Journal Title
XX(X):1–27

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

New BSP/CGM algorithms for spanning trees

Jucele França de Alencar Vasconcellos[1], Edson Norberto Cáceres[1], Henrique Mongelli[1], Siang Wun Song[2], Frank Dehne[3] and Jayme Luiz Szwarcfiter[4]

Abstract

Computing a spanning tree (ST) and a minimum spanning tree (MST) of a graph are fundamental problems in Graph Theory and arise as a subproblem in many applications. In this paper, we propose parallel algorithms to these problems. One of the steps of previous parallel MST algorithms relies on the heavy use of parallel list ranking which, though efficient in theory, is very time-consuming in practice. Using a different approach with a graph decomposition, we devised new parallel algorithms that do not make use of the list ranking procedure. We proved that our algorithms are correct, and for a graph $G = (V, E)$, $|V| = n$ and $|E| = m$, the algorithms can be executed on a BSP/CGM model using $O(\log p)$ communications rounds with $O(\frac{n+m}{p})$ computation time for each round. To show that our algorithms have good performance on real parallel machines, we have implemented them on GPU (Graphics Processing Unit). The obtained speedups are competitive and showed that the BSP/CGM model is suitable for designing general purpose parallel algorithms.

Keywords

spanning tree, minimum spanning tree, parallel algorithm, BSP/CGM model, GPU

Introduction

Computing a spanning tree, 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 Hell

[1]College of Computing, Federal University of Mato Grosso do Sul, Campo Grande, MS, Brazil
[2]Institute of Mathematics and Statistics, University of São Paulo, São Paulo, SP, Brazil
[3]School of Computer Science, Carleton University, Ottawa, Canada
[4]NCE, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

Corresponding author:
Jucele França de Alencar Vasconcellos, Faculdade de Computação, Universidade Federal de Mato Grosso do Sul, Cidade Universitária, CEP 79070-900, Campo Grande - MS - Brazil
Email: jucele@facom.ufms.br

(1985) point out the importance of the minimum spanning tree 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 (2008).

The sequential algorithms use depth-first or breadth-first search to solve these problems efficiently (Kozen, 1992). The parallel solutions for these problems, however, do not use these search methods because they are not easy to parallelize (Reif, 1985). They are based instead on the approach proposed by Hirschberg et al. (1979), 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 (Karp and Ramachandran, 1990). 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 (Karp and Ramachandran, 1990).

The spanning tree algorithm presented by Cáceres et al. (2004) uses as input a bipartite graph since a general graph can be transformed into a corresponding bipartite graph. However, they use a graph decomposition that does not seem suitable to compute the smallest edges of the spanning tree and consequently compute the minimum spanning tree of the input graph.

In this paper, we propose an algorithm in which the first step creates a corresponding bipartite graph by dividing each edge of the original graph and adding a new vertex in the middle of each edge. We observed that the corresponding bipartite graph thus obtained from the original graph is a special bipartite graph where all the vertices of one of the two sets of vertices of the bipartite graph have degree two. Observing that the degree of all added vertices (in the middle of each edge) in the corresponding bipartite graph is two, we devised a new approach to compute the spanning tree of the graph by selecting the smallest edges too, which gave us the minimum spanning tree of the graph.

We propose, in this paper, an approach to obtain a spanning tree and a minimum spanning tree of a given graph that does not need to solve the Euler tour or the list ranking problem.

The parallel algorithms, described in this paper, were designed using the BSP/CGM model and use $O(\log p)$ communication rounds with $O(\frac{n+m}{p})$ local computation time, where p is the number of processors. The algorithm to solve the minimum spanning tree problem was presented in the eighth Workshop on Applications for Multi-Core Architectures (WAMCA 2017) (Vasconcellos et al., 2017), without proofs and many important details. However, here we present a new way to formulate the main results presented by Cáceres et al. (2004), and provide the proofs for both the spanning tree and the minimum spanning tree algorithms. We also present more experimental results.

We also show that our parallel algorithms have good performance when implemented on real parallel machines. We have tested them on several GPU's. The obtained results showed that the algorithms scaled well and had competitive speedups.

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 of the minimum spanning tree algorithm with a recently published efficient algorithm (Mamun and Rajasekaran, 2016), called the Edge Pruned Minimum Spanning Tree (EPMST). Thist algorithm presents better performance compared to the Filter-Kruskal solution (Osipov et al., 2009). 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 the next section, we present the proposed parallel algorithms. We discuss their correctness in the third section. Experimental results are shown in the fourth section and, finally, conclusions are given in the last section.

Related works

Dehne et al. (2002) 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 Dehne et al. (2002) 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.

Cáceres et al. (2004) introduce another solution to the spanning tree and connected component problem based on the integer sorting. This algorithm, also explained in an expanded article (Cáceres et al., 2003), does not need to solve the Euler tour or the list ranking problem. Cáceres et al. (2010), considering a bipartite graph obtained from a transformation of the input graph, present experimental results for this solution on a Beowulf cluster and on a grid.

Some works proposing parallel solutions for MST using GPGPU can be found in the literature, such as those shown by Vineet et al. (2009); Nobari et al. (2012); Nasre et al. (2013). The algorithm proposed by Nobari et al. (2012) is inspired by Prim's algorithm. Vineet et al. (2009) and Nasre et al. (2013) presented parallel solutions based on Borůvka's algorithm.

The main ideas of the parallel algorithms to obtain a spanning tree (ST) and a minimum spanning tree (MST)

We initiate this section presenting basic concepts used in the algorithms, followed by the basic algorithm and details of the steps.

Preliminary concepts

Now we describe some basic concepts used in our algorithms. Consider $G = (V, E)$ a **graph** 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, w_{ij}, v_j), where v_i and v_j are vertices of V and w_{ij} is the edge weight. For simplicity, we sometimes also write an edge as (v_i, v_j) and the weight w_{ij} is implied. For the spanning tree problem, we ignore the weight of the edges.

A **path** in G is a sequence of edges $(v_1, v_2), (v_2, v_3), (v_3, v_4) \ldots, (v_{n-1}, v_n)$ connecting distinct vertices $v_1, \ldots, v_n$ of G. A **cycle** is a path connecting different vertices $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A **connected graph** has at least one path for every vertex pair v_i, v_j, $1 \leq i \neq j \leq n$ in V. A **tree** $T = (V, E)$ is a connected graph with no cycles. A **forest** is a set of trees. A **spanning tree** of $G = (V, E)$ is a tree $T = (V, E')$ which includes all vertices of G and is a subgraph of G, that is, all edges of T belong to G, $E' \subset E$. A spanning tree of G can also be defined as the maximal set of G edges, and $|E'| = |V| - 1$, which contains no cycle. A **minimum spanning tree** is a spanning tree with the minimum possible total edge weight.

A **bipartite graph** is a graph whose vertices can be divided into two disjoint and independent sets V_1 and V_2 such that every edge of the graph connects a vertex in V_1 to one vertex in V_2. Our algorithms read the input graph $G = (V, E)$ and create a bipartite graph $H = (V, U, E')$ based on G, where V and U form the vertex set. Details to obtain the transformed bipartite graph from the original input graph will be shown shortly. The concept denominated **strut** is used in the algorithms, but different from the definition presented in Cáceres et al. (1993). In the context of the present paper, a **strut**, represented by S, is defined as a forest of $H = (V, U, E')$, such that each vertex $v_i \in V$ is incident in S to exactly one edge (v_i, u_j) of E', such that $v_i \in V$ and $u_j \in U$. The details to choose the strut edges depend on whether we are considering the spanning tree problem or the minimum spanning tree problem, and will be given later on. A vertex u_j is considered a **zero-difference vertex** in S if $d_H(u_j) - d_S(u_j) = 0$, where $d_H(u_j)$ denotes the degree of u_j in H and $d_S(u_j)$ the degree of u_j in S.

Our parallel algorithms were designed using the BSP/CGM model (Valiant, 1990; Dehne et al., 1996). This model considers a set of p processors, each one having a local memory of size $O(n/p)$, where n is the input size. An algorithm in this model performs a set of local computation steps (super steps) alternating with global communication phases, separated by a synchronization barrier. The cost of the communication considers the number of super steps required to execute the algorithm. In this model, the parallel algorithm can be executed in a CPU (one node) or in a GPU, since all the tasks in each computation round are independent.

The BSP/CGM model is appropriate for the design and analysis of parallel algorithms where there is much communication between the processes. This is a characteristic of irregular problems, i.e., the input in each round of the program changes and the processors need the information that different processors computed in the last round. The spanning tree and minimum spanning tree problems are in this class, which motivates the use of the model to predict the behavior and complexity of the algorithm.

The mapping of a BSP/CGM algorithm and a distributed memory environment is straightforward. The super steps consist of computation and communication rounds, the computation is done in the nodes and the communication through a network. When using the GPGPU environment, we have a shared memory environment, and we can see the invocations of each CUDA kernel as a super step of the BSP/CGM model as stated by Lima et al. (2016). Parallel execution of each kernel by the various threads created by CUDA constitutes a computation round, which can be alternated by communication between the threads through memory and communication between the GPU and the CPU. Considering this approach, we can predict that our algorithm will have a compatible performance, when implemented on a GPGPU, to its theoretical behavior.

