Trabalho 1 - Programação Paralela
Relatório de Experimentos

Geffté L. S. Caetano[1], **Arthur H. A. Farias**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brazil

{geffte.caetano,arthur.farias}@ufms.br

Abstract. *This document aims to present and compare the results obtained from experiments with an arbitrary-sized clique counting algorithm on an undirected graph. The algorithm was applied serially, parallelized without workload balancing, and parallelized with workload balancing. The comparisons of the three algorithms were performed for the same arbitrary clique size, varying the graphs used. The graphs tested were, in ascending order: citeseer, ca_astroph, dblp. Finally, the time comparisons generate the results of gain or loss in efficiency.*

Resumo. *Este documento tem como fim expor e comparar os resultados obtidos a partir dos experimentos de um algoritmo de contagem de cliques de tamanho arbitrário em um grafo não orientado. O algoritmo foi aplicado de forma serial, paralelizado sem balanceamento de carga de trabalho e paralelizado com balanceamento de carga de trabalho. As comparações dos três algoritmos foram realizadas para o mesmo tamanho de clique arbitrário, variando os grafos utilizados. Os grafos testados foram, em ordem crescente: citeseer, ca_astroph, dblp. Por fim, as comparações de tempo geram os resultados de ganho ou perda de eficiência.*

1. Grafo

De acordo com [Feofiloff et al. 2011], um grafo é um par (V,A), onde V é um conjunto arbitrário e A é um subconjunto de V. V é chamado de **vértices** e seu subconjunto A é chamado de **arestas**. Ainda nessa linha, os elementos de A assumiram a forma v,w, onde v e w pertecem ao conjunto V.

1.1. Cliques, cliques máximas e de tamanhos arbitrários

Segundo [Feofiloff et al. 2011], dado um grafo G, uma **clique** (ou conjunto completo) de G é definido por ser um conjunto de vértices X dois a dois adjacentes, ou seja, deve haver pelo menos uma aresta que conecta todos os vértices de X.

Ainda nessa linha, se há uma clique máxima denotada por $\omega(G)$, é óbvio que haverá pelo menos uma clique de tamanho menor ou igual a k, onde $k \leq \omega(G)$.

Assim, é possível concluir que, desde que haja uma clique máxima que atenda as condições acima, é possível obter cliques de tamanho menores que a máxima.

2. Algortimo de contagem de cliques

Nessa seção, será abordado o algoritmo de contagem de cliques de tamanho arbitrário k, onde $k \leq \omega(G)$. Esse mesmo algoritmo será utilizado nas seções abaixo com outras abordagens.

A principio, foi nos dado o pseudo-código em python, talvez por ser uma linguagem de alto níve. De qualquer forma, segue o algoritmo abaixo:

```
def contagem_de_cliques_serial(g:Grafo, k:int):
    cliques = []
        for each v in g.V(G):
            cliques.append([v])

        contador = 0
        while not empty cliques:
            clique = cliques.pop()

            # A clique atual tem k vertices
            if(len(clique) == k):
                contador+=1
                continue

            ultimo_vertice = clique[len(clique)-1]
            for each vertice in clique:
                for each vizinho in (adjacencia de vertice):
                    if (vizinho not in clique) and
                    (vizinho se conecta a todos os vertices de clique) a
                    (vizinho > ultimo_vertice):
                        nova_clique = clique.clone()
                        nova_clique.append(vizinho)
                        cliques.push(nova_clique)
    return contador
```

Resumidamente, o algoritmo acima, calcula todas as cliques de tamanho k de um grafo G, onde garante que nenhuma clique igual (em termos de vértices) será gerada a partir do mesmo vértices, mas ainda há a necessidade de ter uma estrutura de dados que não permita que duas cliques gêmeas sejam inseridas na estrutura de cliques.

