UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
TEORIA DOS GRAFOS

Algoritmo de Borůvka em Paralelo

Marcus Rodrigues

Pedro Ramos

1. INTRODUÇÃO

Uma **árvore geradora mínima** (AGM) é uma árvore geradora de um grafo conexo e não-direcionado. A AGM conecta todos os vértices da árvore com o menor peso total de arestas possível.

O problema de encontrar-se a AGM em um grafo conexo e não direcionado possui várias aplicações práticas. Um exemplo seria uma empresa de telecomunicações que gostaria de atravessar cabos em uma região da cidade. Se há algumas condições para se colocar cabos apenas em certos caminhos ou ruas, então haveria um grafo representando quais pontos estão conectados por esses caminhos. Alguns desses caminhos podem ser mais caros, pois são mais longos, por exemplo. Estes caminhos seriam representados por arestas com um peso associado, que representaria o custo daquele caminho. Neste caso, uma árvore geradora para este grafo seria um subconjunto de caminhos que não tenham ciclos mas conectem todos os pontos (no caso, todas as casas). Há várias árvores geradores possíveis. Porém, há uma árvore geradora que possui o menor peso possível de todas as arestas. Achá-la seria solucionar o problema da empresa de atravessar os cabos com o menor custo possível.

Todos os algoritmos para se encontrar a AGM são algoritmos gulosos. O primeiro algoritmo para encontrar a árvore geradora mínima foi desenvolvido pelo

cientista checo Otakar Borůvka. Seu objetivo, na época, era achar uma cobertura ótima para a rede elétrica da cidade de Moravia [1][2][3]. Discutiremos na próxima seção seu funcionamento com detalhes, já que trata-se dele o escopo deste trabalho.

Um outro algoritmo proposto, porém anos mais tarde, é o algoritmo de Prim, que foi inventado por Jarnik em 1930 [4] mas apenas redescoberto por Prim em 1957 [5] e Dijkstra em 1959 [6]. De forma geral, o algoritmo de Prim gera uma AGM aresta por aresta. Inicialmente, a árvore A contém um vértice árbitrário. Em cada passo, A é incrementada com a aresta de menor peso ligada àquele componente conexo, de tal forma que a cada passo um novo vértice é adicionado à árvore A [5]. Sua complexidade assintótica é O(m*log(n)) ou O(m + n*log(n)), dependendo-se das estruturas de dados utilizadas.

Outro algoritmo cujo uso é conhecido é o algoritmo de Kruskal, que também tem complexidade O(m*log(n)). De forma geral, o algoritmo de Kruskal ordena as arestas pelo seu peso, e percorre-as escolhendo uma a uma e acrescendo-as à árvore A, que começa vazia. Quando uma aresta é acrescida à A, seu peso é atualizado e seus vértices também. Quando uma nova aresta for selecionada, verifica-se se a adição daquela aresta causará um ciclo, ou seja, se ambos os vértices daquela arestas já estejam em A. Caso isso ocorra, essa aresta é desprezada. O algoritmo seleciona as arestas na ordem crescente de pesos até que se tenha uma árvore A com todos os vértices do grafo original G. Nesse estágio, A é uma AGM, e o algoritmo termina. [7]

Um quarto algoritmo não muito conhecido é o algoritmo "reverse-delete", que é basicamente o algoritmo de Kruskal na ordem reversa, e sua complexidade é $O(m*\log(n)*(\log(\log(n))^3)$.

Todos estes algoritmos supracitados são gulosos, e executam em tempo polinomial.

```
prim(G) # G é grafo
    # Escolhe qualquer vértice do grafo como vértice inicial/de partida
    s ← seleciona-um-elemento(vertices(G))

    para todo v ∈ vertices(G)
        π[v] ← nulo
    Q ← {(0, s)}
    S ← ø

    enquanto Q ≠ ø
        v ← extrair-mín(Q)
        S ← S ∪ {v}

        para cada u adjacente a v
            se u ∉ S e pesoDaAresta(π[u]→u) > pesoDaAresta(v→u)
                Q ← Q \ {(pesoDaAresta(π[u]→u), u)}
                Q ← Q ∪ {(pesoDaAresta(v→u), u)}
                π[u] ← v

    retorna {(π[v], v) | v ∈ vertices(G) e π[v] ≠ nulo}
```

