A parallel algorithm for minimum spanning tree on GPU

Jucele França de Alencar Vasconcellos,
Edson Norberto Cáceres
and Henrique Mongelli
College of Computing
Federal University of Mato Grosso do Sul
Campo Grande, MS, Brazil
Email: jucele, edson, mongelli@facom.ufms.br

Siang Wun Song
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, SP, Brazil
Email: song@ime.usp.br

Abstract—**Computing a minimum spanning tree (MST) of a graph is a fundamental problem in Graph Theory and arises as a subproblem in many applications. In this paper, we propose a parallel MST algorithm and implement it on a GPU (Graphics Processing Unit). One of the steps of previous parallel MST algorithms is a heavy use of parallel list ranking. Besides the fact that list ranking is present in several parallel libraries, it is very time-consuming. Using a different graph decomposition, called *strut*, we devised a new parallel MST algorithm that does not make use of the list ranking procedure. Based on the BSP/CGM model we proved that our algorithm is correct and it finds the MST after $O(\log p)$ iterations (communication and computation rounds). To show that our algorithm has a good performance on real parallel machines, we have implemented it on GPU. The way that we have designed the parallel algorithm allowed us to exploit the computing power of the GPU. The efficiency of the algorithm was confirmed by our experimental results. The tests performed show that, for randomly constructed graphs, with vertex numbers varying from 10,000 to 30,000 and density between 0.02 and 0.2, the algorithm constructs an MST in a maximum of six iterations. When the graph is not very sparse, our implementation achieved a speedup of more than 50, for some instances as high 296, over a minimum spanning tree sequential algorithm previously proposed in the literature.**

I. Introduction

Computing a minimum spanning tree and the connected components of a graph are fundamental problems in Graph Theory and arise as subproblems in many applications. Graham and Pavol [1] point out the importance of this problem in the design of computer and transportation networks, water supply networks, telecommunication networks, and electronic circuitry. A survey on the many facets of the minimum spanning tree problem can be found in the article by Mareš [2].

The sequential algorithms use depth-first or breadth-first search to solve these problems efficiently [3]. The parallel solutions for these problems, however, do not use these search methods because they are not easy to parallelize [4]. They are based instead on the approach proposed by Hirschberg et al. [5], which successively combines super-vertices of the graph into larger super-vertices. The approach gives rise to algorithms for PRAM (Parallel Random-Access Machine) models [6]. The most efficient of these algorithms is on a CRCW (Concurrent Read Concurrent Write) PRAM of $O(\log n)$ time with $O((m+n)\alpha(m,n))/\log n$ processors, where n and m are, respectively, the number of vertices and edges of the input graph, and $\alpha(m,n)$ is the inverse of the Ackermann's function [6].

Dehne et al. [7] present a BSP/CGM algorithm for computing a spanning tree of an unweighted graph that requires $O(\log p)$ communication rounds, where p is the number of processors. The algorithm in [7] requires the calculation of the Euler tour problem which in turn bases itself on the solution of the list ranking problem, which is very time-consuming.

Literature presents some works that propose parallel solutions using GPGPU, such as those shown by [8], [9], [10]. The algorithm proposed by Nobari et al. [9] is inspired by Prim's algorithm. Vineet et al. [8] and Nasre et al. [10] presented parallel solutions based on Borůvka's algorithm.

In this paper, we present an approach to obtain a minimum spanning of a given graph that does not need to solve the Euler tour or the list ranking problem. It has the practical advantage of avoiding the list ranking computation which, in spite of presenting an $O(\log p)$ communication rounds complexity [7], has been shown to require large constants in practical implementations. The proposed algorithm is based on the integer sorting algorithm which can be implemented efficiently on the BSP/CGM model [11]. Its operation resembles the Borůvka's algorithm [1]. We also used a filter idea similar as presented in [12].

We first show how to transform any input graph into a corresponding bipartite graph. We define a particular collection of trees (a forest), called *strut* [13], of the bipartite graph. The algorithm repeats a strut construction step followed by a compaction step until the desired minimum spanning tree is found.

The proposed algorithm is particularly suitable for implementation on a GPU. For instance, in the strut construction step, the computation associated with each vertex of the input graph can be carried out independently, so as to exploit the full computing power of the GPU. We prove that the number of strut construction and compaction steps is at most $O(\log n)$ where n is the number vertices of the input graph. Thus, in theory, we expect an efficient solution, which is in fact confirmed by the experimental results.

