A Parallel Implementation for Computing the Region-Adjacency-Tree of a Segmentation of a 2D Digital Image

Fernando Díaz-del-Río[1], Pedro Real[1(✉)], and Darian Onchis[2]

[1] H.T.S. Informatics' Engineering, University of Seville, Seville, Spain
fdiaz@atc.us.es, real@us.es

[2] Faculty of Mathematics, University of Vienna, Vienna, Austria
darian.onchis@univie.ac.at

Abstract. A design and implementation of a parallel algorithm for computing the Region-Adjacency Tree of a given segmentation of a 2D digital image is given. The technique is based on a suitable distributed use of the algorithm for computing a Homological Spanning Forest (HSF) structure for each connected region of the segmentation and a classical geometric algorithm for determining inclusion between regions. The results show that this technique scales very well when executed in a multicore processor.

Keywords: Digital image · Segmentation · RAG · Parallel algorithm

1 Introduction

An important high level processing task in image understanding is to find a topological and structured description of the segmentation previously performed at low level processing, mostly independent of size and contrast characteristics of the extracted regions. The most usual representations are graph-based and the reasoning used in the region-node connectivity calculus involves two topological properties: adjacency and inclusion. The Region Adjacency Graph (RAG) [18–20] can be considered as the germ notion of all these models and it is composed of a set of nodes, one by region of the image, and there is an edge between two nodes if and only if the two corresponding regions are neighbors. Within the traditional context of square pixel-based 2-dimensional digital images, the most usual adjacency relationships between regions employed for the RAG are 4 or 8-neighborhoods. In order to exclusively highlight the ambient isotopic property "to be surrounded by" as adjacency relationship between regions or boundaries of regions, the notion of *Region-Adjacency Tree* (or RAG tree, for short) (also called *homotopy tree*, *inclusion tree* or *topological tree*) is created [6,20,21]. Restricted to binary 2D digital images, the RAG tree contains all the homotopy type information of the foreground object (black object) but the converse is, in general, not true [21]. Aside from image understanding applications [2],

© Springer International Publishing Switzerland 2016
F. Huang and A. Sugimoto (Eds.): PSIVT 2015 Workshops, LNCS 9555, pp. 98–109, 2016.
DOI: 10.1007/978-3-319-30285-0_9

RAG trees have encountered exploitation niches in geoinformatics, rendering, dermatoscopics image, biometrics,... [3,4].

Common operations for modifying RAGs are splitting and merging of regions. Most of the methods using these split-and-merge techniques are employed in "improving the quality of initial oversegmentations". Some split-and-merge RAG-based methods consider low-level information of the image for the region merging task [10,12,13,22]. Others take into account not only local knowledge of the image but also the extraction of "global" information about image models [1,8,9,23].

The contribution of this paper consists of the design and implementation of an parallel algorithm for computing the RAG of a given segmentation, based on a suitable distributed use of the algorithm for computing an HSF structure for each 4-connected region. The paper has the following sections. Section 2 is devoted to recall the machinery for computing the structures needed for HSF construction already given in [7]. Next, a parallel algorithm for constructing RAG starting from a HSF is showed in Sects. 3 and 4. Section 5 is devoted to show the advantages of our parallel implementation and to present the scalability results, and finally the conclusions are summarized.

2 About HSF Framework for RAG Tree Computation

In this paper, we specify an algebraic-topological technique for parallel computing a RAG of a given segmentation. This modus operandi have already used in [7,16] for developing a topologically consistent framework for parallel computation in 2D digital image context. Succinctly, starting from a 2-dimensional abstract cell (cubical) complex $\mathcal{C}^2$ analogous of a digital image I in which the pixels are the 0-cells, we compute in parallel another cell complex $\mathcal{D}^2$ topologically equivalent to the first one, having the same 0-cells but in which the number of face and coface relationships in the canonically associated partial-order set of cells is reduced to a "minimum". Taking into account that $\mathcal{C}^2$ is contractible, $\mathcal{D}^2$ will have only one cell (critical 0-cell) with no coface neighbors. In [7], it is employed the name of *Morse spanning forest* ($MrSF$) for $\mathcal{D}^2$. Let us note that we use the nickname of $MrSF$ instead of that of MSF, due to the fact that MSF is an usual abbreviation for Minimum Spanning Forest in graph theory. If the concrete strategy for "cutting" coface connexions used in any processing unit associated $\mathcal{P}$ to each 0-cell mainly depends on the localization in $\mathcal{P}$ of the pixels of the "foreground" and the "background" of a given segmentation $\mathcal{S} = \{R_i\}_{i=0}^{k}$ of I, the $MrSF$ $\mathcal{D}_2$ is, in principle, a good candidate structure for topologically analyzing $\mathcal{S}$ and, in particular, to compute its associated RAG. A much better and natural strategy for the RAG-problem is to compute a sort of optimal $MrSF$ (called HSF in [7]) for each region R_i of $\mathcal{S}$ and to find an efficient way for "minimally" connecting those HSF structures (interpreted as connected components) between them using adjacency relationships. By "minimal" connectivity, we mean to use only one edge for connecting two adjacent connected components (nodes of the RAG). Since we use an object-based digital