2.1. Implementação do Algoritmo

Seguindo o pseudo-código, sua implementação foi em C++, e segue exatamente o que foi proposto. Segue o código abaixo:

```
int Graph::contagem_cliques_serial(int k) {
    set<vector<int>> cliques;

    for(auto v: vertices) {
        cliques.insert({v});
    }

    int count = 0;
    while(!cliques.empty()) {
        vector<int> clique = *cliques.cbegin();
        cliques.erase(find(cliques.begin(), cliques.end(), clique));

        int tamanho_clique = clique.size();
        if(tamanho_clique == k) {
            count++;
            continue;
        }

        int ultimo_vertice = clique.back();
        for(int vertice : clique) {
            vector<int> vizinhos_atual = getNeighbours(vertice);

            for(int vizinho: vizinhos_atual) {
                if(vizinho > ultimo_vertice &&
                formar_clique(vizinho, clique)) {
                    vector<int> nova_clique = clique;
                    nova_clique.push_back(vizinho);
                    cliques.insert(nova_clique);
                }
            }
        }
    }

    return count;
}
```

A implementação atual garante que o grafo é não orientado, pois ele foi tratado anteriormente para garantir isso, e também usa da estrutura de dados **set** para garantir que duas cliques gemêas não sejam inseridas no conjunto de cliques. Como o problema é apenas de contagem de cliques, as cliques obtidas não são armazenadas em outra estrutura de dados, logo o tempo não acaba se prolongando tanto. Porém, ainda assim, este algoritmo foi implementado de forma serial, e teóricamente, sua eficiência era para ser menor.

2.2. Implementação do paralelismo

Nesta seção, será abordada a implementação do paralelismo no algoritmo de contagem de cliques. Anteriormente toda a carga de trabalho era posta para uma única thread, agora, a ideia é dividir a carga de trabalho para t threads também arbitrárias. Vale levar em consideração que se o t escolhido for maior que a quantidade de threads suportada pelo computador, as threads irão disputar pelo núcleo e o problema sairá de paralelo para concorrente. Também foi dado um pseudo-código como exemplo, onde segue:

```
def contagem_de_cliques_paralela1(g:Grafo, k:int, t:int):
    cliques = []
    for each v in g.V(G):
        cliques.append([v])

    trabalhos_threads = divida vetor cliques em t
    conjuntos do mesmo tamanho

    Crie t threads com o seguinte codigo:
        contador = 0
        cliques = trabalhos_threads[tid]
        while not empty cliques:
            clique = cliques.pop()

            // A clique atual ja tem k vertices
            if(len(clique) == k):
                contador+=1
                continue

            ultimo_vertice = clique[len(clique)-1]
            for each vertice in clique:
                for each vizinho in (adjacencia de vertice):
                    if (vizinho not in clique) and
                    (vizinho se conecta a todos os vertices de clique) a
                    (vizinho > ultimo_vertice):
                        nova_clique = clique.clone()
                        nova_clique.append(vizinho)
                        cliques.push(nova_clique)
    return contador
```

Com o algoritmo dado, foi implementado sua versão em C++ , que segue os mesmos principios de divisão de carga de trabalho para as t threads. O algoritmo implementado está logo abaixo:

```
int Graph::contagem_cliques_paralela(int k, int n_threads) {
    unsigned int num_threads = n_threads;

    if (num_threads == 0) {
        num_threads = 1;
    }

    vector<vector<int>> cliques_iniciais;
    for (auto v : vertices) {
        cliques_iniciais.push_back({v});
    }

    vector<vector<vector<int>>> cliques_por_thread(num_threads);
    size_t num_cliques = cliques_iniciais.size();
    size_t cliques_por_thread_size = num_cliques / num_threads;
    size_t excesso = num_cliques % num_threads;

    size_t indice = 0;
    for (unsigned int tid = 0; tid < num_threads; ++tid) {

        size_t num_para_thread = cliques_por_thread_size +
                                 (tid < excesso ? 1 : 0);

        for (size_t j = 0; j < num_para_thread; ++j) {
            cliques_por_thread[tid].push_back(
                                     cliques_iniciais[indice++]);
        }
    }

    vector<int> contagens(num_threads, 0);
    vector<thread> threads;

    for (unsigned int tid = 0; tid < num_threads; ++tid) {
        threads.emplace_back([&, tid]() {
            set<vector<int>> cliques;

            cliques.insert(
                cliques_por_thread[tid].begin(),
                cliques_por_thread[tid].end());

            int &count = contagens[tid];

            while (!cliques.empty()) {

                vector<int> clique = *cliques.begin();
                cliques.erase(cliques.begin());
```

```
                    int tamanho_clique = clique.size();
                    if (tamanho_clique == k) {
                        count++;
                        continue;
                    }

                    int ultimo_vertice = clique.back();

                    for (int vertice : clique) {
                        vector<int> vizinhos_atual=getNeighbours(vertice);

                        for (int vizinho : vizinhos_atual) {
                            if (vizinho > ultimo_vertice &&
                            formar_clique(vizinho, clique)) {
                                vector<int> nova_clique = clique;
                                nova_clique.push_back(vizinho);
                                cliques.insert(nova_clique);
                            }
                        }
                    }
                }
            });
        }

        for (auto &th : threads) {
            th.join();
        }

        int total_count = 0;
        for (int c : contagens) {
            total_count += c;
        }

        return total_count;
    }
```

