UM ALGORITMO PARALELO PARA GERAÇÃO DE CIRCUITOS ELEMENTARES EM GRAFOS DIRIGIDOS

Rodrigo Alves Randel
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
rodrigorandel@gmail.com

Thiago Correia Pereira
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
thiagocorreiap@gmail.com

Leandro Rochink Costa
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
rochink@gmail.com

Daniel Aloise
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
aloise@dca.ufrn.br

Caroline Thennecy de Medeiros Rocha
Universidade Federal do Rio Grande do Norte
Campus Universitário Lagoa Nova, CEP 59078-970 | Natal/RN - Brasil
caroline.rocha@ect.ufrn.br

RESUMO

Este trabalho propõe uma paralelização, inédita na literatura, do algoritmo de Johnson, para enumeração de circuitos elementares de um digrafo, produzido em 1975. Mesmo com o aumento do poder de processamento computacional, poucos são os algoritmos encontrados na literatura que exploram o poder da computação paralela. O ganho de tempo e eficiência proporcionado pela computação paralela em problemas no qual o esforço de resolução cresce muito rapidamente com o aumento da entrada torna o trabalho de paralelização importante para aumentar o tamanho das instâncias passíveis de serem abordadas em um tempo computacional razoável. OpenMP tem uma vantagem grande sobre outras APIs devido à facilidade, para o desenvolvedor, em definir regiões paralelas sem a necessidade de realizar muitas alterações no código. Os resultados mostram que o nosso trabalho foi capaz de paralelizar o algoritmo de Johnson com eficiência média de 74%, impactando grandemente no tempo de resolução desses tipos de problemas.

PALAVRAS CHAVE. Computação Paralela, Grafos, Otimização Combinatória.

Área Principal: Otimização Combinatória ; Teoria e Algoritmos em Grafos

ABSTRACT

This work proposes parallelization, unprecedented in literature, of Johnson's algorithm, for enumerating elementary circuits on a digraph, produced in 1975. Although with the increase of computational processing power, few are algorithms that are found in literature that explore the power of parallel computation. The gain of time and efficiency offered by parallel computation on problems in which the effort to solve increases rapidly with the increase of entry, makes parallelization important to increase instances sizes likely to be done in a reasonably computational time. OpenMP have a great advantage between others APIs because of the ease of, to developers, define parallel regions without the need to change large parts of the code. Results show that our work is capable of parallelize Johnson's algorithm with mean efficiency of 74%, impacting greatly the time of resolution of those kinds of problems.

KEYWORDS. Elementary Circuits. Parallel Computation. Graphs.

Main Area: Combinatorial Optimization; Theory and Algorithms on Graphs

1. Introdução

O algoritmo paralelo tema deste trabalho é baseado no algoritmo concebido por Johnson em 1975 para a enumeração de circuitos elementares de grafos dirigidos. Este algoritmo é encontrado na literatura aplicado a diversos problemas de otimização combinatória como, por exemplo: para evitar impasses em sistemas industriais fléxiveis [Zhang and Judd, 2008]; no problema de orientação, do inglês, *Orienteering Problem* [Leifer and Rosenwein, 1994]; no problema de reabastecimento de postos de combustíveis com vários depósitos [Cornillier et al., 2012]; e no problema de reabastecimento de postos de combustíveis [Cornillier et al., 2009].

Em Cornillier et al.(2009), o algoritmo de Johnson é utilizado para encontrar todas as rotas que os veículos podem realizar. Cada cliente é representado por um vértice e cada arco possui um custo associado ao caminho entre dois clientes em um grafo dirigido.

O ganho de tempo e eficiência proporcionado pela computação paralela problemas no qual o esforço de resolução cresce muito rapidamente com o aumento do grafo torna o trabalho de paralelização importante para aumentar o tamanho das soluções passíveis de serem abordadas em um tempo computacional razoável. O avanço das arquiteturas de processadores e núcleos em computadores abriram as portas para computação paralela permitindo que a rotina seja executada, simultaneamente, em múltiplos processos e *threads* em vários processadores. Como bem definido por Crainic (2008), paralelismo é uma decomposição da carga de trabalho total em tarefas para serem executadas em processadores disponíveis. Quando estamos interessados em paralelizar um algoritmo, é comum analisarmos qual é o gargalo de tempo do algoritmo ou, ainda, onde há *loops* que possam ter suas tarefas divididas.

