Parallel Implementations of Gusfield's Cut Tree Algorithm

Jaime Cohen[1,2], Luiz A. Rodrigues[1,3], Fabiano Silva[1], Renato Carmo[1], André L. P. Guedes[1], and Elias P. Duarte Jr.[1]

[1] Federal University of Paraná, Department of Informatics
Curitiba, Brazil
[2] Paraná State University at Ponta Grossa, Department of Informatics
Ponta Grossa, Brazil
[3] Western Paraná State University, Department of Computer Science
Cascavel, Brazil
{jaime,larodrigues,fabiano,elias,renato,andre}@inf.ufpr.br

Abstract. This paper presents parallel versions of Gusfield's cut tree algorithm. Cut trees are a compact representation of the edge-connectivity between every pair of vertices of an undirected graph. Cut trees have many applications in combinatorial optimization and in the analysis of networks originated in many applied fields. However, surprisingly few works have been published on the practical performance of cut tree algorithms. This paper describes two parallel versions of Gusfield's cut tree algorithm and presents extensive experimental results which show a significant speedup on most real and synthetic graphs in our dataset.

Keywords: Graph edge-connectivity, Cut tree algorithms, MPI, OpenMP.

1 Introduction

A cut tree is an important combinatorial structure able to represent the edge-connectivity between all pairs of nodes of undirected graphs. Cut trees have many direct applications, e.g. [21,2,19,23], and algorithms for cut tree construction are used as subroutines to solve other important combinatorial problems in areas such as routing, graph partitioning and graph connectivity, e.g. [22,9,18,14].

Despite of the numerous applications of cut trees, few studies were done on the practical performance of cut tree algorithms. Distributed or parallel implementations of cut tree algorithms are not available and experimental studies of such implementations have not yet been published.

This paper describes two parallel versions of Gusfield's cut tree algorithm and presents experimental results that show a significant speedup on most real and synthetic graphs in our dataset.

The sequential Gusfield's algorithm consists of $n-1$ calls to a maximum flow algorithm and the parallel version optimistically makes those calls in parallel. Even though the iterations may depend on previous ones, the experiments show that this a priori assumption rarely affects the performance of the algorithm and

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 259–270, 2011.

a significant speedup can be achieved. The implementations were tested using real and synthetic graphs representing potential applications.

This paper is organized as follows. In Section 2 we present an overview of the previous research on cut trees. Section 3 gives basic graph theory definitions including the definition of cut trees. Section 4 describes Gusfield's algorithm in its sequential and parallel versions. Section 5 defines the environment and the parameters used in the experiments. The results are presented and discussed in Section 6. Finally, in Section 7, we present the conclusions and future work.

2 Related Work

The concept of cut trees and a cut tree construction algorithm were first discovered by R. E. Gomory and T. C. Hu in 1961 [13]. Another cut tree algorithm was discovered by D. Gusfield in 1990 [15]. Let us call them *GH algorithm* and *Gus algorithm*, respectively. Both algorithms require the computation of $n-1$ minimum s-t-cuts (or, equivalently, maximum flows). We will discuss their differences in Section 4. The fastest deterministic maximum flow algorithm, by A. V. Goldberg and S. Rao [10], runs in time $\tilde{O}(\min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}m)$. An extra $O(n)$ factor gives the worst case time complexity of the best deterministic algorithm to find a cut tree of a weighted undirected graph.

The only published experimental study on cut tree algorithms is the paper "Cut Tree Algorithms" by A. V. Goldberg and K. Tsioutsiouliklis [12]. They compared the GH algorithm and Gus algorithm. They concluded that an optimized version of the GH algorithm is more robust than Gus algorithm, justified by the fact that at a few instances, Gus algorithm had a much worse running time. We note, however, that most of those instances belong to synthetic classes of graphs for which balanced cuts exist by construction. Despite of that, their implementation of Gus algorithm was the fastest one more times than any of their implementations of GH algorithm.

3 Definitions

A *graph* G is a pair $(V(G), E(G))$ where $V(G)$ is a finite set of elements called vertices and $E(G)$ is a set of unordered pairs of vertices called edges. A *capacitated graph* is a graph G associated with a function $c : E(G) \to \mathbb{Z}_+$ defining the *capacities* of the edges in $E(G)$.

