ITERATIVE CONSTRUCTION OF DISTRIBUTED COMPONENT FORESTS

ANIKET RANE
S3290999

university of groningen

faculty of science and engineering

Bachelor Thesis

SUPERVISORS:
S.R.N. Gazagnes
dr. M.H.F. Wilkinson

ABSTRACT

In this paper, we present an implementation to construct Distributed Component Forests in an iterative manner in situations where the number of processor nodes and memory is limited. The input image is first split into N_t different tiles, where N_t is larger than N_p, the maximal number of nodes or processes on the machine used. Components tees are then computed and stored iteratively for each image tile. In order to adapt the full parallel implementation of constructing Distributed Component Forests in an iterative manner, boundary trees are successfully created, merged, combined and updated by iteratively storing and loading the necessary information in files to perform these operations using a limited number of processes. This implementation would help to process very large images on machines with limited memory capacity.

Trees sprout up just about everywhere in Computer Science

— Donald E. Knuth

ACKNOWLEDGMENTS

I would like to thank my supervisors S.R.N. Gazagnes and dr. M.H.F. Wilkinson for their guidance and immense help. A special thanks to my parents, family and friends for their love and support.

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

DCF - Distributed Component Forest
CT_i - Component tree for i^{th} image tile, where $i \geq 1$
BT_i - Boundary tree for i^{th} image tile, where $i \geq 1$

1 INTRODUCTION

As the size of images increase, it gets increasingly difficult to store the hierarchical representation of these images efficiently and apply filters on them. Component tree is a data structure that allows for an efficient representation of a grey-level image by storing information about each component of the image and the connections that exist between these components at sequential grey-levels [4]. Component trees are useful in attribute filtering [1] [8] [6] [9], visualization [10] and multi-scale analysis [5].

Component trees are unsuitable when either the tree or the image don't fit in the memory of a single processor node [3]. Hence, recently a new structure was suggested called *Distributed Component Forest* (DCF) which is a collection of updated component trees for an image where each component tree is per image tile [1] such that filtering these updated component trees gives the exact same results as filtering a single component tree for the whole image [5]. Although usually these DCFs are computed in parallel, there is a need to compute them iteratively in situations where the number of processor nodes and memory is limited. This paper aims to provide a solution to this very problem by presenting an implementation in which the merging and combining of individual component trees is carried out in an iterative manner. The iterative implementation can also then further be scaled to suit more number processor nodes. This implementation can be useful in many fields where often large images need to be processed on low memory like digital telescopes, phones, microscopes etc.

This thesis is organized as follows : In Section 2 background information is provided to understand the implementation of the idea presented. In Section 3 the implementation of the idea is presented. Section 4 presents the tests conducted and the results obtained. Finally Section 5 presents the conclusion and further improvements that can be made.

1 Image tile and tile are henceforth used interchangbly

2

BACKGROUND INFORMATION

2.1 COMPONENT TREES

The concept of component trees was first introduced in 1999 [4]. Component trees are of two types - max-trees and min-trees [8]. Max-trees are where the leaves depict a foreground component at a particular threshold set and vice versa for min-trees [3]. In this paper we will look at max-trees as we can easily use the same implementation for min-trees as well. Hence, whenever we will mention component tree from now on it means a max-tree. As we can observe in Figure 2.1, the input gray-scale image is represented as foreground components of each threshold set and this representation is converted into a max-tree.

Component trees can be built with various algorithms but mainly they are classified into 3 main categories - immersion, flooding and merge [2]. In this paper we are concerned with merge based algorithms. In merge based algorithms the general idea is to divide an image into blocks and compute component trees for these sub-images in the blocks and finally merge them together [2]. In merge based algorithms we use any sequential methods from either immersion or flooding based algorithms to compute component trees for sub-images. There is no one "optimal" algorithm to build component trees as the choice of algorithm depends on hardware size, size of the image and the dynamic range of the image. In general these algorithms build component trees in $\mathcal{O}(GN)$, where G is the number of gray levels in 8-16 bit per pixel case or $\mathcal{O}(n \log n)$ in the 32-bit per pixel or floating point case [7].

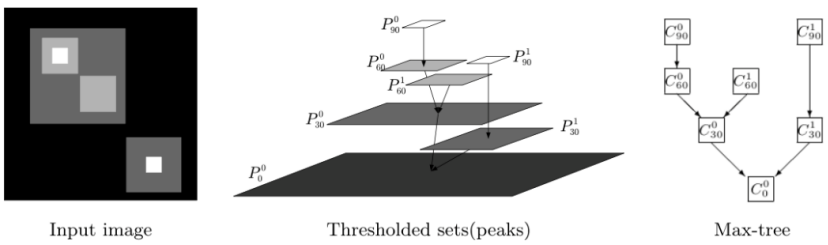

Figure 2.1: Component tree representation from input gray-scale image [3]

2.2 DISTRIBUTED COMPONENT FORESTS

In this subsection we will first look at how to build DCFs in parallel and then we will look at an important structure called *boundary tree* that will help us understand the construction of DCFs better.

2.2.1 *Building DCFs in parallel*

We will first look at a basic 2 tile case and then a complicated 4 tile case which can be extended to more tiles.

2.2.1.1 *2 Process 2 Tiles*

Figure 2.2 shows how to build DCFs in a parallel using 2 processes and 2 image tiles. This is the simplest case to build DCFs. The input image is split into 2 tiles, where the tiles are represented as 0,1. The 2 processes are represented as 0,1. All the processes first load the pixel intensities in the tiles. Processes 0,1 then build component trees for tiles 0,1. Processes 0,1 then build boundary trees for tiles 0,1. After this process 1 sends boundary tree 1 to process 0. Process 0 receives the boundary tree 1 from process 1. Process 0 merges (horizontally) boundary tree 0 and boundary tree 1. At this point, the two boundary tree structures are "linked" together, such that for example some nodes in boundary tree 0 have their parents in boundary tree 1. The two boundary trees are not combined as we only have 2 tiles and we don't need to perform merging with other boundary trees. Process 0 updates boundary tree 0 and boundary tree 1 which ensures that the boundary trees are independent again, such that both of these boundary trees have their parent nodes pointing to themselves. Process 0 then sends back the updated boundary tree 1 to process 1. Process 1 receives the updated boundary tree 1 from process 0. Processes 0 and 1 update their respective component trees using the boundary trees. Finally processes 0 and 1 filter their respective component trees.