OpenMP (Open Multi-Processing) é uma API de programação paralela que utiliza o paradigma de memória compartilhada. OpenMP tem uma vantagem grande sobre outras APIs devido à facilidade, para o desenvolvedor, em definir regiões paralelas sem a necessidade de realizar muitas alterações no código. Ela funciona através de instruções especiais de pré processamento conhecidas como *pragmas*. Os pragmas indicam ao compilador o que ele deve fazer e se ele (o compilador) não suportar estas diretivas, apenas as ignora fazendo com que o código execute normalmente como um programa serial.

Este trabalho, inédito na literatura, apresenta uma abordagem para enumerar todos os circuitos elementares de um grafo através de um algoritmo paralelo. Ele está estruturado da seguinte forma: a Seção 2 descreve o algoritmo de Johnson. Na Seção 3, o algoritmo paralelo de enumeração de circuitos é descrito. Os resultados computacionais são reportados na Seção 4. Finalmente, as considerações finais são dadas na Seção 5.

2. Algoritmo de Johnson

A ideia do algoritmo de Johnson é encontrar todos os Circuitos Elementares (CE) em um digrafo. CE são caminhos em que o primeiro e último vértice são idênticos e todos os outros vértices só aparecem uma única vez. Dois CE são ditos idênticos se um é uma combinação cíclica do outro.

Para encontrar todos os CE, o algoritmo parte de um nó raiz *s* que pertence ao subgrafo induzido (e fortemente conexo) por *s* e todos os vértices maiores do que *s*, evitando assim, repetição de circuitos elementares e buscas infrutíferas (o algoritmo assume que todos os vértices são numericamente identificados de 1 a n). Dado um grafo direcionado $G(V, A)$, sendo V o conjunto de vértices e A o conjunto de arcos, e W outro conjunto de vértices, define-se que F é um subgrafo induzido de G por W se $W \subseteq V$ e $F = (W, \{(u, v)|u, v \in W e (u, v) \in A\})$. F é dito fortemente conexo se para todo $u, v \in W$ existe caminho de *u* para *v* e de *v* para *u*.

A entrada do algoritmo é a representação do digrafo em uma estrutura de adjacência A_G composta por uma lista de adjacência $A_G(v)$ para cada $v \in V$. A lista $A_G(v)$ contém u se, e apenas se, $(v, u) \in A$. Também será necessário possuir uma estrutura de adjacência idêntica a de A_G chamada B que irá armazenar vértices bloqueados para evitar que buscas infrutíferas sejam realizadas repetidas vezes.

O algoritmo, então, procede construindo caminhos elementares a partir do vértice s e, à medida que os outros nós vão sendo visitados, eles são adicionados em uma pilha. Adicionar um vértice v ao circuito elementar é chamar a função $CIRCUIT(v)$. Quando um vértice v é adicionado à pilha, ele é dito como bloqueado ($blocked(v) = true$) para que não seja utilizado novamente na construção dos circuitos. Em seguida é verificado se o nó atual v consegue alcançar o vértice raiz s, formando assim um circuito que pode ser salvo ou impresso. Em seguida, a lista $A_G(v)$ é percorrida e, para cada nó u da lista, se ele não estiver bloqueado, é chamada a função $CIRCUIT(u)$ de forma recursiva. No fim da função $CIRCUIT$, o vértice v é retirado do bloqueio e retirado da pilha. O pseudocódigo do algoritmo de Jonhson é mostrado abaixo.

Retirar do bloqueio não significa apenas fazer $blocked(v) \leftarrow false$. Na linha 16 do algoritmo é mostrada uma possível chamada ao método ***UNBLOCK***. Se a busca encontrou circuitos elementares a partir de v, então ele, e todos os vértices que estão em $B(v)$, podem ser desbloqueados. Em outras palavras, pode ser desbloqueado todo vértice w que não havia gerado circuitos elementares e que possui um arco (w, v) no subgrafo atual.

