J. Parallel Distrib. Comput. 109 (2017) 1–14

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Parallel cut tree algorithms

Jaime Cohen [a], Luiz A. Rodrigues [b], Elias P. Duarte Jr. [c,*]

[a] State University of Ponta Grossa, Department of Informatics, Ponta Grossa, Brazil
[b] Western Paraná State University, Department of Informatics, Cascavel, Brazil
[c] Federal University of Paraná, Department of Informatics, Curitiba, Brazil

## HIGHLIGHTS

- Parallel algorithms for constructing cut trees for undirected weighted graphs are presented and evaluated.
- A new algorithm that combines strategies of both Gomory–Hu and Gusfield algorithms.
- Proof of correctness of the parallel Gomory–Hu algorithm.
- An extensive experimental evaluation of the algorithms.
- A discussion of the trade-offs of the alternative algorithms.

ARTICLE INFO

*Article history:*
Received 13 November 2015
Received in revised form 4 April 2017
Accepted 18 April 2017
Available online 10 May 2017

*Keywords:*
Graph edge-connectivity
Cut tree algorithms
Parallel algorithms

ABSTRACT

A cut tree is a combinatorial structure that represents the edge-connectivity between all pairs of vertices of an undirected graph. Cut trees solve the all pairs minimum $s$–$t$-cut problem efficiently. Cut trees have a large number of applications including the solution of important combinatorial problems in fields such as graph clustering and graph connectivity. They have also been applied to scheduling problems, social network analysis, biological data analysis, among others. Two sequential algorithms to compute a cut tree of a capacitated undirected graph are well known: the Gomory–Hu algorithm and the Gusfield algorithm. In this work three parallel cut tree algorithms are presented, including parallel versions of Gusfield and Gomory–Hu algorithms. A hybrid algorithm that combines techniques from both algorithms is proposed which provides a more robust performance for arbitrary instances. Experimental results show that the three algorithms achieve significant speedups on real and synthetic graphs. We discuss the trade-offs between the alternatives, each of which presents better results given the characteristics of the input graph. On several instances the hybrid algorithm outperformed both other algorithms, being faster than the parallel Gomory–Hu algorithm on most instances.

## 1. Introduction

A cut tree is a combinatorial structure that represents the edge-connectivity between all pairs of vertices of undirected graphs. Cut trees can be used to solve efficiently the *all pairs minimum $s$-$t$-cut problem*. Cut trees are widely used for solving important combinatorial problems in areas such as graph partitioning [47], graph clustering [18,20,24] and graph connectivity [37]. Cut trees also have many direct applications. They have been applied in scheduling problems [45], social network analysis [5], biological data analysis [40,51], among many others.

In a 1961 seminal paper [23], R.E. Gomory and T.C. Hu showed that the *all pairs minimum $s$-$t$-cut problem* on undirected graphs can be solved with $n-1$ calls to a minimum $s$–$t$-cut algorithm instead of the $\binom{n}{2}$ calls required by the straightforward solution. The solution comprises the construction of a tree that encodes one minimum $s$–$t$-cut between every pair of vertices of the input graph. Such a tree is known as a *cut tree* or a *Gomory–Hu tree*.

There are two well known algorithms for building a cut tree of an undirected graph: Gomory–Hu and Gusfield algorithms. We refer to these algorithms as GH and Gus, respectively. The GH and Gus algorithms present similarities and differences. A similarity is that both call a minimum $s$–$t$-cut algorithm $n - 1$ times. A difference is the fact that the Gus algorithm finds the $n - 1$ cuts in the original input graph, whereas the GH algorithm contracts the graph as it progresses. There is clearly a trade-off between the algorithms since Gus consists of a simple loop calling the minimum cut procedure while GH requires the manipulation of

* Corresponding author.
*E-mail addresses:* jcohen@uepg.br (J. Cohen), luiz.rodrigues@unioeste.br (L.A. Rodrigues), elias@inf.ufpr.br (E.P. Duarte Jr).
*URLs:* http://www.deinfo.uepg.br/~jcohen/ (J. Cohen), http://www.inf.unioeste.br/~luiz/ (L.A. Rodrigues), http://www.inf.ufpr.br/elias/ (E.P. Duarte Jr).

http://dx.doi.org/10.1016/j.jpdc.2017.04.007
0743-7315/© 2017 Elsevier Inc. All rights reserved.

nontrivial data structures in order to keep track of the tree being constructed and the contracted graphs, but may decrease the size of the subproblems along the way.

The GH and Gus algorithms have the same worst case time complexity. However, their running times in practice vary widely. The choice of one or the other algorithm for a particular class of graphs is nontrivial: neither algorithm presents the best running times for every instance. After experimenting with both algorithms in many classes of graphs, A.V. Goldberg and K. Tsioutsiouliklis [22] concluded that their optimized version of the GH algorithm is "more robust" than their version of the Gus algorithm. The reason is that for certain instances, the GH algorithm outperforms the Gus algorithm by a large margin, while the opposite never occurs even though the Gus algorithm is the fastest implementation for many families of graphs.

The GH algorithm only outperforms the Gus algorithm when there are balanced cuts in the input graph that allow GH to reduce considerably the size of the graph after contractions are executed. However determining whether there are balanced cuts in the input graph is nontrivial.

In this work we present parallel versions for both the GH and Gus algorithms and also a new hybrid algorithm that combines features of those algorithms. Instead of applying contractions or using the input graph throughout the execution, the hybrid algorithm decides at each step whether there is an advantage to spend time contracting the graph or if it is preferable to simply compute the minimum cut on the original input graph. As the algorithms compute the tree sequentially in a way that each update of the tree depends on previous operations, at first glance it might seem unpromising to parallelize these algorithms. Nonetheless, experimental results show that most cuts computed in parallel can be used by the algorithm to advance the construction of the cut tree.

The strategy employed to parallelize both GH and Gus algorithms is based on parallel computations of the minimum cuts by different processes. As the minimum cuts are computed, the algorithms check whether the cut can be used to modify the cut tree under construction. That attempt will succeed only if the vertices separated by the minimum cut have not been separated by another cut previously found. If another parallel execution has already separated the two vertices in question, the newly found minimum cut is rejected and a new minimum cut must be calculated. We present experimental results that show that significant speedups can be achieved by the proposed parallel algorithms.

The parallel version of the GH algorithm is non-trivial due to a few modifications that are required. In particular, a situation that has to be dealt with is how the tree is updated as cuts computed in parallel may interfere with each other. In the sequential version each new cut cannot cross any of the previously computed cuts. In the parallel version, as a new cut is being computed, the tree may have already been updated as a result of the other cuts which were computed in parallel. We prove that despite the possibility that the new cut crosses two or more other cuts, it is possible to update the tree in an efficient way, without any need to uncross the cuts.

As the running time of neither GH nor Gus dominates the running time of the other [22], the hybrid parallel algorithm is proposed as mentioned above. This algorithm searches for minimum cuts in either the input graph or in the contracted graph. The main advantage of the hybrid algorithm is to avoid the computationally expensive operation of contracting the graph when there is clearly no advantage in doing so. Experimental results show that the hybrid algorithm is frequently faster than GH and for certain instances it is much faster than Gus.

The rest of this paper is organized as follows. Section 2 contains basic definitions used throughout the paper. Section 3 explains the sequential Gus and GH algorithms. Section 4 presents the parallel version of the Gus algorithm. Sections 5 and 6 present, respectively, the parallel GH and hybrid algorithms. In Section 7 the experimental methodology and results are presented. In Section 8 we present related work. Finally, in Section 9, we present the conclusions and future work.

## 2. Cut trees

Let $G$ be a capacitated graph $G = (V_G, E_G, c)$ where function $c : E_G \to \mathbb{Z}_+$ defines the *capacities* of the edges in $E_G$. A *cut* of $G$ is a bipartition of $V_G$. The cut *induced* by a set $X \subset V_G$ is the bipartition $\{X, \overline{X}\}$ of $V_G$ induced by $X$, where $\overline{X} = V_G - X$. The set $E_G(X, \overline{X}) = \{\{u, v\} \in \underline{E} : u \in X, v \in \overline{X}\}$ contains the edges that *cross* the cut $\{X, \overline{X}\}$. The *capacity* of the cut $\{X, \overline{X}\}$ is $c(X, \overline{X}) = \sum_{e \in E_G(X,\overline{X})} c(e)$.

Let $s$ and $t$ be two vertices of $G$. An *$s$–$t$-cut* of $G$ is a cut $\{X, \overline{X}\}$ such that $s \in X$ and $t \in \overline{X}$. A *minimum $s$–$t$*-cut is an $s$–$t$-cut of minimum capacity. A cut $\{\{s\}, V_G - \{s\}\}$ is called a *trivial cut*. The *local connectivity between s and t* in $V_G$, denoted by $\lambda_G(s, t)$, is the capacity of a minimum $s$–$t$-cut. The minimum $s$–$t$-cut problem and the maximum flow problem are dual and $\lambda_G(s, t)$ equals the value of a maximum flow between $s$ and $t$. Any maximum flow algorithm for directed graphs can be used to compute the local connectivity in undirected graphs using the reduction that transforms each undirected edge into two antiparallel edges.

**All pairs minimum connectivity.** Consider the problem of finding the local connectivity between all pairs of vertices of an undirected graph. The naive solution consists of running $\binom{n}{2}$ maximum flow algorithms, one for each pair of vertices. R.E. Gomory and T.C. Hu [23] showed that $n - 1$ maximum flow computations are sufficient. The solution to the problem consists of constructing a weighted tree that represents the values of all pairwise local connectivity.

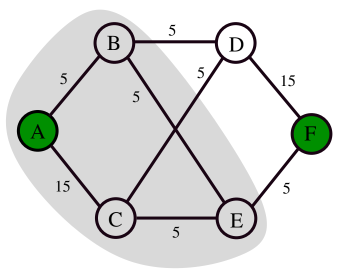
A *flow equivalent tree* of a graph $G = (V_G, E_G, c)$ is a capacitated tree $T$ with vertex set $V_G$ such that for all $u, v \in V_G$, the minimum capacity of an edge on the path between $u$ and $v$ in $T$ is equal to the local connectivity $\lambda_G(u, v)$, i.e., $\lambda_T(u, v) = \lambda_G(u, v)$, for all $u, v \in V_G$.

A *cut tree* is a flow equivalent tree $T$ such that the cut induced by removing an edge of minimum weight from the path between $u$ and $v$ is a minimum $u$-$v$-cut of $G$, for all $u, v \in V_G$. Cut trees are also called *Gomory–Hu trees* [41].

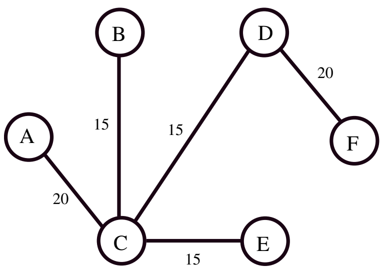
Fig. 1b shows a cut tree of the graph of Fig. 1a. Edge $\{C, D\}$ is the edge with minimum capacity on the tree path between vertices $A$ and $F$. Therefore, a minimum cut between vertices $A$ and $F$ is produced by the removal of edge $\{C, D\}$ from the tree, which gives the partition $\{\{A, B, C, E\}, \{D, F\}\}$ shown in Fig. 1a.

## 3. Sequential cut tree algorithms

In this section we present the two classical sequential algorithms for computing cut trees of capacitated graphs: the *Gomory–Hu* (GH) algorithm [23] and *Gusfield* (Gus) algorithm [27]. Both algorithms use a divide and conquer strategy, and make $n - 1$ calls to a minimum $s$–$t$-cut algorithm. The algorithms differ in their basic approach to obtain a cut tree from an input graph. While the GH algorithm makes a series of contractions on the original graph, the Gus algorithm computes all cuts on the unmodified graph. The descriptions of these sequential algorithms follow. The undirected input graph is denoted by $G = (V_G, E_G, c_G)$ and the tree being constructed is denoted by $T = (V_T, E_T, c_T)$. To avoid ambiguities, the elements of $V_G$ are called *vertices* (of the input graph) and the elements of $V_T$ will be called *nodes* (of the tree).



(a) An undirected weighted graph and a minimum A–F-cut.



(b) A cut tree of the graph.