Let G be a capacitated graph. 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 $E_G(X, \overline{X}) = \{\{u, v\} \in E(G) \colon u \in X, v \in \overline{X}\}$ contains the edges that *cross* the cut $\{X, \overline{X}\}$. The *capacity* of the cut $\{X, \overline{X}\}$ is $c(X, \overline{X}) = \sum_{e \in E_G(X,\overline{X})} c(e)$.

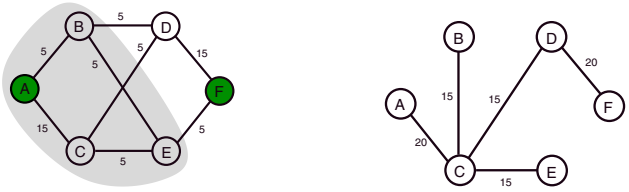
Let s and t be two vertices of G. An *s-t-cut* of G is a cut $\{X, \overline{X}\}$ such that $s \in X$ and $t \in \overline{X}$. A *minimum s-t-*cut is an s-t-cut of minimum capacity. A cut $\{\{s\}, V - \{s\}\}$ is called a *trivial cut*. The *local connectivity between s and t* in

$V(G)$, denoted by $\lambda_G(s,t)$, is the capacity of a minimum s-t-cut. 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 e T. C. Hu [13] showed that only $n-1$ maximum flow computations are necessary. The solution to the problem consists of constructing a weighted tree that represents the values of all pairwise local connectivities.

A *flow equivalent tree* of a graph G 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 equals 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* [20].



(a) Example of an undirected weighted graph and a minimum A-F-cut

(b) Cut tree

Fig. 1. Examples of an undirected graph, a minimum cut and a cut tree

Fig. 1(b) shows a cut tree of the graph of Fig. 1(a). Figure 1(a) shows a minimum cut between vertices A and F induced by the removal of the edge $\{C, D\}$ from the tree.

4 Cut Tree Algorithms

Two cut tree algorithms for weighted undirected graphs are known: the *Gomory-Hu's* algorithm [13] and *Gusfield's* algorithm [15]. Both algorithms make $n-1$ calls to a maximum flow algorithm and they use a divide and conquer approach. The algorithms differ in their data structure: while the former algorithm contracts the original graph, the latter computes all cuts on the input graph. After describing Gus algorithm and comparing it with GH algorithm, we discuss the choice to parallelize Gus algorithm.

4.1 Gusfield's Algorithm - Sequential Version

The sequential Gus algorithm consists of $n-1$ iterations, each of them containing a call to a maximum flow algorithm on the input graph. See the pseudocode in Algorithm 1. 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 maximum flow algorithm, a minimum s-t-cut is found. The tree is reshaped as follows: every neighbor t' of t, $t' > s$, that sits on the side of s of the cut gets disconnected from t and gets connected to s. The algorithm ends when each node from 2 to n has been the source of 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 change in the algorithm causes it produce a cut tree: in line 8, allow any neighbor of t that belongs to X to become a neighbor of s.

Algorithm 1. Sequential Gusfield's 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: $tree_i \leftarrow 1$
// $|V_G| - 1$ maximum flow 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** $T = (V_T, E_T, f)$

4.2 Parallelization of Cut Tree Algorithms

A parallel solution to the cut tree problem involves two choices: the algorithm to implement (Gus or GH algorithm) and the level of parallelism to explore.

First, we will argue that Gus algorithm is a better choice for parallelization. GH algorithm is similar to Gus algorithm but after finding a minimum s-t-cut, it contracts each side of the cut and recurse on each of the graphs obtained.

The existence of balanced minimum cuts favors the GH algorithm because the graph size is reduced at each iteration. Most real graphs and graphs generated by random models such as Erdös-Rényi (ER) model and the preferential attachment model rarely have balanced cuts. That means not only that the subproblems do

not get much smaller, but also that load balancing among processors does not occur. Indeed, in our preliminary experiments with the sequential implementations used in [12], Gus algorithm outperformed GH algorithm on all large real graphs in our dataset and all graphs generated by the ER model and the BA model. This can only happen when balanced cuts are rare or nonexistent.