Em seu artigo, Johnson afirma que, dado um digrafo completo com n vértices, é possível calcular o número exato de CE através da expressão:

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1} (n-i)!$$

E mostra que o algoritmo apresentado executa em tempo $O((n+a)(c+1))$, com a sendo o número de arcos e c o número de CE. A explicação do termo $O(n + a)$ é justamente o tempo consumido entre a saída de dois CE nunca irá exceder o tamanho do grafo. Podemos observar isso já que não podemos ter dois desbloqueios de um mesmo vértice sem que antes um CE tenha sido encontrado. Outra observação é que o tempo para encontrar o componente fortemente conexo na linha 40 pode ser feito em tempo $O(n + a)$. Em relação ao espaço de alocação, ele será limitado em $O(n + a)$ mais o espaço necessário para lidar com a saída (impressão, por exemplo) já que nenhum vértice irá aparecer mais de uma vez em $B - list$.

Já que um vértice apenas é desbloqueado se for retirado da pilha de vértices antes de qualquer chamada ao método CIRCUIT, então nenhum vértice irá aparecer na pilha de vértices mais de uma vez e, consequentemente, a saída do algoritmo são apenas os circuitos elementares.

O algoritmo só irá obter um CE uma única vez. A prova disso se deve ao fato de que, para qualquer pilha $(s, v_1, v_2, \cdots, v_k)$ com v_k no topo da pilha, uma vez que v_k é retirado esta pilha não poderá ocorrer novamente.

3. Algoritmo Paralelo Proposto

Nesta seção será apresentado o algoritmo paralelo de Jonhson. Na subseção 3.1 apresentaremos o escopo das variáveis explicando suas mudanças. Na subseção 3.2 explicaremos com detalhes sobre o uso dos *pragmas* da API de paralelismo OpenMP.

3.1. Escopo das Variáveis

A definição dos escopos das variáveis é fundamental para o resultado correto do algoritmo paralelo. Não podemos permitir que uma *thread* escreva onde não deveria ou que leia informação errada escrita, indevidamente, por outra *thread*.

No Algoritmo 1, o vetor $blocked$ responsável por armazenar os vértices que já foram visitados, possui escopo global ao algoritmo. Vamos imaginar a situação em que duas ou mais *threads* vão executar a linha 3 e 26 ao mesmo tempo, e, como todas elas apontam para o mesmo endereço de memória, não há nenhuma garantia de que as *threads* irão gravar a nova informação uma após a outra, podendo haver sobreescrita e, consequentemente, perda de dados. Então, se o código vai ser executado em paralelo com cada *thread* realizando a construção de CE, de forma independente, a partir de um vértice *v*, então é necessário que cada *thread* possua uma cópia do

Algoritmo 1 Algoritmo de Jonhson

```
1: função CIRCUIT(v)
2:     Adiciona v na pilha
3:     blocked(v) ← true
4:     retorno ← falso
5:     para ∀ w ∈ A_G(v) faça
6:         se w = s então
7:             Armazena o CE composto pelos elementos da pilha e s
8:             retorno ← verdadeiro
9:         senão e se w não está bloqueado então
10:            se CIRCUIT(w) então
11:                retorno ← verdadeiro
12:            fim se
13:        fim se
14:    fim para
15:    se retorno = true então
16:        UNBLOCK(v)
17:    senão
18:        para ∀ w ∈ A_G(v) faça
19:            se v ∉ B(w) então Adiciona v em B(w)
20:        fim para
21:    Remove v da pilha
22:    retorne retorno
23: fim CIRCUIT
24:
25: função UNBLOCK(v)
26:    blocked(v) ← false
27:    para ∀ w ∈ B(v) faça
28:        Deleta w de B(v)
29:        se blocked(w) = true então
30:            UNBLOCK(w)
31:        fim se
32:    fim para
33: fim UNBLOCK
34:
35: função MAIN()
36:    pilha ← ∅
37:    n ← número de vértices
38:    s ← 1
39:    enquanto s < n faça
40:        A_G ← componente fortemente conexo induzido por {s, s + 1, s + 2...}
41:        se A_G ≠ ∅ então
42:            Desbloqueia todos os vértices
43:            CIRCUIT(s)
44:            s = s+1
45:        senão
46:            s = n
47:        fim se
48:    fim enquanto
49: fim MAIN
```

vetor *blocked*, permitindo que as *threads* alterem livremente esse vetor de acordo com os circuitos que são gerados a partir do vértice *v*. Logo, sempre que se realizar uma chamada recursiva à função **CIRCUIT** é necessário informar, também, o vetor $blocked$ que se está trabalhando.