Due to the availability of source code and because it presents a superior performance for not very sparse graphs, we compare the results of our implementation with a recently published efficient algorithm [14]. The Edge Pruned Minimum Spanning Tree (EPMST) [14] algorithm presents better performance compared to the Filter-Kruskal solution [15]. Our parallel algorithm achieves speedups greater than 100 in comparison to the implementation made available by the EPMST authors.

This paper is organized as follows. In Section II we present the proposed parallel algorithm. Its correctness is discussed in Section III. Experimental results are show in Section IV and, finally, conclusions are given in Section V.

II. Parallel algorithm to obtain a minimum spanning tree (MST)

Algorithm 1 gives the main ideas of the proposed parallel MST (minimum spanning tree) algorithm. The solution consists of a parallel algorithm, created using the SIMD paradigm, that is, the steps are executed by several processors finding the solution collaboratively.

Algorithm 1 Main ideas of the proposed MST algorithm

Input: A connected graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n vertices and E is a set of m edges (v_i, v_j), where v_i and v_j are vertices of V. Each edge (v_i, v_j) has a weight denoted by w_{ij}.

Output: A minimum spanning tree of G whose edges are in $SolutionEdgeSet$.

```
1:  $SolutionEdgeSet$ := empty.
2:  Transform the given graph $G$ into a bipartite graph $H = (V, U, E')$.
3:  $condition$ := true.
4:  while $condition$ do
5:      Obtain a strut.
6:      for every strut-edge $(v_i, u_j)$ do
7:          Add the edge $(v_i, v_k)$ to the $SolutionEdgeSet$, where $v_i$ and $v_k$ are adjacent to $u_j$. In other words, add $original\_edge(u_j)$ to the $SolutionEdgeSet$.
8:      end for
9:      Compute the number of zero-difference vertices.
10:     if number of zero-difference vertices = 1 then
11:         $condition$ := false
12:     else
13:         for every strut-edge $(v_i, u_j)$ do
14:             Compact the two vertices adjacent to $u_j$ thereby producing a new bipartite graph.
15:         end for
16:     end if
17: end while
```

A. Transforming the input graph into a bipartite graph

To find a minimum spanning tree of a given graph, we first transform the original graph into a bipartite graph. This transformation can be done by adding a new vertex on each edge of the original graph, thereby subdividing each original edge into two new edges. If we consider the vertices of the original graph as belonging to one partition and the newly added vertices as belonging to a second partition, then we have a resulting bipartite graph.

More formally, consider a connected graph $G = (V, E)$, where V is a set of n vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a set of m edges (v_i, v_j), where v_i and v_j are vertices of V. Each edge (v_i, v_j) has a weight denoted by w_{ij}. We can transform G into a bipartite graph H by considering a set U of m new vertices $(u_1, u_2, \ldots, u_m)$ and substituting each edge (v_i, v_j) of E by two edges (v_i, u_k) and (v_j, u_k), both with weight equal to w_{ij}. Denote by E' the set of edges (v_i, u_k) for all $v_i \in V$ and $u_k \in U$. The graph $H = (V, U, E')$ thus obtained is bipartite.

Figure 1 shows an example. On the left we have the original graph $G = (V, E)$ with $V = \{1, 2, \ldots, 5\}$ and, on the right, the transformed bipartite graph $H = (V, U, E')$ with $U = \{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. Figure 2 shows the same bipartite graph, where the two vertex sets V and U are clearly illustrated. Observe that any vertex $u_k \in U$ which, by construction, was created on an edge of the original graph G, always has degree two and both edges incident to u_k have equal weight. There is a one-to-one correspondence between an edge (v_i, v_j) of the original graph G and the added vertex u_k. We use the notation $original\_edge(u_k)$ to denote the original edge (v_i, v_j). We also say edge (v_i, v_j) is *associated* to u_k.