Algoritmo de PRIM

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3    MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6       A = A ∪ {(u, v)}
7       UNION(u, v)
8 return A
```

Algoritmo de KRUSKAL

1.1. O algoritmo de Borůvka

O Algoritmo de Borůvka procede em uma sequência de etapas. Em cada estágio, o algoritmo identifca uma floresta F que consiste na menor aresta incidente em cada vértice do grafo G, e forma um grafo G' = G - F como entrada para o próximo estágio. Aqui, G - F denota o grafo derivado de G contraindo-se as arestas de F. Pela propriedade de corte, essas arestas agora pertencem à AGM. Cada passo do Borůvka dura um tempo linear. Uma vez que o número de vértices é reduzido para **ao menos** metade em cada passo, o algoritmo tem complexidade assintótica O(m*log(n)). [1]

Neste trabalho, desenvolvemos uma implementação do algoritmo de Borůvka em paralelo e comparamo-a com as implementações sequenciais dos algoritmos de Borůvka , Prim e Kruskal. A principal motivação para este trabalho é que dentre os três, o algoritmo de Borůvka é o que mais possui uma **natureza paralela,** pois é capaz

de operar em conjuntos de nós independentemente, ou seja, não possui uma dependência explícia em cada iteração.

```
0   Input: A connected graph G whose edges have distinct weights
1   Initialize a forest T to be a set of one-vertex trees, one for each vertex of the graph.
2   While T has more than one component:
3     For each component C of T:
4       Begin with an empty set of edges S
5       For each vertex v in C:
6         Find the cheapest edge from v to a vertex outside of C, and add it to S
7       Add the cheapest edge in S to T
8     Combine trees connected by edges to form bigger components
9   Output: T is the minimum spanning tree of G.
```

Algoritmo de BORUVKA

1.2. Paralelismo de dados e paralelismo de tarefas

Há, normalmente, duas formas de se paralelizar um código escrito em linguagem de alto nível: por paralelismo de dados ou por paralelismo de tarefas. No paralelismo de dados, o mesmo código opera paralelamente sobre porções diferentes do mesmo dado. No paralelismo de tarefas, código independente opera paralelamente sobre o mesmo dado, ou código independente e diferente opera de forma paralela.

Neste trabalho optamos por realizar um paralelismo de dados sobre o algoritmo de Borůvka .

2. FORMULAÇÃO DO PROBLEMA

Nesta seção há a formulação do problema de se implementar o algoritmo de Borůvka em paralelo. Para isso, formalizamos o algoritmo sequencial implementado neste trabalho para fins de comparação, e então formalizamos o algoritmo paralelo.

2.1. O algoritmo sequencial

Uma forma de se compreender o algoritmo de Borůvka é que, inicialmente, cada vértice representa um componente conexo. A cada iteração cada componente escolhe a melhor aresta para se conectar a outro componente, e a união desses componentes é realizada. O algoritmo para quando resta apenas 1 componente conexo, que é a próxima AGM.

