university of groningen / faculty of science and engineering

A Parallel Algorithm for the Contour Tree or Tree of Shapes for Self-Dual Image Filtering

Research internship

July 23, 2024

Author:
Tomas de Vries

Primary supervisor:
Michael Wilkinson

Secondary supervisor:
Kerstin Bunte

Abstract

Component trees can help filter images as they are easier to traverse than an image. The most useful tree is the tree of shapes since it works with self-dual filters. However, creating such trees can be a bottleneck. This bottleneck can be reduced if the tree of shapes could be made in parallel. So to allow for better image filtering we will look into the parallel creation of the tree of shapes in this research.

Contents

1 Introduction

Component trees represent the connected components of an image in a hierarchical way as a graph structure. These trees are powerful because there are several different strategies to filter them which can be done more efficiently than filtering the image directly. The most famous component trees are **max-trees** and **min-trees** [1], representing light structures on a dark background and dark structures on a light background, respectively. These are useful trees, however the tree of shapes is even more useful. This can be used for digital image processing [2]. An example can be seen in Figure 2. A more complex application would be identifying galaxies using their structure in the Tree of Shapes representation. In astronomy, images are generally very large which means that an efficient algorithm is essential. What makes the tree of shapes more powerful is that it is self-dual, this means that the tree's structure fundamentally changes if it were to be inverted, from min to max for example. A simple example of different trees can be seen in Figure 1. Self-dual filters are applied to self-dual trees. So these filters are not designed using max-trees and min-trees but have been designed using the tree of shapes. The tree of shapes can simultaneously represent light structures on a dark background and dark structures on a light background. The problem with this is the creation of the trees. Filtering can be done quickly, but the biggest bottleneck is creating the trees. Recent research [1, 3–7] looks into improving the time spent creating trees. Other research [1] has proposed a fast, *sequential* algorithm to create the tree of shapes. This algorithm is quasi-linear, which is quite efficient, but there is a limitation to what can be achieved with sequential algorithms. However, if we were to create an efficient parallel algorithm that decreases computation time when we increase the number of threads, we could still speed up the algorithm a lot.

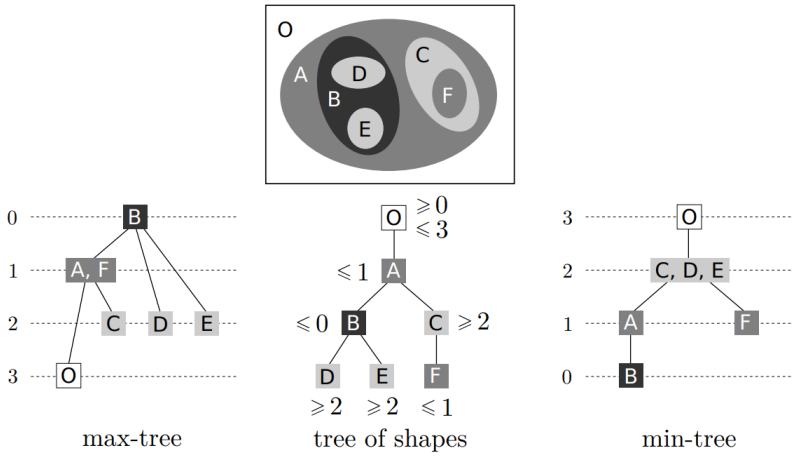


Figure 1: Three morphological trees of the same image, tree of shapes is self-dual. Figure from [1]

.

This leads to the following **research question**: What are the possibilities of adapting current state-of-the-art parallel max-tree algorithms to the tree of shapes?

We will answer this question by implementing and properly altering the research that is covered in Section 2. By doing so, it should result in an efficient parallel algorithm to create the tree of shapes.

In Section 2 we will go over the important research that has already been done and that will be built upon in this research. In Section 3 we will go over how we used the research from Section 2 to create our implementation. In this section we will also go over the changes that were made and the limitations that are present. In Section 4 we will show the results of running our algorithm with different numbers of threads. In Section 5 we will discuss what we have

achieved. In Section 6 we will discuss some possible research that can be done in the future.

(a) Noisy input image

(b) Denoised output image

Figure 2: Input image (a) denoised using the tree of shapes resulting in image (b) [8]

.

2 Background

The creation of component trees has been researched for a few decades [9]. The most famous trees, min-trees and max-trees, have gained a lot of different creation methods [10, 11]. These methods have become quite efficient however, the min-trees and max-trees are not self-dual, so they are not invariant to inversion of contrast. The tree of shapes is invariant to inversion of contrast but the creation is a bit more complex.