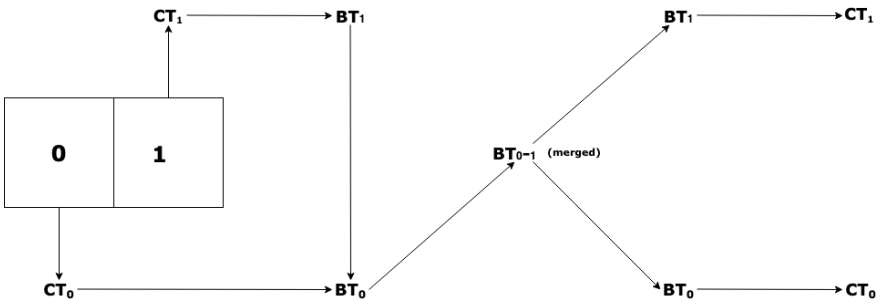

Figure 2.2: Constructing DCF in parallel for 2 image tiles

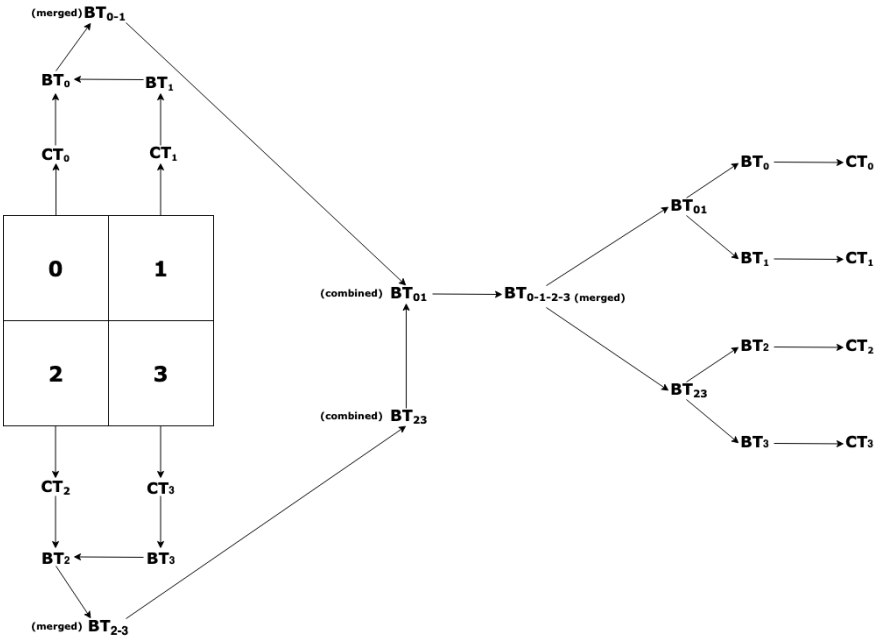

Figure 2.3: Constructing DCF in parallel for 4 image tiles

2.2.1.2 *4 Process 4 Tiles*

Figure 2.3 shows how to build DCFs in a parallel using 4 processes and 4 image tiles. The input image is first split in 4 image tiles, where the tiles are represented as 0,1,2,3. The 4 processes are represented as 0,1,2,3. All the processes first load the pixel intensities in the tiles. Processes 0,1,2,3 then build component trees for tiles 0,1,2,3. Processes 0,1,2,3 then build boundary trees for tiles 0,1,2,3. After this processes 1 and 3 send boundary tree 1 and boundary tree 3 to processes 0 and 2. Process 0 and 2 receive the boundary tree 1 and boundary tree 3 from processes 1 and 3. Process 0 merges (horizontally) boundary tree 0 and boundary tree 1, and process 2 merges boundary tree 2 and 3. Process 0 combines boundary trees 0 and 1 (horizontally), and process 2 combines boundary trees 2 and 3 (horizontally). Process 2 sends the combined boundary tree 2-3 to process 0. Process 0 merges boundary tree 0-1 and boundary tree 2-3 (vertically). Process 0 updates the boundary tree 0-1 and the boundary tree 2-3. Process 0 sends back the updated boundary tree 2-3 to process 2. Process 2 receives the updated boundary tree 2-3 from process 0, and uses it to update the individual boundary trees 2 and 3. Process 0 updates boundary trees 0 and 1 using the updated boundary tree 0-1. Process 0 and 2 send back the updated boundary trees 1 and 3 to process 1 and 3 respectively. Process 0,1,2,3 update the component trees from their tiles using their boundary trees. Process 0,1,2,3 filter the component tree from their tile.

2.2.2 *Boundary trees*

While constructing DCFs it is inefficient to pass the entire component tree from one tile to another for merging. Instead we only choose the nodes that are touching the boundary of the tile as any other nodes don't change while merging [5]. Hence, a structure called a *boundary tree* is created, which is essentially a pruned component tree and passed on to another tile. Boundary tree consists of all the border nodes in a component tree, all of the ancestors of these nodes which include parents until the root and additional metadata [5]. Boundary tree is represented as a $1D$ array of the structure **`Boundary`**.

We will now look at 3 important elements of the structure **`Boundary`** that would help us to understand the implementation presented in the next section. All these 3 elements mentioned below are arrays of the type **`BorderIndex`**. **`BorderIndex`** is a structure that contains a pointer to a boundary tree and an index in the boundary.

- **`border_par`**, represents parents in the boundary tree.
- **`border_ori`**, represents the position in the previous boundary tree (used while combining).
- **`border_lr`**, represents the levelroot at same intensity in the other merged tree (used while merging).

3 IMPLEMENTATION

The main idea to construct DCFs iteratively is by reading and writing structures like component trees and boundary trees into files. Constructing DCF iteratively for n tiles would be expensive and inefficient on memory of m processes where $m < n$ so we read and write structures to mimic the working of n processes.

To completely understand the iterative implementation we will look at the reading and writing functions used and then we will look at the algorithms to construct DCFs iteratively for 1 process 2 tiles and 1 process 4 tiles.

3.1 READING AND WRITING FUNCTIONS

All structures like component trees and boundary trees are stored in files in a local folder. These structures are stored as binary files to reduce their size. The standard C ***fread*** and ***fwrite*** functions are used to read and write the content of these structures. All write functions return **`void`** and all read functions return the structure they are reading from the file.