Outras estruturas que tiveram seu escopo alterados foram, a pilha P que armazena os vértices do CE verificado e a estrutura de adjacência B. A cada iteração da função **CIRCUIT** seus conteúdos mudam, e, como o algoritmo paralelo foi feito de forma que *threads* diferentes construissem CEs diferentes, se estas *threads* modificarem as variáveis P e B com escopo global uma *thread*

pode acabar sobreescrevendo, indevidamente, o conteúdo das variáveis. Para contornar este problema sugerimos que copias das variáveis citadas sejam passadas como parâmetro das funções que as utilizam. Desta forma *threads* que executam instâncias das funções **CIRCUIT** e **UNBLOCK** receberão cópias destas variáveis e portanto poderão modificar seus conteúdos livremente, pois o escopo das variáveis deixou de ser global e passou a ser local.

Para o caso do grafo fortemente conexo A_G não houve alteração de escopo, mesmo com *threads* diferentes utilizando, porque esta variável é apenas de leitura e não sofre alterações por mais de uma *thread*, sendo assim é seguro utiliza-la de forma global.

3.2. Agendamento de Tarefas e Região Paralela

A versão 3.0 do OpenMP introduziu o conceito de *task* à API. *Task* facilita a paralelização de funções recursivas pois é uma ferramenta que permite a criação de tarefas em que as *threads* vão se dividir para executá-las.

Toda *task* criada é adicionada em uma *pool* de tarefas e, sempre que uma *thread* estiver desocupada ela retira uma tarefa dessa *pool* para trabalhar. As *threads* só finalizam seus trabalhos quando não houver mais nenhuma tarefa na *pool*. A criação de tarefas no OpenMP é feita através da diretiva *task*:

#pragma omp task
{
    código da tarefa
}

O escopo das variáveis são definidos exatamente no momento de criação da *task* através das cláusulas *private*, *firstprivate* e *shared*. A única diferença entre *private* e *firstprivate* é que, no escopo *firsprivate*, o valor da variável é copiado antes de se tornar privada. Em nosso algoritmo, o construtor da *task* é algo do tipo:

#pragma omp task firstprivate($blocked$, P e B)

Na tentativa de diminuir o *overhead* produzido pela criação de várias *tasks* devido a recursividade da função **CIRCUIT**, foi utilizada uma cláusula *if* seguida de uma expressão escalar. Esta nova cláusula permite que se satisfeita a expressão escalar, e esta apresente valor *true*, o conteúdo interno a diretiva *task* é executado pela *thread* corrente sem que seja criada uma nova tarefa. Desta forma é possível limitar a quantidade de tarefas que podem ser criadas, solucionando o problema do *overhead*. Esta cláusula é inserida da seguinte forma:

#pragma omp task if (expressão escalar)

Outra diretiva muito importante no conceito de *task* é a **taskwait**. Essa diretiva funciona como uma espécie de barreira para as tarefas criadas indicando que as *tasks* filhas criadas pela *task* atual devem ser finalizadas antes de seguir adiante. Em nosso algoritmo, essa diretiva precisa ser utilizada antes do fim da função **CIRCUIT**, caso contrário, é possível que a função atinja o fim mesmo que alguma *task* filha ainda não tenha sido finalizada.

Esse tipo de agendamento de tarefa é ideal para nosso algoritmo pois podemos modelar a paralelização criando uma tarefa para cada vez que a função **CIRCUIT** é chamada recursivamente.

Mesmo com a declaração das *tasks*, OpenMP exige a criação de uma região paralela. Essa região é onde se define a criação de todas as *threads* que serão executas no algoritmo e o seu construtor é definido por:

#pragma omp parallel num_threads(n)
{
    código executado em paralelo $\cdots$
}

Sempre que esse *pragma* é encontrado, o processo irá criar *n threads* para executar a região paralela, logo vemos a necessidade desta definição se encontrar fora da função **CIRCUIT**. Na próxima seção mostraremos onde foi inserida a diretiva.