2.1 The first algorithm

The first known algorithm that created the tree of shapes is the "Fast Level Line Transform (FLLT)" [12]. Creating the tree of shapes was done by first creating both the max-tree and min-tree of an image and merging both trees. A major drawback is the need to know that a component has a hole so it can be matched with a component of the other tree. They have shown in [13, 14] that this approach gives correct results for nD-images.

2.2 Creating the tree of shapes sequentially

To counter this problem a sequential quasi-linear algorithm was proposed to create the tree of shapes [1]. The algorithm to create max-trees is described in Algorithms 1 and 2, to create the tree of shapes this algorithm can be altered, which we will look into later. The whole algorithm is described by COMPUTE_TREE which makes use of the other functions seen in Algorithms 1 and 2. First, the image u is used in a SORT function to compute the ancestor relationship $\mathcal{R}$. This is represented as an array so that $a\mathcal{R}p \Leftrightarrow index_{\mathcal{R}}(a) < index_{\mathcal{R}}(p)$. With the ancestor relationship $\mathcal{R}$ the union-find algorithm proposed in [7] is used. The algorithm is altered so that it computes the morphological tree as expected [6] while browsing pixels from leaves to root. The last step is to canonicalize the tree. Which is some trivial post-processing described in [6].

Computing the tree of shapes has some modifications. These can be seen in Algorithms 3, 4 and 5. As seen in Algorithm 5, the algorithm is now surrounded by `INTERPOLATE` and `UN-INTERPOLATE`. `INTERPOLATE` adds information in-between pixels of the input image u. u is subdivided, this multiplies the size by 4^n where n is the dimensionality. This subdivided image is interpolated into the Khalimsky grid [16]. For 2D images, every pixel has four points, four edges, and one square. An example of this can be seen in Figure 3. This interpolation is necessary to ensure the correct sorting of pixels.

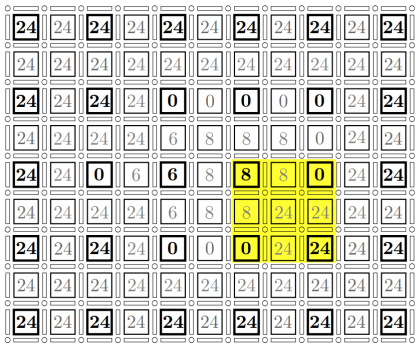
Algorithm 1 Union-Find-based Computation of a Morphological Tree [15]

function UNION_FIND(R)
  for all p **do**
    $zpar(p) \leftarrow$ undef
  end for
  for $i \leftarrow N-1$ **to** 0 **do**
    $p \leftarrow R[i]$
    $parent(p) \leftarrow p$
    $zpar(p) \leftarrow p$
    for all $n \in N(p)$ such as $zpar(n) \neq$ undef **do**
      $r \leftarrow$ FIND_ROOT$(zpar, n)$
      if $r \neq p$ **then**
        $parent(r) \leftarrow p$
        $zpar(r) \leftarrow p$
      end if
    end for
  end for
  return *parent*
end function

Algorithm 2 Find root function and compute tree which results in final tree [15]

function FIND_ROOT(zpar, x)
  if $zpar(x) = x$ **then**
    return x
  else
    $zpar(x) \leftarrow$ FIND_ROOT$(zpar, zpar(x))$
    return $zpar(x)$
  end if
end function
function COMPUTE_TREE(u)
  $R \leftarrow$ sort(u)
  $parent \leftarrow$ UNION_FIND(R)
  CANONICALIZE_TREE(u, R, parent)
  return $(R, parent)$
end function

24	24	24	24	24	24
24	24	0	0	0	24
24	0	6	8	0	24
24	24	0	0	24	24
24	24	24	24	24	24



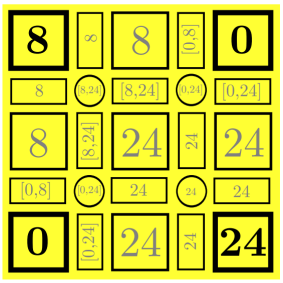


Figure 3: Interpolation of a 2D image. Figure from [1].

Algorithm 5 Compute Tree of Shapes [15]

function COMPUTE_TREE_OF_SHAPES(u)
  $U \leftarrow$ INTERPOLATE(u)
  $(R, u[\ldots]) \leftarrow$ SORT(U)
  $parent \leftarrow$ UNION_FIND(R) CANONICALIZE_TREE(u[. . .], R, parent)
  return UN_INTERPOLATE(R, parent)
end function

Algorithm 3 Priority Queue Operations [15]

```
1: function PRIORITY_PUSH(q, h, U, l)
2:     [lower, upper] ← U(h)
3:     if lower > l then
4:         l' ← lower
5:     else if upper < l then
6:         l' ← upper
7:     else
8:         l' ← l
9:     end if
10:    PUSH(q[l'], h)
11: end function
12: function PRIORITY_POP(q, l)
13:    if q[l] is empty then
14:        l' ← level next to l such as q[l'] is not empty
15:        l ← l'
16:    end if
17:    return POP(q[l])
18: end function
```