3.1.1 *Writing functions*

The function ***write_tree_file*** [A.1] writes the component tree for a tile to file. This function takes as input a component tree, name of the file to write to and the tile number. This function writes everything in the **`Node`** structure to file.

The function ***write_boundary_file*** [A.2] writes the boundary tree to file. This function takes as input a boundary tree, name of the file to write to, tile number and the status of the boundary tree. Status of the boundary tree indicates whether the boundary tree is a *basic* or a *merged* boundary tree.

If the boundary tree is a *basic* boundary tree then we write everything in the **`Boundary`** structure except the arrays **`border_lr`** and **`border_ori`**. While writing **`border_par`** array, we cannot write all the parent indexes stored in it. Hence, we create a sub-array that only contains the indexes of the parents and we write this sub-array in file. In the basic boundary tree we don't have to worry about storing the pointers in the **`border_par`** array as everything points to the boundary tree itself.

If the boundary tree is a *merged* boundary tree then we write everything in the **Boundary** structure except the **border_ori** array. As in the basic case, we use sub-arrays to write the indexes contained in **border_par** and **border_lr** arrays to file. Although here we also have to store the pointers to these indexes for both these arrays as merging changes their addresses. Hence, we can't directly store the address of pointers in file as after reading the file. To bypass this problem we first create a boolean array with the same size as the current boundary tree. This is because we know that these pointers may point to one of two boundary trees i.e the current boundary tree or the boundary tree with which the current boundary tree is merged. Then to write these pointers to file we check if the pointer points to the same tree, if it does we assign **true** to the boolean array element otherwise we assign **false**. We then write this boolean array in file.

The function ***write_boundary_combined_file*** [A.3] writes the combined boundary tree to file. This function takes as input a combined boundary tree, name of the file to write to, tile number, two boundary trees that combine to create the resulting combined boundary tree and the status of the combined boundary tree. Status of the combined boundary tree indicates whether the combined boundary tree is *basic* or *merged*.
For both the *basic* and *merged* case we follow the same procedure as ***write_boundary_file*** except we also write **border_ori** array in both the cases. Again we use a boolean array to store the pointers for the **border_ori**. We assign **true** if the pointer points to one of the boundary trees otherwise we assign **false** if it points to the other. We do this because we know that every element in the **border_ori** array points to either one of the boundary trees that have been combined. We then write this boolean array in file.

3.1.2 *Reading functions*

The function ***read_tree_file*** [A.4] reads the component tree from a file. This function takes as input the name of the file to be read, tile number. This function reads everything in the **Node** structure which was written to file.

The function ***read_boundary_file*** [A.5] reads the boundary tree from a file. This function takes as input the name of the file to be read, tile number and the status of the boundary tree to be read.

If the boundary tree to be read is a *basic* boundary tree then we read everything except for the **border_ori** array. We read the indexes in **border_lr** and **border_par** arrays by assigning for each node the correct indexes. We assign the pointers to the indexes in the **border_par**

the boundary tree itself. We assign the pointers to the indexes in the **border_lr** array **BOTTOM** to initialize them.

If the boundary tree to be read is a *merged* boundary tree then we read everything except for the **border_ori** array. We read the indexes of **border_lr** and **border_par** just as the basic case. In order to read the pointers to indexes of **border_lr** and **border_par** we first read the boolean arrays written to the file. If the value of the element in the boolean array is **true**, then we assign the pointer to the boundary tree being read otherwise we assign it **NULL**. We later correct these pointers pointing to **NULL** by making them point to the correct boundary tree in the function ***correct_boundary_indexes***.

The function ***read_boundary_combined_file*** [A.6] reads the combined boundary tree from file. This function takes as input the name of the file to be read, tile number, two boundary trees that combine to create the resulting combined boundary tree and the status of the combined boundary tree to be read.

For both *merged* and *basic* case we follow the same procedure as ***read_boundary_file*** except we also read **border_ori** array. We use the inverse of the method used to write the **border_ori** array to read it.

3.2 ITERATIVE DCF CONSTRUCTION

3.2.1 *1 Process 2 Tiles*

Algorithm 1 constructs DCF iteratively for an image split into 2 tiles. This is the base case algorithm for the method of iterative construction of DCF. The algorithm is the same as the steps to construct DCF in parallel with 2 tiles except there is only 1 process and we use read and write functions.

3.2.2 *1 Process 4 Tiles*

Algorithm 2 constructs DCF iteratively for an image split into 4 tiles. This algorithm is an extended version of the basic case algorithm and deals with a complicated case of 4 tiles. This algorithm works like the steps to construct DCF in parallel for 4 tiles. The function ***correct_boundary_indexes*** corrects the **border_lr** and **border_par** array pointers of the boundary trees by pointing them to the correct boundary trees. The function does this by checking the arrays and if it comes across an element pointing to **NULL**, then it points it to the other boundary tree the current boundary tree was merged with. The function ***correct_combined_indexes*** does the same for combined boundary trees that are merged.

Algorithm 1 IterativeDcf2Tiles

```
1: i ← 0
2: num_tiles ← 2
3: while i < num_tiles do
4:    lct ← build_local_tree(i)
5:    write_tree_file(lct, i)
6:    bt ← create_boundary(lct, i)
7:    write_boundary_file(bt, i)
8:    free_boundary(bt)
9:    free_tree(lct)
10:   i ← i + 1
11: end while
12: bt_0 ← read_boundary_file(tile_number_0)
13: bt_1 ← read_boundary_file(tile_number_1)
14: merge(bt_0, bt_1, HORIZONTAL)
15: bt_01 ← combine(bt_0, bt_1, HORIZONTAL)
16: if bt_01 == NULL then
17:    update_par_step()
18:    update(bt_0, bt_01)
19:    update(bt_1, bt_01)
20: end if
21: write_boundary_file(bt_0, 0)
22: write_boundary_file(bt_1, 1)
23: free_boundary(bt_0)
24: free_boundary(bt_1)
25: i ← 0
26: while i < num_tiles do
27:    lct ← read_tree_file(tile_number_i)
28:    bt ← read_boundary_file(tile_number_i)
29:    lct ← correct_local_tree(lct, bt)
30:    tree_filtering(lct)
31:    free_boundary(bt)
32:    free_tree(lct)
33:    i ← i + 1
34: end while
```