**Fig. 1.** An undirected weighted graph, a minimum cut and a cut tree of the graph.

## 3.1. *The Gusfield algorithm*

The sequential Gus algorithm consists of $n - 1$ iterations, each of which calls a minimum $s$–$t$-cut algorithm which is executed on the input graph. The pseudocode of Gus algorithm is given as Algorithm 1. Assume the vertices of the input graph are identified by integers from 1 to $n$. Initially, the tree is a star with all vertices pointing to node 1 (lines 1–2). At each iteration (lines 3–6), the algorithm chooses a different source vertex $s$, $s \geq 2$. This choice determines the destination vertex $t$ as the current neighbor of $s$ in the tree. Then, using a minimum cut algorithm, a minimum $s$–$t$-cut $\{X, \overline{X}\}$, with $s \in X$, is found. The tree is updated as follows: every neighbor $u$ of $t$, $u > s$, $u \in X$, gets disconnected from $t$ and gets connected to $s$. The algorithm ends when each node from 2 to $n$ has been used at an iteration. The implementation of the algorithm is simple and requires no changes in the maximum flow algorithm. This version of the algorithm finds a flow equivalent tree. A small modification causes the algorithm to output a cut tree: in line 8, do not restrict the update with $u > s$ and allow any neighbor of $t$ that belongs to $X$ to become a neighbor of $s$.

**Algorithm 1** Sequential Gusfield Algorithm

```
Input: $G = (V_G, E_G, c)$ is a weighted graph
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of $G$
 1: for $i = 1$ to $|V_G|$ do
 2:     $tree_i \leftarrow 1$
    // $|V_G|-1$ minimum $s$-$t$-cut iterations
 3: for $s \leftarrow 2$ to $|V_G|$ do
 4:     $t \leftarrow tree_s$
 5:     $flow_s \leftarrow$ max-flow$(s, t)$
 6:     $\{X, \overline{X}\} \leftarrow$ minimum $s$-$t$-cut
        // update the tree
 7:     for $u \in V_G,\ u > s$ do
 8:         if $tree_u = t$ and $u \in X$ then
 9:             $tree_u \leftarrow s$
    // return the flow equivalent tree
10: $V_T \leftarrow V_G$
11: $E_T \leftarrow \emptyset$
12: for $s \leftarrow 2$ to $|V_G|$ do
13:     $E_T \leftarrow E_T \cup \{s, tree_s\}$
14:     $f(\{s, tree_s\}) \leftarrow flow_s$
15: return $T = (V_T, E_T, f)$
```

## 3.2. *The Gomory–Hu algorithm*

The GH algorithm [23] also consists of $n - 1$ iterations, each of which finds a minimum $s$–$t$-cut, but unlike the Gus algorithm, the minimum cuts are obtained in contracted versions of the input graph. The algorithm starts with a tree $T$ with a single node $S$ identified with all vertices of the input graph, that is, $S = V_G$. At each step, the algorithm picks two vertices $s$ and $t$ of $V_G$ contained in the same node of the tree, say, $X \in V_T$. For each connected component of $T - X$, the algorithm contracts, in $G$, the corresponding vertices into one node. A minimum $s$–$t$-cut is found in the contracted graph. The tree node $X$ is split into two new nodes $X_s$ and $X_t$ containing $s$ and $t$ respectively. The other vertices in $X$ are inserted in $X_s$ or in $X_t$ and the tree edges are rearranged in accordance to the cut just found. The algorithm ends when every node contains only one vertex.

Fig. 2 shows the three first steps of the GH algorithm executed for an example graph. In the beginning, see Fig. 2a, the tree has a single node from which vertices 1 and 2 are taken as $s$ and $t$. The cut separating vertices 1 and 2 induce the tree shown in Fig. 2b where vertices 1 and 3 are picked. The next contracted graph is shown in Fig. 2c as well as the updated tree, after computing the minimum cut that separates vertices 1 and 3. Finally, Fig. 2d shows the contracted graph after vertices 2 and 4 are picked up and also the updated partial cut tree. The algorithm continues for two more steps to obtain the complete cut tree.

Next we describe the GH algorithm formally. Let $P$ be a partition of $V_G$. Let $T = (P, E_T, c)$ be a tree with edge capacities defined by function $c : E_T \to \mathbb{Q}$. The nodes of this tree are sets of vertices of $V_G$ contained in the partition $P$. Each partial tree will correspond to a partition of $V_G$ and the final tree will correspond to the trivial partition $\{\{v\} \mid v \in V_G\}$.

We say that $T$ is a *cut tree of* $G$ *with respect to partition* $P$ if for each edge $e = \{V_u, V_v\} \in E_T$, there exist vertices $u \in V_u$ e $v \in V_v$ such that $\lambda_G(u, v) = c(e)$ and the partition of $V_G$ obtained by the deletion of $e$ from $T$ induces a minimum $u$-$v$-cut of $G$. We denote by $C_T(e)$ the cut of $G$ induced by the components of $T - e$.

In [41] it is shown that if $G$ is a graph, $T$ a cut tree with respect to a partition $P = \{\{v\} \mid v \in V_G\}$ of $V_G$, then $T$ is a cut tree of $G$.

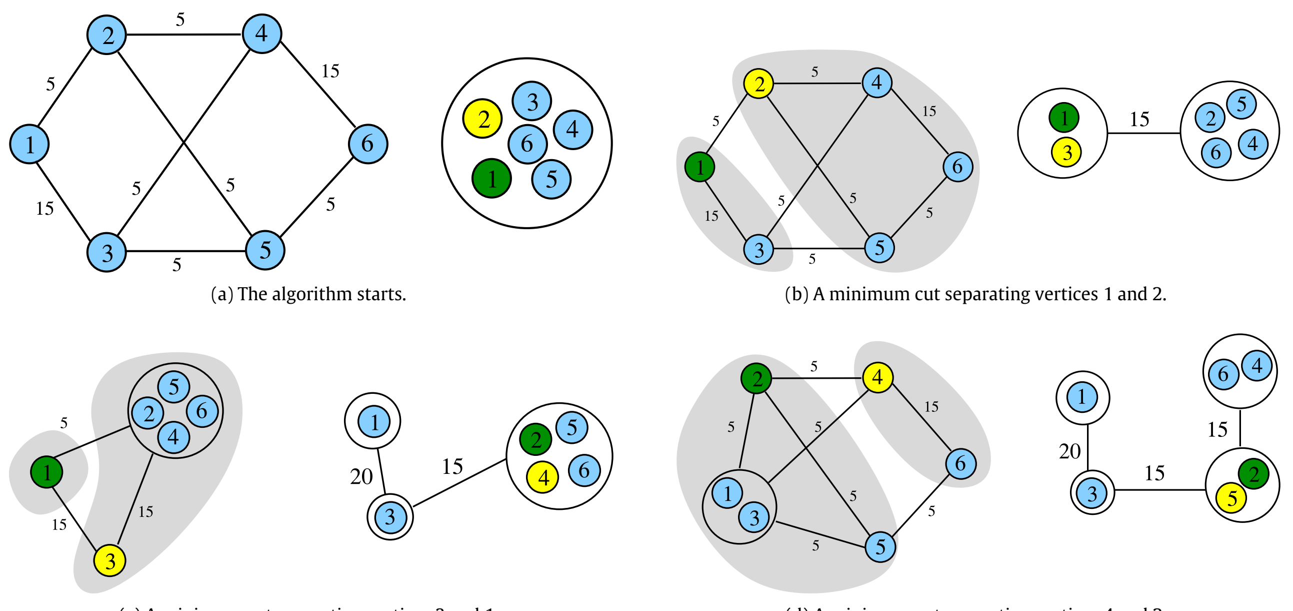
The Gomory–Hu algorithm is based on the following lemma [23]:

**Lemma 1.** *Let $\{X, \overline{X}\}$ be a minimum x-y-cut with $x \in X$. Let $s, t \in X$ and let $\{S, \overline{S}\}$ be a minimum s-t-cut such that $y \notin S$ (substitute $S$ by $\overline{S}$ and s by t if necessary). Then, $\{X \cap S, \overline{X \cap S}\}$ is a minimum s-t-cut.*

This lemma implies that if the set $\overline{X}$ is contracted in $G$, then for any pair of vertices in $X$, there exists a minimum $s$–$t$-cut in the contracted graph that corresponds to a minimum $s$–$t$-cut of the original graph. Let $\{X, \overline{X}\}$ be a minimum cut and let $G/\overline{X}$ be the graph produced by the contraction of the vertices in $\overline{X}$ of $G$. According to Lemma 1, $\lambda_G(s, t) = \lambda_{G/\overline{X}}(s, t)$, that is, a minimum $s$–$t$-cut of $G$ can be found alternatively in $G/\overline{X}$.

Let $X_1, X_2, \ldots, X_k$ be pairwise disjoint subsets of $V_G$. We denote by $G/X_1, X_2, \ldots, X_k$ the graph obtained from $G$ by contracting each $X_i$, $1 \leq i \leq k$. We denote by $G - v$ the graph obtained from $G$ by removing the vertex $v$ and its incident edges.

Algorithm 2 shows the pseudocode of the GH algorithm. The initial tree has one node and corresponds to the partition $P = \{V_G\}$. At each iteration, the GH algorithm picks a node $X \in V_T$ with $|X| > 1$ (line 3). The vertices that correspond to the connected components of $T - X$ are contracted in $G$ to produce a graph $G'$ (lines 4–6). A minimum $s$–$t$-cut $\{Y, \overline{Y}\}$ is found between two



(a) The algorithm starts.

(b) A minimum cut separating vertices 1 and 2.

(c) A minimum cut separating vertices 3 and 1.

(d) A minimum cut separating vertices 4 and 2.

**Fig. 2.** First 3 steps of the GH algorithm.

vertices $s, t \in X$ in the contracted graph (lines 7–8). The tree being constructed is updated: node $X$ is split into two nodes $X_s = Y \cap X$ and $X_t = \overline{Y} \cap X$. Each edge $\{A, X\}$ incident to $X$ is connected to $X_s$ if the contracted vertex containing $A$ is on the $s$ side of the cut; the edge is connected to $X_t$ otherwise (lines 11–19). The algorithm ends when all nodes are singleton sets.

**Algorithm 2** Sequential Gomory–Hu Algorithm

```
1: $T \leftarrow (V_T = \{V_G\},\ E_T = \emptyset)$ // Partial GH tree
2: while $\exists X \in V_T$ such that $|X| > 1$ do
3:     Let $X \in V_T$ such that $|X| > 1$
4:     Let $V_1, V_2, \ldots, V_k$ nodes of the connected components of $T - X$
5:     $X_i \leftarrow \bigcup_{V' \in V_i} V'$, for $1 \le i \le k$ // all vertices in the component
6:     $G' \leftarrow G/X_1, X_2, \ldots, X_k$ // Contracted graph
7:     Let $s, t \in X$
8:     $\{Y, \overline{Y}\} \leftarrow$ minimum $s$-$t$-cut of $G'$
9:     $X_s \leftarrow Y \cap X$
10:    $X_t \leftarrow \overline{Y} \cap X$
       // Update the tree splitting $X$ in $X_s$ and $X_t$
11:    $e \leftarrow \{X_s, X_t\}$
12:    $c(e) \leftarrow d(Y)$ // sets the edge capacity
13:    for all edges $e' = \{A, X\} \in E_T$ incident $X$ in $T$ do
14:        if $A$ is on the $Y$ side of $\{Y, \overline{Y}\}$ then
15:            $E_T \leftarrow E_T \cup \{\{A,\ X_s\}\}$
16:        else
17:            $E_T \leftarrow E_T \cup \{\{A,\ X_t\}\}$
18:    $V_T \leftarrow (V_T \backslash \{X\}) \cup \{X_s,\ X_t\}$ // Split $X$
19:    $E_T \leftarrow E_T \cup \{e\}$
20: return T
```

## 4. A parallel version of the Gusfield algorithm

In this section we describe a parallel implementation of the Gus algorithm. The implementation follows a master/slave model, in which the master is responsible for defining the tasks to be executed by the slaves. These tasks are instances of the minimum cut problem. The algorithm always finds minimum cuts on the input graph. Therefore, each slave process can maintain a copy of the input graph, and this copy does not need to be updated. Messages sent from the master to the slaves need only to carry a source and a destination pair. The slaves compute minimum cuts and return the obtained cuts to the master.