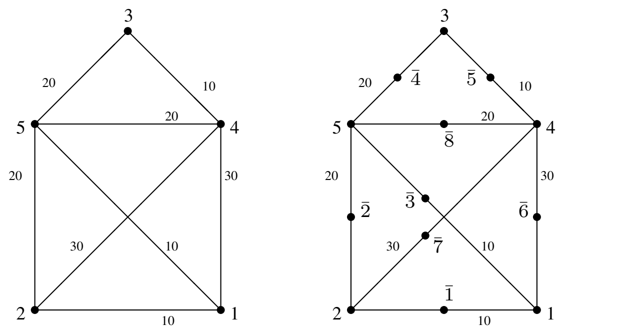

Fig. 1. On the left, the original graph $G = (V, E)$ and, on the right, the transformed bipartite graph $H = (V, U, E')$

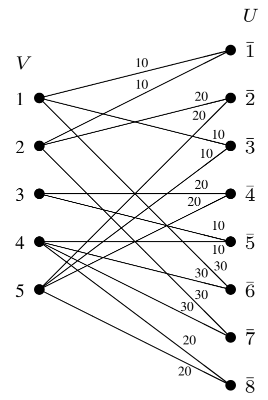

Fig. 2. The transformed bipartite graph $H = (V, U, E')$

B. Definition of strut and zero-difference vertices

Consider the transformed bipartite graph $H = (V, U, E')$ with vertex sets $V = \{v_1, v_2, \cdots, v_n\}$ and $U = \{u_1, u_2, \cdots, u_m\}$, and edge set E' where each edge joins one vertex of V and one vertex of U. For simplicity, let each vertex v_i of V be represented by i, i.e. $V = \{1, 2, \ldots, n\}$. Likewise, let us represent each vertex u_j of U by $\bar{j}$, i.e. $U = \{\bar{1}, \bar{2}, \ldots, \bar{m}\}$.

We define a *strut* in U [13], denoted by S, as a forest in $H = (V, U, E')$ such that each $v_i \in V$ is incident in S to exactly one edge of E', chosen as follows. Among all edges (v_i, u_j) incident to v_i in H, choose the edge with smallest weight and, if there are several edges (v_i, u_k) with the same smallest weight, select the edge (v_i, u_k) with the smallest u_k.

Let us give an example. Consider the transformed bipartite graph $H = (V, U, E')$ of Figure 2, where $V = \{v_1, v_2, \ldots, v_5\} = \{1, 2, \ldots, 5\}$ and $U = \{u_1, u_2, \ldots, u_8\} = \{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. For each vertex v_i of V, consider all the edges (v_i, u_j) incident to v_i and sort them lexicographically by the edge weight and by $\bar{j}$. To illustrate this, we extend the notation of an edge (v_i, u_j) by adding the weight w_{ij} in the middle, as $(v_i \;\; w_{ij} \;\; u_j)$. Thus we illustrate the result of this lexicographical sort as follows, where we have five groups of edges $(v_i \;\; w_{ij} \;\; u_j)$, one group for each $v_i = 1, 2, \ldots, 5$.

$(v_i$	w_{ij}	$u_j)$
(1	10	$\bar{1}$)
(1	10	$\bar{3}$)
(1	30	$\bar{6}$)

$(v_i$	w_{ij}	$u_j)$
(2	10	$\bar{1}$)
(2	20	$\bar{2}$)
(2	30	$\bar{7}$)

$(v_i$	w_{ij}	$u_j)$
(3	10	$\bar{5}$)
(3	20	$\bar{4}$)

$(v_i$	w_{ij}	$u_j)$
(4	10	$\bar{5}$)
(4	20	$\bar{8}$)
(4	30	$\bar{6}$)
(4	30	$\bar{7}$)

$(v_i$	w_{ij}	$u_j)$
(5	10	$\bar{3}$)
(5	20	$\bar{2}$)
(5	20	$\bar{4}$)
(5	20	$\bar{8}$)

By definition, the strut S is composed of the edges corresponding to the first row of each of the five groups above, namely, $(1 \;\; 10 \;\; \bar{1})$, $(2 \;\; 10 \;\; \bar{1})$, $(3 \;\; 10 \;\; \bar{5})$, $(4 \;\; 10 \;\; \bar{5})$, and $(5 \;\; 10 \;\; \bar{3})$. In Figure 3, the strut S obtained is illustrated by the dotted lines. For notation purposes, denote an edge in a strut as a *strut-edge*.

Observe that we could have assumed, without loss of generality, that all the weights of the edges $e \in E$ of the input graph G to be different. We need only to modify the original weight of each edge as follows. Consider an edge $(v_i, v_j) \in E$ and the label of the additional vertex u_k added on this edge to obtain the bipartite graph. If we now consider the new weight of (v_i, v_j) as the concatenation of the original weight and the label u_k, then all the new edge weights of G will be different. For example, in Figure 1, the original weights of edges $(1, 2)$, $(1, 5)$ and $(3, 4)$ are the same (equal to 10). The new weights of these edges can be 101, 103 and 105, respectively. This makes the strut construction step even easier. For each $v_i \in V$, the strut-edge is the edge (v_i, u_j) with the smallest new weight. We will make this assumption in Section III to simplify the correctness proof.