The basic parallel algorithm

Algorithm 1 gives the main ideas of the proposed parallel algorithms. It uses the SIMD paradigm, that is, the steps are executed by several processors finding the solution collaboratively. The algorithms for spanning tree and minimum spanning tree are very similar, both following the Algorithm 1. The major difference consists in the way we construct the strut. As we expect to execute $O(\log p)$ rounds on BSP/CGM model, we establish that, after $\log p$ rounds, if the spanning tree or the minimum spanning tree is not found, the algorithm continues the processing locally on the CPU.

Creating a bipartite graph for the input graph (Algorithm 1 - line 3 to line 9)

To find a spanning tree (or minimum spanning tree) of a given graph, we first create a bipartite graph corresponding to the input graph. This step is executed in parallel and can be done by adding a new vertex on each edge (line 4) of the original graph, thereby subdividing each original edge into two new

Algorithm 1 Spanning Tree (ST)/Minimum Spanning Tree (MST)

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 spanning tree (or a minimum spanning tree) of G whose edges are in $SolutionEdgeSet$.

1: *SolutionEdgeSet* := empty.
2: // Creation of the bipartite graph $H = (V_H, U_H, E_H)$ corresponding to input graph G.
3: **for each** $(e_i(v_a, v_b) \in E)$ **in parallel do**
4:     Add vertex u_i to U_H // u_i is a new vertex associated to edge e_i
5:     Add edge (v_a, u_i, w_{ab}) and (v_b, u_i, w_{ab}) to E_H
6: **end for**
7: **for each** $(v_i \in V)$ **in parallel do**
8:     Add vertex v_i to V_H
9: **end for**
10: // Finding the ST or MST solution.
11: *condition* := true
12: r := 0
13: **while** (($r < \log p$) **AND** (*condition*)) **do** // p is the number of processors
14:     // Obtaining the strut.
15:     **for each** $(u_i \in U_H)$ **in parallel do**
16:         $d_S(u_i) = 0$
17:     **end for**
18:     **for each** $(v_i \in V_H)$ **in parallel do**
19:         Find the lightest edge among all edges of v_i
20:     **end for**
21:     **for each** $(v_i \in V_H)$ **in parallel do**
22:         // considering that (v_i, u_j) is the lightest edge among all edges of v_i
23:         $d_S(u_j) = d_S(u_j) + 1$ // atomic function
24:     **end for**
25:     // Adding edges to the SolutionEdgeSet
26:     **for each** $(u_j \in U_H)$ **in parallel do**   // u_j is the vertex that corresponds to the edge e_j of the original graph, created in line 4
27:         **if** $(d_S(u_j) \geq 1)$ **then**
28:             Add edge e_j, where e_j is $original\_edge(u_j)$, to *SolutionEdgeSet*
29:         **end if**
30:     **end for**
31:     // Calculating the number of zero-difference vertices
32:     *numdiff* = 0   // *numdiff* is the number of zero-difference vertices of U
33:     **for each** $(u_j \in U_H)$ **in parallel do**
34:         **if** $(d_S(u_j) == 2)$ **then**
35:             *numdiff* = *numdiff* +1   // atomic function
36:         **end if**
37:     **end for**
38:     **if** (*numdiff* == 1) **then**
39:         *condition* := false
40:     **else**   // Compaction of graph H
41:         **for each** (vertex u_t, with edges (v_i, u_t) and (v_j, u_t) in E_H) **in parallel do**
42:             **if** $(d_S(u_t) \geq 1)$ **then**
43:                 Contract the adjacent vertices in the strut into a super-vertex v_i, being v_i the smallest label among them
44:                 Update to v_i the edges of E_H incident to the contracted vertices
45:                 Eliminate the other contracted vertices from V_H
46:             **end if**
47:         **end for**
48:         **for each** (vertex u_t, with edges (v_a, u_t) and (v_b, u_t) in E_H) **in parallel do**
49:             **if** $(v_a == v_b)$ **then**   // v_a and v_b were joined in the previous "for"
50:                 Eliminate u_t from U_H
51:                 Eliminate (v_a, u_t) and (v_b, u_t) from E_H
52:             **end if**
53:         **end for**
54:     **end if**
55:     $r := r + 1$
56: **end while**

edges (line 5). 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 create a corresponding bipartite graph H by adding 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 $H = (V, U, E')$, in the algorithm denoted by $H = (V_H, U_H, E_H)$, thus obtained is bipartite. Remember that the weights are only used to find a minimum spanning tree. In the algorithm to compute a spanning tree, we ignore the weight of the edges.

Figure 1 shows an example of this step. On the left, we have the original graph $G = (V, E)$ with $V = \{1, 2, \ldots, 5\}$. In the middle, we represent the created bipartite graph $H = (V_H, U_H, E_H)$ with $U_H = \{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. And on the right, the same bipartite graph is shown in another vision, where the two vertex sets V_H and U_H are illustrated separately. Observe that any vertex $u_k \in U_H$ which, by construction, was created on an edge (v_i, v_j) 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.

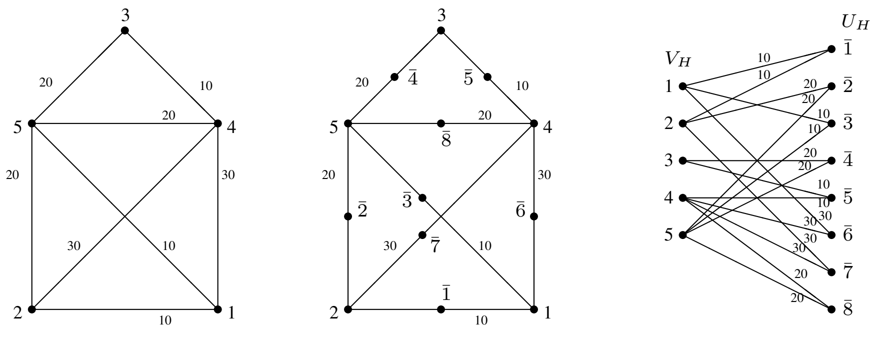

Figure 1. On the left, the original graph $G = (V, E)$, in the middle, the created bipartite graph $H = (V_H, U_H, E_H)$ and, on the right the same bipartite graph separating V_H and U_H.

Obtaining the strut in the calculation of the spanning tree (Algorithm 1 - line 15 to line 24)

Consider the created bipartite graph $H = (V_H, U_H, E_H)$ with vertex sets $V_H = \{v_1, v_2, \cdots, v_n\}$ and $U_H = \{u_1, u_2, \cdots, u_m\}$, and edge set E_H where each edge joins one vertex of V_H and one vertex of U_H. For simplicity, let each vertex v_i of V_H be represented by i, i.e. $V_H = \{1, 2, \ldots, n\}$. Likewise, let us represent each vertex u_j of U_H by $\bar{j}$, i.e. $U = \{\bar{1}, \bar{2}, \ldots, \bar{m}\}$. To compute a spanning tree, the strut is obtained as follows. Among all edges (v_i, u_j) incident to v_i in H, select the edge (v_i, u_k) with the smallest u_k.

Let us give an example. Consider the corresponding bipartite graph $H = (V_H, U_H, E_H)$ of Figure 1, where $V_H = \{v_1, v_2, \ldots, v_5\}$ = $\{1, 2, \ldots, 5\}$ and $U_H = \{u_1, u_2, \ldots, u_8\}$ = $\{\bar{1}, \bar{2}, \ldots, \bar{8}\}$. For each vertex v_i of V_H, consider all the edges (v_i, u_j) incident to v_i. Table 1 illustrates five groups of edges $(v_i\ u_j)$, one group for each $v_i = 1, 2, \ldots, 5$.

Table 1. Edges of graph H in Figure 1. The edges with the smallest u_k chosen to compose the spanning tree are marked with an asterisk ("*").

$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$	$(v_i$	$u_j)$
(1	$\bar{6}$)	* (2	$\bar{1}$)	* (3	$\bar{4}$)	(4	$\bar{6}$)	* (5	$\bar{2}$)
(1	$\bar{3}$)	(2	$\bar{7}$)	(3	$\bar{5}$)	(4	$\bar{7}$)	(5	$\bar{3}$)
* (1	$\bar{1}$)	(2	$\bar{2}$)			(4	$\bar{8}$)	(5	$\bar{8}$)
						* (4	$\bar{5}$)	(5	$\bar{4}$)

By definition, to find a spanning tree, the strut S is composed of the edges with the smallest u_k, for each vertex, marked with "*" in Table 1, namely, $(1\ \bar{1})$, $(2\ \bar{1})$, $(3\ \bar{4})$, $(4\ \bar{5})$, and $(5\ \bar{2})$. In Figure 2, the obtained strut S is illustrated by the dotted lines. For notation purposes, we denote an edge in a strut as a *strut-edge*.