We note that the minimum cut problem is hard to parallelize. Despite extensive research on the problem, experimental studies of parallel min cut algorithms report modest speedups [6,33] due to synchronization requirements. On the other hand, in the parallel version of the Gus algorithm each slave can run the sequential minimum cut algorithm without synchronization. This strategy is optimistic in the sense that the slaves compute their tasks independently, even though those tasks may interfere with each other. Specifically, tree updates may invalidate other pending tasks. This coarse grain strategy to parallelize the algorithm achieves high speedups as shown in Section 7.

The pseudocode of the MPI implementation is given as Algorithm 3. The master process is $proc_0$ and the slaves are $proc_1, .., proc_{p-1}$. Each process maintains a copy of the input graph. We assume that $V_G = \{1, 2, 3, \ldots, |V_G|\}$. The master creates the tasks and sends them to the slaves (lines 5, 15 and 18). Each task contains the source and the destination nodes, $s$ and $t$, used as input to the minimum cut algorithm. When a slave finishes a task, it sends the minimum cut and its value to the master. Based on these data, the master may update the tree if $s$ is still a neighbor of $t$ (line 9). This is done in the same way as the sequential algorithm does. If $s$ and $t$ are not neighbors by the time the task result is processed, then the task is said to have been "unused" and another task having $s$ as the source is produced (line 18). The condition to stop the `while` loop in line 7 guarantees that $|V_G| - 1$ successful receives are executed.

The structure of the graph has an influence on the number of unused tasks. If the $s$–$t$-cut $\{X, \overline{X}\}$ is such that $X$ is small, the tree suffers few changes. The speedup of the parallel execution depends on the number of unused cuts.

## 5. A parallel version of the Gomory–Hu algorithm

In this section we describe a parallel version of the GH algorithm. The algorithm is also based on the master–slave paradigm. The master keeps the tree in construction and the slaves solve minimum $s$-$t$-cut problems independently. Although this strategy is similar to that used in the parallel Gus algorithm, the parallel GH algorithm is more complex. The tree that is built by the algorithm represents a partition of the vertices of the input graph. A tree node corresponds to a set of nodes of the input graph. At each step, the algorithm picks a tree node to be further partitioned.

**Algorithm 3** Parallel Gusfield Algorithm

```
Input: $G = (V_G, E_G, c)$, $proc_j$ processors $(0 \le j < p)$
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of $G$
 1: if $proc_j = 0$ then // master process
 2:     for $i \leftarrow 1$ to $|V_G|$ do
 3:         $tree_i \leftarrow 1$
 4:     for $s \leftarrow 2$ to $p$ do
 5:         send Task(s, $tree_s$) to $proc_{s-1}$
 6:     $s \leftarrow p + 1$
 7:     while $s < p + |V_G|$ do
 8:         receive result $s'$, $t'$, $flow$, $\{X, \overline{X}\}$ from $proc_j$
 9:         if $tree_{s'} = t'$ then // update the tree
10:             $flow_{s'} = flow$
11:             for all $u \in V_G,\ u > s'$ do
12:                 if $tree_u = t'$ and $u \in X$ then
13:                     $tree_u \leftarrow s'$
14:             if $s \le |V_G|$ then
15:                 send a new Task(s, $tree_s$) to $proc_j$
16:             $s \leftarrow s + 1$
17:         else // unused, try again
18:             send Task($s'$, $tree_{s'}$) to $proc_j$
     Build $T$ as in lines 10–14 of the sequential algorithm
19:     return $T$
20: else // slave processes
21:     while more tasks do
22:         receive Task($s$, $t$)
23:         $s$-$t$-cut $\{X, \overline{X}\} \leftarrow$ Min-Cut($s$, $t$)
24:         send $s$, $t$, $\{X, \overline{X}\}$ to $proc_0$
```

We call this node the *pivot*. In our parallel implementation, the master must send to each slave enough information for the slave to build the contracted graph on which the minimum cut algorithm is executed. The master also picks the pair of vertices $s$ and $t$ to be separated. These vertices should be selected with care, as they have an influence on the performance of the algorithm.

The pseudocode of parallel GH is shown in Algorithm 4. The master process is $proc_0$ and the slaves are $proc_1, .., proc_{p-1}$. The algorithm receives as input the capacitated graph $G = (V_G, E_G, c_G)$ and produces as output the capacitated tree $T = (V_T, E_T, c_T)$. Each process maintains a copy of the input graph. We assume that $V_G = \{1, 2, 3, \dots, |V_G|\}$.

The master generates the tasks which are sequentially sent to the slaves (line 6). Each task contains a partition of $V_G$ and the source and the destination nodes, $s$ and $t$. This information is used by the slave to build the contracted graph and to compute the minimum $s$–$t$-cut. When a slave completes a task, it sends the cut and its value back to the master. Based on these data, the master updates the tree if $s$ and $t$ still belong to the same node of the tree (lines 9–20).

Please note that in this description the term *node* is used for a cut tree node, and the term *vertex* refers to vertices of both the input graph and the contracted graph. Given an edge $e = \{U, V\} \in E_T$, $e.V$ denotes the vertex that is the source or destination of the cut from which edge $e$ was obtained and that remained in the same side of the cut as the vertices in $V$.

The tree is updated as follows:

1. create a new node $R$ that contains the vertices of set $X \cap pivot$;
2. remove from $pivot$ the vertices in $X \cap pivot$;
3. remove $pivot$ from the list of pivot candidates if $|pivot| = 1$;
4. insert node $R$ in the list of pivot candidates in case $|R| > 1$;
5. for each edge that is incident on $pivot$, $e = \{V, pivot\}$, remove $e$ from $E_T$ and insert $\{V, R\}$ in $E_T$, if $e.V \in X$;
6. add edge $\{pivot, R\}$ to $E_T$, setting its capacity to the capacity of cut $\{X, \overline{X}\}$.

If $s$ and $t$ have already been separated and belong to different tree nodes, then we say that the task is unused. The minimum

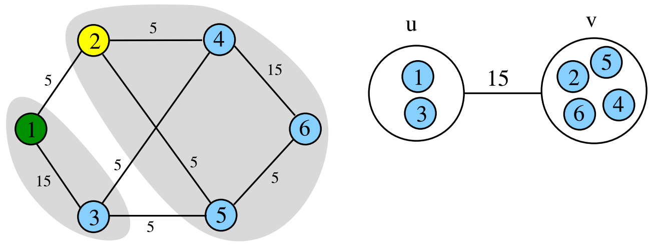


**Fig. 3.** An example of the first step of the execution of the parallel algorithm GH. The tree has two nodes $u = \{1, 3\}$ and $v = \{2, 4, 5, 6\}$. The vertices associated with edge $e = \{u, v\}$ are $e.u = 1$ and $e.v = 2$ which were employed to compute the cut that originated edge $e$.

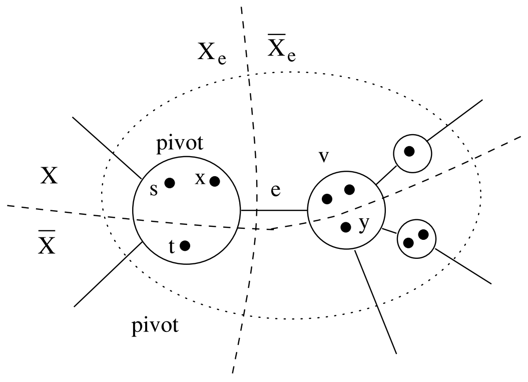


**Fig. 4.** The parallel GH algorithm may cause the minimum $s$–$t$-cut to cross other tree nodes besides the pivot. When the $s$–$t$-cut is processed because of cut $\{X, \overline{X}\}$, $x$ and $y$ had been separated as a result of another minimum $s$–$t$-cut $\{X_e, \overline{X_e}\}$ computed in parallel.

cut is discarded and another task is produced (lines 25–28). The algorithm ends when all tree nodes contain only one vertex.

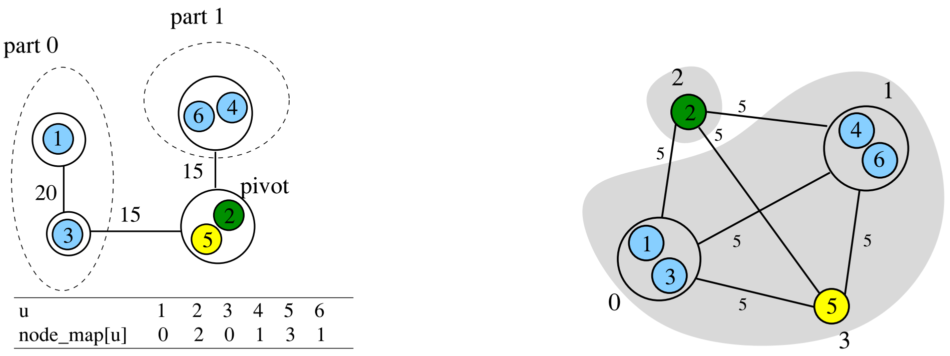
As an example consider Fig. 3. Vertices 1 and 2 were separated and the tree has 2 nodes that correspond to the sets of vertices $u = \{1, 3\}$ and $v = \{2, 4, 5, 6\}$. Edge $e = \{u, v\}$ was created with capacity equal to 15. In this case, $e.u = 1$ e $e.v = 2$ are the vertices used in the definitions of $s$ and $t$ for producing the minimum $s$–$t$-cut that originated edge $e$. As the algorithm continues, $e.u$ and $e.v$ remain the same.

Note that the parallel algorithm updates the tree in a different way from the sequential algorithm. The new strategy is necessary because a node $v$ that is currently adjacent to the pivot might have been part of the pivot when the task was created. Therefore, the $s$–$t$-cut being processed may cross $v$ and/or other nodes in its subtree. See Fig. 4 for an illustration of this case. Solid lines show the current tree and the dashed line represents the pivot at the step in which nodes $s$ and $t$ were selected to be separated. Cut $\{X_e, \overline{X_e}\}$ separated $x$ from $y$, dividing the pivot into two nodes connected by edge $\{pivot, v\}$. Note that the $s$–$t$-cut $\{X, \overline{X}\}$ crosses node $v$. The parallel GH algorithm reconnects edge $e$ to the node that belongs to the same side of the cut in which $y = e.v$ is: if $y \in X$, then $e$ is connected to $new = pivot \cap X$ that contains $s$. Otherwise, the edge remains connected to the $pivot$ node that contains $t$. The correctness of this operation is proved at the end of this section.

More details of the algorithm follow:

*The master process.*

To create a new task, the master invokes the `pick_pivot` subroutine that picks the pivot containing the nodes $s$, $t$ that will be separated. Procedure `build_partition` returns an array $node\_map$ that: (i) associates each vertex in $V_G \setminus pivot$ with one



|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| u | 1 | 2 | 3 | 4 | 5 | 6 |
| node_map[u] | 0 | 2 | 0 | 1 | 3 | 1 |

(a) Partition of the nodes and the *node_map* array.

(b) Contracted graph.

**Fig. 5.** The procedure `build_partition` produces a numbering of the vertices: the *node_map* array. The 2 connected components of $T - pivot$ are mapped to vertices 0 and 1 and the vertices in the pivot are mapped to vertices 2 and 3, respectively.

of the parts of a partition of $V_G \setminus pivot$. The partition is formed by the subset of vertices contained in the nodes of each connected component of $T - pivot$. (ii) Each vertex of *pivot* receives a unique sequential number. These are the vertices that will not be contracted. The array *node_map* is sent to a slave process and together with the input graph contains enough information for the construction of the contracted graph, whose vertices are labeled by the numbers in the *node_map* array.

Fig. 5a shows an example of a *node_map* array. The input graph is the same that appeared in Fig. 2. Vertices 1 and 3 are mapped to 0; vertices 4 and 6 are mapped to 1; vertices 2 and 5 of the pivot are assigned unique sequential numbers 2 and 3, respectively. The contracted graph is shown in Fig. 5b.