We can see that if (v_i, u_j), with weight w_{ij}, is a strut-edge then no edge (v_i, u_k) in E' has weight smaller than the weight w_{ij}.

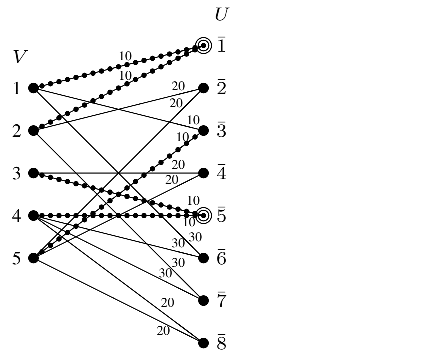

Fig. 3. The strut S represented by dotted lines. Vertices $\bar{1}$, $\bar{3}$ and $\bar{5}$ have at least one incident *strut-edge*. Vertices $\bar{1}$ and $\bar{5}$ have two and are zero-difference vertices.

Consider the bipartite graph $H = (V, U, E')$ and a strut S. Let u_j denote a vertex of U. Then we use the notation $d_H(u_j)$ to denote the degree of u_j in H, and $d_S(u_j)$ to denote the degree of u_j in S. A vertex $u_j \in U$ is called *zero-difference* in the strut S if $d_H(u_j) - d_S(u_j) = 0$. As already seen, the degree of any vertex u_j in H, or $d_H(u_j)$, is always two. Thus, vertex u_j is zero-difference if its degree in S is also two. In Figure 3 the vertices $\bar{1}$ and $\bar{5}$ both have two strut-edges, and thus they are zero-difference vertices. In Figure 3 they are enclosed by double circles around them.

Consider the solution set $SolutionEdgeSet$ with the edges of the desired minimum spanning tree. Initially this set is empty. After obtaining the strut S, the algorithm finds all vertices u_j that are incident to strut-edges (i.e. $d_S(u_j) \geq 1$) and adds $original\_edge(u_j)$ to the solution set. See again Figure 3. Vertices $\bar{1}, \bar{3}$ and $\bar{5}$ are incident to strut-edges. Thus we add to the solution set $original\_edge(\bar{1}), original\_edge(\bar{3}), original\_edge(\bar{5})$ (respectively edges (1, 2), (1, 5) and (3, 4)). Figure 4 shows the edges added to the solution set so far. The added edges are shown as dotted lines.

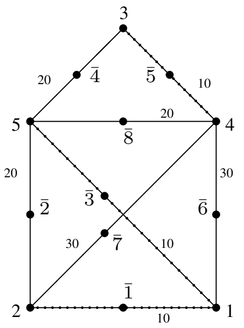

Fig. 4. Partial solution obtained so far.

If there is only one zero-difference vertex in the obtained strut, the problem is solved. See the proof of this in Section III.

C. Compacting the bipartite graph

When there are two or more zero-difference vertices, we must compress the bipartite graph for the execution of a new iteration of the algorithm. To do this, we must analyze each vertex $u_t \in U$ that is incident to a strut-edge (i.e. $d_S(u_t) \geq 1$). The vertices v_i and v_j ($v_i, v_j \in V$, such that $v_i < v_j$) adjacent to u_t must be contracted into a super-vertex v_i (keeping the label of the smallest vertex). Let the edges (v_i, u_t), (v_j, u_t) and (v_k, u_r) be strut-edges, and assume v_i or v_j is adjacent to u_r, then vertices v_i, v_j and v_k will all be contracted into the same super-vertex in the compacted graph. Looking at the example of Figure 3, the strut-edges are $(1, \bar{1})$, $(2, \bar{1})$, $(3, \bar{5})$, $(4, \bar{5})$ and $(5, \bar{3})$. As the other edge incident to $\bar{3}$ is $(1, \bar{3})$, the resulting super-vertices are $\{1, 2, 5\}$, labeled with 1, and $\{3, 4\}$, labeled with 3.