Algorithm 2 IterativeDcf4Tiles

```
1: i ← 0
2: num_tiles ← 4
3: while i < num_tiles do
4:    lct ← build_local_tree(i)
5:    write_tree_file(lct, i)
6:    bt ← create_boundary(lct, i)
7:    write_boundary_file(bt, i)
8:    free_boundary(bt)
9:    free_tree(lct)
10:   i ← i + 1
11: end while
12: i ← 0
13: while i < num_tiles do
14:   a ← i
15:   b ← i + 1
16:   bt_a ← read_boundary_file(tile_number_a)
17:   bt_b ← read_boundary_file(tile_number_b)
18:   merge(bt_i, bt_i + 1, HORIZONTAL)
19:   bt_ab ← combine(bt_a, bt_b, HORIZONTAL)
20:   write_boundary_combined_file(bt_ab, bt_a, bt_b, BASIC)
21:   write_boundary_file(bt_a, tile_numbera, MERGED)
22:   write_boundary_file(bt_b, tile_numberb, MERGED)
23:   free_boundary(bt_a)
24:   free_boundary(bt_b)
25:   free_boundary(bt_ab)
26:   i ← i + 2
27: end while
28: bt_0 ← read_boundary_file(tile_number0, MERGED)
29: bt_1 ← read_boundary_file(tile_number1, MERGED)
30: bt_2 ← read_boundary_file(tile_number2, MERGED)
31: bt_3 ← read_boundary_file(tile_number3, MERGED)
32: i ← 0
33: while i < num_tiles do
34:   a ← i
35:   b ← i + 1
36:   correct_boundary_indexes(bt_a, bt_b)
37:   i ← i + 2
38: end while
```

```
39: bt_01 ← read_boundary_combined_file(tile_number_0, tile_number_1, BASIC)
40: bt_23 ← read_boundary_combined_file(tile_number_2, tile_number_3, BASIC)
41: merge(bt_01, bt_23, VERTICAL)
42: bt_0123 ← combine(bt_01, bt_23, VERTICAL)
43: update_par_step(bt_0123)
44: update(bt_01, bt_0123)
45: update(bt_23, bt_0123)
46: write_boundary_combined_file(bt_01, MERGED)
47: write_boundary_combined_file(bt_23, MERGED)
48: free_boundary(bt_01)
49: free_boundary(bt_23)
50: bt_01 ← read_boundary_combined_file(tile_number_0, tile_number_1, MERGED)
51: bt_23 ← read_boundary_combined_file(tile_number_2, tile_number_3, MERGED)
52: correct_combined_indexes(bt_01, bt_23)
53: i ← 0
54: while i < num_tiles do
55:     a ← i
56:     b ← i + 1
57:     update_par_step(bt_ab)
58:     update(bt_a, bt_ab)
59:     update(bt_a, bt_ab)
60:     write_boundary_file(bt_a, a)
61:     write_boundary_file(bt_b, b)
62:     free_boundary(bt_ab)
63:     free_boundary(bt_a)
64:     free_boundary(bt_b)
65:     i ← i + 2
66: end while
67: i ← 0
68: while i < num_tiles do
69:     lct ← read_tree_file(tile_numberi)
70:     bt ← read_boundary_file(tile_numberi)
71:     lct ← correct_local_tree(lct, bt)
72:     tree_filtering(lct)
73:     free_boundary(bt)
74:     free_tree(lct)
75:     i ← i + 1
76: end while
```

4

TESTS & RESULTS

Tests were conducted on Zeus compute server : 64-core AMD Opteron Processor 6276 (2.3 GHz) with 512 GB of RAM by implementing the iterative algorithms presented in the previous section. Two images were used to carry out the tests : 8-bit per pixel remote sensing image of Haiti of size 1.2 Gigapixels (1.2 GB in storage) and 16-bit per pixel remote sensing image of Haiti of size 1.2 Gigapixels (2 GB in storage). Different tests were run to compare and study the timings between generating DFCs using 1 process n tiles, 1 process 1 tile and n processes n tiles. The speed-up was obtained by taking the ratio $t(1)/t(n)$, where $t(1)$ is the wall-clock time (in seconds) for 1 process 1 tile and $t(n)$ is the wall-clock time (in seconds) for 1 process n tiles. The timings in all the figures in this section are minimum timings recorded over 5 runs. Timings that are reported for steps like tree building, writing trees to file, filtering etc. are the total sum of wall-clock times (in seconds) for those steps over all the tiles present. The time taken for n processes n tiles and 1 process n tiles does not include the time taken to read the input per tile and the time taken to write output files per tile. Different tests were run to compare the memory efficiency by varying the number of tiles and the number of processes. The *top* command was used to check the maximum memory used during a process.

In Figure 4.1 the step numbers indicate the following :

1. Initial component trees built for all tiles.
2. Initial component trees written for all tiles.
3. Initial boundary trees built for all tiles.
4. Writing boundary trees to file for all tiles (including merged and combined).
5. Reading boundary trees from files for all tiles (including merged and combined).
6. (Merging + Combining + Updating) the boundary trees.
7. Tree filtering.

In Figure 4.1 (a) for 8-bit image we can observe that per step it takes more time for 2 tiles as compared to 4 tiles for 1 process except for steps - 4, 5 and 6. This is because as the number of tiles decreases from 4 to 2, the size of the component tree per tile and the corresponding boundary tree generated increases as now more information

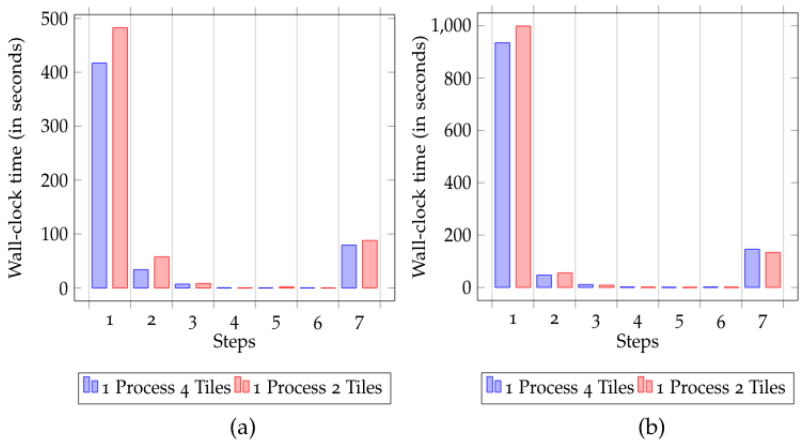

Figure 4.1: Total time to execute 1 process steps for (a) Haiti 8-bit and (b) Haiti 16-bit