**Algorithm 4** Parallel Gomory–Hu Algorithm

```
Input: $G = (V_G, E_G, c_G)$, $proc_j$, $0 \le j < p$, processes
Output: $T = (V_T, E_T, c_T)$ a cut tree of $G$
 1: if $proc_j = 0$ then // master process
 2:     for $s \leftarrow 1$ to $p - 1$ do
 3:         $pivot \leftarrow$ pick_pivot($T$)
 4:         $partition \leftarrow$ build_partition($pivot, T$)
 5:         $(s, t) \leftarrow$ pick_pair_st($pivot$)
 6:         send task $(s, t, partition)$ to $proc_s$
 7:     loop
 8:         receive from $proc_j$ reply $(s, t, X)$, where $\{X, \overline{X}\}$ is a minimum $s$-$t$-cut of
            $G$
 9:         if $s$ and $t$ belong to same $pivot$ of $V_T$ then // update tree
10:             $new \leftarrow pivot \cap X$
11:             $pivot \leftarrow pivot \setminus X$
12:             $V_T \leftarrow V_T \cup \{new\}$
13:             if $|pivot| = 1$ then
14:                 remove $pivot$ from the candidate list
15:             if $|new| > 1$ then
16:                 insert $new$ in the candidate list
17:             for all edges $e = \{pivot, V\} \in E_T$ do
18:                 if $e.V \in X$ then // reconnect $e$
19:                     $E_T \leftarrow (E_T \setminus \{e\}) \cup \{\{new, V\}\}$
20:             add edge $\{pivot, new\}$ to $E_T$ with the capacity of $\{X, \overline{X}\}$
21:         if $|V_T| = |V_G|$ then
22:             send finalization messages to all slave processes
23:             print tree $T$
24:             end
25:         $pivot \leftarrow$ pick_pivot($T$)
26:         $partition \leftarrow$ build_partition($pivot, T$)
27:         $(s, t) \leftarrow$ pick_pair_st($pivot$)
28:         send task $(s, t, partition)$ to $proc_j$
29: else // slave process
30:     loop
31:         receive task $(s, t, partition)$
32:         if Task = end then
33:             halt
34:         $G_c \leftarrow$ build_contracted_graph($G, partition$)
35:         $X \leftarrow$ minimum_cut($G_c, s, t$)
36:         send reply $(s, t, X)$ to $proc_0$
```

**Algorithm 5** `build_contracted_graph`($G$, *partition*)

```
Input: $G = (V_G, E_G, c_G)$, input graph e $partition$
Output: $G_c = (V_c, E_c, c_c)$, contracted graph
1: for each $\{u, v\} \in E_G$ do
2:     if $partition[u] \neq partition[v]$ then
3:         add_edge ($partition[u]$, $partition[v]$, $c_G[\{u, v\}]$, $G_c$)
```

Procedure `pick_st_pair` is responsible for selecting a pair of nodes $s$ and $t$ in *pivot* to be separated by a minimum $s$–$t$-cut. After that, the task can be sent to a slave.

The algorithm completes when the tree contains $|V_G|$ nodes. The master process sends finalization messages to all slave processes (line 22).

*The slave processes.*

A slave process receives tasks from the master process. For each task, the slave builds the contracted graph, computes a minimum $s$–$t$-cut on that graph and sends the result back to the master process (lines 34–36 of Algorithm 4).

The contracted graph is built by iterating through the list of edges of the input graph and deciding whether the edge exists or not in the contracted graph, as shown in Algorithm 5. If the edge connects vertices that belong to the same subtree of $T - pivot$, then the corresponding edge is not included in the contracted graph because it would form a loop. On the other hand, edges connecting vertices in different subtrees of $T - pivot$ or edges that contain at least one vertex in the pivot generate an edge of the contracted graph (or an increase in the capacity of an edge already included). Although the slave does not have access to the tree $T$, the partition vector contains enough information to carry out these tests.

*Correctness of the parallel GH algorithm.*

As discussed previously, the parallel GH algorithm differs from the sequential algorithm in the way it updates the tree. In the sequential algorithm, all tree nodes belonging to the same subtree of $T - pivot$ are necessarily contained on the same side of the $s$–$t$-cut. In the parallel GH algorithm, a pivot node might have been split one or more times at the moment that an $s$–$t$-cut for this node is processed. Fig. 4 illustrates this situation. In the figure, $e.v$ is the vertex $y$, which can either belong to $v$ or to another node in the same subtree of $T - pivot$. As described above, the parallel GH algorithm connects edge $e$ according to the side of the cut to which $y = e.v$ belongs. The next lemma shows that Algorithm 4 correctly updates the tree.

**Lemma 2.** *Algorithm 4 correctly updates the tree.*

**Proof.** Let $G = (V_G, E_G, c_G)$ be the input graph and $T = (V_T, E_T, c_T)$ the tree in construction as an $s$–$t$-cut is received by the master process. We show that the tree update executed by Algorithm 4 is equivalent to the update executed by the sequential algorithm in the current tree. Note that each cut of the contracted graph corresponds to a cut of the input graph with the same capacity.

As shown in Fig. 4, let $pivot \in V_T$ such that $s, t \in pivot$. Let $e = \{pivot, v\} \in E_T$ be an edge incident to $pivot$. Let $X \subseteq V_G$, $s \in X$, such that $\{X, \overline{X}\}$ is the minimum $s$–$t$-cut received by the master process. Let $X_e \subseteq V_G, s, t, x \in X_e$, be the minimum $x$-$y$-cut associated with edge $e$. Assume that cuts $\{X, \overline{X}\}$ and $\{X_e, \overline{X_e}\}$ cross, i.e., $X \cap \overline{X_e} \neq \emptyset$ and $\overline{X} \cap \overline{X_e} \neq \emptyset$ (note that $s \in X \cap X_e$ and $t \in \overline{X} \cap X_e$).

Case I. $y \in \overline{X}$. From Lemma 1 of Section 3, when $y \in \overline{X}$, the set $S = X \cap X_e$ defines a minimum $s$–$t$-cut that does not cross cut $X_e$, either if $x$ belongs to $X$ or to $\overline{X}$. Cut $\{S, \overline{S}\}$ is consistent with the sequential GH algorithm (with respect to edge $e$) and is such that the subtree of $T - pivot$ which contains $v$ belongs to $\overline{S}$. Therefore the sequential algorithm would also update the tree by connecting node $v$ to node $pivot \cap \overline{S}$. As $e.v = y \in \overline{X}$, the parallel GH algorithm does not execute line 19, keeping $v$ connected to the pivot that resulted in set $pivot \cap \overline{X}$ which is the same as $pivot \cap \overline{S}$.

Case II. $y \in X$. When $y \in X$, then Lemma 1 implies that $S = \overline{X} \cap X_e$ is a minimum $s$–$t$-cut. By using this cut the sequential algorithm would connect $v$ to node $pivot \cap \overline{S}$. The parallel algorithm (line 19), connects $v$ to $pivot \cap X$ which is the same as $pivot \cap \overline{S}$.

If cut $\{X, \overline{X}\}$ crosses several subtrees of $T - pivot$, the same analysis must be applied successively to each subtree. The resulting cut does not cross any subtree, because each uncrossing does not interfere in the previous uncrossings as they correspond to set intersection operations.

## 6. A hybrid algorithm for arbitrary instances

In this section we introduce a hybrid algorithm that combines characteristics of GH and Gus in order to present a robust performance for arbitrary input graphs. Note that Gus and GH are either the best choice for some input graphs. Actually for some graph instances one of the two algorithms may present results that are substantially better than the other. In this way, our main purpose in developing the hybrid algorithm is to remove from the user the need to make a single choice.

In order to understand how the parallel GH algorithm compares with the parallel Gus algorithm, it is necessary to understand the notion of balanced/unbalanced cuts. Informally, a cut $\{X, \overline{X}\}$ is unbalanced if either $X$ or $\overline{X}$ is small. This notion plays an important role on the performance of the parallel cut algorithms. The more unbalanced the cut, the less it interferes with other cuts. The Gus parallel algorithm performs well when the graph has unbalanced cuts. On the other hand, the best performances of the GH algorithm are for graphs with balanced cuts. Upon finding a balanced cut, the GH algorithm significantly reduces the size of the graph, causing all subsequent minimum cut computations to be executed on smaller graphs. As a matter of fact, the GH algorithm involves the manipulation of a number of data structures and the only way that this overhead can be compensated for is by reducing the size of the graph. In conclusion, there is no best algorithm for every case. Depending on the input graph either the parallel version of the Gus algorithm or the parallel version of the GH algorithm may be the best. When the input graph has balanced cuts, the GH algorithm is likely to be the best.

The performance of algorithms GH and Gus varies according to the instance to which they are applied. Gus is the best for several instances, but there are instances for which GH performs much better. The main reason is that GH spends time building contracted graphs, a computationally expensive operation, which may or may not be compensated by the amount of time saved as the minimum cuts are computed. Thus we present a hybrid algorithm that combines Gus and GH algorithms and that computes minimum cuts both on the input graph and on a contracted graph. The purpose of this hybrid algorithm is to achieve a robust performance, which does not depend on the particular instance. We actually expect the performance of the hybrid algorithm to be comparable to the best of the both others (GH and Gus). In particular, when Gus is the best choice we expected the hybrid algorithm to take slightly more time, as it involves decisions and employs complex data structures that Gus by itself does not. Next, the hybrid algorithm is described.

The hybrid algorithm maintains the tree under construction as the GH algorithm. The contracted graph is only built if there is a reduction in its size, i.e., if it is sufficiently smaller than the size of the input graph. A threshold $t$, $0 \leq t \leq 1$, is defined, which is employed as follows: after the pivot and the pair of vertices to be separated are selected, if the contracted graph has less vertices than $t$ times the number of vertices of the input graph, then the contracted graph is built. Otherwise the input graph is used to compute the minimum cut.

As shown in the previous section, it is possible to update the tree under construction even when a newly found cut crosses previous cuts. The strategy employed in this case is similar to that described for the parallel GH: (i) the pivot is split according to the cut and (ii) each edge $e = \{pivot, v\}$ incident to the pivot is connected to the node containing $s$ or $t$, according to the side of the cut to which vertex $e.V$ belongs.

**Algorithm 6** Parallel Hybrid Algorithm

```
Input: $G = (V_G, E_G, c_G)$, $proc_j$, $0 \leq j < p$, processes
Output: $T = (V_T, E_T, c_T)$ a cut tree of $G$
1: if $proc_j = 0$ then // master process
2:     for $s \leftarrow 1$ to $p - 1$ do
3:         $pivot \leftarrow$ pick_pivot($T$)
4:         $(s, t) \leftarrow$ pick_pair_st($pivot$)
5:         send task ($s, t, BuildContracted$ = False) to $proc_s$
6:     loop
7:         receive from $proc_j$ reply $(s, t, X)$, where $\{X, \overline{X}\}$ is a minimum $s$-$t$-cut of $G$
8:         if $s$ and $t$ belong to same $pivot$ of $V_T$ then // update tree
9:             update tree as in lines 9–20 of Algorithm 4
10:        if $|V_T| = |V_G|$ then
11:            send finalization messages to all slave processes
12:            print tree $T$
13:            end
14:        $pivot \leftarrow$ pick_pivot($T$)
15:        $partition \leftarrow$ build_partition($pivot, T$)
16:        $(s, t) \leftarrow$ pick_pair_st($pivot$)
17:        $contracted\_size \leftarrow num\_vertices(pivot) + degree(pivot)$
18:        if $contracted\_size < threshold \times |V_G|$ then
19:            send task ($s, t, partition, BuildContracted$ = True) to $proc_j$
20:        else
21:            send task ($s, t, BuildContracted$ = False) to $proc_s$
22: else // slave process
23:     loop
24:         receive task ($s, t, partition, BuildContracted$)
25:         if Task = end then
26:             halt
27:         if BuildContracted then
28:             $G_c \leftarrow$ build_contracted_graph($G, partition$)
29:             $X \leftarrow$ minimum_cut($G_c, s, t$)
30:         else
31:             $X \leftarrow$ minimum_cut($G, s, t$)
32:         send reply $(s, t, X)$ to $proc_0$
```

The pseudocode of the hybrid parallel algorithm is shown in Algorithm 6. The master process is $proc_0$ and the slaves are $proc_1, .., proc_{p-1}$. The algorithm receives as input the capacitated graph $G = (V_G, E_G, c_G)$ and produces as output the capacitated tree $T = (V_T, E_T, c_T)$. Each process maintains a copy of the input graph. We assume that $V_G = \{1, 2, 3, \ldots, |V_G|\}$.

A boolean variable *BuildContracted* is used by the master to signal the slave whether it should or should not compute a contracted graph. Initially, *BuildContracted* is set to false because the pivot contains all vertices of $V_G$ and the associated graph does not have any contracted nodes. When the master receives a response from a slave it updates the cut tree as in lines 9–20 of Algorithm 4. Before preparing a new task, the master computes the size of the contracted graph as the number of vertices in the pivot plus its degree. If the size of the contracted graph is less than the size of $|V_G|$ times the threshold then *BuildContracted* is set to true; otherwise it is set to false. After receiving the task, the slave uses the *BuildContracted* variable to decide whether it will compute the contracted graph or use the input graph.