Para se realizar o algoritmo de Borůvka de forma eficiente podemos usar o algoritmo de *union-find* (união-busca ou conjuntos disjuntos) [8]. Cada componente é um conjunto disjunto, e um vértice não pode estar em mais de um conjunto disjunto ao mesmo tempo, pela própria definição de conjunto disjunto (*"a intersecção de dois conjuntos disjuntos é o conjunto vazio"*). Quando há a união de um conjunto C1 com um conjunto C2, o vértice raiz de um dos dois conjuntos se torna o vértice raiz do novo conjunto C3, que contém os vértices de C1 e C2.

De uma maneira mais detalhada, o algoritmo que implementamos pode ser entendido como a seguinte sequência de passos:

Início: Inicialmente, cada nó é uma componente conexa, e portanto um conjunto disjunto. Cada conjunto possui uma lista de arestas incidentes a ele. A árvore A é inicializada vazia. Cada nó é um conjunto disjunto, portanto há (n) conjuntos disjuntos no começo.

Enquanto número de conjuntos independentes for maior que 1, faça:

Para cada componente c em C:

Passo 1 (Escolha da aresta mais leve): escolha a aresta de menor peso incidente a c. Caso existam duas ou mais candidatas com o peso igual, selecione aquela cujo vértice de destino tenha o menor índice (ordenação lexicográfica). Este detalhe é importante para evitar-se um caso especial do Borůvka, a ser explicado abaixo. A aresta escolhida liga c a um outro componente c'.

Passo 2 (Busca): Busque o nó raiz do componente c e do componente c'.

Passo 3 (União): Caso os nós-raízes de ambos sejam diferentes, então eles são conjuntos disjuntos. Portanto esta aresta escolhida entra para a árvore A, o peso da árvore A é atualizado com o peso desta aresta, e deve haver uma união entre os componentes c e c'. Esta união é feita da seguinte forma: aquele que possuir o nó raiz maior lexicograficamente (com o menor índice) "engole" o outro, ou seja, o nó-raiz de menor índice passa a ser o nó-raiz de todo o conjunto disjunto. A lista de arestas do componente é atualizada somente com as arestas incidentes a este novo conjunto.

Final: ao final, temos apenas um conjunto disjunto com todos os vértices e com a menor soma de peso das arestas possível. Encontramos nossa AGM.

Este algoritmo tem complexidade O(m*log(n)), pois a cada passo no laço externo, o número de conjuntos disjuntos diminui ao menos pela metade. No laço interno, para cada conjunto percorre-se a lista de arestas do conjunto, que no pior caso é (m). Dessa forma o algoritmo é O(m*log(n)) [9].

Um caso especial do Borůvka é quando, no passo 1, encontramos mais de uma aresta candidata com o menor peso. Nesse caso, se uma nova regra de ordenação secundária não seja definida, o algoritmo pode falhar, pois pode ocorrer a criação de ciclos. Para evitar esta falha, ordenamos lexicograficamente as arestas pelos índices dos nós das mesmas.

2.2. O algoritmo paralelo

