Clique máxima: 3 12 13 20 23 26 29
Tempo de execução: 4.6572e-05s
```

- grafo 50 vértices

```
Clique máxima: 19 24 34 35 39 43 48 49
Tempo de execução: 6.4488e-05s
```

- grafo 75 vértices

```
Clique máxima: 28 45 53 54 64 67 71 72 74
Tempo de execução: 8.748e-05s
```

- grafo 100 vértices

```
Clique máxima: 3 54 56 65 69 76 84 85 90 98 99
Tempo de execução: 9.9685e-05s
```

- grafo 200 vértices

```
Clique máxima: 43 64 89 130 162 173 176 188 190 195 198 199
Tempo de execução: 0.000158288s
```

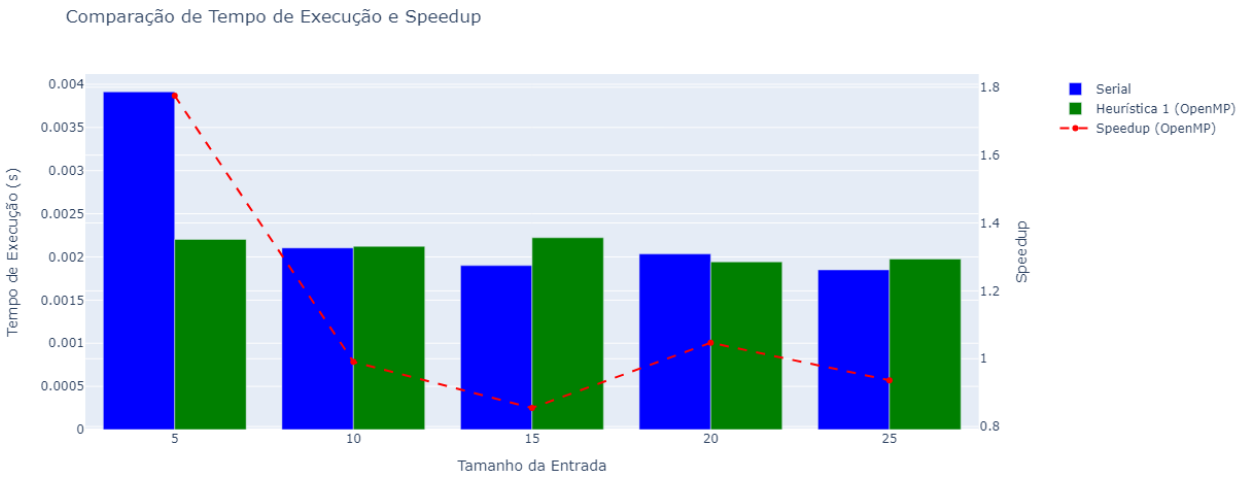
- grafo 500 vértices

```
Clique máxima: 271 391 399 426 440 458 462 464 472 475 488 492 493 498 499
Tempo de execução: 0.000398441s
```

Podemos repetir para outros isso para valores acima de 500 mas o MPi vai começar a falhar um pouco.Logo, o poder computacional para retornar uma solução do MPI em relação as outras abordagens, principalmente a heurística, e bem maior.

Comparações em relação a speedups:

- heuristica da busca_exaustiva (nome serial no gráfico foi uma pequena confusão) e OpenMp

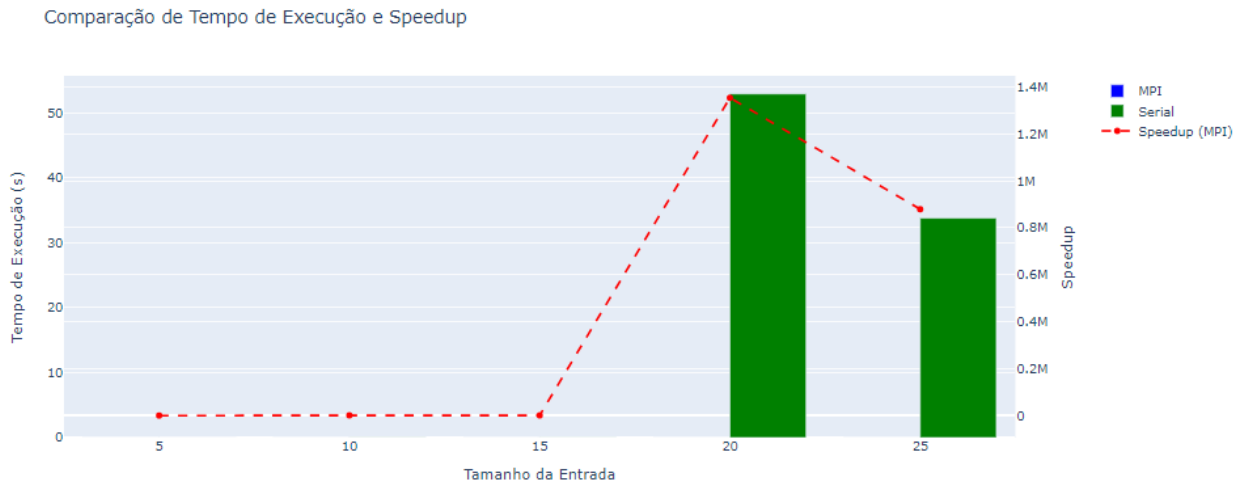


No primeiro o tempo foi parecido mas o OpenMp ja devia ser mais rápido o que aconteceu para alguns vértices.A disparidade de desempenho observada em alguns casos entre a execução sequencial e paralela com OpenMP pode ser atribuída a desafios específicos relacionados à natureza dos dados e ao padrão de acesso a esses dados. Em determinados vértices, a paralelização pode introduzir overheads adicionais, como contenção de recursos compartilhados ou desequilíbrio de carga, que neutralizam os benefícios potenciais da execução paralela. Estes casos particulares podem exigir uma análise mais profunda e ajustes específicos na implementação

paralela para otimizar a distribuição de tarefas e mitigar possíveis gargalos, destacando a complexidade intrínseca da paralelização em cenários específicos.

- heuristica da busca_exaustiva (exaustiva no gráfico) e MPI pelos tempos coletados pelos executáveis.

- heuristica da busca_exaustiva (está como serial no gráfico foi feita uma pequena confusão aqui também em relação a nomes) e MPI pelos tempos coletados via print.



No segundo caso é muito rápido o tempo de execução do MPI e por isso praticamente imperceptível em relação a tempo.

O código disponível em script.py roda o primeiro gráfico e os codigos speedups_MPI.py e speedups_MPI_hard_coded.py rodam a relação do e MPI com a busca exaustiva mostrando a grande disparidade de tempos.

Conclusão

O MPI em relação à heurística resultou em melhorias significativas no tempo de execução, como evidenciado pelo gráfico de desempenho.
O MPI facilitou a distribuição eficiente de tarefas entre diferentes nós de processamento, permitindo a execução simultânea em várias instâncias. Esse paralelismo distribuído revelou-se particularmente eficaz em cenários onde a heurística envolve a avaliação de múltiplos estados independentes.
A capacidade do MPI em coordenar a comunicação entre os processos, em conjunto com a execução simultânea, levou a uma redução substancial no tempo total de processamento.
Este resultado destaca a sinergia eficaz entre a heurística e a abordagem de programação paralela proporcionada pelo MPI, fortalecendo a escalabilidade e o desempenho global do algoritmo em questão