Dessa forma, cada thread irá trabalhar no subconjunto de cliques que lhe foi atribuida. Mas ainda resta o desafio de fazer com que a divisão, mesmo que desigual, seja balanceada, já que neste algoritmo não há a garantia de que os subconjuntos contém a mesma quantidade de cliques, e assim as threads vão receber essa mesma quantidade. De forma sucinta, mesmo que o subconjunto tenha o mesmo tamanho, uma thread pode receber menos cliques e terminar primeiro, assim, ficando ociosa.

2.3. Implementação de balanceamento de carga

Por fim, essa seção trata do balanceamento de carga de trabalho das threads, onde a inteção é a thread que terminou seu trabalho e está ociosa acabar "roubando" carga de trabalho de outras threads. Neste caso, temos mais um parâmetro que é o r, que é a quantidade de cliques que serão "roubadas" de outras threads.

Também foi proposto um pseudo-código, que segue:

```
def contagem_de_cliques_paralela1(g:Grafo, k:int, t:int, r:int):
    cliques = []
    for each v in g.V(G):
        cliques.append([v])

    trabalhos_threads = divida vetor cliques em t conjuntos
    do mesmo tamanho

    Crie t threads com o seguinte codigo:
        contador = 0
        cliques = trabalhos_threads[tid]
        while not empty cliques:
            clique = cliques.pop()

            // A clique atual ja tem k vertices
            if(len(clique) == k):
                contador+=1
                continue

            ultimo_vertice = clique[len(clique)-1]
            for each vertice in clique:
                for each vizinho in (adjacencia de vertice):
                    if (vizinho not in clique) and
                    (vizinho se conecta a todos os vertices de clique)
                    and (vizinho > ultimo_vertice):
                        nova_clique = clique.clone()
                        nova_clique.append(vizinho)
                        cliques.push(nova_clique)
            if empty cliques:
                cliques = r cliques roubadas de outra thread
    return contador
```