Algorithm 4 Sort for Tree of Shapes Computation [15]

```
1: function SORT(U)
2:     for all h do
3:         deja_vu(h) ← false
4:     end for
5:     i ← 0
6:     PUSH(q[l_∞], p_∞)
7:     deja_vu(p_∞) ← true
8:     l ← l_∞
9:     while q is not empty do
10:        h ← PRIORITY_POP(q, l)
11:        u^b[h] ← l
12:        R[i] ← h
13:        for all n ∈ N(h) such as deja_vu(n) = false do PRIORITY_PUSH(q, n, U, l)
14:            deja_vu(n) ← true
15:        end for
16:        i ← i + 1
17:    end while
18:    return (R, u)
19: end function
```

The major difference between computing a max-tree or the tree of shapes is the sorting step. The max-tree is sorted such that $i < i' \Rightarrow u(\mathcal{R}[i]) \leq u(\mathcal{R}[i'])$, while the tree of shapes is sorted such that it goes from the "external" shapes to the "internal" shapes. The sorting, as seen in Algorithm 4, is done using a hierarchical queue [17], denoted by q, and the current level is denoted by l. There are some important things to note, one of which is how the hierarchical queue handles the values in between pixels. These values are intervals but are handled as a single value that is closest to l. Secondly, it is important to note that when the queue at the current level, $q[l]$ is empty and q is not, it needs to be decided what the next level to be processed is. The possibilities are either taking the level less or greater than l, such that the queue at that level is not empty. So basically it is choosing between going up or down the levels, which does not change the resulting tree but changes which subtree is created first. Lastly, it should be noted that using a hierarchical queue with high dynamic images could cause issues since it would mean having a large amount of levels in the queue.

For the tree of shapes a temporary image u^b is introduced. This is used in the sorting step to memorize the enqueueing level of faces. Furthermore, it is used in the `CANONICALIZE_TREE` step to check if an element h is canonical, meaning that it checks if $u^b(parent(h)) \neq u^b(h)$. So it checks if the parent's value is the same as the current value.

This describes the complete algorithm. The complexity of it is quasi-linear, which is pretty good. But this is still a sequential algorithm. We would like to look into the parallelization of this algorithm.

2.3 Creating the tree of shapes in parallel

Creating a tree of shapes in parallel has been researched before [15]. In this section, we will look into this research methodology and results.

The parallel implementation builds upon the sequential quasi-linear algorithm described above and also in [1]. The function is modified such that it looks like the algorithm seen in Algorithm 6. All steps are parallelized, except for the canonicalization, and the union-find function is replaced

by a parallel max-tree computation algorithm. The following algorithm does not specify a max-tree algorithm, this is because most max-tree algorithms are compatible. The paper [15] uses an old max-tree algorithm [18] for testing. This algorithm is not compatible with high-dynamic range images. This is therefore not likely to work well with large images that output high values.

Algorithm 6 Computing the tree of shapes in parallel [15]

```
function ComputeTree(f, p_∞)
    F ← ParallelImmerse(f)
    Q ← list of queues
    λ ← Mean(F(p_∞))
    Q[λ] ← p_∞
    F^ord ← ParallelSort(F, Q, λ, 0)
    par ← ParallelMaxTree(F^ord)
    return Canonicalize(par, F^ord)
end function
```

The `ParallelImmerse` function describes the interpolation of the image f as described in the quasi-linear algorithm [1] but computed in parallel. This is done by computing the valuation of each face in parallel, which is possible because the valuation of each face only requires information about its local configuration.

The `ParallelSort` function describes the sorting as described in the quasi-linear algorithm [1] but with some modifications to allow for parallel computation. The algorithm can be seen in Algorithm 7. The sorting of a level λ is initially done normally. After the propagation of a level λ the parallelization is achieved by creating another thread that sorts the set of all values in the interpolated image, also known as **faces**, below the current one S_λ^-, mathematically described by $S_\lambda^- = \{x \in \mathcal{S} | \mathcal{F}^b(x) < \lambda\}$. Of course, there is also the set of all faces above the current one $S_\lambda^+ = \{x \in \mathcal{S} | \mathcal{F}^b(x) > \lambda\}$, the current thread will use this set. So the algorithm describes a parallel recursion where `ParallelSort` is run on separate threads. This works correctly because of the following property:

Property 1 *After each propagation step each unvisited distinct sub-tree of the Tree of Shapes corresponds to a distinct connected component of the remaining pixels [15].*

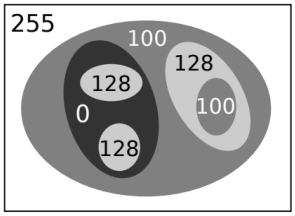
The output of the `ParallelSort` function is not $\mathcal{R}$ like it was in the sequential algorithm [1]. Instead, it returns $\mathcal{F}^{ord}$, which describes the face with its level on the tree of shapes. So the face now has a value based on how deep in the image it is in terms of its inclusion level. An example of this can be seen in Figure 4. As we can see, the max-tree of (b) is the same as the tree of shapes of (a). So thanks to the sorting procedure, we can now use any parallel max-tree computation for the`ParallelMaxTree` function to compute the tree of shapes.