image analysis, the design of an HSF framework for 8-connected digital objects can straightforwardly be done simply by slightly changing the external aspect of the processing units employed. An important topological property in the 2D digital setting is that, given a black region-of-interest R of an image, a white maximal connected region surrounded by R ("no there" or hole of R in homology theory) is always 4-connected independently whether R is 4 or 8-connected. We suppose here that all the regions R_i of $\mathcal{S}$ are 4-connected. Although this is, in principle, a strong topology restriction that almost never is accomplished in real image segmentation, a suitable mathematical morphological preprocessing of the image can solve this obstacle.

Algorithms of the next section part from the information structures that contain the HSF and the contours of the regions of interest. They can be computed (see [7]) on a time complexity order (for a parallel execution) of *O(log c)*, being c the number of corners detected on a 2D digital image. To begin with, a brief summary of the extraction of the HSF and the contours follows.

From now on, in order to favor the computational methods and techniques allowing the practical implementation of a parallel architecture, we reduce the amount of mathematical results to a minimum.

In [7] the processing was focused on detecting in an exhaustive manner the outer and inner contours of the different curves that are present at a presegmented image. Thus, global information could be extracted. The implementation in [7] was essentially a sort of CCL algorithm (Connected-Component Labeling, see [11,15]) only applied to those pixels that form corners. This process was divided in three sections. The first part built the MrSF forest and determine some matrices containing the image corners and their characteristics. The second part was devoted to discover the outer and inner contour curves by following consecutive corners in vertical and horizontal directions. The result was two matrices H_{cycle}, V_{cycle}, containing the corner tags of the cycles that stood for the detected curves, and other structures with the information about the sink and sources cells (for the present work they have been comprised into the vector *crit_idx*). This part seeks for the global information relating the different corners that form the objects inside an image. Finally, a third section finished with the construction of a HSF for the foreground ROI of a binary image, by finding the appropriate pairs of sources and sinks, and by doing the arrow reversing process in parallel for each curve and each line. With previous processing, the contour information of the different image regions is comprised into two matrices. Matrix $C_{row,col,type}$ contains one row per corner and three columns. Column 1 indicates the row index of the corner (or alternatively the Y coordinate). Column 2 holds the column index (or alternatively the X coordinate), and the last column contains the type (number of quadrant, with negative sign if it is external to the current curve, that is, a number among 1, 2, 3, 4, -1, -2, -3, -4). The matrix with the information of closed curves is H_{cycle}. This matrix can be reduced to have one row per curve and as many columns as corners have each curve. The numbers contained in the elements of this matrix are indexes to the rows of $C_{row,col,type}$. Finally vector *crit_idx* indicates the row index of each critical cell (according to

the order given by Matrix $C_{row,col,type}$). Two counters are obtained by previous algorithms: *nof_curves*, which is the number of curves detected, and the vector *nof_corn_curve*, whose elements contain the number of corners of each curve.

In order to clarify this notation, a basic image with two closed curves (each one with a hole) and five curves (one of them is the image border) is depicted in Fig. 1. In this case, *nof_curves* is 5 and the five elements of *nof_corn_curve* contains 4. The elements of these matrices are given in Fig. 2. One of the rows of H_{cycle} includes its corner sequence (16,11,10,5); in particular the first row. Its critical 0-cell has been represented by corner 5. Similarly for the other curves.