3.3. Algoritmo Proposto

Aplicando as modificações propostas nas subseções anteriores, chegamos aos algoritmos:

Algoritmo 2 Função CIRCUIT

```
1:  função CIRCUIT(v, B, blocked, P)
2:    Adiciona v na pilha P
3:    blocked(v) ← true
4:    f ← false
5:    para ∀ w ∈ A_G(v) faça
6:      se w = s então
7:        Armazena o CE composto pelos elementos da pilha e s
8:        f ← true
9:      senão e se w não está bloqueado então
10:       #pragma omp task firstprivate(blocked, B, P) if (máximo de tarefas)
11:       {
12:         f ← f||CIRCUIT(w, B, blocked, P)
13:       }
14:     fim se
15:   fim para
16:   #pragma omp taskwait
17:   se f = true então
18:     UNBLOCK(v, blocked, B)
19:   senão
20:     para ∀ w ∈ A_G(v) faça
21:       se v ∉ B(w) então Adiciona v em B(w)
22:     fim para
23:   fim se
24:   Remove v da pilha P
25:   retorne f
26: fim CIRCUIT
```

O Algoritmo 2 mostra a inserção da diretiva ***task*** - imediatamente antes da chamada recursiva - no algoritmo serial e apresenta todas as modificações necessárias para tornar o algoritmo paralelo. A primeira mudança é na linha 1 onde é necessário receber o vetor $blocked$, a pilha P e a estrutura de adjacência B. Falamos na seção 3.1 que é necessário criar uma cópia das variáveis $blocked$, P e B antes de enviar na chamada da função ***CIRCUIT*** (linha 12), porém esse vetor foi adicionado na cláusula *firstprivate* do construtor da ***task*** e uma cópia já será criada automaticamente. Também é possível observar o uso da diretiva ***taskwait*** na linha 16 indicando que todas as tarefas filhas criadas pela tarefa atual devem ser finalizadas antes do fim da função.

Analisando mais detalhadamente o Algoritmo 2:

(a) na linha 1, a função **CIRCUIT** recebe o vetor $blocked$, a pilha P e a estrutura de adjacência B;

(b) entre as linhas 2 e 4, v é adicionado em P, o v é bloqueado e f tem o valor *false* atribuído;

(c) entre as linhas 5 e 15, serão realizadas operações sobre cada um dos vértices adjacentes a v;

(d) na linha 6, se w forma um circuito, então armazena o CE(linha 7) e atribuí *true* a f(linha 8);

(e) senão, na linha 9, uma nova tarefa é criada para continuar a busca por CE;

(f) na linha 10, a diretiva *task* cria um nova tarefa para ser processada paralelamente. No entanto, isso só ocorre se o número máximo de tarefas já criadas não estiver sido alcançado. Quando ao *firstprivate*, ele cria uma cópia de $blocked$, P e B antes de tornar-las privadas a thread;

(g) na linha 12, a função **CIRCUIT** é chamada pela *task* passando como parametros w, $blocked$, P e B. Além disso, o valor de f é atualizado levando em consideração o que foi retornado de **CIRCUIT**;

(h) na linha 16, a diretiva *taskwait* faz com que o algoritmo esperes as tarefas filhas serem concluídas;

(i) na linha 17, se algum CE foi encontrado, então, desbloqueie v (linha 18);
(j) caso contrário (linha 19) nas linhas 20 até 23 adicione v, se já não estiver, a lista de buscas infrutíferas de cada vértice adjacente dele;
(k) Finalmente, v é removido da pilha P, na linha 24, e f é retornado na linha 25;

Algoritmo 3 Função UNBLOCK

```
1: função UNBLOCK(v, blocked, B)
2:     blocked(v) ← false
3:     para ∀ w ∈ B(v) faça
4:         Deleta w de B(v)
5:         se blocked(w) = true então
6:             UNBLOCK(W, blocked, B)
7:         fim se
8:     fim para
9: fim UNBLOCK
```

Para que a função ***UNBLOCK***(Algoritmo 3) funcionasse com a abordagem paralela foi necessário que a sua chamada fosse alterada. A chamada desta função agora precisa exigir como parâmetros a referência do vetor *blocked* da tarefa que invocou o ***UNBLOCK***, a estrutura de adjacência que armazena os vértices infrutíferos, B, também desta tarefa, e, finalmente, o vértice v.

O Algoritmo 3 pode ser analisado da seguinte forma:

(a) na linha 1, a função **UNBLOCK** recebe o vertice v, o vetor $blocked$ e a estrutura de adjacência B;
(b) na linha 2, o vertice v é desbloqueado;
(c) nas linhas 3 até 8, são realizadas operações sobre cada um dos vértices adjacentes a v em B;
(d) na linha 4, o vértice w da adjacencia de v em B;
(e) se w estiver bloqueado (linha 5), então **UNBLOCK** é chamado para os parametros w, $blocked$ e B.