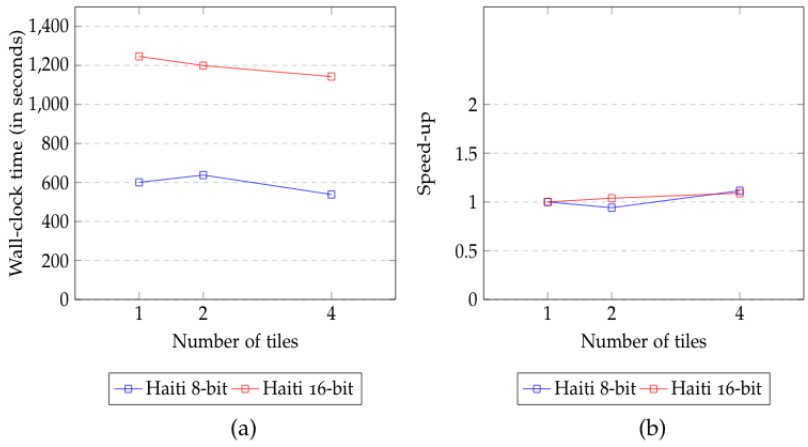

Figure 4.2: (a) Timings and (b) speed-up for 1 process with Haiti 8-bit and Haiti 16-bit

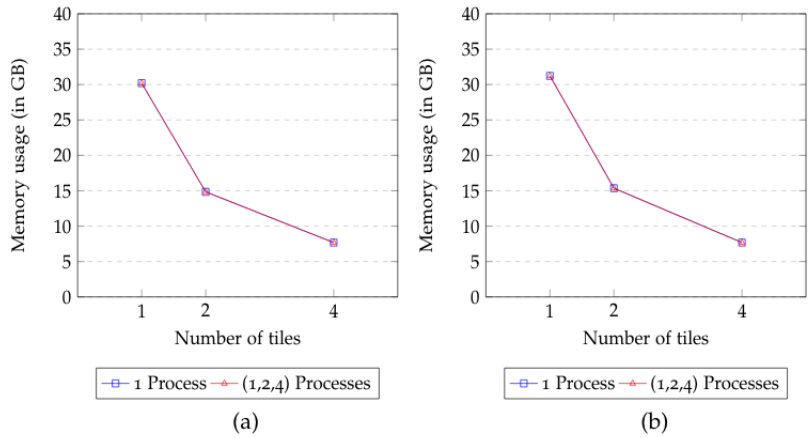

Figure 4.3: Maximum memory usage per process for (a) Haiti 8-bit and (b) Haiti 16-bit

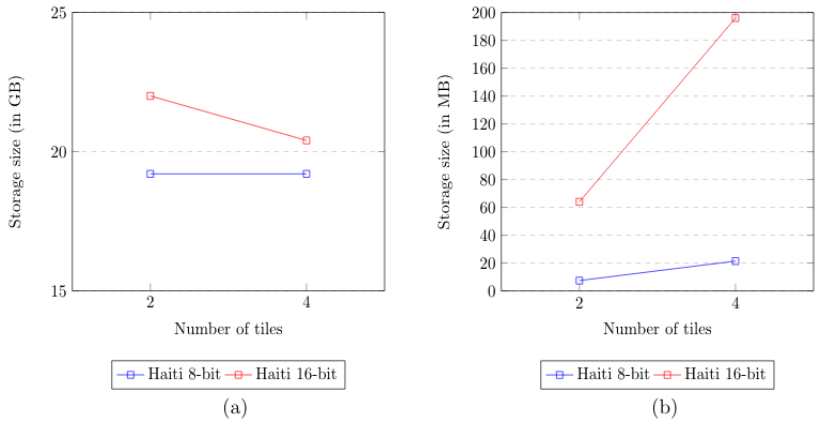

Figure 4.4: Size of (a) component tree and (b) boundary tree saved files

is stored per tile. Hence, with larger component tree and boundary tree size, the time taken to write it initially also increases. However for the 4 tiles case there are more than 4 boundary trees written, it also includes the time to write the merged and the combined trees. That's why steps 4, 5 and 6 take more time for 4 tile cases as the number of structures to read and write are simply more than the 2 tile case. So by increasing the number of tiles we save time for the most time consuming steps which are building, writing and filtering component trees. We can observe the same results in Figure 4.1 (b) for a 16-bit image of Haiti. But as the bit depth has increased we can also observe comparing Figure 4.1 (a) and Figure 4.1 (b) that the difference between 2 tiles and 4 tiles steps has decreased.

In Figure 4.2 (a) we can observe the timing required by 1 process varying the number of tiles and in Figure 4.2 (b) we can observe its corresponding speed-up plot. The time required by 1 process 4 tiles is faster than the time required by 1 process 1 tile for both 8-bit and 16-bit images. The time taken for 1 process 2 tiles is slower than 1 process 1 tile for 8-bit image but it's faster in 16-bit image case. This similarly reflects in the speed-up.

In Figure 4.3 we can observe for both 8-bit (a) and 16-bit (b) images, the memory usage per process scales down linearly as the number of processes increases. It takes the same amount of memory per process for 1 process n tiles as it does for n processes n tiles, where $n = 2, 4$. This is an important result as it shows with the iterative implementation 1 process performs exactly the same as n processes for n tiles memory wise.

Figure 4.4 (a) shows a non increasing trend in the size of the component tree files per tile. This is because the size of the component tree files per tile written decreases as the number of tiles increases. In Figure 4.4 (b) as the number of tiles increases, the number of boundary trees increases (including combined boundary trees) and hence their size too.

5 CONCLUSION & FURTHER IMPROVEMENTS

5.1 CONCLUSION

In this thesis an iterative implementation to construct DCF is presented with the help of reading and writing structures like component trees and boundary trees for 2 and 4 tiles. The implementation presented for 1 process uses $1/n^{th}$ the total memory used by n processes, where $n = 2, 4$. The time taken by 1 process and n tiles gets faster compared to the time needed by 1 process and 1 tile as n increases, where $n = 2, 4$ except for the 8-bit case where 1 process 2 tiles performs slightly worse than 1 process 1 tile. Hence, overall this iterative implementation constructs DCFs by using memory equivalent to n processes while taking less time in general compared to 1 process 1 tile.