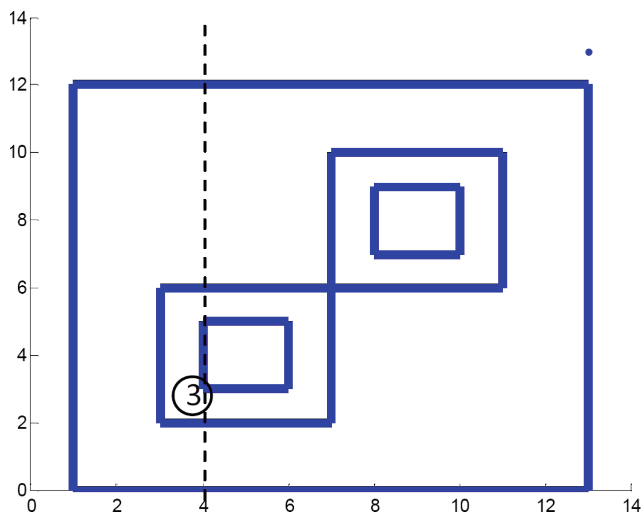
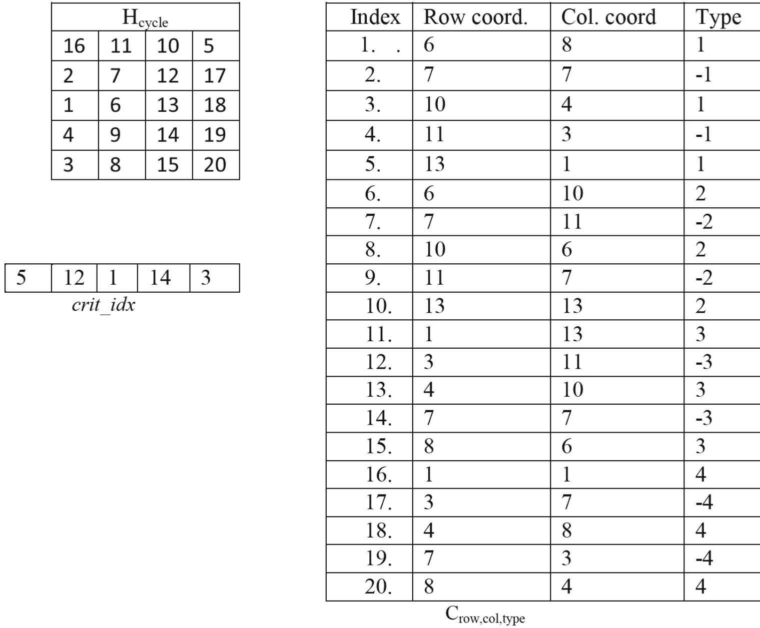


Fig. 1. A simple image with three curves (one of them is the image border). The crosses of the different segments with a dotted vertical line that parts from critical cell (tagged as 3) serve to detect inclusions.

An important fact of H_{cycle} is that the corner tags are stored in a sequential order following a "contour" along the curve. This will permit us to find the number of times that a set of segments move in one direction or in the opposite. Moreover, the first hop of this contour is always a horizontal segment, which is important to extract a set of only horizontal pairs of corners (segments).

3 Parallel Computation of the Inclusion Relations for the MrSF Structures

Using these matrices, the parallelization of an inclusion searching algorithm can be done in an efficient way, as depicted by the pseudo code of Algorithm 1 (see Fig. 3). Our implementation is inspired by the classical ray-crossing algorithm for the point in polygon problem [14]. However in our case, this problem can



H_{cycle}			
16	11	10	5
2	7	12	17
1	6	13	18
4	9	14	19
3	8	15	20

5	12	1	14	3

crit_idx

Index	Row coord.	Col. coord	Type
1. .	6	8	1
2.	7	7	-1
3.	10	4	1
4.	11	3	-1
5.	13	1	1
6.	6	10	2
7.	7	11	-2
8.	10	6	2
9.	11	7	-2
10.	13	13	2
11.	1	13	3
12.	3	11	-3
13.	4	10	3
14.	7	7	-3
15.	8	6	3
16.	1	1	4
17.	3	7	-4
18.	4	8	4
19.	7	3	-4
20.	8	4	4

$C_{row,col,type}$

Fig. 2. Input matrices for a simple image with three curves.

be computed more efficiently because instead of arbitrary polygons, only horizontal and vertical concatenated segments exist. Thus only simple computer operations like comparisons are sufficient to detect crossings. Furthermore, the parallel processing becomes more evident.