Lastly, the canonicalization step is modified so it can be performed without using the ancestor relationship $\mathcal{R}$. To do this, the function is split into two passes that find the correct representatives of points. We assume that the max-tree computation returns a *par* function that describes each face of the tree of shapes $\mathcal{F}$ along with its corresponding parent on the tree. After the canonicalization step, the artificial faces have been removed and the final tree of shapes is obtained.

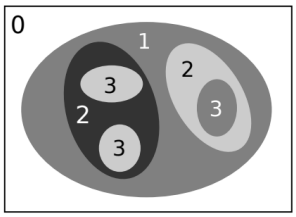
So this algorithm describes the creation of the tree of shapes in parallel. In this research we will build upon this research by using a proper parallel max-tree algorithm that will be described in the next section.

Algorithm 7 Parallel Sort Procedure [15]

procedure ParallelSort($\mathcal{F}, Q, \mathcal{F}^{ord}, \lambda, ord$)
  $Q[\lambda] \leftarrow p_\infty$
  while any queue in Q is not empty **do**
    while $Q[\lambda]$ is not empty **do**
      $p \leftarrow \text{POP}(Q[\lambda])$
      $\mathcal{F}^{ord}(p) \leftarrow ord$
      for all $n \in N_4(p)$ not visited yet **do**
        if $\lambda \in \mathcal{F}(n)$ **then**
          $\text{PUSH}(Q[\lambda], n)$
        else if $\lambda < \min(\mathcal{F}(n))$ **then**
          $\text{PUSH}(Q[\min(\mathcal{F}(n))], n)$
        else
          $\text{PUSH}(Q[\max(\mathcal{F}(n))], n)$
        end if
      end for
      $ord \leftarrow ord + 1$
    end while
    $S_\lambda^+ \leftarrow Q[\lambda \dots \text{max value}]$
    $S_\lambda^- \leftarrow Q[0 \dots \lambda]$
    $\lambda' \leftarrow$ highest level having faces on S_λ^-
    parallel { ParallelSort($\mathcal{F}, S_\lambda^-, \mathcal{F}^{ord}, \lambda_0, \text{ord}$) on another thread }
    $Q \leftarrow S_\lambda^+$
    $\lambda \leftarrow$ smallest level having faces on S_λ^+
  end while
  wait all child threads
end procedure



(a) Original image



(b) Re-valued image

Figure 4: Original image (a) and the associated $\mathcal{F}^{ord}$ (b) computed by the `ParallelSort` function [15]

.

2.4 Parallel max-tree algorithm

To properly use the parallel tree of shapes algorithm described in the section above, we need a parallel max-tree algorithm. There have been several parallel max-tree algorithms proposed in research [3–5, 19]. Most research achieves increased performance, however this is only up to 16 bits per pixel. In this section we will look into a parallel max-tree algorithm proposed in [20] that can be used to efficiently build max-trees of high-dynamic range images or very large images.

The first step of the parallel max-tree algorithm is computing a quantized version $\overline{f}$ of the input image f. This quantized image will be used to build a **pilot** max-tree. This pilot max-tree can eventually be used for correct attribute computation and merging of the sub-trees. The tree can be built in parallel using any existing algorithm. Which algorithm is not very relevant due to the low number of values in the quantized image $\overline{f}$. The paper uses the parallel version [18] of the sequential algorithm in [21] because they found it to be faster than other methods on such low-quantized values. A relation is enforced between the level roots (introduced as canonical elements in section 2.2) of the pilot tree and those of the final tree. The level roots of the pilot tree are a subset of those of the final tree. To do this, the only change made concerning the max-tree implementation in [18] is a stricter definition of the level root of a connected component. The canonical element node where the attributes are accumulated now corresponds to the pixel with the lowest coordinate. The algorithm that builds the max tree can be seen in Algorithms 8 and 9.