Assim, segue o código com a implementação em C++ do balanceamento de carga.

```
int Graph::contagem_cliques_paralela_balanceada(int k,
    int n_threads, long unsigned int roubo_carga) {

    unsigned int num_threads = n_threads;

    if (num_threads == 0) {
        num_threads = 1;
    }

    vector<vector<int>> cliques_iniciais;
    for (auto v : vertices) {
        cliques_iniciais.push_back({v});
    }

    vector<vector<vector<int>>> cliques_por_thread(num_threads);
    size_t num_cliques = cliques_iniciais.size();
    size_t cliques_por_thread_size = num_cliques / num_threads;
    size_t excesso = num_cliques % num_threads;

    size_t indice = 0;
    for (unsigned int tid = 0; tid < num_threads; ++tid) {

        size_t num_para_thread = cliques_por_thread_size +
                                 (tid < excesso ? 1 : 0);

        for (size_t j = 0; j < num_para_thread; ++j) {
            cliques_por_thread[tid].push_back(
                cliques_iniciais[indice++]);
        }
    }

    vector<int> contagens(num_threads, 0);
    vector<thread> threads;

    vector<mutex> mutexes(num_threads);
    atomic<int> threads_trabalhando(num_threads);
    atomic<bool> trabalho_ativo(true);

    auto roubar_cliques = [&](unsigned int tid) -> bool {
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dist(0, num_threads - 1);
        set<int> visitados;

        for (unsigned int tentativa = 0;
            tentativa < num_threads; ++tentativa) {
```

```
            unsigned int outro_tid = dist(gen);

            if (visitados.find(outro_tid) !=
            visitados.end() || outro_tid == tid) {
                continue;
            }

            visitados.insert(outro_tid);

            if (cliques_por_thread[outro_tid].size() <
                2 * roubo_carga)
                continue;

            {
                lock_guard<mutex> lock(mutexes[outro_tid]);
                    {
                        lock_guard<mutex> lock_roubo(mutexes[tid]);

                        for (long unsigned int i = 0;
                            i < roubo_carga &&
                            !cliques_por_thread[outro_tid].empty();
                            ++i)
                            {
                            cliques_por_thread[tid].push_back(
                                cliques_por_thread[outro_tid].back());
                            cliques_por_thread[outro_tid].pop_back();
                            }
                    }
                    return true;
            }
        }
        return false;
    };

    for (unsigned int tid = 0; tid < num_threads; ++tid) {
        threads.emplace_back([&, tid]() {
            set<vector<int>> cliques;

            cliques.insert(
            cliques_por_thread[tid].begin(),
            cliques_por_thread[tid].end());

            int &count = contagens[tid];

            while (!cliques.empty()) {
```

```
                vector<int> clique = *cliques.begin();
                cliques.erase(cliques.begin());


                int tamanho_clique = clique.size();
                if (tamanho_clique == k) {
                    count++;
                    continue;
                }

                int ultimo_vertice = clique.back();

                for (int vertice : clique) {
                    vector<int> vizinhos_atual=getNeighbours(vertice);

                    for (int vizinho : vizinhos_atual) {
                        if (vizinho > ultimo_vertice &&
                        formar_clique(vizinho, clique)) {
                            vector<int> nova_clique = clique;
                            nova_clique.push_back(vizinho);
                            cliques.insert(nova_clique);
                        }
                    }
                }
            }
            if(!roubar_cliques(tid)){
                threads_trabalhando--;
            }
        });
    }

    for (auto &th : threads) {
        th.join();
    }

    int total_count = 0;
    for (int c : contagens) {
        total_count += c;
    }

    return total_count;
}
```

No caso, ficou a critério do autor escolher o método de escolha para decidir qual(is) thread(s) terá(ão) suas cliques roubadas. O método proposto foi a escolha randômica da thread. Como é um problema indeterminado, pois a partir de um vértice a thread pode morrer ou ramificar indefinidamente, todos os vértices tem potencial de crescimento. E, mesmo considerando o volume de carga de trabalho não faz tanta diferença qual vai ser a thread escolhida.

3. Resultados

Por fim, esta seção mostrará os resultados obtidos, sendo que as comparação feitas foram variando o tamanho k das cliques e os datasets para cada algoritmo. Também foi contabilizado o tempo de execução de cada um para saber se houve ganho ou perda. Por fim, houve uma variação no valor arbitrário de r cliques "roubadas", a fim de investigar se a partir de um determinado valor, não há mais ganho de desempenho em roubar carga de trabalho de outras threads. Então, a primeira tabela os valores das cliques obtidas nos três algoritmos. Naturalmente há apenas uma tabela pois os valores precisam serem os mesmos. Nas tabela seguintes, temos o tempo de execução gasto para a contagem de cliques em segundos.

Table 1. Quantidade de cliques obtidas

Dataset	Tamanho das Cliques (k)	Quantidade de Cliques
citeseer	3	1166
	4	255
	5	46
	6	4
ca_astroph	3	1351441
	4	9580415
	5	64997961
	6	400401488
	7	2218947958
dblp	3	2224385
	4	16713192
	5	262663639
	6	4221802226

A tabela abaixo representa o tempo gasto em segundos para a contagem das cliques usando o algoritmo de contagem serial.