Taking into account previous considerations, the codification style and the notation of Algorithm 1, is different from the usual iterative/procedural conventions to describe a pseudo-code. Most studies use an OpenMP-like notation or the *for each* paradigm to express more clearly the thread parallelism that can be exploited (see references along this work). As our aim is to describe the inherent parallelism that can be exploited for extracting the RAG Tree, our notation follows that of OCTAVE/MATLAB, so time complexity orders can be easily computed.

The last stage to compute the RAG Tree consists of the conversion of matrix T_{inc} into another simplified matrix TTM that holds the topological tree (also in a matrix form). This is done by Algorithm 2 (Fig. 4) in a few steps.

The first line computes the number of columns of the TTM, which is given by those curves that have no other inside, by counting the number of columns of T_{inc} that are empty. If the level of "depth" of a curve is the number of curves that it encloses, then the maximum level of "depth" of all the curves plus one gives the number of rows (tree levels) of the TTM (see step 1). Step 2 contains the key point that will serve to order the rows of T_{inc}, in order to compute the RAG tree: each row of matrix C_{idx} contains the right sequences. This is due that

Algorithm 1: Pseudo-code for detecting curve inclusion.

Input: H_{cycle} (1:*nof_curves, 1:nof_corn_curve*): matrix that contain the consecutive corner tags of the cycles that stand for the detected curves
$C_{row,col,type}$ (1:*nof_corn,* 3): matrix with the information of corners of quadrants 1, 2, 3, 4. Each matrix contains the row and column indexes and the type (number of quadrant, with negative sign if it is external to the current pixel)
crit_idx (1:*nof_ curves*): vector containing the corner index of each critical cell.

Outputs: T_{inc} (1:*nof_curves,* 1:*nof_curves*): square inclusion matrix. The element (*i, j*) is set to *j* if the curve *i* is included into the curve *j*.
$T_{inc_min_row}$(1:*nof_curves,* 1:*nof_curves*): minimum row coordinates of all the segments of a curve that surrounds another one.
rows_to_be_deleted(1:*nof_curves*): vector with the number of times that a curve include any other.

Notation: Comments are preceded with symbol '%'. Matrices are represented by uppercase letters, vectors or variables by cursive lowercase letters.

1
 a. $F_{extreme} = H_{cycle}$ (:, 1: 2: *nof_corn_curve*) ;
 b. $S_{extreme} = H_{cycle}$ (:, 2: 2: *nof_corn_curve*);

2
 a. $C_{left} = C_{row,col,type}$ ($F_{extreme}$, 2) ;
 b. $C_{rigth} = C_{row,col,type}$ ($S_{extreme}$, 2);

3 **for** *crit_cell* = 2: *nof_curves* % curve 1 is the image border, so it is not included in no other. This loop is inserted here to prevent the use of tensors and other complications in the notation. Note that it can be avoided in a parallel implementation because any critical cell can be processed in parallel.

4
 a. *r_crit_cell* = $C_{row,col,type}$ (*crit_idx* (*crit_cell*), 1) ;
 b. *c_crit_cell* = $C_{row,col,type}$ (*crit_idx* (*crit_cell*), 2) ;

5
 a. CW_{idx} = find(*c_crit_cell* > C_{left} && *c_crit_cell* < C_{rigth});
 b. AW_{idx} = find(*c_crit_cell* < C_{left} && *c_crit_cell* > C_{rigth});
 c. $E_{idx,left}$ = find(*c_crit_cell* == C_{left});

6
 a. ECW_{idx} = find(*c_crit_cell* > C_{left} ($E_{idx,left}$-1) && *c_crit_cell* < C_{rigth}($E_{idx,left}$));
 b. EAW_{idx} = find(*c_crit_cell* < C_{left} ($E_{idx,left}$-1) && *c_crit_cell* > C_{rigth}($E_{idx,left}$));

7
 a. $CW_{idx_tag} = F_{extreme}$ ([CW_{idx}; ECW_{idx}]);
 b. $AW_{idx_tag} = F_{extreme}$ ([AW_{idx}; EAW_{idx}]); % idem if $S_{extreme}$ were chosen as horizontal segments are used.