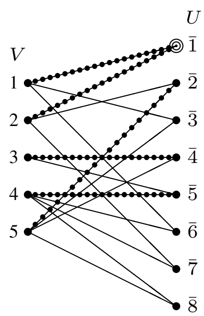

Figure 2. The strut S represented by dotted lines. Vertices $\bar{1}$, $\bar{2}$, $\bar{4}$ and $\bar{5}$ have at least one incident *strut-edge*. Vertex $\bar{1}$ has two incident *strut-edge* and is a zero-difference vertex. (spanning tree case)

Obtaining the strut in the calculation of the minimum spanning tree (Algorithm 1 - line 15 to line 24)

Considering the calculation of a minimum spanning tree, now the strut is obtained 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 see an example of strut construction considering the corresponding bipartite graph $H = (V_H, U_H, E_H)$ of Figure 1. For each vertex v_i of V, consider all the edges (v_i, u_j) incident to v_i illustrated in Table 2 To illustrate this, we extend the notation of an edge (v_i, u_j) by adding the weight w_{ij} in the middle, as $(v_i\ w_{ij}\ u_j)$. We use Table 2 to ilustrate five groups of edges $(v_i\ w_{ij}\ u_j)$, one group for each $v_i = 1, 2, \ldots, 5$.

By definition used in the calculation of a minimum spanning tree, the strut S is composed of the lightest edge, considering the weight and the label of vertex u. Table 2 shows the chosen edges for each

Table 2. Edges of graph in Figure 1, considering the calculation of a minimum spanning tree. The smallest edges chosen to compose the spanning tree are marked with an asterisk ("*").

$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$	$(v_i$	w_{ij}	$u_j)$
(1	30	$\bar{6}$)	* (2	10	$\bar{1}$)	(3	20	$\bar{4}$)	(4	30	$\bar{6}$)	(5	20	$\bar{2}$)
(1	10	$\bar{3}$)	(2	30	$\bar{7}$)	* (3	10	$\bar{5}$)	(4	30	$\bar{7}$)	* (5	10	$\bar{3}$)
* (1	10	$\bar{1}$)	(2	20	$\bar{2}$)				(4	20	$\bar{8}$)	(5	20	$\bar{8}$)
									* (4	10	$\bar{5}$)	(5	20	$\bar{4}$)

vertex marked with "*", 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.

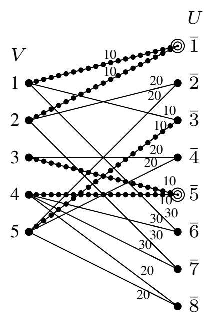

Figure 3. The strut S represented by dotted lines. Vertices $\bar{1}$, $\bar{3}$ and $\bar{5}$ have at least one incident *strut-edge*. Vertices $\bar{1}$ and $\bar{5}$ have two and are zero-difference vertices. (minimum spanning tree case)

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 Discussion of the algorithms 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_H has weight smaller than the weight w_{ij}.

Adding edges to the SolutionEdgeSet (Algorithm 1 - line 26 to line 30)

Consider the solution set $SolutionEdgeSet$ with the edges of the desired spanning tree or minimum spanning tree. Initially this set is empty (line 1). 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 2. Vertices $\bar{1}, \bar{2}, \bar{4}$ and $\bar{5}$ are incident to strut-edges. Thus we add to the solution set $original\_edge(\bar{1}), original\_edge(\bar{2}), original\_edge(\bar{4})$ and $original\_edge(\bar{5})$ (respectively edges (1, 2), (2, 5), (3, 5) and (3, 4)). Figure 4 shows the edges added to the solution edge set so far, in the algorithm first round. The added edges are shown as dotted lines.

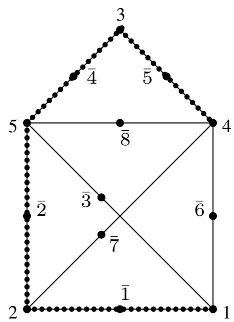

Figure 4. Solution edge set composed by *strut-edges*, after the algorithm first round. (spanning tree case)

For the example of Figure 3, vertices $\bar{1}, \bar{3}$ and $\bar{5}$ are incident to strut-edges. So, we add to the solution edge set $original\_edge(\bar{1}), original\_edge(\bar{3})$ and $original\_edge(\bar{5})$ (respectively edges (1, 2), (1, 5) and (3, 4)). Figure 5 shows the edges added to the solution edge set, after the algorithm first round, as dotted lines.

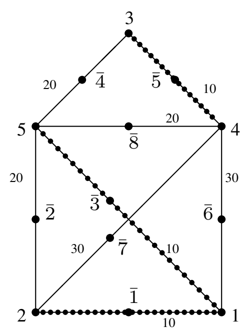

Figure 5. Intermediate solution edge set, resulting from the algorithm first round, composed by *strut-edges*. (minimum spanning tree case)

Calculating the number of zero-difference vertices (Algorithm 1 - line 32 to line 37)

Consider the bipartite graph $H = (V_H, U_H, E_H)$ and a strut S. Let u_j denote a vertex of U_H. 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 $u_j \in U_H$ 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 2 the vertex $\bar{1}$ has two strut-edges, and thus it is zero-difference vertex. In Figure 3 the vertices $\bar{1}$ and $\bar{5}$ both have two strut-edges, and thus they are zero-difference vertices. In Figure 2 and Figure 3 the zero-difference vertices are enclosed by double circles around them.

If there is only one zero-difference vertex in the obtained strut, the problem is solved. See the proof of this in Section Discussion of the algorithms. So, the solution edge set of Figure 4 is complete and represents a spanning tree for the input graph. However, the solution edge set presented in Figure 5 is not a minimum spanning tree yet. The algorithm will need another round to find the complete solution.

Compacting the bipartite graph (Algorithm 1 - line 41 to line 53)

When there are two or more zero-difference vertices, we must compress the bipartite graph for the execution of a new iteration (round) of the algorithm. To do this, we must analyze each vertex $u_t \in U_H$ 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_H that are adjacent to contracted vertices of V_H. This is illustrated with the example of Figure 3, where vertices $\bar{1}$, $\bar{2}$, $\bar{3}$ and $\bar{5}$ must be suppressed. Figure 6 (a) 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 $H' = (V_{H'}, U_{H'}, E'_{H'})$, resulting from compaction, $|V_{H'}|$ is equal to the number of zero-difference vertices in the strut S (see Lemma 3 in Section Discussion of the algorithms).

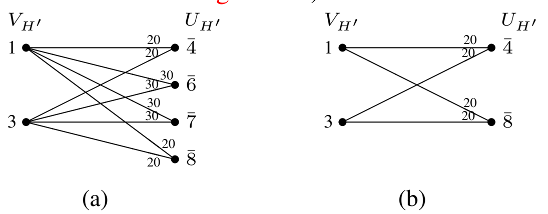

Figure 6. (a) Bipartite graph H' after compaction. (b) Bipartite graph H' after optimized compaction. (minimum spanning tree case)

The resulting graph after the compaction can have multiple $U_{H'}$ vertices that are adjacent to the same pair of vertices of $V_{H'}$. As in the example of Figure 6 (a), 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 $U_{H'}$ vertices that have heavier edges. Figure 6 (b) shows the resulting compact graph for this optimization.

Finalizing the algorithm (another iteration of Algorithm 1)

In the compaction step, we update the vertices and edges of H, where called H'. In the second iteration (round), the algorithm obtains a new strut for H', updates the solution edges 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 $V_{H'}$, consider all the edges (v_i, u_j) incident to v_i and choose the smallest edges. Considering the graph of Figure 6 (a), the result of this choice is illustrated in Table 3, 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.)

In the BSP/CGM model, we expect to execute the algorithm in $O(\log p)$ rounds, so, we establish that, after $\log p$ rounds, if the spanning tree or the minimum spanning tree is not found, the algorithm continues the processing locally in the CPU.

Table 3. Edges of graph in Figure 6 (a)

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

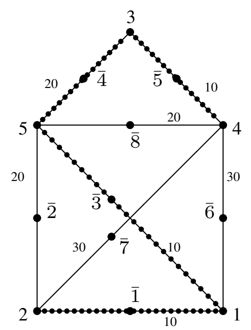

Figure 7. The resulting minimum spanning tree for the example of Figure 1. (minimum spanning tree case)

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 $(3\ \ 20\ \ \bar{4})$. Having obtained the strut, we add $original\_edge(\bar{4})$ to the solution set. Observe that now we have only one zero-difference vertex and thus the algorithm terminates. Figure 7 shows the obtained minimum spanning tree of the original graph. The edges added to the solution set are shown as dotted lines.

Parallel algorithm to compute a spanning tree (ST)

The Algorithms 2 and 3 present more details of the parallel GPU algorithm to compute a spanning tree. As previously mentioned, the algorithm was designed using the BSP/CGM model (Valiant, 1990; Dehne et al., 1996), and performs a set of local computation steps (super steps) alternating with global communication phases, separated by a synchronization barrier. Notice that in Algorithm 1, we can identify many steps that are executed in parallel, for example, from line 3 to line 9. These lines are an example of a super step of our algorithm. The Algorithm 2 presents the kernel calls representing the steps described in Algorithm 1. The Algorithm 3 details some of the kernel functions called in Algorithm 2.