The algorithm also performs the suppression of vertices of U that are adjacent to contracted vertices of V. This is illustrated with the example of Figure 3, where vertices $\bar{1}$, $\bar{2}$, $\bar{3}$ and $\bar{5}$ must be suppressed. Figure 5 shows the result of the compaction, where the original vertices 2 and 5 were joined to vertex 1, and the original vertex 4 was joined to vertex 3. In the new bipartite graph $H' = (V', U', E'')$, resulting from compaction, $|V'|$ is equal to the number of zero-difference vertices in the strut S (see Lemma 2 in Section III).

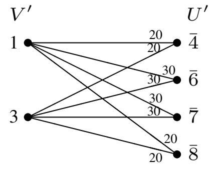

Fig. 5. Bipartite graph H' after compaction

The resulting graph after the compaction can have multiple U' vertices that are adjacent to the same couple of vertices of V'. As in the example of Figure 5, vertices $\bar{4}$, $\bar{6}$, $\bar{7}$ and $\bar{8}$ are adjacent to vertices 1 and 3. To optimize the algorithm, we can remove the U vertices that have greater weight. Figure 6 shows the resulting compact graph for this optimization.

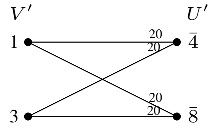

Fig. 6. Bipartite graph H' after optimized compaction

D. Finalizing the algorithm

After the compaction, we obtain a new bipartite graph H'. The algorithm obtains a new strut for H', updates the solution set, and repeats the entire process until there is only one zero-difference vertex. To obtain a strut for this compacted graph, we repeat the same procedure as before. For each vertex v_i of V', consider all the edges (v_i, u_j) incident to v_i and sort them lexicographically by the edge weight and by $\bar{j}$. The result of this lexicographical sort is illustrated as follows, where we now have two groups (v_i w_{ij} u_j), one group for each v_i = 1 and 3. (Notice that here vertex 1 represents the compaction of the original vertices 1 , 2 and 5, and vertex 3 represents the compaction of the original vertices 3 and 4.)

(v_i	w_{ij}	u_j)	(v_i	w_{ij}	u_j)
(1	20	$\bar{4}$)	(3	20	$\bar{4}$)
(1	20	$\bar{8}$)	(3	20	$\bar{8}$)
(1	30	$\bar{6}$)	(3	30	$\bar{6}$)
(1	30	$\bar{7}$)	(3	30	$\bar{7}$)

The new strut is composed by the edges corresponding to the first row of each of the two groups above, namely, $(1\ \ 20\ \ \bar{4})$, and $(3\ \ 20\ \ \bar{4})$. Figure 7 shows the strut obtained, illustrated by the dotted lines. In the figure, the zero-difference vertex is shown enclosed by double circles. Having obtained the strut, we add $original\_edge(\bar{4})$ to the solution set. Notice now we have only one such vertex and thus the algorithm terminates.

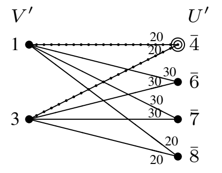

Fig. 7. New strut obtained, now with only one zero-difference vertex.

Figure 8 shows the obtained minimum spanning tree of the original graph. The edges added to the solution set are shown as dotted lines.

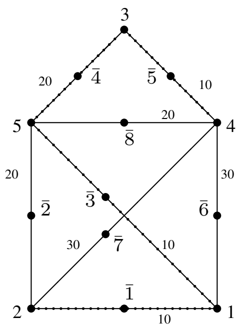

Fig. 8. Minimum spanning tree.

III. Discussion of the Algorithm

In this section we address the correctness of the proposed algorithm. Demonstrations can be safely obtained. We omit the evidence due to space constraints but they are available at https://github.com/jucele/MinimumSpanningTree.

Lemma 1. *Consider a connected graph $G = (V, E)$. Let $H = (V, U, E')$ be the transformed bipartite graph and S be a strut in U, obtained at step 5 of Algorithm 1. Let G' be the graph obtained by adding the edges associated to vertices u (i.e. $original\_edge(u)$) from U such that $d_S(u) \geq 1$ (step 7 of Algorithm 1). Then G' is acyclic. More over, if S contains exactly one zero-difference vertex then G' is a spanning tree of G.*

Note that if S has more than one zero-difference vertex, the generated graph G' will have less than $|V| - 1$ edges and will not be a spanning tree. Thus, the algorithm will need more iterations to complete the graph G'.