Algoritmo 4 Função MAIN

```
1: função MAIN()
2:     pilha ← ∅
3:     n ← número de vértices
4:     s ← 0
5:     #pragma omp parallel num_threads(threads)
6:     {
7:         #pragma omp single
8:         {
9:             enquanto s < n faça
10:                A_G ← componente fortemente conexo induzido por {s, s+1, s+2...}
11:                Desbloqueia todos os vértices, limpa a estrutura de adjacência B
12:                CIRCUIT(s, B, blocked, P)
13:                s = s + 1
14:            fim enquanto
15:        }
16:    }
17: fim MAIN
```

A diretiva ***parallel***, na linha 5, é responsável por criar a região paralela para o número de *threads* passado. A diretiva ***single*** utilizada na linha 7 indica que apenas uma *thread* irá executar o trecho de código definido. As demais $n-1$ *threads* irão aguardar até que alguma tarefa seja inserida no *pool* de para começar a trabalhar.

No Algoritmo 4 os seguintes passos são feitos:

(a) nas linhas 2 a 4, a pilha $pilha$, o número de vértices n e o vértice inicial do CE s são inicializados;
(b) na linha 5, a região paralela (da linha 7 a 15) com o número de *threads* $threads$ é então criada com o uso da diretiva ***parallel*** ;
(c) na linha 7 a diretiva ***single*** é usada e terá influência nas linhas 9 a 14;
(d) linhas 9 até 14, o laço de repetição enquanto é executado somente uma vez devido a diretiva da linha 7;
(e) então para cada um dos vértices do dígrafo, o componemte fortemente conexo induzido é criado na linha 10, primeiramente por s depois por $s + 1$ e assim por diante;
(f) também para cada um dos vértices do dígrafo, na linha 11 os vérices bloqueados são desbloqueados e os vértices infrutíferos são limpos;
(g) na linha 12, a função ***CIRCUIT*** é chamada informando $s, B, blocked, P$;
(h) na linha 13, s é incrementado em uma unidade;

4. Resultados

Neste capítulo, serão apresentados os resultados obtidos após a execução do algoritmo inédito proposto aplicado a grafos completos de tamanhos 2 a 13, porém para tamanhos do grafo de 2 a 7 o algoritmo demonstrou-se muito rápido e seus tempos se tornaram muito pequenos e portanto insignificantes para a analise do algoritmo paralelo. Portanto, apresentaremos apenas os tempos para os tamanhos de 8 a 13. Apresentaremos e comentaremos os resultados obtidos e, por último, iremos calcular a eficiência do algoritmo paralelo nas subseções posteriores.

4.1. Desempenho

As instâncias foram executadas no ambiente mostrado pela Tabela 1.

Arquitetura:	64 bits
Processador:	6 Intel(R) Xeon(R) CPU X5650 2.67GHz
Memória RAM:	32 Gb
Sistema Operacional:	Linux Ubuntu 14.04 64-bit

Tabela 1: Ambiente de execução das instâncias

Executamos nosso algoritmo calculando o tempo de execução entre as linhas 8 e 15 do Algoritmo 4 e, para obter uma melhor precisão, cada teste foi executado 10 vezes, sendo o resultado final a média aritmética dos tempos. Além disso, o algoritmo foi executado serialmente e com 2 a 6 *threads*. Será com base nesses resultados que iremos calcular a eficiência do algoritmo na próxima seção.

A Tabela 2 mostra o número de CE encontrados e o tempo de execução, em segundos, para cada grafo completo.