The second step of the parallel max-tree algorithm is the **refinement** stage. This stage shapes the pilot tree into the max-tree of the original image. This stage is a parallel version of the algorithm proposed by Berger et al. [6]. The complete algorithm of this stage is shown in Algorithms 10 and 11. All nodes that will be stored refer to a pixel, similar to the pilot max-tree, with the structure of parent pointer, area attribute value, and pixel intensity value after filtering. The final tree is created in parallel using K threads that work on S partitions on the original pixel values of f. The number of partitions S is the same as the number of threads K. Every thread T_i retrieves the sorted pixels belonging to its corresponding partition S_i. Each one has the same quantized intensity, in descending order. The computation starts from the pixels that correspond to the maxima from the partition S_i. For every pixel p retrieved from the sorted array and belonging to S_i, the set of neighbour pixels is calculated. Only the pixels with intensity greater than or equal to the current pixel's intensity could have been processed so only the already processed neighbours are considered. This leaves two possible situations, where H_i describes the set of intensity values in S_i.

Possibility 1: Intensity $\overline{f}(q) \in H_i$

> In this case, the intensity $\overline{f}(q)$ of the neighbour pixel q belongs to the set H_i. So the computation continues as in the Berger algorithm [6]. These steps are performed in lines 22-29 of Algorithm 10.

Possibility 2: Intensity $\overline{f}(q) \notin H_i$

> Since pixels must be processed in decreasing order of intensity, if $\overline{f}(q) < h_i$, then neighbour q is considered not visited, and the computation continues by retrieving the next neighbour. However, if $\overline{f}(q) \geq h_{i+1}$, then q is considered visited. Since $\overline{f}(q) \notin H_i$, this means that the neighbour q belongs to a partition S_j with $i < j$. Thus, the sections being built by threads T_i and T_j must now be merged. Here, the pilot max-tree is used to retrieve the attribute of the closest descendant of the component that p belongs to in the pilot max-tree and drive the merging of both sub-trees and attributes. These steps are performed in lines 9-21 of Algorithm 10.

So this algorithm describes the creation of a max-tree in parallel. In this research, we will combine

the elements described above. The main part of the algorithm is described by algorithm 6. Starting with the parallel immerse function. This can be done by After that, we will use the parallel sort algorithm seen in algorithm 7 to convert the interpolated image so it can be used by the max tree algorithm. The max tree algorithm is described by algorithms 8, 9, 10, and 11. After that the artificial faces are removed using the canonicalize function. use the parallel version (Section 2.3) from the sequential tree of shapes algorithm (Section 2.2) along with the parallel max-tree algorithm (Section 2.4) that we will apply on the new image representation. This should result in an efficient computation of the tree of shapes in parallel.

Algorithm 8 `Flood_Pilot` Builds the pilot Max-Tree [20]

```
1: procedure FLOOD_PILOT(Level lev, Partition P, Attribute thisarea)
2:     area ← thisarea
3:     while Queue at level lev is not empty do
4:         Extract pixel p from the Queue at level lev
5:         area ← area + 1
6:         for all neighbours q of p with q ∈ P do
7:             if isVisited[q] = false then
8:                 isVisited[q] ← true
9:                 if levelroot[g(q)] = not set then
10:                    levelroot[g(q)] ← q
11:                else
12:                    KEEPLOWESTLEVELROOT(q)
13:                end if
14:                Add q to the Queue at level f(q)
15:                if f̄(q) > lev then
16:                    childarea ← 0
17:                    fq ← f̄(q)
18:                    while fq > lev do
19:                        fq ← FLOOD_PILOT(f̄(q), P, childarea)
20:                    end while
21:                    area ← area + childarea
22:                end if
23:            end if
24:        end for
25:    end while
26:    m ← lev − 1
27:    while m ≥ 0 ∧ levelroot[m] = not set do
28:        m ← m - 1
29:    end while
30:    if m ≥ 0 then
31:        node_qu[levelroot[lev]].parent ← levelroot[m]
32:    end if
33:    qu[levelroot[lev]].Area ← area
34:    levelroot[lev] ← not set
35:    thisarea ← area
36:    return m
37: end procedure
```

Algorithm 9 Keep Lowest Level Root [20]

```
1: procedure KEEP_LOWEST_LEVEL_ROOT(Pixel q)
2:     cond1 ← f(q) < f(levelroot[f̄(q)])
3:     cond2 ← f(q) = f(levelroot[f̄(q)]) ∧ q < levelroot[f̄(q)]
4:     if cond1 ∨ cond2 then
5:         node_qu[levelroot[f̄(q)]] : parent ← q
6:         levelroot[f̄(q)] ← q
7:     end if
8:     node_qu[q].parent ← levelroot[f̄(q)]
9: end procedure
```