No algoritmo paralelo, cada thread é responsável por um conjunto de nós. Dessa forma, exploramos a natureza paralela intrínseca ao algoritmo de Borůvka , que permite que o passo 1 do algoritmo (de escolha da melhor aresta) possa ser executado concomitantemente em cada um dos componentes conexos.

Nossa implementação do algoritmo paralelo baseia-se também no *union-find*, portanto a operação de procura de arestas e união de componentes é a mesma do sequencial. A grande diferença, que o torna paralelo, reside em uma nova estrutura chamada de *task*. Cada thread é responsável por uma *task*. Uma task é responsável por um conjunto de nós. Dessa forma, cada thread executa o *union-find* em um conjunto diferente de nós ao mesmo tempo. Nossos experimentos mostraram que essa paralelização torna o algoritmo 15% mais rápido, pois nas iterações iniciais, há muita oportunidade de paralelismo. Discutiremos este aspecto mais tarde.

Dessa forma, ao definir-se o número de *threads*, o programdor define em quantos subgrupos o conjunto de nós será dividido. Cada *thread* opera separadamente percorre a lista de arestas incidentes somente àqueles nós pelas quais ela está responsável.

De forma genérica, a cada iteração o conjunto de componentes é subdividido entre as *threads*. O algoritmo paralelo tem complexidade O(m*log(n)), pois continua fazendo como operação principal o *union-find*.

Início: Cada componente é um nó. Defina o número de *threads.*

Enquanto há mais de 1 componente, **fazer em paralelo:**

Passo 1: Dividir os componentes entre as *threads* disponíveis. Cada *thread* fica responsável por um conjunto de componentes.

Passos 1 - 4: Percorre as arestas e realiza a operação de União e Busca, como nos passos 1 a 3 do algoritmo sequencial.

Final: o conjunto disjunto resultante é a AGM.

3. IMPLEMENTAÇÃO

Implementamos os três algoritmos sequenciais (Prim, Kruskal e Borůvka) e a versão paralela do Borůvka na linguagem C++14.

Para a implementação do algoritmo sequencial, que é baseado no algoritmo *union-find*, as principais estruturas de dados são:

- **Grafo**: é a entrada do problema. Uma estrutura de grafo conexo não-direcionado foi implementada por lista de adjacências.
- **Conjuntos disjuntos (*union-find*):** Cada componente possui uma lista de arestas incidentes a ele e um nó raiz que representa a raiz do componente.

Para a implementação do algoritmo paralelo, além da utilização das estruturas mencionadas acima para o *union-find*, há também:

- **Task:** Uma região que mantém dois índices, i e j. Uma task é responsável por componentes do índice i ao índice j, ou seja, representa o intervalo de componentes pelo qual uma *thread* é responsável em tempo de execução.

A seguir descreveremos os experimentos e discutimos os resultados obtidos.

4. EXPERIMENTOS E DISCUSSÃO