The drawback of this iterative implementation is that as we increase the number of tiles, the size of this folder to store the component and boundary trees increases by a large factor. For example, an image of 1.2 Gigapixels results in a folder size of 20 GB in storage.

5.2 FURTHER IMPROVEMENTS

The iterative implementation of 1 process 4 tiles can be further extended to 1 process n tiles and m processes n tiles where $m < n$. Also the read and write functions can be more efficiently written to reduce the size of the folder where the files are saved. Further tests can be conducted on more image sets and on Peregrine cluster with 162 standard nodes with 24 Intel Xeon 2.5 GHz cores and 128 GB of RAM each. This would help to get a more accurate measure of memory being used.

APPENDIX

A.1 READ AND WRITE FUNCTIONS

Listing A.1: Function to write component tree to file

```
void write_tree_file(Node *tree, const char *fname, int num) {
  asprintf(&fname, "%s_%d", fname, num);
  FILE *fp = fopen(fname, "wb");
  fwrite(&tree->size_init, 1, sizeof(ulong), fp);
  fwrite(&tree->size_curr, 1, sizeof(ulong), fp);
  fwrite(&tree->size_attr, 1, sizeof(ulong), fp);
  fwrite(tree->offsets, 3, sizeof(ulong), fp);
  fwrite(tree->border, 6, sizeof(bool), fp);
  fwrite(tree->parent, tree->size_curr, sizeof(idx) , fp);
  fwrite(tree->attribute, tree->size_curr, tree->size_attr, fp);
  fwrite(tree->gval, tree->size_curr, sizeof(value), fp);
  fclose(fp);
}
```

Listing A.2: Function to write boundary tree to file

```
void write_boundary_file(Boundary *b, const char *fname, int num,
    Status s) {
  asprintf(&fname, "%s_%d", fname, num);
  FILE *fp = fopen(fname, "wb");
  fwrite(&b->size_init, 1, sizeof(ulong), fp);
  fwrite(&b->size_curr, 1, sizeof(ulong), fp);
  fwrite(&b->size_attr, 1, sizeof(ulong), fp);
  fwrite(b->offset, 7, sizeof(ulong), fp);
  fwrite(b->dims, 3, sizeof(ulong), fp);
  fwrite(b->merge_idx, b->offset[6], sizeof(idx), fp);
  fwrite(b->array, b->size_curr, sizeof(BoundaryNode), fp);
  fwrite(b->attribute, b->size_curr, b->size_attr, fp);
  if(s == BASIC) {
    idx *par_idx = malloc(b->size_curr*sizeof(idx));
    for (ulong i = 0; i < b->size_curr; i++) {
      par_idx[i] = b->border_par[i].i; // extracting the parents
          indexes
    }
    fwrite(par_idx, b->size_curr, sizeof(idx), fp);
  }
  else if(s == MERGED) {
    bool *partree = malloc(b->size_curr * sizeof(bool));
    idx *par_idx = malloc(b->size_curr*sizeof(idx));
    for(ulong i = 0; i < b->size_curr; i++) {
```

```
    if(b->border_par[i].b == b) {
     partree[i] = true;
    }
    else {
    partree[i] = false;
    }
   }
   fwrite(partree, b->size_curr, sizeof(bool), fp);
   for (ulong i = 0; i < b->size_curr; i++) {
    par_idx[i] = b->border_par[i].i;
   }
   fwrite(par_idx, b->size_curr, sizeof(idx), fp);
   if(b->border_lr != NULL) {
    bool *lrtree = malloc(b->size_curr * sizeof(bool));
    idx *lr_idx = malloc(b->size_curr*sizeof(idx));
    for(ulong i = 0; i < b->size_curr; i++) {
     if(b->border_lr[i].b == b) {
      lrtree[i] = true;
     }
     else {
      lrtree[i] = false;
     }
    }
    fwrite(lrtree, b->size_curr, sizeof(bool), fp);
    for (ulong i = 0; i < b->size_curr; i++) {
     lr_idx[i] = b->border_lr[i].i;
    }
    fwrite(lr_idx, b->size_curr, sizeof(idx), fp);
   }
  }
  fclose(fp);
}
```

Listing A.3: Function to write combined boundary tree to file

```
void write_boundary_combined_file(Boundary *b, const char *fname,
     int num, Boundary *b1, Boundary *b2, Status s) {
  asprintf(&fname, "%s%d", fname, num);
  FILE *fp = fopen(fname, "wb");
  fwrite(&b->size_init, 1, sizeof(ulong), fp);
  fwrite(&b->size_curr, 1, sizeof(ulong), fp);
  fwrite(&b->size_attr, 1, sizeof(ulong), fp);
  fwrite(b->offset, 7, sizeof(ulong), fp);
  fwrite(b->dims, 3, sizeof(ulong), fp);
  fwrite(b->merge_idx, b->offset[6], sizeof(idx), fp);
  fwrite(b->array, b->size_curr, sizeof(BoundaryNode), fp);
  fwrite(b->attribute, b->size_curr, b->size_attr, fp);
  if(s == BASIC) {
   idx *par_idx = malloc(b->size_curr*sizeof(idx));
   for(ulong i = 0; i < b->size_curr; i++) {
    par_idx[i] = b->border_par[i].i;
   }
```

```
    fwrite(par_idx , b->size_curr , sizeof(idx) , fp);
    if(b->border_lr != NULL) {
      idx *lr_idx = malloc(b->size_curr*sizeof(idx));
      for (ulong i = 0; i < b->size_curr; i++) {
        lr_idx[i] = b->border_lr[i].i;
      }
      fwrite(lr_idx , b->size_curr , sizeof(idx) , fp);
    }
  }
  else if(s == MERGED) {
    bool *partree = malloc(b->size_curr * sizeof(bool));
    idx *par_idx = malloc(b->size_curr*sizeof(idx));
    for(ulong i = 0; i < b->size_curr; i++) {
      if(b->border_par[i].b == b) {
        partree[i] = true;
      }
      else {
        partree[i] = false;
      }
      par_idx[i] = b->border_par[i].i;
    }
    fwrite(partree, b->size_curr, sizeof(bool), fp);
    fwrite(par_idx , b->size_curr , sizeof(idx) , fp);
    if(b->border_lr != NULL) {
      bool *lrtree = malloc(b->size_curr * sizeof(bool));
      idx *lr_idx = malloc(b->size_curr*sizeof(idx));
      for(ulong i = 0; i < b->size_curr; i++) {
        if(b->border_lr[i].b == b) {
          lrtree[i] = true;
        }
        else {
          lrtree[i] = false;
        }
      }
      fwrite(lrtree, b->size_curr, sizeof(bool), fp);
      for (ulong i = 0; i < b->size_curr; i++) {
        lr_idx[i] = b->border_lr[i].i;
      }
      fwrite(lr_idx, b->size_curr, sizeof(idx), fp);
    }
  }
  if(b->border_ori != NULL) {
    bool *oritree = malloc(b->size_curr * sizeof(bool));
    idx *ori_idx = malloc(b->size_curr * sizeof(idx));
    for(ulong i = 0; i < b->size_curr; i++) {
      if(b->border_ori[i].b == b1) {
        oritree[i] = true;
      }
      else if(b->border_ori[i].b == b2) {
        oritree[i] = false;
      }
    }
```

```
    fwrite(oritree, b->size_curr, sizeof(bool), fp);
    for (ulong i = 0; i < b->size_curr; i++) {
71    ori_idx[i] = b->border_ori[i].i;
    }
    fwrite(ori_idx, b->size_curr, sizeof(idx), fp);
  }
  fclose(fp);
76 }
```