		Nº Threads					
Grafo Completo	**Nº de CE**	**1**	**2**	**3**	**4**	**5**	**6**
8	16064	0.0808	0.0565	0.0366	0.0289	0.0234	0.0210
9	125664	0.6656	0.4298	0.2895	0.2235	0.1907	0.1694
10	1112073	6.2789	4.1065	2.5998	2.2671	1.7883	1.4815
11	10976173	71.2962	40.9000	28.5714	21.7909	17.1640	15.3024
12	119481284	816.3316	461.8581	308.3115	230.8198	220.1661	170.5517
13	1421542628	9623.9950	5239.6090	3759.0720	2851.4460	2587.1570	2148.1580

Tabela 2: Resultado dos tempos de execução do algoritmo em segundos

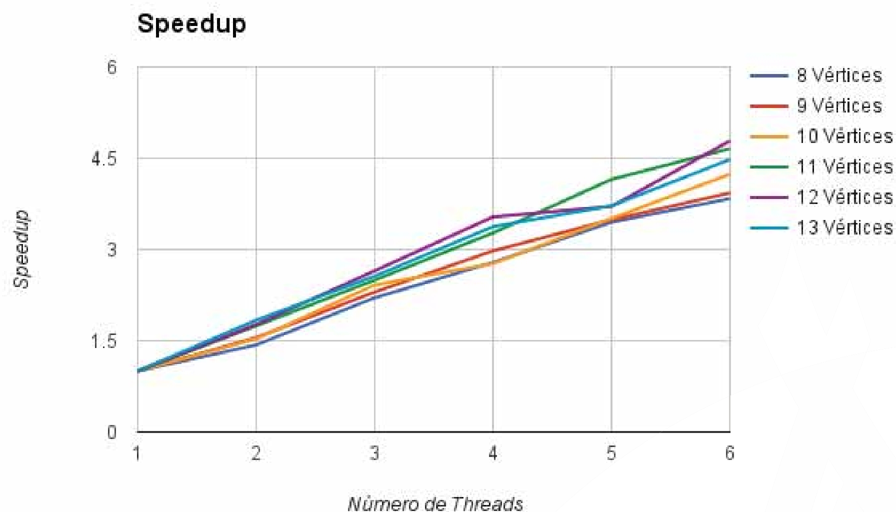

Figura 1: *Speedup* do algoritmo paralelo

Em uma análise rápida, é possível observar um bom desempenho do algoritmo que encontra todos os CE em tempos relativamente curtos. Em relação ao paralelismo, também é possível visualizar um decaimento proporcional quando aumentamos o número de *threads*.

4.2. *Speedup* e Eficiência

Para medir o ganho de desempenho relativo ao tempo de execução serial do algoritmo nós utilizamos o *speedup*. O cálculo do *speedup* é a relação entre o tempo de execução serial e o tempo de execução do algoritmo paralelo, que representa quantas vezes o algoritmo paralelo foi mais rápido em relação ao tempo serial. A Figura 1 apresenta o *speedup* do algoritmo.

Na Figura 1 é possível perceber que para todos os tamanhos do problema houve um ganho no *speedup* a medida que aumentamos o número de *threads*, observando fica evidente que aumentando o tamanho do problema há um ganho maior com o aumento do número de *threads*, este comportamento pode melhor ser observado no gráfico de eficiência que será exposto a seguir.

Quando se deseja avaliar o desempenho de um algoritmo paralelo, é comum calcular a eficiência do ganho de tempo quando o número de *threads* aumenta. Essa eficiência é calculada através da expressão abaixo:

$$E = \frac{T_{serial}}{p{*}T_{paralelo}},$$

com p sendo o número de *threads* e T_{serial} e $T_{paralelo}$ os tempos de execução do algoritmo serial e com p *threads*, respectivamente. A análise da eficiência nos fornece uma importante