Gus algorithm always executes maximum flow algorithms on the same input graph. Therefore, each process can simply start with a copy of the graph and no further transmissions of the input graph are necessary. The interprocess communication only comprises of source and destination pairs, cuts and connectivity values. On the other hand, a parallel GH algorithm would require more interprocess communication because either the graphs or the contracted subsets of vertices should be sent from the master to the slaves threads or processes.

For all these reasons, it seems that Gus algorithm is more amenable to parallelization than GH algorithm. Nonetheless, further research can be done to explore possible ways to parallelize GH algorithm to produce a faster algorithm for graphs that contain balanced cuts. A hybrid algorithm is another possibility.

With respect to the level of parallelism to explore, we note that the maximum flow problem is hard to parallelize. Despite of extensive research on the problem, experimental studies of parallel max flow algorithms report very modest speedups [3,16] due to the synchronization requirements. On the other hand, Gus algorithm can run one sequential maximum flow per thread or process, without synchronization, and achieve high speedups, as we will see in Section 6.

4.3 MPI Version

MPI has emerged in the early 1990s as a set of libraries for process management and message exchange in distributed memory architectures [6]. The advantage of this solution is scalability, because it uses independent computers that can be easily connected through the network. However, in comparison with OpenMP, MPI requires a larger reorganization of the code sequence to obtain the parallel solution, usually based on the master/slave model.

The pseudocode of the MPI implementation appears in Algorithm 2. 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, inputs of the maximum flow algorithm. When a slave finishes a task, it sends the value of the maximum flow and the cut 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. If s and t are not neighbors by the time of processing the task result, then we say that the task "failed" and another task having s as the source is produced (line 18). The stop condition of the `while` loop in line 7 guarantees that $|V_G| - 1$ successful receives are executed.

The structure of the graph influences the number of failed 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 tasks that fails.