Table 2. Tempo de execução - algoritmo serial

Dataset	Tamanho das Cliques (k)	Tempo (s)
citeseer	3	0.0522312
	4	0.0650045
	5	0.0687923
	6	0.0699239
ca_astroph	3	17.3828
	4	221.228
	5	2220.54
	6	+4hrs
	7	+4hrs
dblp	3	36.1096
	4	286.3
	5	5454.28
	6	+4hrs

Na tabela abaixo, temos os valores dos tempos de execução da contagem de cliques em minutos do algoritmo parelelo sem balanceamento de carga.

Table 3. Tempo de execução - algoritmo paralelo

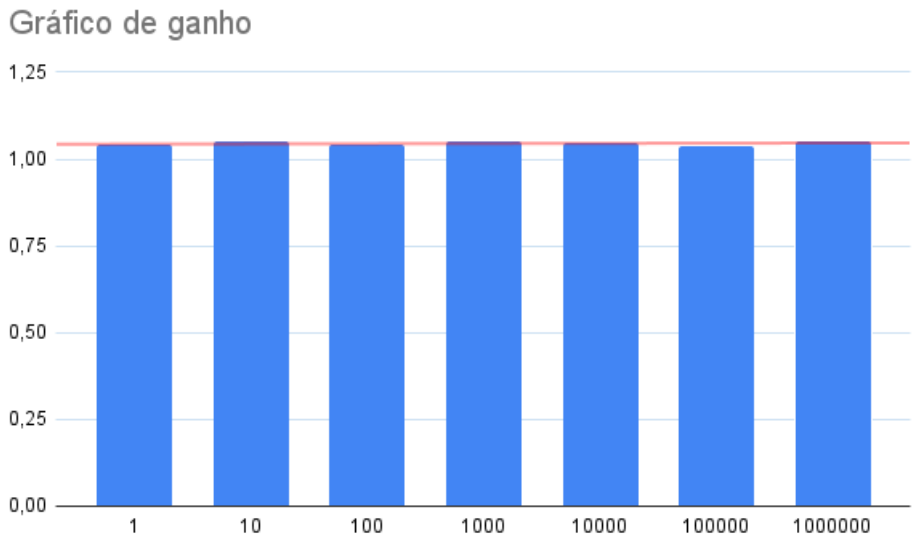
Dataset	Tamanho das Cliques (k)	Tempo (s)
citeseer	3	0.0112153
	4	0.0184929
	5	0.012463
	6	0.0125436
ca_astroph	3	12.9792
	4	165.054
	5	1782.02
	6	+4hrs
	7	+4hrs
dblp	3	18.9649
	4	200.95
	5	1833.37
	6	+4hrs

Na tabela abaixo, temos os valores dos tempos de execução da contagem de cliques em minutos do algoritmo parelelo sem balanceamento de carga. Porém, a fim de concluir os experimentos, esse foi executado em um i5 de 10ºgeração da Dell, com 8 threads.

Table 4. Tempo de execução - algoritmo paralelo com balanceamento

Dataset	Tamanho das Cliques (k)	Tempo (s)
citeseer	3	0.0297959
	4	0.0303518
	5	0.0406894
	6	0.0407241
ca_astroph	3	10.6048
	4	158.169
	5	1694.71
	6	14962.4
	7	+4hrs
dblp	3	18.2536
	4	175.129
	5	1723.25
	6	+4hrs

Por fim, para investigar o comportamento do tempo ao aumentar o valor r variamos seu valor, assim podemos ver se há aumento significativo de ganho, estabilidade ou perda de desempenho. Para esse experimento, temos no eixo y o ganho de desempenho em relação ao algoritmo paralelo, e no eixo x a quantidade de roubo de carga (r).



Nota-se um comportamento de que mesmo aumentando 10x a cada iteração de r, não há ganhos significativos no tempo.

References

Feofiloff, P., Kohayakawa, Y., and Wakabayashi, Y. (2011). Uma introdução sucinta à teoria dos grafos.