8
 a. $CW_{row} = C_{row,col,type}$ (CW_{idx_tag}, 1)
 b. $AW_{row} = C_{row,col,type}$ (AW_{idx_tag}, 1)

9
 a. *sup_crosses* = nnz (CW_{row} >= *r_crit_cell*) - nnz (AW_{row} >= *r_crit_cell*);
 b. *inf_crosses* = nnz (CW_{row} <= *r_crit_cell*) - nnz (AW_{row} <= *r_crit_cell*) ;
 c. *min_row* = min(CW_{row} , AW_{row});

10
 a. *inclusion* = abs (*sup_crosses* .* *inf_crosses*);

11
 a. T_{inc} (*crit_cell* , :) = *inclusion* .* (1:*nof_curves*);
 b. $T_{inc_min_row}$ (*crit_cell* , :) = *inclusion*.* *min_row* ;
 c. *rows_to_be_deleted* = *rows_to_be_deleted* + *inclusion*;

12 **end**

Fig. 3. Algorithm 1: Pseudo code for the algorithm that computes the inclusion relation of a set of curves.

the minimum row coordinates of a group of nested curves identify the order of inclusions (when sorting them).

Next line reorders T_{inc} using these sequences C_{idx}, meanwhile step 4 adds the own curve numbers in a new column. Finally the last step 5 extracts those lines that are the leaves of the RAG tree and transpose the result to obtain TTM. Thus, those curves that do contain any other curve are forgotten in the final TTM matrix.

Algorithm 2: Pseudo-code for the conversion of the inclusion matrix into a RAT.

Input: T_{inc} (1:*nof_curves,* 1:*nof_curves*): square inclusion matrix. The element (*i, j*) is set to *j* if the curve *i* is included in the curve *j*.

$T_{inc_min_row}$(1:*nof_curves,* 1:*nof_curves*): minimum row coordinates of all the segments of a curve that surrounds another one.

rows_to_be_deleted(1:*nof_curves*): vector with the number of times that a curve include any other.

Outputs: TTM (1:*nof_curves, 1:nof_corn_curve*): matrix that contains the topological tree or RAT

Notation: Comments are preceded with symbol '%'. Matrices are represented by uppercase letters, vectors or variables by cursive lowercase letters.

```
% curve 1 is the image border, so it is not included in no other
1
    a. nof_col_TTM (1:nof_curves)= nof_curves - nnz(sum(T_inc (:, 1:nof_curves) ));
    b. nof_row_TTM (1:nof_curves)= 1 + max (sum(T_inc (1:nof_curves, :) ));

2
    a. [dummy, C_idx]= sortrows (T_inc_min_row);

3
    b. TTM_extra= T_inc (C_idx);

4
    c. TTM_extra (:, 1+nof_curves) = 1:nof_curves;
5
    d. TTM = TTM_extra (not(rows_to_be_deleted), :)' ;
```

Fig. 4. Algorithm 2: Pseudo code for the algorithm that computes the inclusion relation of a set of curves.

4 An Example of RAG Tree Parallel Computation

In order to explain those Algorithms, let us examine the case of curve 5, whose critical cell is tagged with a 3 (Fig. 1). The searching of the curves that surround any other is based on detecting the crosses of a vertical line with the critical cell of each curve. In Fig. 1, the vertical line is depicted on the column 4, where critical cell of curve 5 was placed by the HSF construction [7].

First step extracts the tags of both extremes of any possible segment, which give us:

$F_{extreme}$ = (16 10 ; 2 12 ; 1 13 ; 4 14 ; 3 15)

$S_{extreme}$ = (11 5 ; 7 17 ; 6 18 ; 9 19 ; 8 20)

After that, those tags are converted into matrices C_{left}, C_{rigth}, which are filled with the column indexes of all the pair of corners that represent horizontal segments. In our case:

C_{left} = (1 13 ; 7 11 ; 8 10; 3 7; 4 6)

C_{rigth} = (13 1; 11 7; 10 8; 7 3; 6 4)

Step 4 extract the tag of the critical cell of this curve, that is *crit_cell* (5)=3. Coordinates *r_crit_cell*=10 and *c_crit_cell*=4 are obtained by accessing to $C_{row,col,type}$. Step 5 gives us the indexes over C_{left} and C_{rigth} (for each curve) of those segments that cross the vertical line on clockwise (CW) or anticlockwise (AW) directions. In our case, the *find* function returns:

CW_{idx} = (1; Empty matrix; Empty matrix; 1; Empty matrix)

AW_{idx} = (2; Empty matrix; Empty matrix; 2; Empty matrix)

Matrices ECW_{idx}, EAW_{idx} are provided for those cases where the column coordinate is the same for the critical cell and a segment extreme. For those cases, the proper corners that determine this crossing are selected (step 6). At next step 7, the tags of the crosses are found, by indexing on $F_{extreme}$ and $S_{extreme}$. For our example, CW_{idx_tag} =(10; Empty matrix; Empty matrix; 14; Empty matrix); AW_{idx_tag} =(6; Empty matrix; Empty matrix; 4; Empty matrix)

The row coordinates of previous tags CW_{idx_tag} and AW_{idx_tag} are found by accessing the first column of $C_{row,col,type}$ in the operations of step 8. This gives us:

CW_{row} =(13; Empty matrix; Empty matrix; 7; Empty matrix)

AW_{row} =(1; Empty matrix; Empty matrix; 11; Empty matrix)

The first two operations of step 9 count the number of crosses (non zero elements). Function *nnz* returns the values:

sup_crosses = (1 ; 0 ; 0 ; 1 ; 0)

inf_crosses = (1 ; 0 ; 0 ; 1 ; 0)

The other operation saves the minimum row coordinates of all the segments, which will be used to find the order in which a curve is surrounded by others (see next algorithm). Finally step 10 computes the vector *inclusion*, that is, the absolute of the element-by-element product (*sup_crosses .* inf_crosses*), because *sup_crosses* and *inf_crosses* can be 1 or -1 when a segment crosses the vertical line. Note that a curve include the current cell, if and only if there is one cross with this vertical line above this cell and only one below it. Four our case, the fifth row of *inclusion* tells us that curves 1 and 4 surround the current curve 5:

inclusion (5, :) = (1 ; 0 ; 0 ; 1 ; 0)

This vector is multiplied element-by-element by the numbers (1: *nof_curves*), so at step 11 the matrix T_{inc} is marked with these numbers. In our example, the fifth row is set to (1 0 0 4 0). In addition, the matrix $T_{inc_min_row}$ is filled with the minimum row coordinates of all the segments of surrounding curves. At the same time, these values serve to fill a vector *rows_to_be_deleted* with the

number of times that a curve include any other. This vector will help to delete those rows that must not appear at the final RAG tree. In our example:

T_{inc} =

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 \end{pmatrix}$$

This means that curve 2 is surrounded by 1 (the image borders, in row 1), curve 3 by 1 and 2 (in row 3), curve 4 by 1, and curve 5 by 1 and 4 (row 7).

Using Algorithm 2, it is obvious that the resultant matrix for this example TTM results:

$$\begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 5 \end{pmatrix}$$

5 Implementation and Testing Results

A complete implementation in OCTAVE/MATLAB has been built following previous pseudo codes. In order to check its scalability, a non-fully functional implementation of Algorithm 1 (because its time computing is very much bigger than that of Algorithm 2) has been done in C++ using OpenMP directives. Using previous notation, the next advantages are done evident:

(a) It can be easily demonstrated that Algorithm 1 scales well whit the number of Processing Elements of the target parallel architecture (like multi or manycore, GPUs, SIMD kernels, ...). This also eases the codification for a parallel computer. For similar reasons, those sentences that can be executed in parallel are grouped in the same step (using lines preceded with letters).
(b) OpenMP codes can be written directly through this notation, in most of the cases by converting the matrix processing into nested loops, and preceding the outer loop (usually devoted to the rows) with the directive *#pragma omp parallel for*.
(c) The memory access pattern (which is in most occasions [5] a critical point for the performance that can be achieved in GPUs) can be clearly observed.
(d) Matrix operations that cannot be done in an element-by-element manner (like matrix inversions, matrix multiply, etc.) are avoided.
(e) Only plain matrices have been used, preventing complex data structures (trees, chained lists, stacks, etc.), very inefficient when running on massive data parallel computer architectures.
(f) Finally, Algorithms have no conditional sentences. The avoidance of conditional sentences in the hot spot zones prevents the so-called thread divergence for GPUs, which is one the main reasons why the performance on this platforms diminishes [5].