Algorithm 2 High Level Implementation

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

SolutionEdgeSet := empty.
// Creation of the bipartite graph $H = (V_H, U_H, E_H)$ corresponding to input graph G.
copy original graph vertices and edges to GPU
call CreateBipartiteGraph()
// Finding the ST solution.
condition := true
r := 0
while ($r < \log p$) **AND** (*condition*) **do** // p is the number of processors
  // Obtaining the strut S.
  call InicializeLightestEdges()
  call FindLightestEdges()
  // Obtaining the strut and calculating the number of zero-difference vertices
  numdiff = 0
  call ObtainStrutANDCalculateNumdiff()
  // Adding edges with $d_S(u) \geq 1$ to the SolutionEdgeSet
  call UpdateSolutionSet()
  if (*numdiff* == 1) **then**
    condition := false
  else // Compaction of graph H
    call ComputeConnectedComponents()
    call MarkEdgesForDeletion()
    call ReorganizeEdges()
    call UpdateVerticesV()
    call UpdateVerticesU()
    call UpdateEdges()
  end if
  $r := r + 1$
end while
copy $SolutionEdgeSet$ to CPU

Parallel algorithm to compute a minimum spanning tree (MST)

The proposed parallel algorithm used to compute a minimum spanning tree is almost the same presented to calculate the spanning tree in Algorithm 2. The difference between them consists in the way we select the edges to compound the strut, using the weights. Algorithm 4 makes clear this distinction detailing the Procedures $CreateBipartiteGraph$ and $FindLightestEdges$. All other steps of Algorithm 2 are the same for the minimum spanning tree.

Algorithm 3 ST Algorithm - Description of the Procedures

Procedure CreateBipartiteGraph
// $thread\_id$ is the identification of each thread during a kernel execution
// nOG is the number of vertices of the original graph
// mOG is the number of edges of the original graph
// consider each edge of the original graph represented by vertices v1 and v2
// consider each edge of the bipartite graph represented by vertices v and u
if $(thread\_id < mOG)$ **then**
  if $(thread\_id < nOG)$ **then**
    $V_H[thread\_id].id$ = $V[thread\_id].id$;
  end if
  $U_H[thread\_id].id = thread\_id$;
  $E_H[2 * thread\_id].v = E[thread\_id].v1$;
  $E_H[2 * thread\_id].u = U_H[thread\_id]]).id$;
  $E_H[2 * thread\_id + 1].v = E[thread\_id].v2$;
  $E_H[2 * thread\_id + 1].u = U_H[thread\_id]]).id$;
end if
EndProcedure

Procedure FindLightestEdges
// $thread\_id$ is the identification of each thread during a kernel execution
// mBG is the number of edges of the bipartite graph
// $lightest\_edge[v]$ stores the identification of the lightest edge for the vertex v
if $(thread\_id < mBG)$ **then**
  $v = E_H[thread\_id].v$;
// Begin atomic function
  if $(E_H[lightest\_edge[v]].u > E_H[thread\_id].u)$ **then**
    $lightest\_edge[v] = thread\_id$
  end if
// End atomic function
end if
EndProcedure

Procedure ObtainStrutANDCalculateNumdiff
// $thread\_id$ is the identification of each thread during a kernel execution
// nGB is the number of vertices of the bipartite graph
if $(thread\_id < nBG)$ **then**
  $v = V_H[thread\_id]$;
  $S[thread\_id].v = v$;
  $S[thread\_id].u = E_H[lightest\_edge[v]].u$;
  $d_S[E_H[lightest\_edge[v]].u] + +$;
  if $(d_S[E_H[lightest\_edge[v]].u] == 2)$ **then**
    $numdiff + +$
  end if
end if
EndProcedure

Procedure UpdateSolutionEdgeSet
// $thread\_id$ is the identification of each thread during a kernel execution
// mGB is the number of edges of the bipartite graph
if $(thread\_id < (mBG/2))$ **then**
  if $(d_S[thread\_id] \geq 1)$ **then**
  // Begin atomic function
    $SolutionEdgeSet[SolutionLenght] = U_H[thread\_id].id$
    $SolutionLenght + +$
  // End atomic function
  end if
end if

Algorithm 4 MST Algorithm - Description of the Procedures

```
Procedure CreateBipartiteGraph
// thread_id is the identification of each thread during a kernel execution
// nOG is the number of vertices of the original graph
// mOG is the number of edges of the original graph
// consider each edge of the original graph represented by vertices v1 and v2 and weight w
// consider each edge of the bipartite graph represented by vertices v and u and weight w
if (thread_id < mOG) then
    if (thread_id < nOG) then
        V_H[thread_id].id = V[thread_id].id;
    end if
    U_H[thread_id].id = thread_id;
    E_H[2 * thread_id].v = E[thread_id].v1;
    E_H[2 * thread_id].w = E[thread_id].w;
    E_H[2 * thread_id].u = U_H[thread_id]]).id;
    E_H[2 * thread_id + 1].v = E[thread_id].v2;
    E_H[2 * thread_id + 1].w = E[thread_id].w;
    E_H[2 * thread_id + 1].u = U_H[thread_id]]).id;
end if
EndProcedure

Procedure FindLightestEdges
// thread_id is the identification of each thread during a kernel execution
// mBG is the number of edges of the bipartite graph
// lightest_edge[v] stores the identification of the lightest edge for the vertex v
if (thread_id < mBG) then
    v = E_H[thread_id].v;
// Begin atomic function
    if (E_H[lightest_edge[v]].w > E_H[thread_id].w) OR
     ((E_H[lightest_edge[v]].w == E_H[thread_id].w) AND (E_H[lightest_edge[v]].u > E_H[thread_id].u)) then
        lightest_edge[v] = thread_id
    end if
// End atomic function
end if
EndProcedure
```

Discussion of the algorithms

In this section we address the correctness of the proposed algorithms. As noted earlier, we now assume, without loss of generality, that all the weights of the edges $e \in E$ of the input graph $G = (V, E)$ to be different.

Lemma 1. *Consider a connected graph $G = (V, E)$. Let $H = (V, U, E')$ be the correspondent bipartite graph and S be a strut in U, obtained at Procedure ObtaingStrut of Algorithm 3. Let G' be the graph obtained by adding the edges associated to vertices u (i.e. $original\_edge(u)$) from U such that $d_S(u) \geq 1$ (line 53 of Algorithm 3). Then G' is acyclic. More over, if S contains exactly one zero-difference vertex then G' is a spanning tree of G.*

Proof. By definition of the strut, S is a forest in $H = (V, U, E')$, so that there is only one incident edge of E' for each vertex $v_i \in V$. The selected edge (v_i, u_a) is the one with the smallest label u_a among the vertices of U connected to v_i. For the vertex $u_x \in U$ with the smallest label, both incident edges will be in S, so S has at least one zero-difference vertex.

Therefore, we can conclude that there will be at most $|V| - 1$ vertices $u \in U$ with degree equal to or greater than 1 in S or $d_S(u) \geq 1$. This means that at most $|V| - 1$ edges will be added to G', an edge incident on each vertex of V, resulting in an acyclic graph. If S has only a zero-difference vertex, we wil have $|V| - 2$ vertices with degree 1. Each one is associated to an edge in the generated graph G', resulting in a spanning tree of G. □

Lemma 2. *Consider a connected graph $G = (V, E)$. Let $H = (V, U, E')$ be the correspondent bipartite graph and S be a strut in U, obtained at Procedure $ObtaingStrutANDCalculateNumdiff$ of Algorithm 3, after Procedure $FindLightestEdges$ of Algorithm 4 to choose the edges. Let G' be the graph obtained by adding the edges associated to vertices u (i.e. $original\_edge(u)$) from U such that $d_S(u) \geq 1$. Then G' is acyclic. More over, if S contains exactly one zero-difference vertex then G' is a spanning tree of G.*

Proof. This proof is basically the same as presented for Lemma 1. By definition, S is a forest in $H = (V, U, E')$ such that there is exactly one edge of E' incident to each vertex $v_i \in V$, being chosen the edge (v_i, u_x) with the smallest weight (line 28 of Algorithm 4). Let w be the smallest weight considering all edges of G and, consequently, of H. Considering the strut construction process, there will be at least one vertex $u_t \in U$ that is incident to edges with weight w. For this vertex u_t both its incident edges will be in S, thus S has at least one zero-difference vertex.

Therefore, we can conclude that there will be at most $|V| - 1$ vertices u of U with degree equal to or greater than 1 in S, or $d_S(u) \geq 1$. This means that at most $|V| - 1$ edges will be added to G', one incident edge in each vertex of V, resulting in an acyclic graph. If S has only a zero-difference vertex, we wil have $|V| - 2$ vertices with degree 1. Each one is associated to an edge in the generated graph G', resulting in 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'.

Before we prove the following theorem, we give some definitions. A cut$(W, V\backslash W)$ of a graph $G = (V, E)$, with $W \subseteq V$, is a partition of V. An edge of E *crosses* the cut$(W, V\backslash W)$ if one of its endpoints is in W and the other endpoint is in $V\backslash W$. Let T be a spanning tree of G. Then the removal of any edge $e \in T$ will result in the components $(W, V\backslash W)$, where one endpoint of e is in W and the other in $V\backslash W$.