In the next section we present experimental results comparing the three parallel algorithms. Results confirm that the hybrid algorithm presents robust performance for real and random generated graphs.

## 7. Experimental results

In this section we describe experimental results obtained with the implementation of the three proposed parallel cut tree algorithms: GH, Gus and the hybrid algorithm. The experiments were executed on a multiuser shared cluster with 18 machines each of which with 32 Intel(R) Xeon(R) CPU E5-2670 cores at 2.60 GHz with 128 Gbytes memory and 20,480 Kbytes of cache, interconnected by a Gigabit Ethernet network. The code was written in C/C++ language and compiled with `gcc` (optimization level `-O3`), using OpenMPI. Our implementations are based on the push-relabel maximum flow algorithm [21] code HIPR,[1] developed by B.V. Cherkassky and A.V. Goldberg [14]. The input graph is represented as a list of edges and it is converted to an adjacency list to compute minimum cuts. The list of edges is maintained to help the construction of the contracted graphs. All outputs had been validated against the output of the sequential implementation.

### 7.1. The datasets

We executed experiments with two datasets. The first dataset included 10 graphs (Table 1) from different domains. The second dataset contained random generated graphs in which the existence of balanced cuts can be controlled by a parameter of the graph generating algorithm. These sets of graphs allowed us to explore the differences among the three investigated algorithms in a controlled way. See Table 2.

The dataset with 10 graphs is composed as follows. The first four graphs are based on real data: a road network of the city of Rome (1999) [50], one collaboration network [8], a power grid network [52] and a network of blogs [1]. Two networks were generated using random models: the Erdö s–Rényi (ER) model [11] and the preferential attachment model [2]. The other 4 graphs are synthetic graphs of different types that have been used as benchmarks for min cut and cut tree algorithms [13,22].

The NOI random graphs were proposed by Nagamochi et al. [42] to test minimum cut algorithms [13,42]. Finding graphs from concrete domains that present minimum $s$–$t$-cuts with varying factors of balance is hard. The NOI graph generator is particularly important for understanding the practical aspects of cut tree algorithms exactly because it produces graphs with varying balance factors. The NOI graphs are instances described as algorithmic centered in the book "A Guide to Experimental Algorithmics" by Catherine C. McGeoch [39] as their generator is devised "with parameters that

[1] Owned by IG Systems, Inc. Copyright 1995–2004. Freely available for research purposes.

**Table 1**
Sizes of the graphs in the dataset 1.

| Graph | $\lvert V_G\rvert$ | $\lvert E_G\rvert$ |
|---|---|---|
| ROME99 | 3,353 | 8,879 |
| GEOCOMP | 3,621 | 9,461 |
| POWERGRID | 4,941 | 6,594 |
| POLBLOGS | 1,222 | 16,714 |
| BA | 10,000 | 49,995 |
| ER | 2,000 | 9,995 |
| DCYC | 1,024 | 2,048 |
| NOI | 1,000 | 99,900 |
| PATH | 2,000 | 21,990 |
| TREE | 2,000 | 21,990 |

**Table 2**
Sizes of the graphs in the dataset 2: NOI graphs.

| Graph | $\lvert V_G\rvert$ | $\lvert E_G\rvert$ | Parameter $k$ |
|---|---|---|---|
| NOI | 1000 | 99,900 | 1, 2, 3, 5, 10, 15, 20, 30, 40, 50, 100, 200, 300, 400, 500 |

exercise algorithm mechanisms"; they contrast with the "reality centered generators" to which the BA graphs belong.

The algorithm for generating a NOI graph receives a parameter $k$ that determines the number of heavy components (clusters) of the graph. The algorithm starts with a cycle of $n$ nodes colored randomly with $k$ colors. Edges are added to the graph until the expected density is achieved. If an edge connects nodes of the same color, it receives a large random weight, otherwise it receives a small random weight. The parameter $k$ controls the existence of balanced cuts. Intermediate values of $k$ produce graphs with balanced cuts. For extreme values, i.e., for $k$ equal to 1 or for large $k$, the generated graphs do not contain balanced cuts.

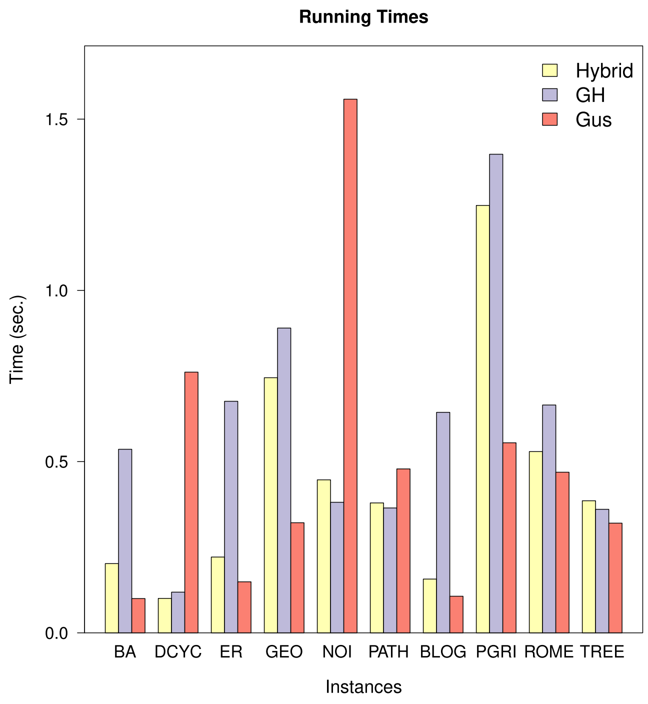
### 7.2. Total running times and profiling

We measured the total running times of the algorithms. The hybrid algorithm used a threshold of 0.7, meaning that the graph is contracted only if the number of nodes is less than 70% of the number of nodes of the input graph. Preliminary experiments were executed with threshold values of 0.1, 0.3, 0.5, 0.7 and 0.9. These experiments showed that 0.7 gives a good trade-off and also that the hybrid algorithm is robust with respect to changes on the threshold.

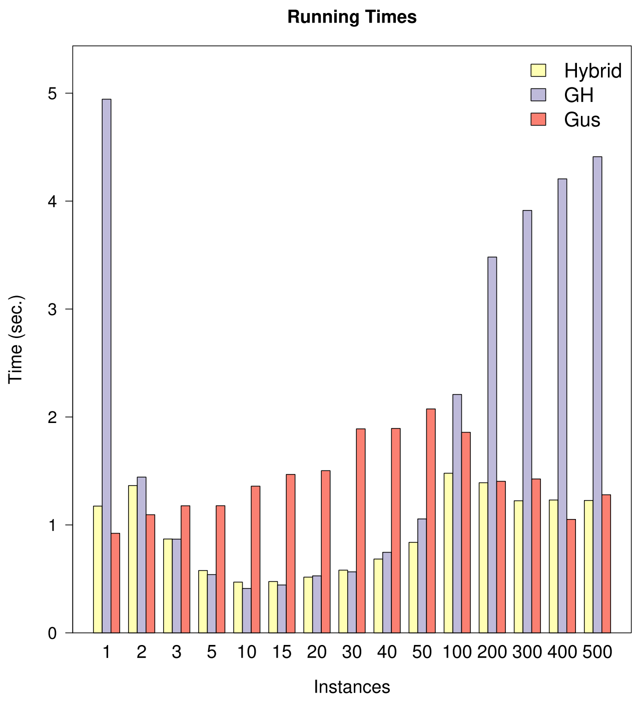
Fig. 6a shows the mean running times of 10 executions of Gus, GH and the hybrid algorithms executed on all 10 instances running 18 processes on 18 different hosts/processors.

The experiment shows that neither GH nor Gus dominates each other. The GH algorithm is faster than Gus for instances DCYC, NOI and PATH. Executed on these instances, the hybrid algorithm shows a similar performance as the faster of the two algorithms. On instances BA, ER, BLOG, ROME and TREE, the Gus algorithm is faster than the GH algorithm and again the hybrid algorithm produced running times that are close to the faster of the two algorithms. On the instances GEO and PGRI the Gus algorithm was faster than the GH algorithm and the hybrid algorithm performed better than GH algorithm but not as fast as the Gus algorithm. The Gus algorithm was 5.8 times faster than the GH algorithm for instance BLOG.

Detailed information on the execution of the algorithms are presented in Table 3. The columns show, respectively, the number of clusters ($k$), the name of the algorithms, initialization times, times for updating the tree, times to construct the contracted graph, times to compute the minimum cuts on the contracted graph and on the input graph, the total running times, the number of discarded cuts and the average size of the graphs on which minimum cuts were found. Note that the total running time is the wall-clock time of the parallel execution, measured at the master process, and the other times are a summation of the times all



(a) Mean total running times with 18 processors of the 10 graph dataset.



(b) Mean total running times with 18 processors of the NOI graphs dataset.

**Fig. 6.** Running times.

processes (slaves) take to execute the task. For example, the time to compute minimum cuts is reported as the summation of the computation of every minimum cut by all processes.

The third column of Table 3 shows the time in seconds to initialize the master process, which includes initializing MPI plus loading the input graph. The initialization takes little time when compared to the other phases shown in the profile, it is always below 0.027 s, except for one input graph: NOI, that has close to 100,000 edges; in this case it took 0.057 s. The fourth column shows the tree update times, which includes the time to test whether a cut can be used or not, plus the time to update the tree. We can see that the tree update time is closely related both to the number of vertices of the input graph and the number of unused cuts (shown in the column before the last). For most graphs the tree update time is on the order of tens of milliseconds, while for three graphs (GEO, PGRI, ROME) it takes up to two hundred milliseconds. The next column is key to understand the advantage of the hybrid algorithm when compared to GH alone; note that no result is presented for Gus as it does not contract graphs. It is easy to see that the hybrid algorithm spends less time contracting graphs in comparison with GH. However, in the last column we can see that the mean sizes of the contracted graphs are very similar for both algorithms (GH and Hybrid).

Columns 6 and 7 show the time spent computing minimum cuts, both on the contracted (column 6) and input graph (column 7). These two columns are important to understand how the hybrid algorithm compares with Gus. For most graphs it is easy to see that there is a significant reduction of the time spent computing minimum cuts by the hybrid algorithm (on a contracted graph) in comparison with Gus on the input graph. However for one case (BA graph) the situation inverts, the hybrid algorithm spent more time than Gus. The reason is that the hybrid algorithm discarded more than two hundred unused cuts, but this number was zero for Gus. For another graph (ER) hybrid takes slightly more time (roughly 13%) than Gus, and we believe this can be explained by the number of unused cuts. For a third input graph (BLOG) hybrid took 6% more time than Gus computing minimum cuts. In this case we cannot explain the figures with the number of unused cuts, we believe this slight increase is due to the fact that hybrid does have an overhead in comparison with Gus, due to the data structures employed.

The last column of Table 3 shows that the maximum reduction of the graph by contractions was near 90% for the NOI and PATH instances. The total time to compute minimum cuts was reduced up to 20 times on the instance NOI.

The initialization times and the time to update the tree under construction are not significant.

The NOI graphs were generated with 1000 nodes, edge density of 20% and parameter $k$ in $\{1, 2, 3, 5, 10, 15, 20, 30, 40, 50, 100, 200, 300, 400, 500\}$. Fig. 6b shows the mean running times of executions of Gus, GH and the hybrid algorithm on these NOI instances.

The Gus algorithm was faster than GH for $k$ equal to 1, 2, 100, 200, 300, 400 and 500. For the remaining 8 instances, the GH algorithm was faster than Gus. For this instance with intermediary values of $k$, the GH algorithm was able to find the balanced cuts and, therefore, it reduced the average size of the graph achieving better running times. With respect to the two algorithms, the slowest performance was 187% worse than the best performance on average, with a maximum of 436% in the worst case.

The hybrid algorithm presented consistent performance on all instances. It was the fastest among the three algorithms on instances with $k$ equal to 3, 20, 40, 50, 100, 200, 300 and 500. On average, the hybrid algorithm was only 14% slower than the best algorithm for the 7 instances for which it was not the fastest.