Algorithm 2. MPI Gusfield's Algorithm

Input: $G = (V_G, E_G, c)$, $proc_j$ processors $(0 \leq j < P)$
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 ← 1 to |V_G| do
 3:       tree_i ← 1
 4:    for s ← 2 to P do
 5:       send Task(s, tree_s) to proc_{s−1}
 6:    s ← P + 1
 7:    while s < P + |V_G| do
 8:       receive result s', t', flow, {X, X̄} from proc_j
 9:       if tree_{s'} = t' then // update the tree
10:          flow_{s'} = flow
11:          for all u ∈ V_G, u > s' do
12:             if tree_u = t' and u ∈ X then
13:                tree_u ← s'
14:          if s ≤ |V_G| then
15:             send a new Task(s, tree_s) to proc_j
16:          s ← s + 1
17:       else // failed, 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:       flow, {X, X̄} ← MaxFlow(s, t)
24:       send s, t, flow, {X, X̄} to proc_0
```

4.4 OpenMP Version

OpenMP is an API (Application Programming Interface) designed for parallel programming on shared memory architectures (SMP). This API offers policies that can be added to code sequences in Fortran, C and C++ which define how work is shared among threads to be executed on different processors/cores and how data on shared memory is accessed [6].

The adaptation of Gusfield's algorithm to OpenMP was done by the parallelization of the main loop which performs the $n-1$ calls to the maximum flow routine. See Algorithm 3. Let k be a predefined maximum number of threads. The algorithm uses an optimistic strategy and finds k minimum s-t-cuts in parallel. Each of these cuts is a candidate for changing the tree. After computing its s-t-cut, each thread verifies if the destination t is still a neighbor of s in the tree. If it is not, meaning that the cut of a previous successful thread separated s from t, then we say that the thread "failed" and another s-t-cut is computed in order to separate s from its new neighbor. If the thread succeeds in separating s from t, it proceeds to update the tree. The test and the operations on the tree are done inside a critical region, what guarantees the correctness of the algorithm.

Algorithm 3. OpenMP Gusfield's Algorithm

Input: $G = (V_G, E_G, c)$, P the number of processors
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of G
1: **for** $i \leftarrow 1$ **to** $|V_G|$ **do**
2: $tree_i \leftarrow 1$
// $|V_G|-1$ maximum flow iterations
3: **for** $s \leftarrow 2$ to $|V_G|$ **in parallel do**
4: $proc_j$ is an idle process
5: $succeed \leftarrow false$
6: **repeat**
7: $t \leftarrow tree_s$
8: $proc_j$ executes $flow$, $\{X, \overline{X}\} \leftarrow$ MaxFlow(s, t)
// omp critical region
9: **if** $tree_s = t$ **then**
10: $proc_j$ updates the $tree$
11: $succeed \leftarrow true$
12: **until** $succeed$
// Build T as in lines 10-14 of the sequential algorithm
13: **return** T

5 Experimental Setup

The experiments with MPI ran on a cluster with 15 Intel Core 2 Quad 2.4 GHz, 2 Gbyte memory and 4096 Kbyte cache, interconnected by a Gigabit Ethernet network. The experiments with OpenMP used a Quad-Core 2.8 GHz AMD Opteron workstation with 8 cores, 8 Gbyte memory and 512 Kbyte cache. The code was written in C language and compiled with `gcc` (optimization level `-O3`). Our implementations are based on the push-relabel maximum flow algorithm [11] code HIPR[1], developed by B.V. Cherkassky and A.V. Goldberg [8]. We implemented the flow equivalent tree version of Gus algorithm.