Theorem 1. *Consider a connected graph $G = (V, E)$. Let $H = (V, U, E')$ be the correspondent bipartite graph and S be a strut in U, obtained at Procedure $ObtaingStrutANDCalculateNumdiff$ of Algorithm 3, after Procedure $FindLightestEdges$ of Algorithm 4 to choose the edges. Let G' be the graph obtained by adding the edges associated to vertices u (i.e. $original\_edge(u)$) from U such that $d_S(u) \geq 1$ (line 53 of Algorithm 3). If S contains exactly one zero-difference vertex then G' is a minimum spanning tree of G.*

Proof. By Lemma 2 it is known that G' is a spanning tree of G. Consider a vertex $v_i \in V$ and the edge $(v_i, v_j, w) \in E$ such that the weight of $(v_i, v_j, w) \in E$ is the smallest among all edges incident to v_i. By step 53 of Algorithm 3, edge (v_i, v_j, w) is added to the set of edges of the spanning tree. Consider the cut $(\{v_i\}, V \backslash \{v_i\})$. Assume by contradiction that edge (v_i, v_j, w) is not part of the minimum spanning tree. Then there is another edge $(v_i, v_k, z) \in E$, among those edges that cross the cut, that connects v_i to the minimum spanning tree. However, the weight of edge (v_i, v_k, z) is greater than that of edge (v_i, v_j, w). Therefore, if we remove edge (v_i, v_k, z) and add edge (v_i, v_j, w), the total edge weights would be smaller. This is a contradiction. Therefore, we conclude that G' is a minimum spanning tree of G. $\square$

Lemma 3. *Let V_H and U_H be the partitions of H right before the compaction (described in steps 20-24 of Algorithm 2) and let $V_{H'}$ and $U_{H'}$ be the partitions of the compacted graph H' right after step 24. Let k be the number of zero-difference vertices in the strut S obtained using the Procedure $ObtaingStrutANDCalculateNumdiff$, then the number of vertices in $V_{H'}$ is k.*

Proof. Algorithm 2 adds to the solution set any edge associated to a vertex $u \in U$ such that $d_S(u) \geq 1$. With this, all vertices of V that are interconnected by the added edges will be united or combined into a single component, in the compaction step. Each such component will be a vertex of $V_{H'}$. Thus, each zero-difference vertex of U represents a new vertex of $V_{H'}$ in the compacted graph and, therefore, $V_{H'}$ will have at least k vertices, i.e. $|V_{H'}| \geq k$.

We now prove $|V_{H'}| = k$. Suppose, by contradiction, $|V_{H'}| > k$. We have k zero-difference vertices in S, that will give rise, after compaction, to k vertices in $V_{H'}$. (Notice that $2k$ vertices of V are required to produce the k zero-difference vertices in S.) If $|V_{H'}| > k$, then we have at least one vertex of $V_{H'}$ that is formed by vertices of V that are not interconnected to zero-difference vertices of S. This means there must exist x vertices of V, $1 \leq x \leq |V| - 2k$, identified as vertices of the set $V_x = \{v_1, v_2, \ldots, v_x\}$, that are connected in the strut to x vertices of U, identified as vertices of the set $U_x = \{u_1, u_2, \ldots, u_x\}$, where each u_i, $1 \leq i \leq x$, has $d_S(u_i) = 1$. Since in graph H all vertices of U have degree two, the other vertex that is connected to one of the vertices of U_x should be one of V_x.

Consider the vertex in $U_x = \{u_1, u_2, \ldots, u_x\}$ which is incident to edges with the smallest weight. Call this vertex u_a. Let $v_a \in V_x$ and $v_b \in V_x$ be the vertices connected to u_a in H. Let the edge (v_a, u_a) be a strut-edge of S. See Figure 8 (a).

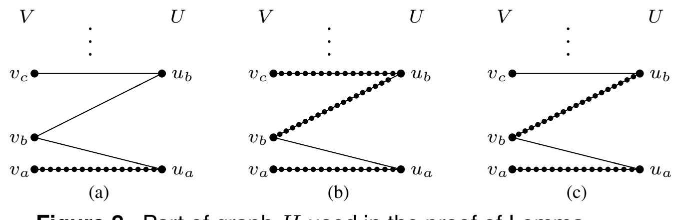

Figure 8. Part of graph H used in the proof of Lemma.

Consider the strut-edge incident to vertex v_b. It cannot be the edge (v_b, u_a), for otherwise u_a would be a zero-difference vertex. Let (v_b, u_b) the strut-edge. Now we have two cases: either u_b is incident to two strut-edges (Figure 8 (b)) or is incident only to one strut edge (Figure 8 (c)). The first case is impossible, since $v_b \in V_x$ cannot be interconnected to zero-difference vertices. The second case implies $u_b \in U_x$. Recall that u_a was chosen to possess the smallest weight in U_x. Then the strut-edge from v_b, by definition of strut, should be (v_b, u_a). This contradiction proves the lemma. $\square$

Theorem 2. *The number of zero-difference vertices in $U_{H'}$ after step 24 of Algorithm 2 is at least divided by 2 in each iteration.*

Proof. Let V and U be the partitions right before the compaction and let $V_{H'}$ and $U_{H'}$ be the partitions right after step 24. Let k be the number of zero-difference vertices in U. By Lemma 2, the number of vertices in $V_{H'}$ is also k.

Since each zero-difference vertex in $U_{H'}$ have degree 2, the number of zero-difference in $V_{H'}$ is at most $|V_{H'}|/2$, i.e., $k/2$. □

After $O(\log p)$ rounds the number of zero difference vertices will be n/p, with $O(m/p)$ edges. Then we can move the remaining compacted bipartite graph to the CPU and finish the algorithm. The number of $\log n$ iterations needed for the algorithm convergence occurs in the worst case. In practice, as shown in our experiments, the number of iterations is much smaller, as we can see in second column of Table 7.

It is worth noting that the same idea present in the algorithms proposed in this paper can also be used to calculate the related components of a graph. This was also studied in the doctoral thesis of Jucele Vasconcellos (in preparation).

Experimental analysis

We presented two parallel algorithms, one to compute the spanning tree and another one to compute the minimum spanning tree. Since the computation of the minimum spanning tree with all the edges with equal weight one is actually a spanning tree of an unweighted graph, we only tested the minimum spanning tree algorithm. Also, we did not find any implementation of a spanning tree algorithm in order to compare our results.

To show the efficiency of our solution, we implement two versions of the Algorithm 4. We developed a CPU version 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. It is noteworthy that our parallel implementation is a simple CUDA implementation without exploiting the various high-performance features available for GPGPU architectures. The main objective of our work was to present an efficient algorithm in the BSP/CGM model that could be easily implemented in a real parallel machine.

The CUDA version implements the algorithm steps using 12 *kernel* functions. One function is to create the bipartite graph immediately after reading the input graph and transferring the data to the GPU. We created two functions to find the smallest edge for each vertex. Two functions are used to obtain the strut. Another function was implemented to compute the connected components where we use the proposal presented by Hawick et al. (2010). And six functions have the responsibility to mark the items to be removed and to create the new bipartite graph to be used in the next iteration.

The implementation used in the experimental tests at (Vasconcellos et al., 2017) has a function to optimize the compaction as exemplified in Figure 6. As this function uses a specific data structure, consuming more memory space, we suppress it to permit conducting tests with larger input graphs. Therefore, the performance presented in this paper is different from the results showed in (Vasconcellos et al., 2017).

We compare the results obtained by our implementation with a recently published efficient algorithm, Edge Pruned Minimum Spanning Tree (EPMST) (Mamun and Rajasekaran, 2016). EPMST chooses a subset of edges using random sampling. It uses the idea of Kruskal's algorithm (Kruskal, 1956) on the small subset of edges and, if necessary, Prim's algorithm (Prim, 1957) on a compacted graph. EPMST implementation is available for download at GitHub, which made it easier to compare results. As EPMST is a sequential solution, we have developed a CPU implementation and another parallel version of our MST algorithm. It is noteworthy that the sets of input data used to perform the tests for this article are different from those used in (Mamun and Rajasekaran, 2016), possibly producing different results.

Table 4 presents the characteristics of the three execution environments that we have used. Comparison of execution times with some CUDA solutions that were published recently was difficult because these algorithms used input graphs for specific problems and different computational resources. However, we were able to compare our results with the parallel solution proposed by Manoochehri et al. (2017) since we can execute tests in a similar environment (NVIDIA Tesla K40) and with the same subset of input graphs (a subset of Ninth DIMACS Implementation Challenge input graphs).

Table 4. Test environments characteristics

	Environment one		Environment two		Environment three	
	CPU	GPU	CPU	GPU	CPU	GPU
#Devices	8	1	40	1	8	1
Manufacturer	Intel	NVIDIA	Intel	NVIDIA	Intel	NVIDIA
Model	E5-1620 v3	Quadro-M4000	E5-2650 v3	Tesla-K40M	i7-4790S	GeForce-GTX 745
#Cores	4	1664	10	2880	4	384
Memory	32 GB	8 GB	126 GB	12 GB	15 GB	4 GB

The first set of input graphs used in our experimental tests was generated using a random graph generator (Johnsonbaugh and Kalin, 1991) available at http://condor.depaul.edu/rjohnson/source/graph_ge.c. We generated 27 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 seven graphs with 15,000 vertices (*graph15*) with densities 0.02, 0.05, 0.1, 0.15, 0.2, 0.5 and 0.75 (see the information of this set of graphs in Table 5).