Algorithm 10 Pseudo-Code of the Refinement Stage [20]

```
1: procedure REFINEMENT(Thread i, Partition S_i)
2:     lwb ← min(S_i)
3:     upb ← max(S_i)
4:     qi ← quantized intensity managed by T_i
5:     for j = upb to lwb do
6:         p ← SortedArray[j]
7:         zpar[p] ← p
8:         for all neighbours q of p do
9:             if f̄(q) > qi then
10:                desc ← DESCENDROOTS(q, i)
11:                if node_ref[desc].parent =not set then
12:                    node_ref[desc].parent ← p
13:                    node_ref[p] : Area ← node_ref[p].Area + node_qu[desc].Area
14:                else
15:                    z ← FINDROOT(node_ref[desc].parent)
16:                    if z ≠ p then
17:                        node_ref[z].parent ← p
18:                        zpar[z] ← p
19:                        node_ref[p].Area ← node_ref[p].Area + node_ref[z]
20:                    end if
21:                end if
22:            else if f̄(q) = qi then
23:                if zpar[q] ≠ −1 then
24:                    r ← FINDROOT(q)
25:                    if r ≠ p then
26:                        node_ref[r].parent ← p
27:                        zpar[r] ← p
28:                        node_ref[p].Area ← node_ref[p].Area + node_ref[r].Area
29:                    end if
30:                end if
31:            end if
32:        end for
33:    end for
34: end procedure
```

Algorithm 11 Descend Roots [20]

```
1: procedure DESCEND_ROOTS(Pixel q, int i)
2:     c ← q
3:     while f̄(node_qu[c].parent) > i do
4:         c ← node_qu[c].parent
5:     end while
6:     return c
7: end procedure
```

3 Implementation

For the implementation, we focused on combining the algorithms described in Sections 2.3 and 2.4. The implementation is split up in different phases: Interpolation, image converting, sorting, creating quantized image, making the quantized tree, refining the tree, and filtering. The first 2 phases have been implemented in C based on the pseudocode from [15], the other phases were present in Moschini's source code [20] and made compatible during this research.

3.1 Interpolation

As described in section 2.2, interpolation is the process of assigning values between pixels. To create a parallel implementation we evenly split up the image over the specified number of threads. So each thread is responsible for its distinct region. We add the desired number of *faces* for every pixel. In 2D, this means that an additional 15 values are added. There are multiple options for which in-between values are managed per pixel, as long as every pixel can operate independently. The in-between values are managed as seen in Figure 5. An important thing to note here is the high memory cost. For a 2D image, a grid will be created that is 16 times the size of the input, for 3D it is even 64 times the size of the input. In this grid, intervals are used which means that 2 values can be held per in-between value. This means that the total memory for the interpolated image is the input image size times 32 for 2D and times 128 for 3D. Since we want to use big data for the parallel tree of shapes, the memory cost can be quite expensive. This can be seen as a bottleneck for this algorithm.

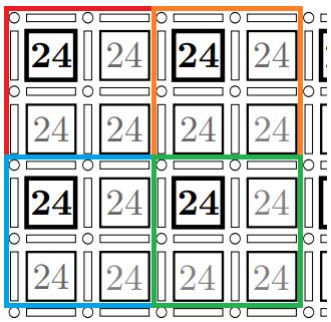


Figure 5: The values are divided in this way when running interpolation in parallel. For every **primary pixel** the corresponding values are calculated.

3.2 Converting the image

The algorithm of the image conversion that we are focusing on in this section can be seen in Algorithm 7. As described in section 2.3, the goal of this phase is to convert the input image such that the tree of shapes of the input image is the same as the max tree of the output image. This means that flat zones deeper in the image should have higher values. We have implemented this algorithm and looked at the results. Our implementation of the algorithm correctly converts the input image to be compatible with the parallel max tree algorithm that we will discuss further in section 3.3. Examples of the conversions can be seen in Figure 6. However, some difficulties arose when using this algorithm. This has caused us to make some changes to the algorithm. In this section, we will go over some bottlenecks of this algorithm and the changes that were made to work around them.

The first bottleneck is the recursive nature of the algorithm. In theory, recursion works perfectly fine and can result in an algorithm that visits all pixels with optimal time efficiency. In reality, recursion has some disadvantages. In the worst-case scenario, this algorithm creates threads in the order of n where n is the size of the input image. This can be inefficient in both memory