The dataset was composed of 10 graphs as shown in Table 1. The first four graphs come from real data: 2 collaboration networks [17,4], a power grid network [24] and a peer-to-peer network [17]. Two networks were generated by random models: the Erdös-Rényi (ER) model [5] and the preferential attachment model [1]. The other 4 graphs are synthetic graphs of different types that have been used as benchmarks for min cut and cut tree algorithms [7,12].

Table 1. Sizes of the graphs in the dataset

Graph	$\|V\|$	$\|E\|$
CA-HepPh	11,204	235,238
GeoComp	3,621	9,461
PowerGrid	4,941	6,594
P2P-Gnutella	10,876	39,994
BA	10,000	49,995

Graph	$\|V\|$	$\|E\|$
ER	10,000	49,841
Dblcyc	1,024	2,048
NOI	1,500	562,125
Path	2,000	21,990
Tree	1,500	563,625

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

Speedups were calculated as $S = T_S/T_P$, where T_S is the time of the sequential implementation of Gusfield's algorithm and T_P is the execution time in parallel on P processes. The efficiency was calculated using $E = S/P$.

6 Experimental Results

Initial tests were performed in order to define the best scheduling strategies for the tasks. For MPI, the best results were achieved running the master process and one slave in one machine and the other slaves in individual machines. For OpenMP, the loop scheduling with best performance was the OpenMP dynamic scheduling that distributes tasks during runtime in order to balance the load. All further experiments used these strategies.

Results with MPI. Let P be the number of processes. For $P = 2$, the algorithm runs the master and 1 slave process. Because the slave waits for a task, the execution is sequential and, therefore, $S_2 \leq 1$ and $E_2 \leq 0.50$. For $P > 2$, at least one of the $P-1$ slaves waits while the master tries to update the tree. Let us approximate the sequential time by $T_s = T_F + T_T$, where T_F is the time to compute $|V| - 1$ maximum flows and T_T is the time to update the tree. A lower bound on the parallel time can be computed by assuming the best case where the maximum flow work is divided evenly by the $P-1$ slave processes. Each slave takes $\frac{T_F}{P-1}$ time to compute maximum flows and waits idle approximately $\frac{T_T}{P-1}$ time for new tasks. Therefore, the parallel running time with P processes is bounded by $T_P \geq \frac{T_F+T_T}{P-1}$. The speedup can be upper bounded by $P-1$, because

$$S_P = \frac{T_s}{T_P} \leq \frac{T_F + T_T}{\frac{T_F+T_T}{P-1}} = P - 1.$$

An upper bound for the efficiency is $E_P = \frac{S_P}{P} \leq \frac{P-1}{P}$.

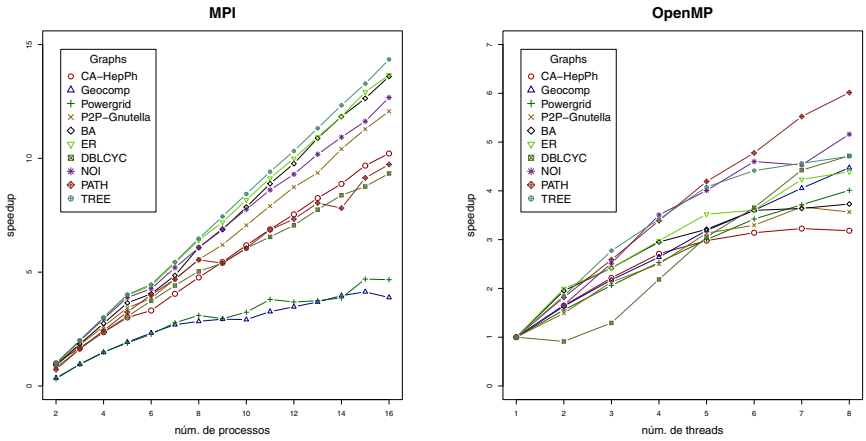


Fig. 2. Speedups with MPI e OpenMP

Table 2. MPI results with the running times, speedups (S) and efficiency (E) of each graph in the dataset

Num procs	CA-HepPh			Geocomp			Powergrid			P2P-Gnutella			BA		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
sequential	475.01	-	-	2.25	-	-	3.85	-	-	65.23	-	-	87.71	-	-
2	479.93	0.99	0.49	6.45	0.35	0.17	12.31	0.31	0.16	73.24	0.89	0.45	95.49	0.92	0.46
3	283.42	1.68	0.56	2.37	0.95	0.32	4.01	0.96	0.32	36.25	1.80	0.60	47.58	1.84	0.61
4	201.56	2.36	0.59	1.53	1.47	0.37	2.57	1.50	0.37	24.88	2.62	0.66	31.94	2.75	0.69
5	157.89	3.01	0.60	1.17	1.93	0.39	2.04	1.89	0.38	19.01	3.43	0.69	24.04	3.65	0.73
6	143.35	3.31	0.55	0.97	2.32	0.39	1.69	2.27	0.38	16.94	3.85	0.64	21.66	4.05	0.67