Graphs of the United States road networks compound the second input set, made available in the Ninth DIMACS Implementation Challenge. The Ninth DIMACS Implementation Challenge (9DIMACS), presented at http://www.dis.uniroma1.it/challenge9/, provides twelve graphs of road networks in the United States. Since our implementation works with non-directed graphs, and the available graph files have the duplicate edges (one to represent the arc between vertex a and b and another to symbol the link between b and a), we reduce the number of edges of the graphs by half. Table 6 shows the information of this set of graphs. A significant difference between the input sets is the density of the graphs. In the second set, the densities are much smaller.

For each input graph, we executed the CPU and CUDA implementations 20 times and collected each runtime result. 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 7 presents the obtained tests results for both input sets using the environment one (see Table 4). Each row of the table shows, for each input graph, the number of iterations of our algorithm, the runtime of CPU implementation, the runtime of CUDA implementation, the runtime of EPMST implementation, speedup of our CPU implementation compared with EPMST, speedup of our CUDA

Table 5. Artificially generated input graphs characteristics.

Input Graph	n (number of vertices)	m (number of edges)	density	m/n
graph10a	10,000	1,000,000	0.020	100.0
graph10b	10,000	2,500,000	0.050	250.0
graph10c	10,000	5,000,000	0.100	500.0
graph10d	10,000	7,500,000	0.150	750.0
graph10e	10,000	10,000,000	0.200	1,000.0
graph15a	15,000	2,500,000	0.020	166.7
graph15b	15,000	5,500,000	0.050	366.7
graph15c	15,000	11,500,000	0.100	766.7
graph15d	15,000	17,000,000	0.150	1,133.3
graph15e	15,000	22,500,000	0.200	1,500.0
graph15f	15,000	56,300,000	0.500	3,753.3
graph15g	15,000	84,350,000	0.750	5,623.3
graph20a	20,000	4,000,000	0.020	200.0
graph20b	20,000	10,000,000	0.050	500.0
graph20c	20,000	20,000,000	0.100	1,000.0
graph20d	20,000	30,000,000	0.150	1,500.0
graph20e	20,000	40,000,000	0.200	2,000.0
graph25a	25,000	6,200,000	0.020	248.0
graph25b	25,000	15,500,000	0.050	620.0
graph25c	25,000	32,000,000	0.100	1,280.0
graph25d	25,000	47,000,000	0.150	1,880.0
graph25e	25,000	62,500,000	0.200	2,500.0
graph30a	30,000	9,000,000	0.020	300.0
graph30b	30,000	22,500,000	0.050	750.0
graph30c	30,000	45,000,000	0.100	1,500.0
graph30d	30,000	67,500,000	0.150	2,250.0
graph30e	30,000	90,000,000	0.200	3,000.0

Table 6. Basic characteristics of the ninth DIMACS Implementation Challenge graphs, considering undirected graphs and eliminating duplicate edges.

Input Graph	n (number of vertices)	m (number of edges)	density	m/n
USA-road-d.NY	264,346	366,648	0.0000105	1.4
USA-road-d.BAY	321,270	399,652	0.0000077	1.2
USA-road-d.COL	435,666	527,767	0.0000056	1.2
USA-road-d.FLA	1,070,376	1,354,681	0.0000024	1.3
USA-road-d.NW	1,207,945	1,417,704	0.0000019	1.2
USA-road-d.NE	1,524,453	1,946,326	0.0000017	1.3
USA-road-d.CAL	1,890,815	2,325,452	0.0000013	1.2
USA-road-d.LKS	2,758,119	3,438,289	0.0000009	1.2
USA-road-d.E	3,598,623	4,382,787	0.0000007	1.2
USA-road-d.W	6,262,104	7,609,574	0.0000004	1.2
USA-road-d.CTR	14,081,816	17,120,937	0.0000002	1.2
USA-road-d.USA	23,947,347	29,120,580	0.0000001	1.2

implementation compared with EPMST and speedup of our CUDA implementation compared with our CPU implementation. By Theorem 2 we know that the algorithm needs $\log n$ iterations in the worst case; however, we can see that in practice this number is much smaller (see the second column of Table 7). Similarly Table 8 and Table 9 present the obtained tests results for both input sets using the environments two and three, respectively.

We aim to present a parallel algorithm that is efficient in the BSP/CGM model ($O(\log p)$ rounds (computation/communication), where p is the number of processors) and suitable to work in real parallel

Table 7. Tests results using environment one.

Input Graph	# iterations	CPU Time (s)	CUDA Time (s)	EPMST Time (s)	Speedup CPU x EPMST	Speedup CUDA x EPMST	Speedup CUDA x CPU
graph10a	5	0.858	0.763	0.068	0.079	0.089	1.125
graph10b	4	1.852	0.827	0.260	0.140	0.315	2.241
graph10c	3	2.568	0.910	1.301	0.507	1.429	2.821
graph10d	4	4.445	1.100	3.908	0.879	3.554	4.042
graph10e	3	4.686	1.117	9.517	2.031	8.520	4.195
graph15a	5	2.397	0.863	0.414	0.173	0.479	2.777
graph15b	5	4.676	1.065	1.262	0.270	1.185	4.392
graph15c	3	5.884	1.189	6.807	1.157	5.723	4.947
graph15d	3	8.322	1.406	19.812	2.381	14.092	5.919
graph15e	3	10.524	1.649	48.508	4.609	29.412	6.381
graph15f	2	15.070	2.366	92.278	6.123	38.993	6.368
graph15g	2	22.122	3.228	205.924	9.309	63.798	6.854
graph20a	5	3.993	0.967	0.527	0.132	0.545	4.131
graph20b	4	7.458	1.276	4.057	0.544	3.180	5.846
graph20c	4	12.533	1.788	20.737	1.655	11.600	7.010
graph20d	3	14.474	1.955	61.792	4.269	31.601	7.402
graph20e	3	18.050	2.298	153.033	8.478	66.593	7.854
graph25a	5	5.950	1.114	1.257	0.211	1.128	5.339
graph25b	4	10.845	1.565	9.700	0.894	6.197	6.929
graph25c	4	20.116	2.454	52.785	2.624	21.513	8.199
graph25d	2	13.251	2.007	151.694	11.447	75.589	6.603
graph25e	3	28.272	3.177	414.316	14.655	130.431	8.900
graph30a	6	10.176	1.459	2.657	0.261	1.821	6.973
graph30b	5	18.385	2.218	20.321	1.105	9.163	8.289
graph30c	3	22.840	2.619	104.559	4.578	39.921	8.721
graph30d	3	33.143	3.536	332.890	10.044	94.154	9.374
graph30e	3	40.840	4.448	988.326	24.200	222.193	9.182
USA-road-d.NY	9	0.158	0.730	0.082	0.515	0.112	0.217
USA-road-d.BAY	10	0.167	0.729	0.080	0.480	0.110	0.229
USA-road-d.COL	9	0.209	0.742	0.084	0.403	0.114	0.282
USA-road-d.FLA	10	0.540	0.809	0.520	0.964	0.643	0.667
USA-road-d.NW	10	0.559	0.821	0.332	0.593	0.404	0.681
USA-road-d.NE	10	0.819	0.856	0.760	0.927	0.888	0.957
USA-road-d.CAL	11	0.984	0.892	0.955	0.970	1.071	1.104
USA-road-d.LKS	11	1.466	0.988	1.766	1.204	1.787	1.484
USA-road-d.E	11	1.831	1.066	2.584	1.411	2.424	1.718
USA-road-d.W	11	3.233	1.323	5.224	1.616	3.949	2.444
USA-road-d.CTR	12	9.778	2.331	26.405	2.700	11.329	4.195
USA-road-d.USA	12	12.599	3.112	71.107	5.644	22.848	4.048

environments. Our implementations used only standard resources of the C language and CUDA library. Besides that, we found that our algorithm has better speedup when we have a significant number of edges since our minimum spanning tree implementation uses more than 40% of the time in the bipartite graph creation. If the graph does not have at least 3,000,000 edges, approximately, the spanning tree is computed very fast and the time spent in the bipartite graph creation dominates the total time. We tested it with different GPUs environments to see if the power of the GPU would be an issue. As we found, the behavior of the algorithm in the three environments are very similar using GPUs with different computational power. We also observed that the algorithm has a better speedup with larger graphs.

Table 8. Some tests results using environment two.