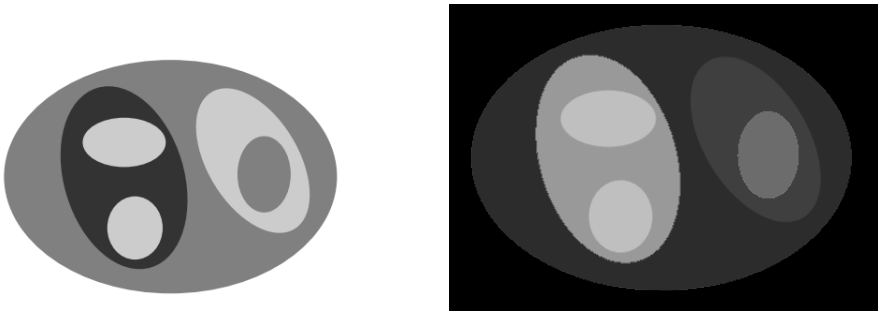
and time complexity. Different threads are responsible for different subtrees, but if these subtrees are not big enough, the overhead of creating the thread is not worth handling the small subtree. And if more threads are running than there are cores on the machine, there will be less speedup. Not only that, but every thread needs its own set of queues. Which can cost a lot of memory. We will talk about the memory cost of queues in the next paragraph. So in short, such a high number of threads is undesirable. To work around this problem we changed the algorithm. The changed algorithm uses a main thread that fires child threads. These child threads handle subtrees and once they are done, new threads can be fired. The downside of this approach is that we do not know what part of the data the child threads are handling. This can result in a load-balancing problem where one child thread handles almost all of the image. This can also be seen in the results in section 4. Since this approach was not optimal we tried another approach. In this approach, we would keep track of how many threads are running and not use a main thread. This way when one thread is done, another thread can fire from any thread. Theoretically, this helps with the load-balancing problem that we had. However, using more threads resulted in a longer computation time with this approach. This was likely caused by the overhead of small subtrees and constant checking of the number of threads. Because the computation time increases with this approach, the previous approach is used where one thread handles most of the image. Theoretically, this problem can be solved by figuring out where the 'holes' in the image are and assigning threads to those areas. This will cost some computation time but allows for better parallelization. This task is currently out of scope for this research and therefore has not been looked into further.

Another bottleneck that ties in with the high number of threads is the high memory cost of queues. In the queues seen in Algorithm 7, we store the pixels to be pushed. It is impossible to know beforehand which pixels are going to be pushed and at what value, so the queues for all values should be created beforehand. This holds for every thread. This means that every thread needs a queue of the size of the interpolated image. Since the interpolation already has a high memory cost, this causes the memory cost to increase even further. This could only be prevented by knowing which flat zones you have beforehand. Since we have not researched this further, this research adjusts the input image size to still allow us to see the increase in performance by parallelization but uses a reasonable amount of memory on a high-performance cluster. The amount of memory that will be used will be in the range of 100GB to 1TB.

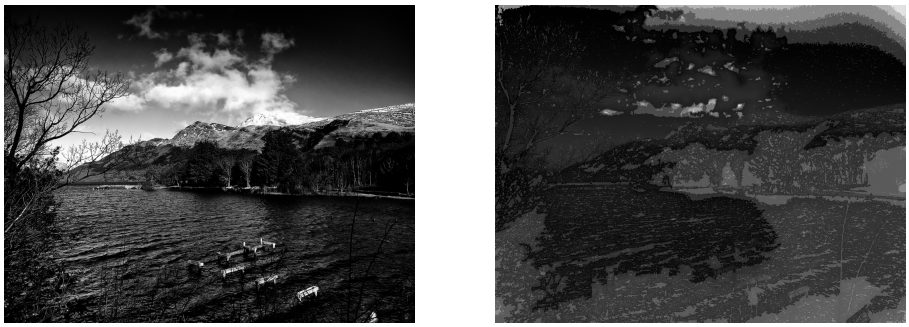
We can conclude that this conversion algorithm has some difficulties with both memory and time complexity when applied in practice. This can cause this phase to be a bottleneck in the complete algorithm.

3.3 Max-tree

This section goes over the remaining phases that were mentioned in the introduction of Section 3: sorting, creating a quantized image, making the quantized tree, refining the tree, and filtering. This is all part of the parallel max tree algorithm from [20]. The main thing about implementing the max tree algorithm in this tree of shapes algorithm is adjusting the input, which is done in the previous phases. This input can simply be used in the max-tree algorithm like normal. The final step in Algorithm 6 is canonicalizing, which compresses the paths of the tree and de-interpolates the output. In this research, we do not need to worry about this since Moschini et al's algorithm [20] already applies path compression in FIND_ROOT as seen in Algorithm 2. This is beneficial since the canonicalize function from Crozets algorithm [15] did not run in parallel. The output of this max-tree algorithm is the final tree of shapes.