Nossos testes foram executados em ambiente Linux (Ubuntu 14.04 LTS), em uma máquina Core i5 5200u 2.9Ghz (2 cores físicos + 2 cores simulados), Cache 3MB, com 8GB de RAM e SSD de 512GB. Para testar nossa implementação paralela do Borůvka , inicialmente geramos grafos completos com diferentes números de nós e pesos das arestas como números aleatórios entre 1 e 500. O intuito inicial era somente comparar os métodos sequenciais implementados (Borůvka , Prim e Kruskal) e determinar o melhor número de threads com o qual executar o Borůvka em paralelo.

Como observado na Figura 1, nossa implementação do Borůvka foi a mais rápida entre as três sequenciais. No eixo X temos a complexidade assintótica dos três algoritmos. O intuito deste grafo é observar a reta que possui a menor inclinação, de forma a detectar o algoritmo mais rápido entre os três. O número de nós dos grafos foi de n=100 a n=64000, sempre grafos completos.

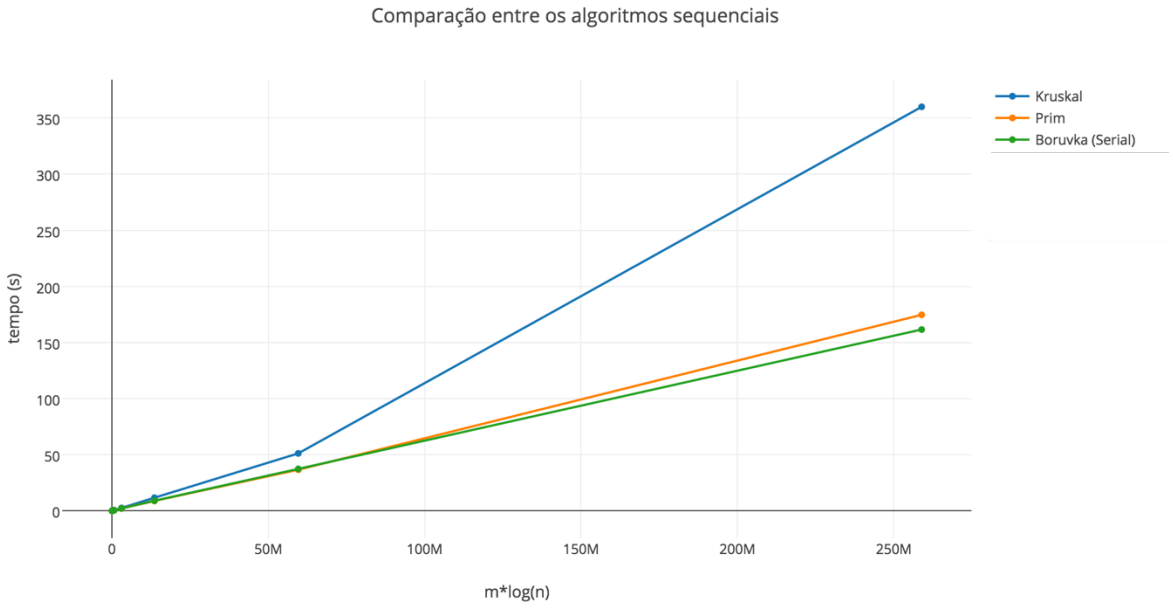

Figura 1: Comparação entre os algoritmos sequenciais

Após feito este experimento, variamos o número de *threads* do algoritmo paralelo para observar a eficiência do paralelismo sobre o problema. Como implementamos os algoritmos em C++ e utilizamos a biblioteca *pthread*, há um custo na criação e manutenção de *threads* no ambiente de execução. Executamos o algoritmo

paralelo com 2, 4, 8, 16, 32, 64 e 128 threads. Para grafos muito pequenos, com um número pequeno de nós, o número de threads não altera muito o desempenho do algoritmo. Observamos uma melhoria quando os grafos começam a ter muitos nós. Acreditamos que o paralelismo no Borůvka é máximo no início do algoritmo, e tende a zero à medida em que conjuntos disjuntos se unem e há menos componentes conexos. Discutiremos mais tarde este aspecto.

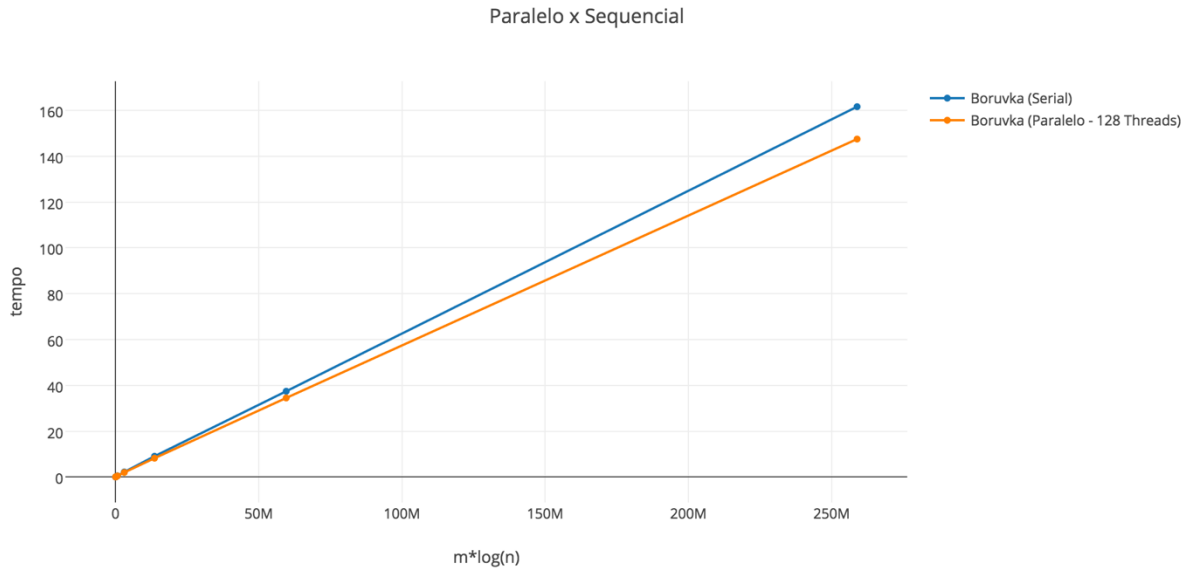

Figura 2: Comparação do Borůvka paralelo com o sequencial para grafos completos aleatórios.

Embora extremamente paralelizado, não conseguimos observar um ganho muito grande de tempo em relação ao algoritmo serial. Acreditamos que isso ocorra pois o paralelismo disponível no algoritmo de Borůvka não é linear.

Inicialmente, quando cada componente é apenas um vértice, há muito paralelismo no algoritmo, pois aproximadamente metade dos nós pode escolher a melhor aresta independentemente. Após a primeira escolha, a floresta composta pelos componentes se torna mais densa e há menos componentes, então o paralelismo disponível decresce rapidamente, pois menos componentes podem executar o estágio de escolha da melhor aresta.

A Figura 3 mostra o paralelismo para uma entrada de 10 mil nós. No começo do algoritmo, há muito paralelismo disponível, mas a medida que o algoritmo itera, os componentes diminuem e o paralelismo diminui consequentemente.

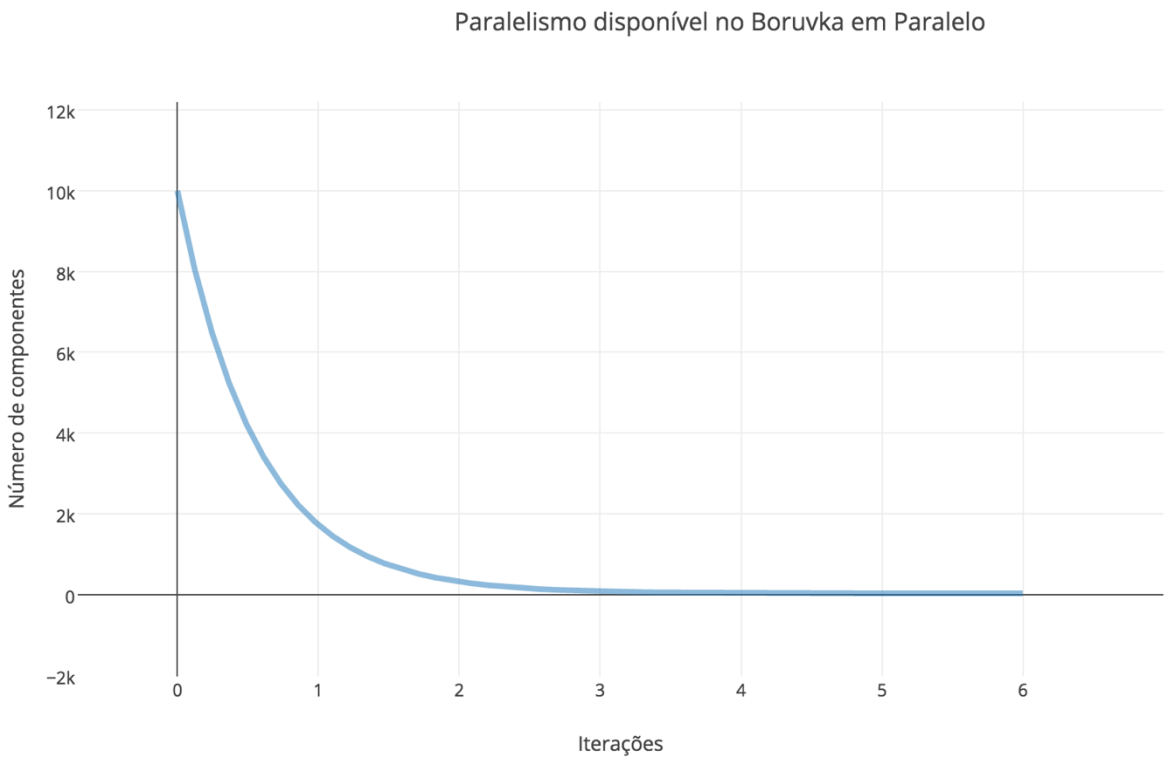

Figura 3: Paralelismo disponível no algoritmo de Borůvka

5. CONCLUSÃO

O algoritmo de Borůvka tem uma natureza mais paralela que os outros algoritmos conhecidos (Kruskal e Prim), pois um de seus passos envolve a escolha da melhor aresta incidente a um componente. Este passo pode ser executado de forma independente por cada componente, e a operação de união dos componentes é atômica.