Input Graph	# iterations	CPU Time (s)	CUDA Time (s)	EPMST Time (s)	Speedup CPU x EPMST	Speedup CUDA x EPMST	Speedup CUDA x CPU
graph10a	5	1.084	0.898	0.083	0.077	0.093	1.207
graph10b	4	2.494	1.036	0.317	0.127	0.306	2.407
graph10c	3	3.601	1.163	1.607	0.446	1.382	3.097
graph10d	4	7.104	1.476	4.814	0.678	3.263	4.814
graph10e	3	7.916	1.540	11.754	1.485	7.634	5.141
graph15a	5	4.025	1.061	0.507	0.126	0.478	3.795
graph15b	5	8.153	1.375	1.558	0.191	1.133	5.929
graph15c	3	10.323	1.644	8.407	0.814	5.114	6.279
graph15d	3	14.497	2.061	24.510	1.691	11.890	7.033
graph15e	3	18.024	2.467	59.959	3.327	24.303	7.305
graph15f	2	24.700	3.866	113.986	4.615	29.481	6.388
graph15g	2	35.827	5.430	255.183	7.123	46.993	6.598
graph20a	5	7.227	1.220	0.651	0.090	0.534	5.922
graph20b	4	13.800	1.733	5.009	0.363	2.890	7.962
graph20c	4	21.801	2.718	25.658	1.177	9.441	8.022
graph20d	3	25.414	3.029	76.408	3.007	25.226	8.390
graph20e	3	30.611	3.598	189.168	6.180	52.582	8.509
graph25a	5	10.854	1.521	1.557	0.143	1.023	7.134
graph25b	4	19.792	2.289	11.994	0.606	5.240	8.648
graph25c	4	35.413	3.873	65.263	1.843	16.849	9.143
graph25d	2	22.130	3.389	187.733	8.483	55.393	6.530
graph25e	3	48.612	5.330	470.622	9.681	88.294	9.120
graph30a	6	19.145	2.062	3.282	0.171	1.592	9.286
graph30b	5	33.556	3.460	25.107	0.748	7.256	9.698
graph30c	3	41.477	4.433	129.282	3.117	29.162	9.356
graph30d	3	59.575	6.167	403.514	6.773	65.431	9.660
graph30e	3	70.957	7.528	1095.140	15.434	145.484	9.426
USA-road-d.NY	9	0.185	0.924	0.105	0.569	0.114	0.201
USA-road-d.BAY	10	0.196	0.950	0.097	0.495	0.102	0.207
USA-road-d.COL	9	0.270	0.965	0.114	0.421	0.118	0.280
USA-road-d.FLA	10	0.762	1.027	0.720	0.945	0.701	0.742
USA-road-d.NW	10	0.800	1.062	0.491	0.613	0.462	0.753
USA-road-d.NE	10	1.197	1.118	1.062	0.887	0.950	1.071
USA-road-d.CAL	11	1.451	1.210	1.322	0.911	1.093	1.200
USA-road-d.LKS	11	2.220	1.369	2.712	1.221	1.980	1.621
USA-road-d.E	11	2.800	1.504	3.902	1.393	2.594	1.862
USA-road-d.W	11	5.010	1.981	8.133	1.623	4.104	2.528
USA-road-d.CTR	12	16.590	3.773	39.092	2.356	10.362	4.398
USA-road-d.USA	12	19.476	5.493	99.591	5.114	18.131	3.546

With 15,000 vertices, we generated seven graphs with different densities, namely 0.02, 0.05, 0.10, 0.15, 0.20, 0.50 and 0.75. 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 compared to the EPMST solution is better, emphasizing that the graph uses the logarithmic scale on the y-axis. For density 0.02, our implementations have worse results than the EPMST, but above density 0.05 our CUDA implementation already overcomes the EPMST time. Our CPU implementation is already better than the EPMST from density 0.1. The speedup of CUDA implementation compared to EPMST increases substantially as the density of the graph is raised, ranging from 0.479 to 63.798 in the case of graphs with 15,000 vertices.

Table 9. Some tests results using environment three.

Input Graph	# iterations	CPU Time (s)	CUDA Time (s)	EPMST Time (s)	Speedup CPU x EPMST	Speedup CUDA x EPMST	Speedup CUDA x CPU
graph10a	5	0.778	0.315	0.061	0.079	0.195	2.470
graph10b	4	1.617	0.440	0.234	0.145	0.532	3.680
graph10c	3	2.207	0.707	2.272	0.531	1.657	3.121
graph10d	4	3.799	1.165	3.521	0.927	3.023	3.262
graph10e	3	3.995	1.100	8.575	2.147	7.793	3.630
graph15a	5	2.099	0.554	0.192	0.091	0.346	3.788
graph15b	5	4.091	1.050	1.136	0.278	1.081	3.894
graph15c	3	4.994	1.243	6.133	1.228	4.934	4.018
graph15d	3	7.070	1.927	17.855	2.525	9.267	3.670
graph15e	3	9.723	2.578	43.677	4.492	16.945	3.772
graph15f	2	17.001	insufficient memory	83.129	4.890	-	-
graph15g	2	25.282	insufficient memory	196.735	7.782	-	-
graph20a	5	3.499	0.931	0.475	0.136	0.510	3.757
graph20b	4	6.453	1.466	3.654	0.566	2.492	4.401
graph20c	4	10.752	3.269	18.691	1.738	5.718	3.289
graph20d	3	12.076	3.984	55.723	4.615	13.987	3.031
graph20e	3	15.631	4.938	139.219	8.907	28.191	3.165
graph25a	5	5.051	1.206	1.132	0.224	0.939	4.188
graph25b	4	9.177	2.660	8.742	0.953	3.286	3.450
graph25c	4	16.969	5.377	47.599	2.805	8.852	3.156
graph25d	2	11.345	4.678	137.808	12.147	29.461	2.425
graph25e	3	29.636	insufficient memory	382.659	12.912	-	-
graph30a	6	8.845	1.811	2.392	0.270	1.320	4.883
graph30b	5	15.884	4.477	18.309	1.153	4.177	3.572
graph30c	3	23.203	5.884	94.278	4.063	16.022	3.943
graph30d	3	36.268	insufficient memory	316.135	8.717	-	-
graph30e	3	45.430	insufficient memory	839.328	18.475	-	-
USA-road-d.NY	9	0.132	0.443	0.067	0.511	0.152	0.298
USA-road-d.BAY	10	0.138	0.441	0.069	0.498	0.156	0.314
USA-road-d.COL	9	0.180	0.459	0.076	0.424	0.166	0.391
USA-road-d.FLA	10	0.474	0.578	0.439	0.927	0.760	0.819
USA-road-d.NW	10	0.494	0.578	0.302	0.611	0.523	0.856
USA-road-d.NE	10	0.724	0.675	0.677	0.934	1.002	1.073
USA-road-d.CAL	11	0.876	0.735	0.809	0.923	1.100	1.191
USA-road-d.LKS	11	1.306	0.927	1.523	1.166	1.643	1.409
USA-road-d.E	11	1.631	1.096	2.226	1.365	2.030	1.488
USA-road-d.W	11	2.877	1.573	4.519	1.571	2.873	1.829
USA-road-d.CTR	12	8.835	3.452	23.815	2.695	6.899	2.560
USA-road-d.USA	12	12.687	4.554	63.047	4.969	13.845	2.786

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. As we can observe in the data presented in Table 7, Table 8 and Table 9, for about half of the USA-road graphs our CUDA implementation was slower than EPMST solution. The USA-road graphs are very sparse graphs. Since the EPMST algorithm uses a heuristics, the final minimum spanning tree is almost found after the application of this heuristic, since in several situations, there is only one road connecting two cities. Our algorithm uses a step that is very time

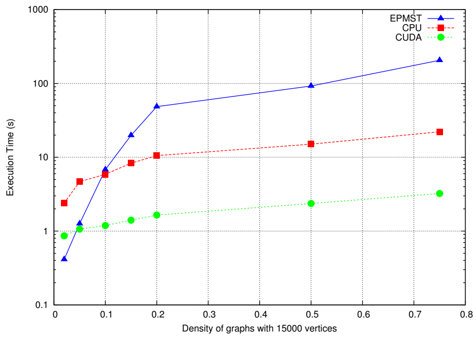

Figure 9. Execution time for graphs with 15,000 vertices using environment one.

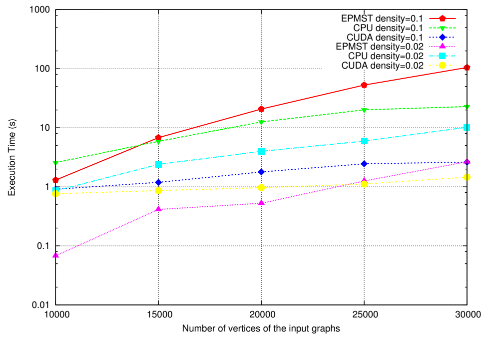

Figure 10. Execution time for graphs with density 0.02 and 0.1 using environment one.

consuming (bipartite graph creation), and if the number of edges is smaller than approximately 3,000,000 edges, this step dominates the final execution time. When we compare with huge graphs, the time used in the bipartite graph creation is compensated.

The standard minimum spanning tree algorithm has a lot of interdependence among the sequential steps. This problem belongs to a class of problems with a very strong relationship among its input data. We devised an efficient parallel algorithm where we can deal with all these dependencies in parallel. Even so, we have to synchronize at the end of each computation round in order to assure the correctness of the input of the next round. To accomplish this, we need a lot of communication in all process. Our implementation only uses the standard resources of the CUDA library. We aimed to show that the algorithm can be implemented in real machines. When compared with one the fastest sequential algorithm (using -O3 compilation directive), our algorithm reached speedups of 222 for synthetic graphs and 22.8 for real graphs. These occurred when we have graphs with more than 29,000,000 and 90,000,000 edges, respectively. We also implemented our parallel algorithm using a CPU (one node). The GPU results were much better than the CPU times, and we reached a speedup of 9 when the graph has a considerable amount of edges. Our parallel algorithms behave better when using a GPU environment.