Theorem 1. *Consider a connected graph $G = (V, E)$. Let $H = (V, U, E')$ be the transformed bipartite graph and S be a strut in U, obtained at step 5 of Algorithm 1. Let G' be the graph obtained by adding the edges associated to vertices u (i.e. $original\_edge(u)$) from U such that $d_S(u) \geq 1$ (step 7 of Algorithm 1). If S contains exactly one zero-difference vertex then G' is a minimum spanning tree of G.*

Lemma 2. *Let V and U be the partitions of H right before the compaction (described in step 14 of Algorithm 1) and let V' and U' be the partitions of the compacted graph H' right after step 14. Let k be the number of zero-difference vertices in the strut S obtained in step 5, then the number of vertices in V' is k.*

Theorem 2. *The number of zero-difference vertices in U' after step 14 is at least divided by 2 in each iteration.*

IV. Experimental results

The input graphs, used in experimental tests, were generated using a random generator [16] available at http://condor.depaul.edu/rjohnson/source/graph_ge.c. We generated 28 random connected graphs. The input graphs named *graph10a, graph10b, graph10c, graph10d* and *graph10e* have 10,000 vertices and density 0.02, 0.05, 0.1, 0.15 and 0.2, respectively. The graphs identified with *graph20, graph25* and *graph30* have similar characteristics but with 20,000, 25,000 and 30,000 vertices, respectively. We generated eight graphs with 15,000 vertices (*graph15*) with densities 0.02, 0.05, 0.1, 0.15, 0.2, 0.5, 0.7 and 1. The input data set was not available due to its size (around 23GB).

Comparison of execution times with tree algorithms using CUDA that were published recently was not possible because these algorithms used input graphs for specific problems and different computational resources

To show the efficiency of our solution, we compare the results obtained by our implementation with a recently published efficient algorithm, Edge Pruned Minimum Spanning Tree (EPMST) [14]. EPMST chooses a subset of edges using random sampling. It uses the idea of Kruskal's algorithm [17] on the small subset of edges, and, if necessary, Prim's algorithm [18] on a compacted graph. EPMST implementation is available for download at GitHub, which made it easier to compare results. It is noteworthy that the set of input data used to perform the tests for this article were generated randomly and are different from those used in [11], possibly producing different results.

As EPMST is a sequential solution, we developed a sequential version of our algorithm using ANSI C and a parallel version, for GPGPU (General Purpose Graphics Processing Unit), using CUDA (Compute Unified Device Architecture). Both implementations are available for download at https://github.com/jucele/MinimumSpanningTree. The execution environment is a desktop which has eight processors Intel® Xeon® E5-1620 v3 @ 3.50GHz, with 10 MB of cache, 32 GB of memory and a GPGPU Nvidia Quadro M4000 (1,664 cores and eight GB of memory).

For each input graph, we executed the sequential and CUDA implementations 20 times and collected each runtime result. In these collected results we applied the Shapiro-Wilk [19] statistical test to ensure that the obtained sample came from a normally distributed population. By the test decision rule, for a calculated W_c, we must have $W_c > W(0.05; 20) = 0.905$, with the P value greater than $\alpha = 0.05$. Thereby, we can assert with a significance level of 5% that the sample comes from a normal population. When the statistical test gives $W_c < 0.905$, i.e. $\alpha \leq 0.05$, we discard the sample and execute new tests. For the valid samples, we used the mean of the runtime to analyze the experimental behavior of the algorithm. For the EPMST algorithm, we also executed 20 times and used the lesser runtime obtained.

Table I presents the obtained tests results, where each row shows the number of our algorithm iterations, the runtime of sequential implementation, the runtime of CUDA implementation, the runtime of EPMST implementation, speedup of our sequential implementation compared with EPMST, speedup of our CUDA implementation compared with EPMST and speedup of our CUDA implementation compared with our sequential implementation.