7	117.33	4.05	0.58	0.84	2.69	0.38	1.39	2.77	0.40	13.92	4.69	0.67	18.07	4.85	0.69
8	99.64	4.77	0.60	0.79	2.84	0.35	1.24	3.10	0.39	11.73	5.56	0.69	14.42	6.08	0.76
9	87.25	5.44	0.60	0.77	2.94	0.33	1.30	2.96	0.33	10.52	6.20	0.69	12.77	6.87	0.76
10	76.84	6.18	0.62	0.77	2.91	0.29	1.19	3.24	0.32	9.24	7.06	0.71	11.16	7.86	0.79
11	69.05	6.88	0.63	0.69	3.27	0.30	1.01	3.80	0.35	8.25	7.91	0.72	9.87	8.89	0.81
12	63.01	7.54	0.63	0.65	3.48	0.29	1.04	3.69	0.31	7.47	8.73	0.73	8.98	9.77	0.81
13	57.54	8.26	0.64	0.61	3.69	0.28	1.03	3.75	0.29	6.97	9.36	0.72	8.06	10.89	0.84
14	53.50	8.88	0.63	0.57	3.97	0.28	0.99	3.87	0.28	6.27	10.41	0.74	7.41	11.83	0.85
15	49.06	9.68	0.65	0.54	4.14	0.28	0.82	4.70	0.31	5.78	11.29	0.75	6.94	12.63	0.84
16	46.52	10.21	0.64	0.58	3.88	0.24	0.82	4.67	0.29	5.41	12.06	0.75	6.45	13.60	0.85
Num procs	**DBLCYC**			**ER**			**NOI**			**PATH**			**TREE**		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
sequential	11.13	-	-	104.23	-	-	384.84	-	-	5.42	-	-	236.78	-	-
2	13.83	0.81	0.40	109.90	0.95	0.47	385.61	1.00	0.50	7.52	0.72	0.36	237.10	1.00	0.50
3	6.88	1.62	0.54	52.40	1.99	0.66	194.59	1.98	0.66	3.32	1.63	0.54	117.89	2.01	0.67
4	4.66	2.39	0.60	34.82	2.99	0.75	130.75	2.94	0.74	2.22	2.44	0.61	78.53	3.02	0.75
5	3.65	3.05	0.61	26.17	3.98	0.80	99.07	3.88	0.78	1.67	3.25	0.65	58.90	4.02	0.80
6	2.98	3.74	0.62	23.70	4.40	0.73	90.11	4.27	0.71	1.35	4.00	0.67	53.11	4.46	0.74
7	2.52	4.41	0.63	19.30	5.40	0.77	74.08	5.20	0.74	1.16	4.69	0.67	43.45	5.45	0.78
8	2.21	5.04	0.63	16.27	6.40	0.80	63.36	6.07	0.76	0.98	5.54	0.69	36.59	6.47	0.81
9	2.06	5.39	0.60	14.49	7.19	0.80	55.65	6.92	0.77	1.01	5.39	0.60	31.79	7.45	0.83
10	1.84	6.05	0.60	12.75	8.17	0.82	49.59	7.76	0.78	0.89	6.06	0.61	28.07	8.43	0.84
11	1.70	6.55	0.60	11.42	9.13	0.83	44.70	8.61	0.78	0.79	6.86	0.62	25.16	9.41	0.86
12	1.58	7.06	0.59	10.43	10.00	0.83	41.38	9.30	0.77	0.74	7.33	0.61	22.94	10.32	0.86
13	1.44	7.75	0.60	9.53	10.93	0.84	37.81	10.18	0.78	0.67	8.04	0.62	20.91	11.32	0.87
14	1.33	8.38	0.60	8.80	11.84	0.85	35.21	10.93	0.78	0.69	7.81	0.56	19.20	12.33	0.88
15	1.27	8.76	0.58	8.07	12.92	0.86	33.10	11.63	0.78	0.59	9.14	0.61	17.83	13.28	0.89
16	1.19	9.34	0.58	7.63	13.66	0.85	30.37	12.67	0.79	0.56	9.73	0.61	16.50	14.35	0.90

Results of the MPI implementation appear in Table 2 and in Fig. 2. The running times are the average of 10 runs. Efficiencies for the instances TREE and ER achieve the upper bound $\frac{P-1}{P}$ for P between 2 and 5 and they are not far from the maximum for greater values of P.

The speedups are consistently high for all graphs but POWERGRID and GEOCOMP which are the easiest instances for the sequential algorithm. The efficiency drops as the number of fails increases as shown in Fig. 3.

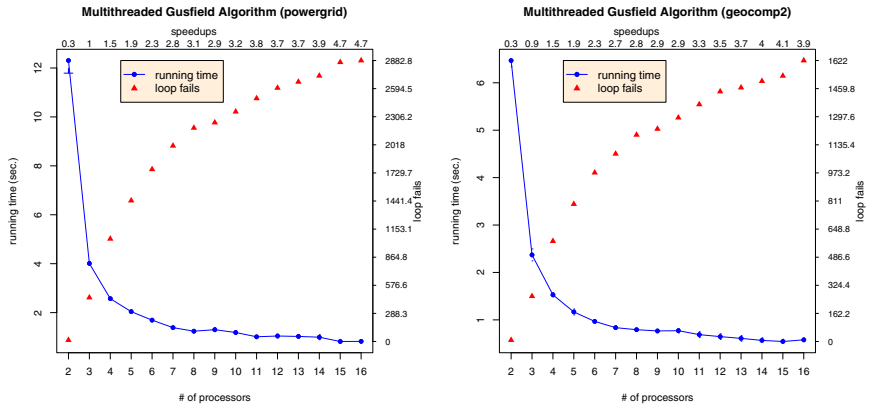
Results with OpenMP. Results for the parallel Gusfield's algorithm with OpenMP on a 8-core computer are reported in Table 3 and in Fig. 2. Efficiency was above 0.50 on most executions.

The best speedup obtained with OpenMP and 8 threads in real graphs was 4.48 and in synthetic graphs was 6.01. The worst speedup in real and synthetic graphs for 8 threads were 3.18 and 3.73, respectively. We report other positive results on a 16-core computer where the implementation achieved the best speedup of 9.4 running 16 threads on the NOI5 graph. An experiment on an ER

Table 3. OpenMP results with the running times, speedups (S) and efficiency (E) of each graph in the dataset