Manoochehri et al. (2017) present an efficient transaction-based implementation of the Borůvka's algorithm on GPU. One of the test environments used by them is based on a NVIDIA Tesla K40, similar to our environment two. Table 10 presents the comparison of our results and the available results in (Manoochehri et al., 2017). Again, as we can see, our implementation presents better results for larger input graphs. The creation of the bipartite graph causes our algorithm to present gains for large input graphs with at least 3,000,000 edges.

Conclusions and Future Works

In this work, we presented parallel algorithms to compute a spanning tree and a minimum spanning tree. A CPU and a parallel versions of the MST algorithm were implemented and compared with the results of other recent efficient algorithm, named Edge Pruned Minimum Spanning Tree (EPMST) (Mamun and Rajasekaran, 2016). The experiments show that the proposed algorithm presents a good performance for

Table 10. Comparing some of the execution times of our CUDA implementation with data available in (Manoochehri et al., 2017).

Input Graph	Manoochehri et al. (2017) Implementation using TeslaK40 GPU	Our implementation using environment 2	Speedup
USA-road-d.NY	0.217	0.924	0.235
USA-road-d.FLA	0.736	1.027	0.717
USA-road-d.E	1.959	1.504	1.302
USA-road-d.W	3.451	1.981	1.742
USA-road-d.USA	13.407	5.493	2.441

not very sparse graphs, resulting in very good speedups. For larger inputs graphs, the execution time of our algorithm is also better than the solution presented by (Manoochehri et al., 2017).

From the experiments, it was evident that the efficiency of our algorithm increases when we have not very sparse graphs or with a large amount of vertices. Nowadays, where we have an enormous volume of information to be treated, this is a great advantage.

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 us 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. It is also part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, CAPES proc. 88887.136422/2017-00, and FAPESP proc. 14/50937-1 and FAPESP proc. 15/24485-9.

References

Cáceres EN, Dehne F, Mongelli H, Song SW and Szwarcfiter JL (2003) A coarse-grained parallel algorithm for spanning tree and connected components. Technical Report RT-MAC-2003-06 .

Cáceres EN, Dehne F, Mongelli H, Song SW and Szwarcfiter JL (2004) A coarse-grained parallel algorithm for spanning tree and connected components. In: Proceedings of Euro-Par 2004. Lecture Notes in Computer Science, volume 3149. Springer Berlin Heidelberg, pp. 828–831.

Cáceres EN, Deo N, Sastry S and Szwarcfiter JL (1993) On finding Euler tours in parallel. Parallel Processing Letters 3(3): 223–231.

Cáceres EN, Mongelli H, Nishibe C and Song SW (2010) Experimental results of a coarse-grained parallel algorithm for spanning tree and connected components. In: 2010 International Conference on High Performance Computing Simulation. pp. 631–637.

Dehne F, Fabri A and Rau-Chaplin A (1996) Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. International Journal on Computational Geometry & Applications 6(3): 298–307.

Dehne F, Ferreira A, Cáceres E, Song SW and Roncato A (2002) Efficient parallel graph algorithms for coarse grained multicomputers and BSP. Algorithmica 33(2): 183–200.

Graham RL and Hell P (1985) On the history of of the minimum spanning tree problem. In: Annals of the History of Computing, volume 7. pp. 43–57.

Hawick KA, Leist A and Playne D (2010) Parallel graph component labelling with GPUs and CUDA. Parallel Computing 36(12): 655–678.

Hirschberg DS, Chandra AK and Sarwate DV (1979) Computing connected components on parallel computers. Comm. ACM 22(8): 461–464.

Johnsonbaugh R and Kalin M (1991) A graph generation software package. In: Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education, volume 23. pp. 151–154.

Karp RM and Ramachandran V (1990) Handbook of Theoretical Computer Science, volume A. Cambridge, MA: The MIT Press.

Kozen D (1992) The Design and Analysis of Algorithms. Springer.

Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society 7(1): 48–50.

Lima ACd, Branco RG, Ferraz S, Cáceres EN, Gaioso RRA, Martins WS and Song SW (2016) Solving the maximum subsequence sum and related problems using BSP/CGM model and multi-GPU CUDA. Journal of The Brazilian Computer Society (Online) 22: 1–13.

Mamun A and Rajasekaran S (2016) An efficient minimum spanning tree algorithm. In: Computers and Communication (ISCC), 2016 IEEE Symposium on. IEEE, pp. 1047–1052.

Manoochehri S, Goodarzi B and Goswami D (2017) An Efficient Transaction-Based GPU Implementation of Minimum Spanning Forest Algorithm. 2017 International Conference on High Performance Computing & Simulation (HPCS) : 643–650DOI:10.1109/HPCS.2017.100.

Mareš M (2008) The saga of minimum spanning trees. Computer Science Review 2(3): 165–221.

Nasre R, Burtscher M and Pingali K (2013) Morph algorithms on GPUs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 147–156.

Nobari S, Cao TT, Karras P and Bressan S (2012) Scalable parallel minimum spanning forest computation. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12. pp. 205–214.

Osipov V, Sanders P and Singler J (2009) 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), pp. 52–61.

Prim RC (1957) Shortest connection networks and some generalizations. Bell Labs Technical Journal 36(6): 1389–1401.

Reif JH (1985) Depth-first search is inherently sequential. Information Processing Letters 20(5): 229–234.

Valiant LG (1990) A Bridging Model for Parallel Computation. Commun. ACM 33(8): 103–111.

Vasconcellos JFdA, Cáceres EN, Mongelli H and Song SW (2017) A parallel algorithm for minimum spanning tree on gpu. In: Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2017 International Symposium on. IEEE, pp. 67–72.

Vineet V, Harish P, Patidar S and Narayanan PJ (2009) Fast minimum spanning tree for large graphs on the GPU. In: Proceedings of the Conference on High Performance Graphics 2009, HPG '09. pp. 167–171.

Author biographies

Jucele França de Alencar Vasconcellos is a doctoral student in Computer Science at Federal University of Mato Grosso do Sul (UFMS). She received a BSc degree in Computer Science in 1995 from UFMS, Brazil, and an MSc degree in Computer Science in 1998 from the State University of Campinas (UNICAMP), Brazil. She is an Assistant Professor of the College of Computing, UFMS, Campo Grande, Brazil, since 2013. Her research interests are parallel algorithms, graph theory, bioinformatics and software engineering.

Edson Norberto Cáceres is a Full Professor of the School of Computer Science of the Federal University of Mato Grosso do Sul, where he has been a former Pro-Rector of Undergraduate Studies and International Affairs Coordinator. He holds a Ph.D. in Computer Science obtained at the Federal University of Rio de Janeiro in 1992. His research interests include the design of parallel algorithms, especially graph algorithms, cloud computing, and Metaheuristics. He was the Director of Publicity and Marketing and Director of Education of the Brazilian Computer Society. He also held a former position with the Brazilian Ministry of Education as the General Coordinator of Student Relations and was the Adjunct Coordinator of the Computing Committee of Graduate Studies from CAPES/Ministry of Education.

Henrique Mongelli is a Full Professor of the Faculty of Computing, Federal University of Mato Grosso do Sul, Brazil, where he is a former dean of the Faculty of Computing. He holds a PhD in Computer Science obtained at University of São Paulo in 2000. His area of interest is the design of parallel algorithms.

Siang Wun Song is a Full Professor of the Department of Computer Science, University of São Paulo, Brazil, where he has been a former dean of the Institute of Mathematics and Statistics. He holds a Ph.D. in Computer Science obtained at Carnegie Mellon University in 1981. He was on the editorial boards of Parallel Computing, Parallel Processing Letters, Parallel and Distributed Computing Practices, Scalable Computing: Practice and Experience, and Journal of the Brazilian Computer Society. His area of interest is the design of parallel algorithms.

Frank Dehne received an M.C.S. (Dipl. Inform.) from RWTH Aachen University, Germany and a Ph.D. from the University of Würzburg, Germany. He is currently Chancellor's Professor of Computer Science at Carleton University in Ottawa, Canada. His research interests include the design and implementation of efficient parallel algorithms, the interrelationship between the theoretical analysis of parallel algorithms and the performance observed on current parallel architectures, and the use of efficient parallel algorithms for large-scale data analytics and computational biology/bioinformatics. Dr. Dehne is serving or has served on the Editorial Boards of IEEE Transaction on Computers, Information Processing Letters, Journal of Bioinformatics Research and Applications, and International Journal of Data Warehousing and Mining. He is a Fellow of the IBM Centre for Advanced Studies Canada, a member and the former vice-chair of the IEEE Technical Committee on Parallel Processing, a member of the ACM Symposium on Parallel Algorithms & Architectures Steering Committee, and a co-founder of the Algorithms and Data Structures Symposium (WADS).

Jayme L. Szwarcfiter is an Emeritus Professor at the Federal University of Rio de Janeiro, Brazil. He completed his Ph.D. at the University of Newcastle upon Tyne, England, in 1975. Afterwards, he visited University of California, Berkeley, USA, 1979-80; University of Cambridge, England, 1984-85; and University of Paris XI (Orsay), France, 1992-94. Besides, he has visited many different institutions, for short periods. His interests are mainly in algorithms and discrete mathematics, areas where is an author of over 150 journal papers and has supervised more than 40 Ph.D.students, Professor Szwarcfiter is a member of the Brazilian Academy of Sciences.