TABLE I
TESTS RESULTS

Input Graph	# iterations	Seq. Time (s)	CUDA Time (s)	EPMST Time (s)	Speedup Seq. x EPMST	Speedup CUDA x EPMST	Speedup CUDA x Seq.
graph10a	5	0.804	0.729	0.068	0.085	0.094	1.103
graph10b	4	1.306	0.761	0.260	0.199	0.342	1.716
graph10c	3	1.786	0.816	1.301	0.728	1.594	2.189
graph10d	4	2.066	0.856	3.908	1.892	4.564	2.413
graph10e	3	2.363	0.923	9.517	4.028	10.308	2.559
graph15a	5	1.850	0.773	0.414	0.224	0.535	2.392
graph15b	5	2.786	0.844	1.262	0.453	1.495	3.301
graph15c	3	3.791	0.958	6.807	1.796	7.102	3.955
graph15d	3	4.384	1.077	19.812	4.519	18.403	4.072
graph15e	3	5.051	1.203	48.508	9.604	40.333	4.200
graph15f	2	10.268	2.008	92.278	8.987	45.961	5.114
graph15g	2	14.909	2.7035	205.924	13.812	76.1683	5.515
graph15h	2	21.628	3.868	407.260	18.830	105.303	5.592
graph20a	5	3.061	0.823	0.527	0.172	0.641	3.719
graph20b	4	4.887	0.954	4.057	0.830	4.253	5.123
graph20c	4	6.298	1.161	20.737	3.292	17.867	5.427
graph20d	3	7.597	1.382	61.792	8.134	44.703	5.496
graph20e	3	8.730	1.620	153.033	17.530	94.461	5.389
graph25a	5	4.695	0.903	1.257	0.268	1.393	5.201
graph25b	4	7.408	1.096	9.700	1.310	8.853	6.761
graph25c	4	9.499	1.440	52.785	5.557	36.657	6.597
graph25d	2	10.195	1.786	151.694	14.879	84.946	5.709
graph25e	3	13.456	2.155	414.316	30.791	192.265	6.244
graph30a	6	6.732	0.978	2.657	0.395	2.715	6.881
graph30b	5	10.241	1.247	20.321	1.984	16.290	8.209
graph30c	3	13.202	1.750	104.559	7.920	59.742	7.543
graph30d	3	16.436	2.279	332.890	20.254	146.088	7.213
graph30e	3	21.944	3.318	988.326	45.039	297.865	6.613

By Theorem 2 we know that the algorithm needs $\log n$ iterations in the worst case; however, we can see that in practice these number is much smaller (see the second column of Table I).

With 15,000 vertices, we generated eight graphs with different densities, namely 0.02, 0.05, 0.10, 0.15, 0.20, 0.50, 0.75 and 1.0 (a complete graph). Figure 9 illustrates the runtime of the tested implementations for these graphs. We can see that as the density increases the performance of our algorithm is better. For density 0.02, our implementations have worse results than the EPMST, but above density 0.05 our CUDA implementation already overcomes EPMST time. Our sequential implementation is already better than the EPMST from density 0.1. We can see the corresponding speedup for the tests using input graphs with 15,000 vertices. The speedup of CUDA implementation compared to EPMST increases substantially as the density of the graph is raised.

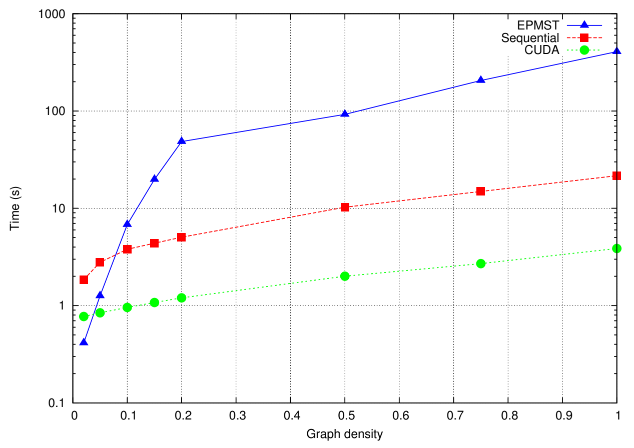

Fig. 9. Running time for graphs with 15,000 vertices.

Analyzing Figure 10, which presents the runtimes for graphs with density 0.02 and 0.1, it is possible to see that EPMST has a better performance for the input graphs with 10,000, 15,000 and 20,000 vertices, but for the input graphs with 25,000 and 30,000 vertices, our CUDA implementation takes less time to execute. For input graphs with density from 0.1, the runtimes of our implementations are better than the EPMST times in practically all tests.