Two final considerations must be done: (1) the only *for* loop encountered in this algorithm can be avoided in a parallel implementation. In fact its iterations are fully independent, because any critical cell can be processed in parallel using the same input matrices. The reason to insert the loop for is the avoidance of complications in the notation (as the use of tensors and indexations that are not supported by OCTAVE/MATLAB). (2) The only procedures that are not strictly (element-by-element) parallel are those of precompiled functions like *find, nnz, max*, etc. The non-strictly parallel sections of these functions are reduction operations, which can be coded in a binary tree fashion and present a time complexity order of *O(log p)*, being p the number of elements involved in the procedure.

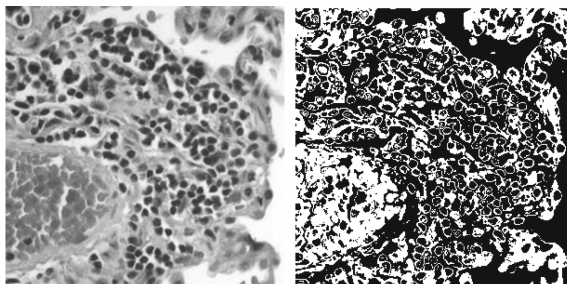


Fig. 5. A 300×300 fragment of a real H & E stained lung tissue sample taken from an end-stage emphysema patient.

For a modest 4-core desktop PC, scalability is almost perfect. Additional tests have been carried out in a server with an Intel Xeon E5 2650 v2, whose main characteristics are: 2.6 GHz, 8 cores (up to 16 threads), 8×32 KB data caches, Level 2 cache size 8×256 KB, Level 3 cache size 20 MB, maximum bandwidth to RAM: 59.7 GB/s. For this machine, all tests behave similarly: execution times scale very well for 8 threads (the number of real cores), meanwhile some additional speedup can be achieved above 9 threads, as usual for simultaneous multi-threading technologies. Due that the effects of multitasking in this server are relevant, the OpenMP guided scheduling was preferred.

Timing and speed results (baseline time is that of one thread) are represented in Fig. 6 for the following image from Wikimedia [17]: a 300×300 fragment of a real H and E (haematoxylin and eosin) stained lung tissue sample taken from an end-stage emphysema patient (Fig. 5, left). This image is segmented (Fig. 5, right) by converting into gray and then posterizing it for the intervals [0,59], [60,119], [120,179], [180,255]. This implementation received input from the output of algorithms in [7], which return a total of 750 curves, 20370 corners (a considerable number, w.r.t to the 90000 pixels of the original image).

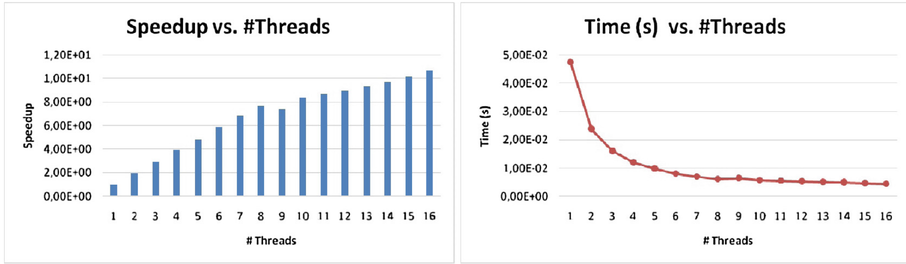


Fig. 6. Scalability testing for an Intel Xeon E5 2650 v2. Left: time vs. number of threads. Right: Speedup vs. number of threads

6 Conclusions

One important aspect of the RAG tree computation in the algorithm presented here is that we have adopted the MrSF structures so as to determine in a parallel fashion which regions are included or surrounded by others, transforming exhaustively a classical geometric algorithm. From a topological perspective, the intuition seems to lead to the conclusion that the notion of RAG is independent of the 2-dimensional embedding of the image I and it can be efficiently computed in parallel using exclusively global homology information of each region and the local topology of the 1-cells of the cracks between regions (without need to turn to geometric algorithms). To give a correct answer in that direction would allow us to appropriately extend in a topological consistent way the notion of RAG to segmentations of n-dimensional ($n \geq 3$) digital images in terms of 0 and $n-1$ homology generators of the different regions.