Detailed profiling of these executions is presented in Table 4. The last column shows a maximum reduction of the graph of 90%, for $k = 30$. And for $k = 15$, the total time to compute minimum cuts was reduced 16.3 times.

The table also shows that the time to compute the contracted graph is much larger on inputs that do not contain balanced cuts and those times may significantly affect the total running time of the GH algorithm. For these NOI instances the number of discarded cuts was not crucial in determining total running times.

Fig. 7 shows stacked bar plots for the three most important factors identified in the table: the contraction time, the time to

**Table 3**
Running time statistics for the 3 algorithms on the 10 instances.

| Instance | Algorithm | Init | Tree update | Contrac-tions | Mincut contracted | Mincut Input | Total runtime | Unused cuts | Graph size |
|---|---|---|---|---|---|---|---|---|---|
| BA | **Hybrid** | 0.010 | 0.037 | 0.000 | 0.000 | 1.932 | 0.20 | 216 | 2000 |
| | **ParGH** | 0.012 | 0.058 | 6.427 | 1.605 | | 0.54 | 214 | 2000 |
| | **ParGus** | 0.012 | 0.058 | | | 1.279 | 0.10 | 0 | 2000 |
| DCYC | **Hybrid** | 0.007 | 0.017 | 0.089 | 0.105 | 0.920 | 0.10 | 99 | 892 |
| | **ParGH** | 0.019 | 0.010 | 0.547 | 1.000 | | 0.12 | 93 | 876 |
| | **ParGus** | 0.019 | 0.010 | | | 12.557 | 0.76 | 278 | 1024 |
| ER | **Hybrid** | 0.014 | 0.049 | 0.129 | 0.000 | 2.358 | 0.22 | 216 | 1996 |
| | **ParGH** | 0.016 | 0.054 | 7.923 | 2.456 | | 0.68 | 203 | 1993 |
| | **ParGus** | 0.016 | 0.054 | | | 2.077 | 0.15 | 171 | 2000 |
| GEO | **Hybrid** | 0.016 | 0.110 | 2.932 | 0.960 | 1.189 | 0.74 | 1234 | 1718 |
| | **ParGH** | 0.011 | 0.106 | 8.606 | 2.399 | | 0.89 | 949 | 1759 |
| | **ParGus** | 0.011 | 0.106 | | | 4.288 | 0.32 | 2444 | 3621 |
| NOI | **Hybrid** | 0.051 | 0.003 | 2.514 | 0.305 | 1.338 | 0.45 | 95 | 108 |
| | **ParGH** | 0.057 | 0.004 | 4.289 | 1.293 | | 0.38 | 97 | 103 |
| | **ParGus** | 0.057 | 0.004 | | | 25.944 | 1.56 | 109 | 1000 |
| PATH | **Hybrid** | 0.016 | 0.011 | 2.753 | 1.287 | 0.288 | 0.38 | 144 | 189 |
| | **ParGH** | 0.020 | 0.005 | 3.191 | 1.650 | | 0.36 | 144 | 190 |
| | **ParGus** | 0.020 | 0.005 | | | 7.577 | 0.48 | 155 | 2000 |
| BLOG | **Hybrid** | 0.015 | 0.013 | 0.310 | 0.000 | 1.582 | 0.16 | 175 | 1148 |
| | **ParGH** | 0.026 | 0.022 | 8.526 | 1.733 | | 0.64 | 155 | 1118 |
| | **ParGus** | 0.026 | 0.022 | | | 1.489 | 0.11 | 174 | 1222 |
| PGRI | **Hybrid** | 0.014 | 0.176 | 3.066 | 0.958 | 1.113 | 1.25 | 2038 | 1888 |
| | **ParGH** | 0.014 | 0.209 | 7.202 | 2.153 | | 1.40 | 1789 | 1963 |
| | **ParGus** | 0.014 | 0.209 | | | 7.337 | 0.55 | 4924 | 4941 |
| ROME | **Hybrid** | 0.014 | 0.102 | 1.427 | 0.597 | 2.360 | 0.53 | 949 | 2367 |
| | **ParGH** | 0.015 | 0.121 | 5.565 | 2.044 | | 0.67 | 867 | 2216 |
| | **ParGus** | 0.015 | 0.121 | | | 6.888 | 0.47 | 2374 | 3353 |
| TREE | **Hybrid** | 0.014 | 0.015 | 3.038 | 1.148 | 0.143 | 0.39 | 139 | 246 |
| | **ParGH** | 0.022 | 0.008 | 3.388 | 1.237 | | 0.36 | 136 | 239 |
| | **ParGus** | 0.022 | 0.008 | | | 4.944 | 0.32 | 14 | 2000 |

**Table 4**
Running time statistics for the 3 algorithms on NOI instances.

| Instance | Algorithm | Init | Tree update | Contrac-tions | Mincut contracted | Mincut Input | Total runtime | Unused cuts | Graph size |
|---|---|---|---|---|---|---|---|---|---|
| 1 | **Hybrid** | 0.053 | 0.011 | 0.000 | 0.000 | 16.079 | 1.17 | 114 | 1000 |
| | **ParGH** | 0.053 | 0.011 | 69.049 | 13.399 | | 4.94 | 112 | 1000 |
| | **ParGus** | 0.053 | 0.011 | | | 15.232 | 0.92 | 123 | 1000 |
| 10 | **Hybrid** | 0.053 | 0.005 | 2.885 | 0.273 | 1.243 | 0.47 | 100 | 131 |
| | **ParGH** | 0.056 | 0.004 | 4.774 | 1.220 | | 0.41 | 100 | 129 |
| | **ParGus** | 0.056 | 0.004 | | | 22.403 | 1.36 | 74 | 1000 |
| 30 | **Hybrid** | 0.050 | 0.006 | 2.937 | 0.699 | 2.543 | 0.58 | 117 | 112 |
| | **ParGH** | 0.055 | 0.005 | 6.205 | 2.432 | | 0.57 | 112 | 104 |
| | **ParGus** | 0.055 | 0.005 | | | 31.338 | 1.89 | 136 | 1000 |
| 40 | **Hybrid** | 0.050 | 0.003 | 3.616 | 1.137 | 3.189 | 0.68 | 134 | 126 |
| | **ParGH** | 0.063 | 0.001 | 7.949 | 3.742 | | 0.75 | 138 | 120 |
| | **ParGus** | 0.063 | 0.001 | | | 31.672 | 1.89 | 124 | 1000 |
| 50 | **Hybrid** | 0.056 | 0.004 | 4.934 | 1.686 | 4.226 | 0.84 | 162 | 169 |
| | **ParGH** | 0.054 | 0.006 | 11.587 | 5.317 | | 1.06 | 175 | 157 |
| | **ParGus** | 0.054 | 0.006 | | | 33.979 | 2.07 | 122 | 1000 |
| 100 | **Hybrid** | 0.054 | 0.011 | 9.727 | 3.911 | 7.848 | 1.48 | 174 | 344 |
| | **ParGH** | 0.051 | 0.006 | 25.498 | 10.787 | | 2.21 | 199 | 304 |
| | **ParGus** | 0.051 | 0.006 | | | 30.479 | 1.86 | 122 | 1000 |
| 200 | **Hybrid** | 0.058 | 0.013 | 3.882 | 1.328 | 14.488 | 1.39 | 142 | 634 |
| | **ParGH** | 0.056 | 0.014 | 43.815 | 13.868 | | 3.48 | 163 | 549 |
| | **ParGus** | 0.056 | 0.014 | | | 23.303 | 1.40 | 109 | 1000 |
| 300 | **Hybrid** | 0.051 | 0.018 | 0.735 | 0.001 | 16.734 | 1.22 | 117 | 735 |
| | **ParGH** | 0.054 | 0.004 | 50.311 | 14.623 | | 3.91 | 152 | 647 |
| | **ParGus** | 0.054 | 0.004 | | | 23.416 | 1.43 | 163 | 1000 |
| 400 | **Hybrid** | 0.053 | 0.016 | 0.633 | 0.000 | 16.544 | 1.23 | 124 | 808 |
| | **ParGH** | 0.054 | 0.009 | 55.245 | 14.746 | | 4.21 | 141 | 751 |
| | **ParGus** | 0.054 | 0.009 | | | 17.352 | 1.05 | 73 | 1000 |
| 500 | **Hybrid** | 0.049 | 0.010 | 0.593 | 0.000 | 17.007 | 1.23 | 120 | 847 |
| | **ParGH** | 0.055 | 0.007 | 58.601 | 14.727 | | 4.41 | 133 | 800 |
| | **ParGus** | 0.055 | 0.007 | | | 21.216 | 1.28 | 142 | 1000 |

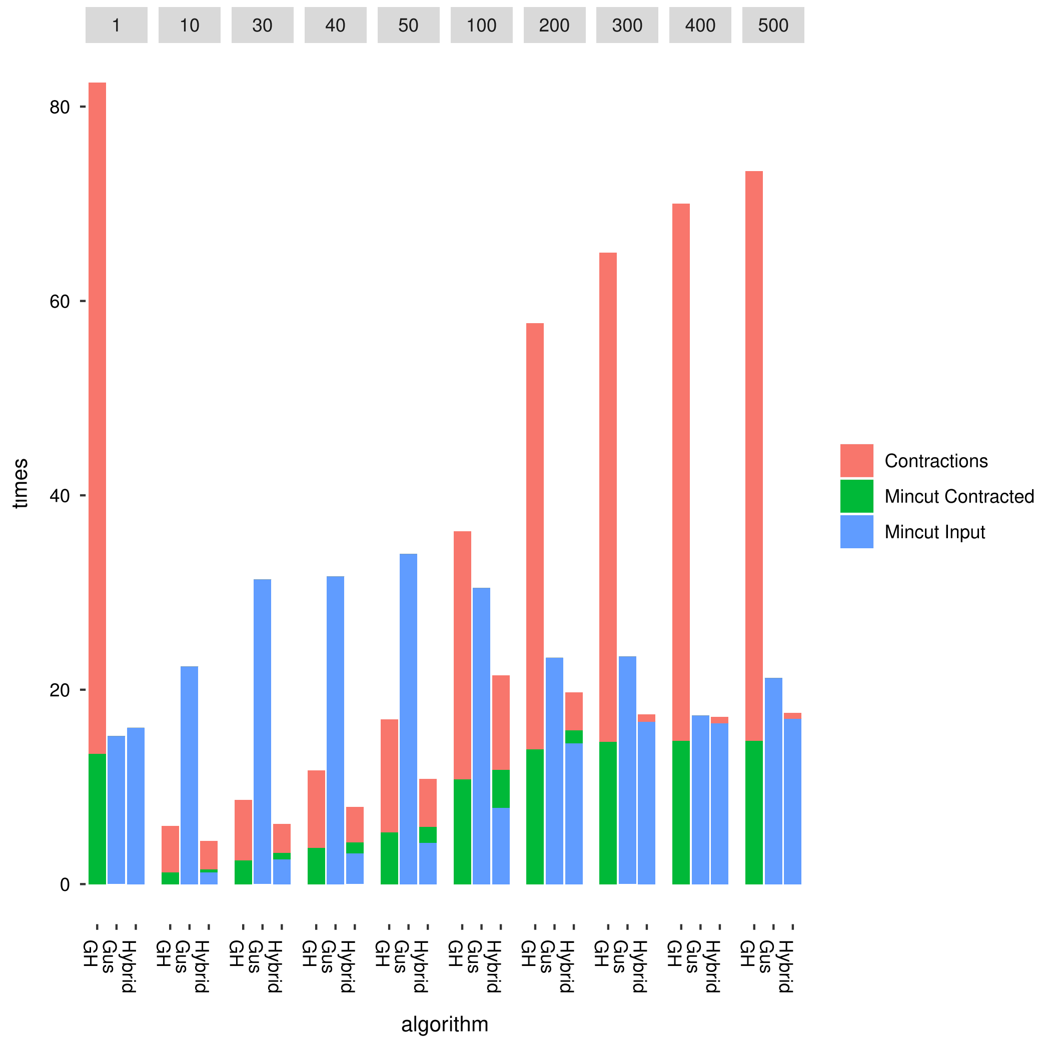


**Fig. 7.** Running times divided by the most time consuming tasks: contractions, minimum cut on contracted graph and minimum cut on the input graph.

compute the min-cut in the contracted graph, and the time to compute the min-cut in the original graph. The plot helps to understand what happens when the hybrid algorithm presents the best overall results. For instance, consider $k = 50$. In this case Gus takes most of the time to compute the minimum cut—more time than both GH and hybrid take to complete the execution. GH takes more time contracting the graph than the hybrid algorithm. The same pattern repeats for $k = 10, 30, 40, 50$. Then a different pattern appears in which GH takes a long time to contract the graphs, while Gus presents good results for computing the minimum cuts, and the hybrid algorithm presents even better results for the same task.