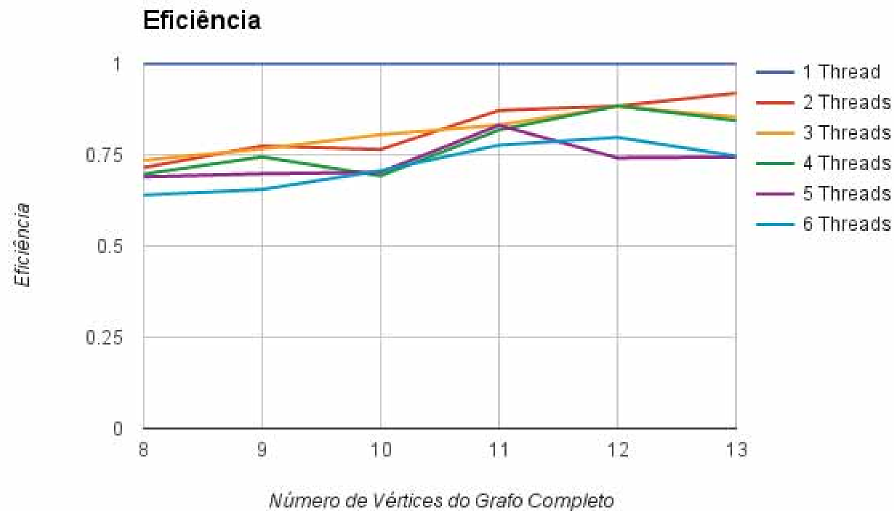

Figura 2: Eficiência do algoritmo paralelo

informação a respeito do ganho de tempo de execução, podendo ser fator fundamental para se tomar decisões referentes ao uso da aplicação, máquina em que a rotina/software irá rodar, custo benefício em adquirir uma máquina melhor, etc.

Para melhor ilustrar a eficiência obtida para as instâncias do problema abordado, foi feito o gráfico de eficiência que pode ser observado na Figura 2.

É possível notar que a eficiência do algoritmo foi aumentando a medida que o tamanho do problema aumentou, o que mostra que para tamanhos maiores do problema o algoritmo se torna mais eficiente e acaba trazendo um ganho de tempo considerável.

4.3. Escalabilidade

Quando tratamos de um algoritmo paralelo é comum verificar quais são os impactos causados no desempenho quando o tamanho do problema cresce. Existem dois tipos de escalabilidade que podem classificar um problema:

- **Fracamente Escalável:** Se aumentarmos o tamanho do problema à medida que aumentamos o número de *threads*, a eficiência do algoritmo deve permanecer constante.
- **Fortemente Escalável:** Se fixarmos o tamanho do problema, aumentarmos o número de *threads*, a eficiência do algoritmo deve permanecer constante.

Analisando as Figuras 1 e 2, é possível observar que a eficiência cai quando fixamos o tamanho do problema e aumentamos o número de *threads*, sendo assim o algoritmo é não fortemente escalável. Por outro lado se aumentarmos o tamanho do problema à medida que aumentamos o número de *threads*, percebemos que a eficiência tende a permanecer constante por volta de 74% como é possível perceber na Figura 2, classificando assim o algoritmo como fracamente escalável.

5. Considerações Finais

Esse trabalho apresentou um algoritmo e pseudo-códigos de sua implementação, capaz de encontrar todos os circuitos elementares de um grafo fortemente conexo de forma exata e paralela.

Podemos destacar também a eficiência média do algoritmo paralelo em 74% o que impacta grandemente no tempo de problemas desta natureza quando utilizamos para instâncias maiores. A partir dos bons resultados obtidos, é possível perceber que podemos destacar a contribuição

desse trabalho à comunidade cientifica por se tratar de um algoritmo paralelo para resolver problemas em grafos, sendo escasso na literatura trabalhos dessa natureza e inédito quando aplicado ao algoritmo de Johnson. Este trabalho pode ser aprimorado se comparado ao uso de uma ferramenta, muito semelhante ao OpenMP, chamada Cilk Plus que apresentou melhor desempenho em resultados recentes com sua função *spawning* quando comparada a função *tasks* do OpenMP, por gerar menos *overhead*. Além disso podem ser feitas analises do impacto do paralelismo em trabalhos que já utilizam o algoritmo de Johnson.

Referências

ARB, O. (2015). The openmp api specification for parallel programming.

Cornillier, F., Boctor, F., and Renaud, J. (2012). Heuristics for the multi-depot petrol station replenishment problem with time windows. *European Journal of Operational Research*, 220(2):361 – 369.

Cornillier, F., Laporte, G., Boctor, F. F., and Renaud, J. (2009). The petrol station replenishment problem with time windows. *Computers and Operations Research*, 36(3):919 – 935.

Crainic, T. (2008). Parallel solution methods for vehicle routing problems.

Johnson, D. B. (1975). Finding all the elementary circuits of a direct graph.

Leifer, A. C. and Rosenwein, M. B. (1994). Strong linear programming relaxations for the orienteering problem. *European Journal of Operational Research*, 73(3):517 – 523.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. An Introduction to Parallel Programming. Morgan Kaufmann.

Zhang, W. and Judd, R. P. (2008). Deadlock avoidance algorithm for flexible manufacturing systems by calculating effective free space of circuits. *International Journal of Production Research*, 46(13):3441–3457.