Listing A.4: Function to read component tree from file

```
Node *read_tree_file(const char *fname, int num) {
  Node *tree = malloc(1* sizeof(Node));
  asprintf(&fname, "%s_%d", fname, num);
4 FILE *fp = fopen(fname, "rb");
  fread(&tree->size_init, 1, sizeof(ulong), fp);
  fread(&tree->size_curr, 1, sizeof(ulong), fp);
  fread(&tree->size_attr, 1, sizeof(ulong), fp);
  fread(tree->offsets, 3, sizeof(ulong), fp);
9 fread(tree->border, 6, sizeof(bool), fp);
  tree->parent = malloc(tree->size_curr*sizeof(idx));
  tree->attribute = malloc(tree->size_curr*tree->size_attr);
  tree->gval = malloc(tree->size_curr*sizeof(value));
  fread(tree->parent , tree->size_curr , sizeof(idx) , fp );
14 fread(tree->attribute , tree->size_curr , tree->size_attr , fp )
      ;
  fread(tree->gval , tree->size_curr , sizeof(value) , fp);
  fclose(fp);
  return tree;
}
```

Listing A.5: Function to read boundary tree from file

```
Boundary *read_boundary_file(const char *fname, int num, Status s
    ) {
2 asprintf(&fname, "%s_%d", fname, num);
  FILE *fp = fopen(fname, "rb");
  Boundary *b = calloc(1, sizeof(Boundary));
  fread(&b->size_init, 1, sizeof(ulong), fp);
  fread(&b->size_curr, 1, sizeof(ulong), fp);
7 fread(&b->size_attr, 1, sizeof(ulong), fp);
  b->size_alloc = b->size_curr;
  fread(b->offset, 7, sizeof(ulong), fp);
  fread(b->dims, 3, sizeof(ulong), fp);
  b->array = malloc(b->size_alloc * sizeof(BoundaryNode));
12 b->merge_idx = malloc(b->offset[6] * sizeof(ulong));
  b->attribute = malloc(b->size_alloc * b->size_attr); check_alloc
      (b->attribute, 220);
  b->border_par = malloc(b->size_alloc * sizeof(BorderIndex));
      check_alloc(b->border_par, 221);
  b->border_lr = malloc(b->size_curr * sizeof(BorderIndex));
```

```
  fread(b->merge_idx, b->offset[6], sizeof(idx), fp);
  fread(b->array, b->size_curr, sizeof(BoundaryNode), fp);
  fread(b->attribute, b->size_curr, b->size_attr, fp);
  size_t size = b->size_curr;
  if(s == BASIC) {
    idx *par_idx = malloc(b->size_curr * sizeof(idx));
    fread(par_idx , b->size_curr , sizeof(idx) , fp );
    for (size_t i = 0; i < size; i++) {
      b->border_par[i] = (BorderIndex) {.b = b, .i = par_idx[i]};
      b->border_lr[i] = (BorderIndex) {.b = b, .i = BOTTOM};
    }
  }
  else if(s == MERGED) {
    bool *partree = malloc(b->size_curr * sizeof(bool));
    idx *par_idx = malloc(b->size_curr * sizeof(idx));
    bool *lrtree = malloc(b->size_curr * sizeof(bool));
    idx *lr_idx = malloc(b->size_curr * sizeof(idx));
    fread(partree, b->size_curr, sizeof(bool), fp);
    fread(par_idx, b->size_curr, sizeof(idx), fp);
    fread(lrtree, b->size_curr, sizeof(bool), fp);
    fread(lr_idx, b->size_curr, sizeof(idx), fp);
    for(size_t i = 0; i < size; i++) {
      if(partree[i] == true) {
        b->border_par[i] = (BorderIndex) {.b = b, .i = par_idx[i]};
      }
      else {
        b->border_par[i] = (BorderIndex) {.b = NULL, .i = par_idx[i
            ]};
      }
    }
    for(size_t i = 0; i < size; i++) {
      if(lrtree[i] == true) {
        b->border_lr[i] = (BorderIndex) {.b = b, .i = lr_idx[i]};
      }
      else {
        b->border_lr[i] = (BorderIndex) {.b = NULL, .i = lr_idx[i
            ]};
      }
    }
  }
  fclose(fp);
  return b;
}
```