Nestre trabalho, implementamos três algoritmos sequenciais para procurar a árvore geradora mínima (Kruskal, Prim e Borůvka), e implementamos também um algoritmo de Borůvka paralelo. Comparamos os algoritmos sequenciais e observamos que o Borůvka é o mais rápido entre os sequenciais. Acreditamos que isso ocorre pois o algoritmo foi implementado com uma estrutura de dados *union-find* em ponteiros, ao contrário dos outros algoritmos em que utilizamos estruturas mais complexas da STL do C++. Ao comparar o Borůvka paralelo com o sequencial, observamos um pequeno ganho, principalmente com entradas maiores (grafos com mais de 5 mil nós).

Infelizmente o paralelismo disponível no algoritmo de Borůvka decresce rapidamente após as iterações iniciais. Isso ocorre pois logo após a primeira iteração temos no máximo metade do número de componentes original disponível, que representa o fator log(n) na complexidade assintótica do algoritmo. Portanto, como o paralelismo do Borůvka envolve a escolha independente da melhor aresta em cada componente, à medida que há menos componentes disponíveis, há menos paralelismo disponível.

Para trabalhos futuros, propomos a implementação de um algoritmo misto, que comece com o Borůvka em paralelo, mas que após o paralelismo da operação de *find* diminua, outra estratégia para paralelizar seja utilizada.

Também pretendemos tornar o parâmetro de número de *threads* dinâmico. Atualmente é estático e definido pelo programador, mas no futuro planejamos variá-lo de acordo com o paralelismo disponível no decorrer do programa. Acreditamos que isto

talvez evite que o overhead de manutenção de *threads* em tempo de execução seja muito grande quando há poucos componentes disponíveis para cada *thread*.

REFERÊNCIAS

[1] Pettie, Seth, and Vijaya Ramachandran. "An optimal minimum spanning tree algorithm." *Journal of the ACM (JACM)* 49.1 (2002): 16-34.

[2] Borůvka, Otakar. "O jistém problému minimálním." (1926): 36-58.

[3] Borůvka, Otakar. "Prıspevek k rešenı otázky ekonomické stavby elektrovodnıch sıtı (contribution to the solution of a problem of economical construction of electrical networks)." *Elektronický obzor* 15 (1926): 153-154.

[4] Jarník, Vojtech. "O jistém problému minimálním." *Práca Moravské Prírodovedecké Spolecnosti* 6 (1930): 57-63.

[5] Prim, Robert Clay. "Shortest connection networks and some generalizations."*Bell system technical journal* 36.6 (1957): 1389-1401.

[6] Dijkstra, Edsger W. "A note on two problems in connexion with graphs."*Numerische mathematik* 1.1 (1959): 269-271.

[7] Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." *Proceedings of the American Mathematical society* 7.1 (1956): 48-50.

[8] Cormen, T. H., et al. "21. Data structures for disjoint sets." *Introduction to Algorithms,* (2009): 498-524.

[9] Chung, Sun, and Anne Condon. "Parallel implementation of Bouvka's minimum spanning tree algorithm." *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*. IEEE, 1996.