### 7.3. Speedups

Speedups were computed as $S = T_S/T_P$, where $T_S$ is the running time of the sequential execution of the algorithms and $T_P$ is the execution time for the parallel implementations on $p$ processes. The efficiency was calculated using $E = S/p$. All experiments consisted of 10 runs on each instance.

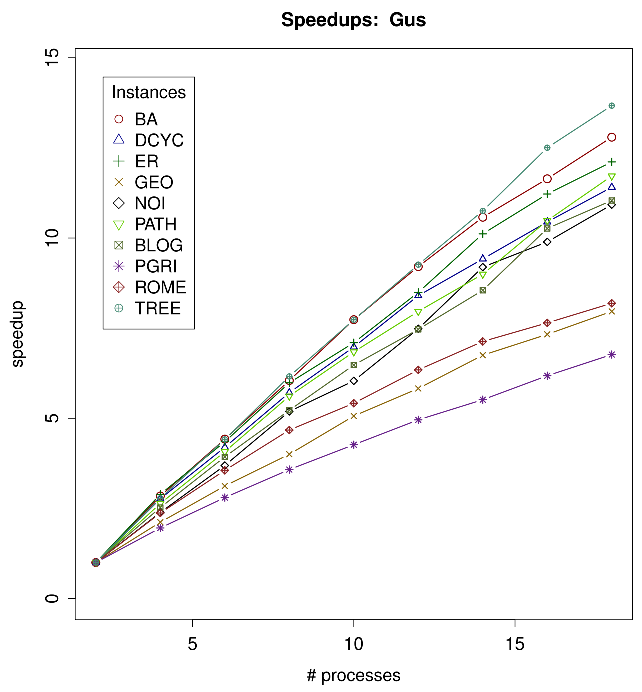
The speedups of the three algorithms were similar. The Gus algorithm had slightly better speedups. Figs. 8a and 8b show the speedups of the Gus and hybrid algorithms, respectively. With 18 processes, Gus presented speedups between 6.7 and 13.6 while the hybrid algorithm had speedups between 4.6 and 11.3.

We also executed the algorithms for large NOI graphs with 100,000 nodes and 400,000 edges. By running the hybrid algorithm on 34 processors we computed a speedup equal to 24. In terms of running time, the reduction was from nearly 8 h (7 h 57 min and 22 s to be precise) for the sequential version to nearly 19 min (18 min and 56 s to be precise) for the parallel version executed on 34 processors.

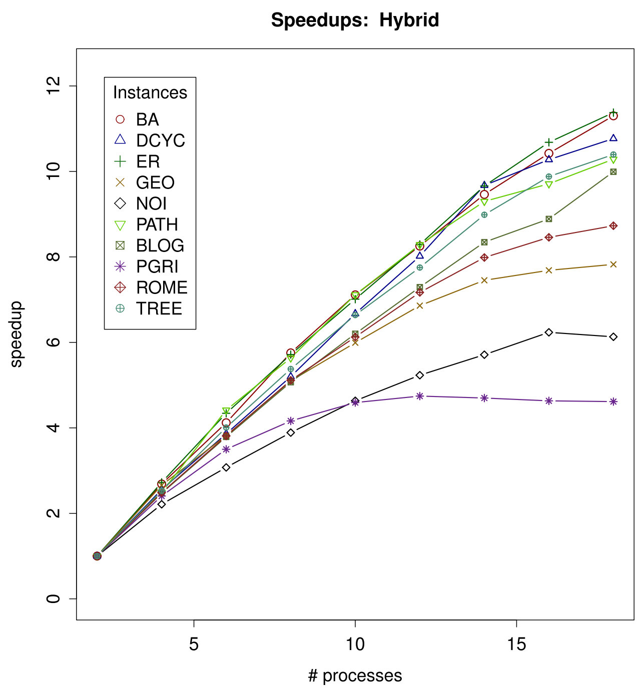
## 8. Related work

Both the GH and Gus algorithms accept as input undirected capacitated multigraphs. For specific cases, more efficient algorithms have been proposed. For instance, A. Bhalgat et al. [29] proposed a randomized algorithm that builds a cut tree from a non-capacitated graph with time complexity of $O(mn)$. We are not aware of any implementation of this algorithm. In the case of capacitated planar graphs, an $O(n.\log^4 n)$ algorithm was introduced by G. Borradaile et al. [12].

A.V. Goldberg and K. Tsioutsiouliklis [22] present the first experimental study of cut tree algorithms. Their work aims at comparing the efficiency of the Gomory–Hu and Gusfield algorithms. The authors' conclusion was that an optimized version of the GH algorithm was more robust than the Gusfield algorithm because the latter was significantly slower than the former in some instances. However, it is worth noting that most instances for which the GH algorithm is much faster than the Gus algorithm are synthetic graphs where balanced cuts exist by construction. On the other hand, Gus algorithm was faster in many classes of graphs.



(a) Speedups of the parallel Gus algorithm.



(b) Speedups of the hybrid algorithm.

**Fig. 8.** Speedups of the parallel Gus and hybrid algorithms.

The problem of finding a partial cut tree that represents all edge cuts with cardinality at most $k$ for non-capacitated graphs was studied by Hariharan et al. [28]. They presented a randomized algorithm with expected complexity of $O(m + nk^3)$ to solve that problem.

Arikati et al. [4] present efficient algorithms to the all-pair minimum cut problem for graphs with bounded *treewidth*, planar graphs and sparse graphs.

The concept of cut trees cannot be directly applied for the edge-connectivity of directed graphs, as shown by Benczúr [9]. The author shows a directed graph that has a quadratic number of $s$–$t$-cuts of distinct capacities. Actually, this result revealed an error in the work of C.P. Schnorr [49] that presented (incorrect) evidence for the existence of cut trees of directed graphs. It also invalidated the work by D. Gusfield and D. Naor [26] that, based on Schnorr's result, presented an (incorrect) algorithm for the construction of such trees.

Minimum vertex cuts cannot be represented by a cut tree [9]. In the more general case of graphs with costs associated with vertices R. Hassin and A. Levin [32] showed that there is no compact representation of the vertex cuts with minimum cost, as there exist up to $\Theta(n^2)$ possible values for minimum cuts.

Separations are a generalization of the concept of vertex cuts. An $s$–$t$-separation is a set of vertices that may be a vertex-cut or it may contain either $s$ or $t$. R. Hassin and A. Levin [32] showed the existence of a cut tree for separations in undirected graphs with arbitrary cost functions as well as an algorithm to compute it. Another generalization of cuts is the node–edge cuts that may contain both vertices and edges. X. Zhang et al. [53] showed the existence of flow equivalent trees to node–edge capacitated undirected planar graphs.

The concept of cut trees was generalized to matroids by D. Hartvigsen in [31]. He showed that computing a cut tree for an undirected graph is a particular case of the same problem applied for matroids.

M. Conforti et al. [17] generalized the concept of flow equivalent trees to allow the efficient retrieval of $\lambda(s, t)$ edge-disjoint paths between any pair of vertices of a graph.

Kabadi S.N. et al. [36] study the existence of cut trees to the maximum flow problem between a pair of vertices with multiple routes (*multiroute flows*).

The problem of finding minimum cuts between all pairs vertices of a graph was also studied in the parametric case where the weights of some edges may vary. P. Berthomé et al. [10] show that from $2^k$ cut trees it is possible to efficiently determine the value of the maximum flow between any pair of vertices in a graph with up to $k$ edges with varying capacities. The same authors show in [7] how to find minimum cuts between all pairs of vertices for the same case. In [3], another parametric version of the same problem is studied.

Cut trees and flow-equivalent trees have been applied in solutions for several different types of problems. Some of these applications are presented below.

Given a graph $G = (V_G, E_G)$ and a set $T \subseteq V_G$, where $|T|$ is even, a $T$-cut in $G$ is a cut $\{X, \overline{X}\}$ such that $|T \cap X|$ is odd. Algorithms for the $T$-cut problem have several applications. They are used, for example, as heuristics to accelerate exact solutions to the Traveling Salesman Problem (TSP) [37]. A minimum $T$-cut of a graph can be found with an algorithm that first finds a cut tree for the graph [43].

Given a graph with non-negative edge weights, the minimum $k$-cut problem consists of finding a set of edges of the graph which removal leaves the graph with $k$ connected components. A 2-approximation algorithm for the minimum $k$-cut problem was proposed by H. Saran and V.V. Vazirani [48] and it consists of removing the $k-1$ edges of minimum capacities of a cut tree of the graph. Generalizations of the $k$-cut problem that are also solved by means of cut trees were proposed by R. Engelberg et al. [19].

Graph clustering algorithms have many applications, for example for finding groups in social networks [38], for gene expression [35], for the classification of web pages [20], among others. Graph clustering algorithms aim at partitioning the vertex set of a graph such that the classes of vertices are internally highly connected but with low connectivity among the classes. The concept of cuts and its variations are naturally related to graph clustering. E. Hartuv and R. Shamir [30] present a clustering algorithm based on minimum cuts and compare their method with others on biological data. G.W. Flake et al. [20] present a graph clustering algorithm that adds a vertex to the graph and partition it with the aid of a cut tree. Their algorithm provably satisfies certain quality guarantees. They apply the algorithm to the classification of scientific papers and web pages. More recently, B. Saha and P. Mitra [46] presented a variation of the clustering algorithm by G.W. Flake et al. for dynamic graphs with node and edge insertions and deletions. That work was extended and corrected by R. Görke et al. [24,25]. Another practical application of graph clustering based on cut trees is on software modularization [34].

Our two previous conference publications introduce (1) the parallel Gus algorithm [15] and the parallel GH algorithm [16]. In this paper we present the problem of computing cut tress in parallel in a unified framework, not only including the parallel Gus and GH algorithms but also introducing the hybrid algorithm and presenting a comprehensive empirical evaluation of all three algorithms.

A heuristic to the parallel GH algorithm was also presented in [16]. The heuristic consists in enumerating all (up to a limit) of the minimum $s$–$t$-cuts in order to choose the most balanced one. The cut enumeration scheme is based on [44] and it is applied on the residual network produced by the maximum flow algorithm. The enumeration algorithm takes linear time to produce each cut. In graphs where a minimum $s$–$t$-cut is likely to be unique, this heuristic may increase the running time of the whole algorithm without any gain. However, if minimum $s$–$t$-cuts are not unique, this procedure is necessary if we expect the algorithm to find balanced cuts, since the maximum flow algorithm tends to output very unbalanced vertex sets.

## 9. Conclusion

Cut trees are widely used combinatorial structures. In this work three parallel cut tree algorithms were presented. Initially, parallel versions of both Gusfield and Gomory–Hu algorithms were described. Although parallelizing Gus algorithm does not present significant obstacles, the same is not true for GH algorithm. However we prove that it is very efficient to update the tree using multiple cuts computed in parallel even if those cuts cross each other. Comparing Gus and GH, there is no best solution, as each algorithm performs better than the other for certain inputs. We propose a hybrid algorithm that can be successfully used for arbitrary inputs. Our main purpose in developing the hybrid algorithm is to remove from the user the need to make a choice of which algorithm to use. Note that the hybrid algorithm tries to make the best decision at each step, it is not about taking a single decision of which algorithm to use.

Experimental results are presented that show that significant speedups can be achieved by the proposed parallel algorithms. We show that the three algorithms achieve significant speedups on real and synthetic graphs. We discuss the trade-offs between the alternatives, each of which presents better results given the characteristics of the input graph. Experimental results comparing the performance of GH algorithm and Gusfield algorithm have been presented. The parallel GH algorithm presents better performance on graphs that contain balanced cuts. The hybrid algorithm proved to have a very consistent performance. It outperformed both other algorithms on several instances, and was faster than the parallel Gomory–Hu algorithm on most instances.