Num threads	CA-HepPh			Geocomp			Powergrid			P2P-Gnutella			BA		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	517.07	-	-	2.10	-	-	3.03	-	-	65.79	-	-	87.66	-	-
2	314.32	1.65	0.82	1.29	1.63	0.82	1.94	1.56	0.78	44.10	1.49	0.75	44.94	1.95	0.98
3	233.42	2.22	0.74	0.97	2.17	0.72	1.47	2.06	0.69	30.87	2.13	0.71	36.24	2.42	0.81
4	190.77	2.71	0.68	0.80	2.64	0.66	1.20	2.53	0.63	26.31	2.50	0.63	29.71	2.95	0.74
5	173.56	2.98	0.60	0.66	3.19	0.64	1.01	3.00	0.60	21.00	3.13	0.63	27.29	3.21	0.64
6	164.59	3.14	0.52	0.58	3.60	0.60	0.88	3.42	0.57	19.96	3.30	0.55	24.35	3.60	0.60
7	160.22	3.23	0.46	0.52	4.05	0.58	0.82	3.72	0.53	17.93	3.67	0.52	24.08	3.64	0.52
8	162.48	3.18	0.40	0.47	4.48	0.56	0.76	4.01	0.50	18.44	3.57	0.45	23.50	3.73	0.47

Num threads	DBLCYC			ER			NOI			PATH			TREE		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	10.12	-	-	115.16	-	-	585.37	-	-	5.60	-	-	315.09	-	-
2	11.09	0.91	0.46	57.71	2.00	1.00	352.84	1.66	0.83	3.08	1.82	0.91	170.51	1.85	0.92
3	7.83	1.29	0.43	47.71	2.41	0.80	232.72	2.52	0.84	2.16	2.59	0.86	113.66	2.77	0.92
4	4.63	2.18	0.55	38.69	2.98	0.74	166.95	3.51	0.88	1.65	3.39	0.85	92.18	3.42	0.85
5	3.31	3.06	0.61	32.70	3.52	0.70	146.06	4.01	0.80	1.33	4.19	0.84	77.23	4.08	0.82
6	2.77	3.66	0.61	31.96	3.60	0.60	127.23	4.60	0.77	1.17	4.78	0.80	71.38	4.41	0.74
7	2.29	4.43	0.63	27.19	4.23	0.60	129.23	4.53	0.65	1.01	5.52	0.79	69.02	4.57	0.65
8	2.14	4.72	0.59	26.23	4.39	0.55	113.44	5.16	0.65	0.93	6.01	0.75	66.93	4.71	0.59



(a) Powergrid (b) Geocomp

Fig. 3. Some graphs with results of the MPI executions. Dots represent real time. Triangles represent the number of loop fails. The speedups appear on the top margin of each graph.

graph with 3000 nodes and average degree 5, achieved 9.2 of speedup running 10 threads with 0.92 of parallel efficiency.

The OpenMP implementation is not as scalable as the MPI implementation because on the multi-core computers loop fails are not the only factor affecting performance and memory access becomes a bottleneck as the number of threads increases. Also, since each thread requires its own copy of the graph, memory

usage increases. Our implementation with OpenMP scales well up to 10 to 16 threads depending on the instance.

7 Conclusion

Cut trees are a widely used combinatorial structure. Two sequential cut tree algorithms are known, but no parallel implementation of them has been reported in the past. This paper presents results of parallel implementations of Gusfield's cut tree algorithm using OpenMP and MPI. The results show that parallel versions of the algorithm can achieve high speedups. The parallel solution is relatively simple, requiring few changes in the original code, particularly with OpenMP. While OpenMP allows a greater control over the running threads, MPI provides more scalability. The implementations are complementary as they can explore the benefits of multi-core machines and computer clusters.

Future work includes a formal analysis of the running times and the scalability of the solutions and an experimental comparison of Gusfield's Algorithm with Gomory-Hu's Algorithm. Heuristics to improve efficiencies and the scalability of the solutions can be explored.