Listing A.6: Function to read combined boundary tree from file

```
Boundary *read_boundary_combined_file(const char *fname, int num,
    Boundary *b1, Boundary *b2, Status s) {
  asprintf(&fname, "%s%d", fname, num);
  FILE *fp = fopen(fname, "rb");
  Boundary *b = calloc(1, sizeof(Boundary));
  fread(&b->size_init, 1, sizeof(ulong), fp);
```

```
  fread(&b->size_curr, 1, sizeof(ulong), fp);
  fread(&b->size_attr, 1, sizeof(ulong), fp);
  b->size_alloc = b->size_curr;
  fread(b->offset, 7, sizeof(ulong), fp);
  fread(b->dims, 3, sizeof(ulong), fp);
  b->array = malloc(b->size_alloc * sizeof(BoundaryNode));
  b->merge_idx = malloc(b->offset[6] * sizeof(ulong));
  b->attribute = malloc(b->size_alloc * b->size_attr); check_alloc
      (b->attribute, 220);
  b->border_par = malloc(b->size_alloc * sizeof(BorderIndex));
      check_alloc(b->border_par, 221);
  b->border_lr = malloc(b->size_curr * sizeof(BorderIndex));
  b->border_ori = malloc(b->size_curr * sizeof(BorderIndex));
  fread(b->merge_idx, b->offset[6], sizeof(idx), fp);
  fread(b->array, b->size_curr, sizeof(BoundaryNode), fp);
  fread(b->attribute, b->size_curr, b->size_attr, fp);
  size_t size = b->size_curr;
  if(s == BASIC) {
   idx *par_idx = malloc(b->size_curr * sizeof(idx));
   idx *lr_idx = malloc(b->size_curr * sizeof(idx));
   bool *oritree = malloc(b->size_curr * sizeof(bool));
   idx *ori_idx = malloc(b->size_curr * sizeof(idx));
   fread(par_idx, b->size_curr, sizeof(idx), fp);
   fread(lr_idx, b->size_curr, sizeof(idx), fp);
   fread(oritree, b->size_curr, sizeof(bool), fp);
   fread(ori_idx, b->size_curr, sizeof(idx), fp);
   for(size_t i = 0; i < size; i++) {
      b->border_par[i] = (BorderIndex) {.b = b, .i = par_idx[i]};
   }
   for(size_t i = 0; i < size; i++) {
      b->border_lr[i] = (BorderIndex) {.b = b, .i = lr_idx[i]};
   }
   for(size_t i = 0; i < size; i++) {
    if(oritree[i] == true) {
      b->border_ori[i] = (BorderIndex) {.b = b1, .i = ori_idx[i
          ]};
    }
    else {
      b->border_ori[i] = (BorderIndex) {.b = b2, .i = ori_idx[i
          ]};
    }
   }
  }
  else if(s == MERGED) {
   bool *partree = malloc(b->size_curr * sizeof(bool));
   idx *par_idx = malloc(b->size_curr * sizeof(idx));
   bool *lrtree = malloc(b->size_curr * sizeof(bool));
   idx *lr_idx = malloc(b->size_curr * sizeof(idx));
   bool *oritree = malloc(b->size_curr * sizeof(bool));
   idx *ori_idx = malloc(b->size_curr * sizeof(idx));
   fread(partree, b->size_curr, sizeof(bool), fp);
   fread(par_idx, b->size_curr, sizeof(idx), fp);
```

```
    fread(lrtree, b->size_curr, sizeof(bool), fp);
    fread(lr_idx, b->size_curr, sizeof(idx), fp);
    fread(oritree, b->size_curr, sizeof(bool), fp);
    fread(ori_idx, b->size_curr, sizeof(idx), fp);
    for(size_t i = 0; i < size; i++) {
      if(partree[i] == true) {
        b->border_par[i] = (BorderIndex) {.b = b, .i = par_idx[i]};
      }
      else {
        b->border_par[i] = (BorderIndex) {.b = NULL, .i = par_idx[i
            ]};
      }
    }
    for(size_t i = 0; i < size; i++) {
      if(lrtree[i] == true) {
        b->border_lr[i] = (BorderIndex) {.b = b, .i = lr_idx[i]};
      }
      else {
        b->border_lr[i] = (BorderIndex) {.b = NULL, .i = lr_idx[i
            ]};
      }
    }
    for(size_t i = 0; i < size; i++) {
      if(oritree[i] == true) {
        b->border_ori[i] = (BorderIndex) {.b = b1, .i = ori_idx[i
            ]};
      }
      else {
        b->border_ori[i] = (BorderIndex) {.b = b2, .i = ori_idx[i
            ]};
      }
    }
  }
  fclose(fp);
  return b;
}
```

BIBLIOGRAPHY

[1] Edmond J Breen and Ronald Jones. "Attribute openings, thinnings, and granulometries." In: *Computer vision and image understanding* 64.3 (1996), pp. 377–389.

[2] Edwin Carlinet and Thierry Géraud. "A comparative review of component tree computation algorithms." In: *IEEE Transactions on Image Processing* 23.9 (2014), pp. 3885–3895.

[3] S. Gazagnes and M. H. F. Wilkinson. "Distributed Component Forests in 2-D: Hierarchical Image Representations Suitable for Tera-Scale Images." In: *International Journal of Pattern Recognition and Artificial Intelligence* 33.11 (2019), p. 1940012.

[4] R. Jones. "Connected filtering and segmentation using component trees." In: *Computer Vision and Image Understanding* 75.3 (1999), pp. 215–228.

[5] J. J. Kazemier, G. K. Ouzounis, and M. H. F. Wilkinson. "Connected morphological attribute filters on distributed memory parallel machines." In: *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer. 2017, pp. 357–368.

[6] Arnold Meijster, Michel A Westenberg, and Michael HF Wilkinson. *Interactive shape preserving filtering and visualization of volumetric data*. University of Groningen, Johann Bernoulli Institute for Mathematics and ..., 2002.

[7] Laurent Najman and Michel Couprie. "Building the component tree in quasi-linear time." In: *IEEE Transactions on Image Processing* 15.11 (2006), pp. 3531–3539.

[8] Philippe Salembier, Albert Oliveras, and Luis Garrido. "Antiextensive connected operators for image and sequence processing." In: *IEEE Transactions on Image Processing* 7.4 (1998), pp. 555–570.

[9] Erik R Urbach, Niek J Boersma, and Michael HF Wilkinson. "Vector-attribute filters." In: *Mathematical Morphology: 40 Years On*. Springer, 2005, pp. 95–104.

[10] Michel A Westenberg, Jos BTM Roerdink, and Michael HF Wilkinson. "Volumetric attribute filtering and interactive visualization using the max-tree representation." In: *IEEE Transactions on Image Processing* 16.12 (2007), pp. 2943–2952.