Future work includes investigating further optimizations of the GH algorithm. In particular, strategies for choosing the next pair of vertices to separate should have a positive impact on the performance. While the master process of the Gus algorithm executes few instructions per cut, the GH master process does more work. Therefore, the master process of the GH algorithm is more likely to become a bottleneck as the number of processes increases. Devising strategies to tackle this problem seems to be relevant, as is devising optimized communication strategies between master and slaves. Another item for future work is improving the algorithm for building contracted graphs efficiently. Finally evaluating the algorithms on very large graphs consisting of hundreds of thousands of vertices is also left as future work.

## Acknowledgments

This work was partially supported by FINEP through project CT-INFRA/UFPR. Jaime Cohen was supported by Fundação Araucária/SETI fellowship under Order No. 16/2008. Luiz A. Rodrigues was also supported by Fundação Araucária/SETI, project 19836. This work was partially supported by grant 309143/2012-8 from the Brazilian Research Agency (CNPq).

## References

[1] L.A. Adamic, N. Glance, The political blogosphere and the 2004 U.S. election: divided they blog, in: Proceedings of the 3rd International Workshop on Link Discovery, LinkKDD '05, ACM, New York, NY, USA, 2005, pp. 36–43.
[2] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Rev. Modern Phys. 74 (1) (2002) 47–97.
[3] Y.P. Aneja, R. Chandrasekaran, K.P.K. Nair, Parametric min-cuts analysis in a network, Discrete Appl. Math. 127 (3) (2003) 679–689.
[4] S.R. Arikati, S. Chaudhuri, C.D. Zaroliagis, All-pairs min-cut in sparse networks, J. Algorithms 29 (1) (1998) 82–110.
[5] L. Backstrom, C. Dwork, J. Kleinberg, Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography, in: Proceedings of the 16th Int'L Conference on World Wide Web, WWW '07, ACM, NY, USA, 2007.
[6] D.A. Bader, V. Sachdeva, A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic, in: ISCA International Conference on Parallel and Distributed Computing Systems, 2005, pp. 41–48.
[7] D. Barth, P. Berthomé, M. Diallo, Effects of Capacities Variations on Maximum Flows, Minimum Cuts and Edge Saturation, 2004.
[8] V. Batagelj, A. Mrvar, Pajek datasets, 2006. URL http://vlado.fmf.uni-lj.si/pub/networks/data/.
[9] A.A. Benczúr, Counterexamples for directed and node capacitated cut-trees, SIAM J. Comput. 24 (3) (1995) 505–510.
[10] P. Berthomé, M. Diallo, A. Ferreira, Generalized parametric multi-terminal flows problem, in: H.L. Bodlaender (Ed.), WG, in: Lecture Notes in Computer Science, vol. 2880, Springer, 2003, pp. 71–80.
[11] B. Bollobás, Random Graphs, second ed., Cambridge University Press, 2001.
[12] G. Borradaile, P. Sankowski, C. Wulff-Nilsen, Min st-cut oracle for planar graphs with near-linear preprocessing time, in: Annual IEEE Symposium on Foundations of Computer Science, 2010, pp. 601–610.
[13] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine, C. Stein, Experimental study of minimum cut algorithms, in: SODA '97: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997, pp. 324–333.
[14] B.V. Cherkassky, A.V. Goldberg, On implementing the push-relabel method for the maximum flow problem, Algorithmica 19 (4) (1997) 390–410.
[15] J. Cohen, L. Rodrigues, F. Silva, R. Carmo, A. Guedes, E. Duarte Jr., Parallel implementations of gusfield cut tree algorithm, in: Y. Xiang, A. Cuzzocrea, M. Hobbs, W. Zhou (Eds.), Algorithms and Architectures for Parallel Processing, in: Lecture Notes in Computer Science, vol. 7016, Springer Berlin/Heidelberg, 2011, pp. 258–269.
[16] J. Cohen, L.A. Rodrigues, E.P. Duarte Jr., A parallel implementation of Gomory-Hu cut tree algorithm, in: The 24th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD'2012, New York, NY, 2012, pp. 124–131.
[17] M. Conforti, R. Hassin, R. Ravi, Reconstructing edge-disjoint paths, Oper. Res. Lett. 31 (4) (2003) 273–276.
[18] C. Doll, T. Hartmann, D. Wagner, Fully-dynamic hierarchical graph clustering using cut trees, in: F. Dehne, J. Iacono, J.-R. Sack (Eds.), Algorithms and Data Structures, in: Lecture Notes in Computer Science, vol. 6844, Springer, Berlin, Heidelberg, 2011, pp. 338–349.
[19] R. Engelberg, J. Könemann, S. Leonardi, J.S. Naor, Cut problems in graphs with a budget constraint, J. Discrete Algorithms 5 (2007) 262–279.
[20] G.W. Flake, R.E. Tarjan, K. Tsioutsiouliklis, Graph clustering and minimum cut trees, Internet Math. 1 (4) (2004) 385–408.
[21] A.V. Goldberg, R.E. Tarjan, A new approach to the maximum-flow problem, J. ACM 35 (1988) 921–940.

[22] A.V. Goldberg, K. Tsioutsiouliklis, Cut tree algorithms, in: SODA '99: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, USA, 1999.
[23] R.E. Gomory, T.C. Hu, Multi-terminal network flows, J. Soc. Ind. Appl. Math. 9 (4) (1961) 551–570.
[24] R. Görke, T. Hartmann, D. Wagner, Dynamic graph clustering using minimum-cut trees, in: F. Dehne, M. Gavrilova, J.-R. Sack, C. Tóth (Eds.), Algorithms and Data Structures, in: LNCS, vol. 5664, Springer, Heidelberg, 2009, pp. 339–350.
[25] R. Görke, T. Hartmann, D. Wagner, Dynamic graph clustering using minimum-cut trees, J. Graph Algorithms Appl. 16 (2) (2012) 411–446.
[26] D. Gusfield, A little knowledge goes a long way: Faster detection of compromised data in 2-*D* tables, in: IEEE Symposium on Security and Privacy, 1990, pp. 86–94.
[27] D. Gusfield, Very simple methods for all pairs network flow analysis, SIAM J. Comput. 19 (1990) 143–155.
[28] R. Hariharan, T. Kavitha, D. Panigrahi, Efficient algorithms for computing all low $s$–$t$ edge connectivities and related problems, in: SODA '07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007, pp. 127–136.
[29] R. Hariharan, T. Kavitha, D. Panigrahi, A. Bhalgat, An $O(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs, in: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, in: STOC '07, ACM, New York, NY, USA, 2007, pp. 605–614.
[30] E. Hartuv, R. Shamir, A clustering algorithm based on graph connectivity, Inf. Process. Lett. 76 (4–6) (2000) 175–181.
[31] D. Hartvigsen, Generalizing the all-pairs min cut problem, Discrete Math. 147 (1–3) (1995) 151–169.
[32] R. Hassin, A. Levin, Flow trees for vertex-capacitated networks, Discrete Appl. Math. 155 (4) (2007) 572–578.
[33] B. Hong, Z. He, An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic, IEEE Trans. Parallel Distrib. Syst. 22 (6) (2010) 1025–1033.
[34] C. Jermaine, Computing program modularizations using the $k$-cut method, in: Proceedings of the Sixth Working Conference on Reverse Engineering, 1999, pp. 224–234.
[35] D. Jiang, C. Tang, A. Zhang, Cluster analysis for gene expression data: A survey, IEEE Trans. Knowl. Data Eng. 16 (11) (2004) 1370–1386.
[36] S.N. Kabadi, R. Chandrasekaran, K.P. Nair, Multiroute flows: Cut-trees and realizability, Discrete Optim. 2 (3) (2005) 229–240.
[37] A. Letchford, G. Reinelt, D. Theis, Odd minimum cut-sets and $b$-matchings revisited, SIAM J. Discrete Math. 22 (4) (2008) 1480–1487.
[38] C.F. Mann, D.W. Matula, E.V. Olinick, The use of sparsest cuts to reveal the hierarchical community structure of social networks, Social Networks 30 (3) (2008) 223–234.
[39] C.C. McGeoch, A Guide To Experimental Algorithmics, first ed., Cambridge University Press, New York, NY, USA, 2012.
[40] A. Mitrofanova, M. Farach-Colton, B. Mishra, Efficient and robust prediction algorithms for protein complexes using Gomory-Hu trees, in: R.B. Altman, A.K. Dunker, L. Hunter, T. Murray, T.E. Klein (Eds.), Pacific Symposium on Biocomputing, 2009, pp. 215–226.
[41] H. Nagamochi, T. Ibaraki, Algorithmic Aspects of Graph Connectivity, Cambridge University Press, New York, NY, USA, 2008.
[42] H. Nagamochi, T. Ono, T. Ibaraki, Implementing an efficient minimum capacity cut algorithm, Math. Program. 67 (1) (1994) 325–341.
[43] M.W. Padberg, M.R. Rao, Odd minimum cut-sets and $b$-matchings, Math. Oper. Res. 7 (1) (1982) 67–80.
[44] J.S. Provan, D.R. Shier, A paradigm for listing $(s, t)$-cuts in graphs, Algorithmica 15 (4) (1996) 351–372.
[45] G. Rao, H. Stone, T. Hu, Assignment of tasks in a distributed processor system with limited memory, IEEE Trans. Comput. 28 (1979) 291–299.
[46] B. Saha, P. Mitra, Dynamic algorithm for graph clustering using minimum cut tree, in: Proceedings of the Seventh SIAM International Conference on Data Mining, 2007.
[47] H. Saran, V.V. Vazirani, Finding $k$ cuts within twice the optimal, SIAM J. Comput. 24 (1) (1995) 101–108.
[48] H. Saran, V.V. Vazirani, Finding $k$ cuts within twice the optimal, SIAM J. Comput. 24 (1) (1995) 101–108.
[49] C.-P. Schnorr, Bottlenecks and edge connectivity in unsymmetrical networks, SIAM J. Comput. 8 (2) (1979) 265–274.
[50] G. Storchi, P. Dell'Olmo, M. Gentili, Road network of the city of Rome, Contributed collection - 9th DIMACS Implementation Challenge: Shortest Paths, 1999.
[51] N. Tuncbag, F.S. Salman, O. Keskin, A. Gursoy, Analysis and network representation of hotspots in protein interfaces using minimum cut trees, Proteins: Structure, Function, and Bioinformatics 78 (10) (2010) 2283–2294.
[52] D.J. Watts, S.H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.
[53] X. Zhang, W. Liang, H. Jiang, Flow equivalent trees in undirected node-edge-capacitated planar graphs, Inf. Process. Lett. 100 (3) (2006) 110–115.

**Jaime Cohen** is Associate Professor at State University of Paraná at Ponta Grossa, Brazil. He obtained his Ph.D. in Computer Science from the Federal University of Paraná, UFPR, Brazil, 2013, M.Sc. in Computer Science from Rutgers University, U.S., 2002, and The State University of Campinas (Unicamp), Brazil, 1995, and B.Sc. in Computer Science from the Federal University of Paraná, 1993. His main research interests include computer networks, parallel systems and graph algorithms.

**Luiz A. Rodrigues** is Associate Professor at Western Paraná State University, Brazil. He obtained his Ph.D. in Computer Science from the Federal University of Paraná, 2014, M.Sc. in Computer Science from the Federal University of Rio Grande do Sul, Brazil 2006, and B.Sc. in Computer Science from the Western Paraná State University, 2003. His main research interests include computer networks, fault tolerance, distributed and parallel systems.

**Elias P. Duarte Jr.** is Associate Professor at Federal University of Paraná (UFPR), Curitiba, Brazil, and chair of the Brazilian National Laboratory on Computer Networks (LARC). He has a Ph.D. degree in Computer Science from Tokyo Institute of Technology, Japan, 1997, M.Sc. Telecommunications from the Polytechnical University of Madrid, Spain, 1991, and B.Sc. and M.Sc. in Computer Science from Federal University of Minas Gerais, Brazil, 1987 and 1991, respectively. Research interests include Computer Networks and Distributed Systems, their Dependability and Algorithms. Prof. Elias has published more than 100 peer-reviewed papers, and has supervised more than 100 students, including graduate and undergraduate levels. He has served on several Editorial Boards, and chaired and served on the TPC of several conferences and workshops in his fields of interest. He chaired the Special Interest Group on Fault Tolerant Computing of the Brazilian Computing Society (2005–2007); the Graduate Program in Computer Science of UFPR (2006–2008). He is a member of the Brazilian Computer Society and the IEEE.