Acknowledgments. This work was partially supported by FINEP through project CT-INFRA/UFPR. Jaime Cohen was on a paid leave of absence from UEPG to conclude his Ph.D. and was supported by a 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 grants 304013/2009-9 and 308692/2008-0 from the Brazilian Research Agency (CNPq).

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74(1), 47–97 (2002)
2. Backstrom, L., Dwork, C., Kleinberg, J.: 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 2007. ACM, NY (2007)
3. Bader, D.A., Sachdeva, V.: A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In: ISCA PDCS (2005)
4. Batagelj, V., Mrvar, A.: Pajek datasets (2006), `http://vlado.fmf.uni-lj.si/pub/networks/data/`
5. Bollobás, B.: Random Graphs, 2nd edn. Cambridge University Press, Cambridge (2001)
6. Chapman, B., Jost, G., Van der Pas, R.: Using OpenMP: portable shared memory parallel programming. MIT Press, Cambridge (2008)
7. Chekuri, C.S., Goldberg, A.V., Karger, D.R., Levine, M.S., Stein, C.: Experimental study of minimum cut algorithms. In: SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms. pp. 324–333. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)

8. Cherkassky, B.V., Goldberg, A.V.: On Implementing the Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19(4), 390–410 (1997)
9. Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph clustering and minimum cut trees. Internet Mathematics 1(4) (2003)
10. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. J. ACM 45, 783–797 (1998)
11. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM 35, 921–940 (1988)
12. Goldberg, A.V., Tsioutsiouliklis, K.: Cut tree algorithms. In: SODA 1999: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, USA (1999)
13. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics 9(4), 551–570 (1961)
14. Görke, R., Hartmann, T., Wagner, D.: Dynamic Graph Clustering Using Minimum-Cut Trees. In: Dehne, F., Gavrilova, M., Sack, J.R., Tóth, C. (eds.) WADS 2009. LNCS, vol. 5664, pp. 339–350. Springer, Heidelberg (2009)
15. Gusfield, D.: Very simple methods for all pairs network flow analysis. SIAM J. Comput. 19, 143–155 (1990)
16. Hong, B., He, Z.: An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic. IEEE Transactions on Parallel and Distributed Systems 99 (2010)
17. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. In: ACM Trans. on Knowledge Discovery from Data, ACM TKDD (2007)
18. Letchford, A., Reinelt, G., Theis, D.: Odd minimum cut-sets and b-matchings revisited. SIAM Journal on Discrete Mathematics 22(4) (2008)
19. Mitrofanova, A., Farach-Colton, M., Mishra, B.: Efficient and robust prediction algorithms for protein complexes using gomory-hu trees. In: Altman, R.B., Dunker, A.K., Hunter, L., Murray, T., Klein, T.E. (eds.) Pacific Symposium on Biocomputing, pp. 215–226 (2009)
20. Nagamochi, H., Ibaraki, T.: Algorithmic Aspects of Graph Connectivity. Cambridge University Press, New York (2008)
21. Rao, G., Stone, H., Hu, T.: Assignment of tasks in a distributed processor system with limited memory. IEEE Transactions on Computers 28, 291–299 (1979)
22. Saran, H., Vazirani, V.V.: Finding k cuts within twice the optimal. SIAM J. Comput. 24(1), 101–108 (1995)
23. Tuncbag, N., Salman, F.S., Keskin, O., Gursoy, A.: Analysis and network representation of hotspots in protein interfaces using minimum cut trees. Proteins: Structure, Function, and Bioinformatics 78(10), 2283–2294 (2010)
24. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393(6684), 440–442 (1998)