Acknowledgments. The first author gratefully acknowledges the support of the Spanish Ministry of Science and Innovation (project Biosense, TEC2012-37868-C04-02), the second author the support of the V Plan Propio de la Universidad de Sevilla, project number 2014/753, and the last author the support of the Austrian Science Fund(FWF): project number P27516.

References

1. Barbu, A., Zhu, S.-C.: Graph partitioning by Swendsen-Wang cuts. In: Ninth IEEE International Conference on Computer Vision, Nice, France, vol. 1, pp. 320–329 (2003)
2. Cucchiara, R., Grana, C., Prati, A., Seidenari, S., Pellacani, G.: Building the topological tree by recursive FCM color clustering. In: Proceeding International Conference on Pattern Recognition, vol. 116(1), pp. 759–762 (2002)
3. Cohn, A., Bennett, B., Gooday, J., Gotts, N.: Qualitative spacial representation and reasoning with the region connection calculus. GeoInformatica **1**(3), 275–316 (1997)
4. Costanza, E., Robinson, J.: A region adjacency tree approach to the detection and design of fiducials. In: Video, Vision and Graphics, pp. 63–99 (2003)

5. CUDA C Best Practices Guide Version. NVIDIA. http://developer.nvidia.com/. Accessed 14 July 2015
6. Damiand, G., Bertrand, Y., Fiorio, C.: Topological model for two-dimensional image representation: definition and optimal extraction algorithm. Comput. Vis. Image Underst. **93**(2), 111–154 (2004)
7. Díaz-del-Río, F., Real, P., Onchis, D.: A parallel homological spanning forest framework for 2D topological image analysis. Submitted to Pattern Recognition Letters, July 2015
8. Duarte, A., Sánchez, A., Fernandez, F., Montemayor, A.S.: Improving image segmentation quality through effective region merging using a hierarchical social metaheuristic. Pattern Recogn. Lett. **27**, 1239–1251 (2006)
9. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient graph-based image segmentation. Int. J. Comput. Vis. **59**(2), 167–181 (2004)
10. Gothandaraman, A.: Hierarchical image segmentation using the watershed algorithm with a streaming implementation. Ph.D. thesis, University of Tennessee, USA (2004)
11. Gupta, S., Palsetia, D., Patwary, M.M.A., Agrawal, A., Choudhary, A.N.: A new parallel algorithm for two-pass connected component labeling. In: 2014 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 1355–1362 (2014)
12. Haris, K., et al.: Hybrid image segmentation using watersheds and fast region merging. IEEE Trans. Image Process. **7**(12), 1684–1699 (1998)
13. Hernández, S.E., Barner, K.E.: Joint region merging criteria for watershed-based image segmentation. In: International Conference on Image Processing, vol. 2, pp. 108–111 (2000)
14. Hormann, K., Agathos, A.: The point in polygon problem for arbitrary polygons. Comput. Geom. **20**(3), 131–144 (2001)
15. Kalentev, O., Rai, A., Kemnitz, S., Schneider, R.: Connected component labeling on a 2D grid using CUDA. J. Parallel Distrib. Comput. **71**(4), 615–620 (2011)
16. Molina-Abril, H., Real, P.: Homological spanning forest framework for 2D image analysis. Ann. Math. Artif. Intell. **4**(64), 385–409 (2012)
17. Emphysema, H., E. Commons.wikimedia.org File: Emphysema H and E.jpg. Accessed 14 September 2015
18. Pavlidis, T.: Structural Pattern Recognition. Springer, New York (1977)
19. Sonka, M., et al.: Image Processing, Analysis and Machine Vision, 2nd edn. PWS, Pacific Grove (1999)
20. Rosenfeld, A.: Adjacency in digital pictures. Inf. Control **26**, 24–33 (1974)
21. Serra, J.: Image Analysis and Mathematical Morphology. Academic Press, Orlando (1982)
22. Sarkar, A., Biswas, M.K., Sharma, K.M.S.: A simple unsupervised MRF model based image segmentation approach. IEEE Trans. Image Process. **9**(5), 801–811 (2000)
23. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Trans. Pattern Anal. Machine Intel. **22**(8), 888–905 (2000)