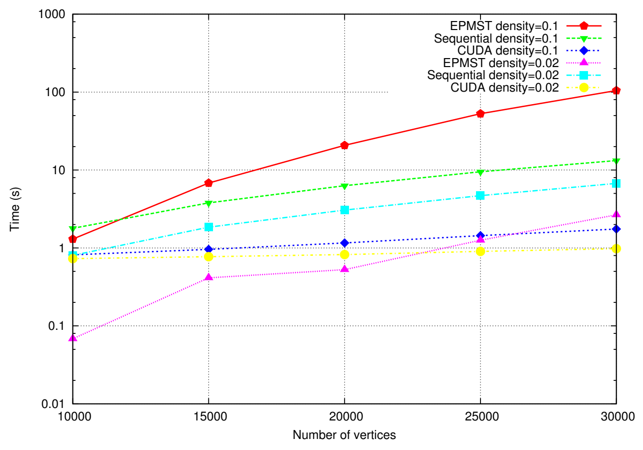

Fig. 10. Running time for graphs with density 0.02 and 0.1.

V. Conclusions and Future Works

In this work, we presented a parallel algorithm for the minimum spanning tree. A sequential and a parallel versions of the algorithm were implemented and compared with the results of other recent efficient algorithm, named Edge Pruned Minimum Spanning Tree (EPMST) [14]. The experiments show that the proposed algorithm presents a good performance for not very sparse graphs, resulting in very good speedups.

As future work, we intend to develop a pruning approach to reduce the input set of edges, which can improve the algorithm results and enable to work with greater input graphs. Another possibility is to evaluate the performance of the algorithm with the use of multiple GPUs to reduce the runtime. We also have the purpose of using real graphs as input data to observe the performance of the algorithm.

Acknowledgment

This research was partially supported by CNPq Proc. No. 482736/2012-7, 30.2620/2014-1 and 465446/2014-0, and Proc. FAPESP 2014/50937-1.

References

[1] R. L. Graham and H. Pavol, “On the history of of the minimum spanning tree problem,” in *Annals of the History of Computing*, vol. 7, no. 1, Jan. 1985, pp. 43–57.

[2] M. Mareš, “The saga of minimum spanning trees,” *Computer Science Review*, vol. 2, no. 3, pp. 165–221, December 2008.

[3] D. Kozen, *The Design and Analysis of Algorithms*. Springer, 1992.

[4] J. H. Reif, “Depth-first search is inherently sequential,” *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, June 1985.

[5] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, “Computing connected components on parallel computers,” *Comm. ACM*, vol. 22, no. 8, pp. 461–464, Aug. 1979.

[6] R. M. Karp and V. Ramachandran, *Handbook of Theoretical Computer Science*. Cambridge, MA: The MIT Press, 1990, vol. A.

[7] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato, “Efficient parallel graph algorithms for coarse grained multicomputers and BSP,” *Algorithmica*, vol. 33, no. 2, pp. 183–200, 2002.

[8] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG ’09, 2009, pp. 167–171.

[9] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, “Scalable parallel minimum spanning forest computation,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’12, 2012, pp. 205–214.

[10] R. Nasre, M. Burtscher, and K. Pingali, “Morph algorithms on GPUs,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 147–156.

[11] A. Chan and F. Dehne, “A note on coarse grained parallel integer sorting,” *Parallel Processing Letters*, vol. 9, no. 4, pp. 533–538, 1999.

[12] A. Kershenbaum and R. V. Slyke, “Computing minimum spanning trees efficiently,” in *Proceedings of the ACM annual conference*, vol. 1. ACM, Aug. 1972, p. 518–527.

[13] E. N. Cáceres, N. Deo, S. Sastry, and J. L. Szwarcfiter, “On finding Euler tours in parallel,” *Parallel Processing Letters*, vol. 3, no. 3, pp. 223–231, 1993.

[14] A. Mamun and S. Rajasekaran, “An efficient minimum spanning tree algorithm,” in *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE, 2016, pp. 1047–1052.

[15] V. Osipov, P. Sanders, and J. Singler, “The filter-kruskal minimum spanning tree algorithm,” in *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics (SIAM), 2009, pp. 52–61.

[16] R. Johnsonbaugh and M. Kalin, “A graph generation software package,” in *Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education*, vol. 23, no. 1, Mar. 1991, pp. 151–154.

[17] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[18] R. C. Prim, “Shortest connection networks and some generalizations,” *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[19] S. S. Shaphiro and M. B. Wilk, “An analysis of variance test for normality,” *Biometrika*, vol. 52, no. 3/4, pp. 591–611, Dec. 1965.