(a) Original image (b) Re-valued image



(c) Original image (d) Re-valued image

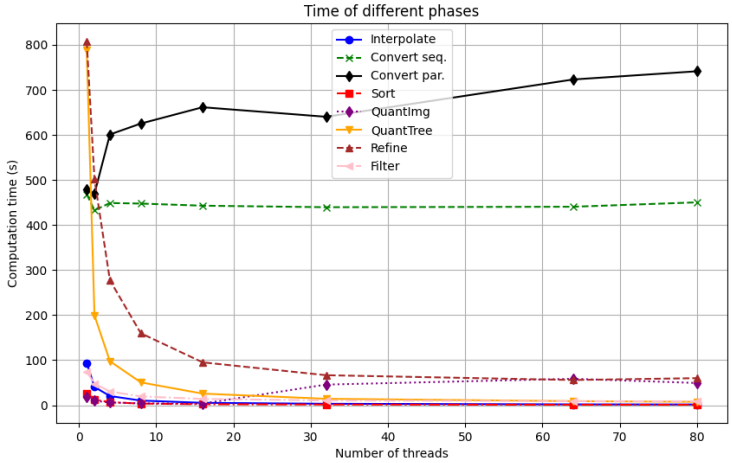
Figure 6: Original images (a, c) and the associated $\mathcal{F}^{ord}$ (b, d) computed by the `ParallelSort` function [15]

.

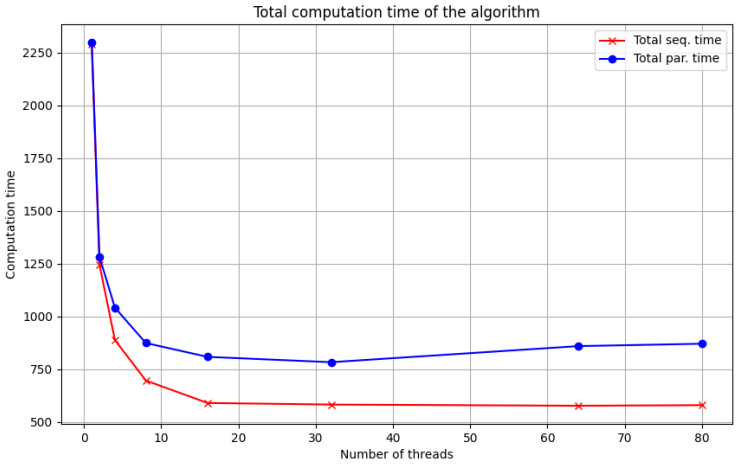
4 Results

In this section, we will focus on the speedup of the algorithm. Checking the time it takes to complete each phase with different numbers of threads. The results of the algorithm are shown in Figure 7. The computation times of all phases are shown in Figure 7a, the total is shown in Figure 7b. For these results, we looked into the computation times of using different numbers of threads, namely: 1, 2, 4, 8, 16, 32, 64, and 80. Doubling the number of threads can effectively show a difference in computation time. Lastly, we used 80 threads because that was the number of cores available on the node. To minimize outliers we ran the algorithm twice and took the minimum computation time. Lastly, we calculated the speedup of the total computation time as seen in Figure 7c.

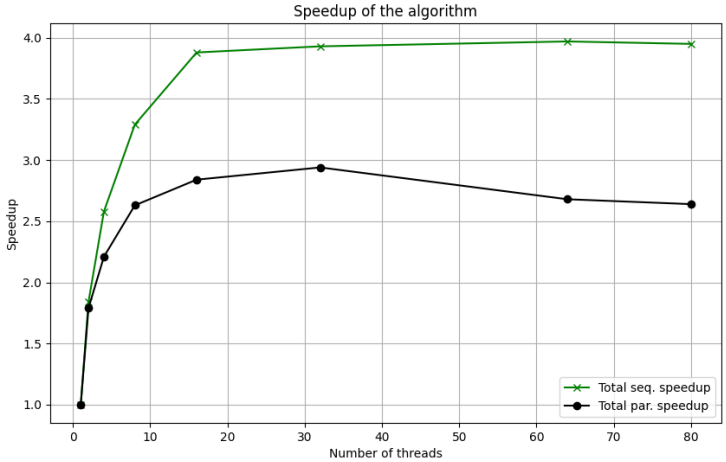
As seen in Figure 7b using a higher number of threads decreases the total computation time. This also holds for most phases as seen in Figure 7a. Due to the bottleneck issues discussed in Section 3.2 and the overhead of creating threads, the computation time for the conversion phase increases with the total number of threads. This is seen in the black plot in Figure 7a. Therefore we also show the sequential approach for this phase. That is why one plot of the conversion phase is almost flat and the other increases. The small changes in the sequential plot would decrease if we take an average of runs, but it is clear that the performance of this phase does not change much, since it always uses the same sequential approach. Another thing to note is that generating the quantized image takes longer after 16 threads. When creating the quantized image, it will first be decided how many levels the quantized image will be. If the number of quantized levels is lower than the number of threads, the algorithm then uses another method that sacrifices load balancing. This causes the increase in computation time seen in Figure 7a. After 32 threads the total computation time starts to increase. Most phases already run with a very low computation time, this means that there is not enough speedup to compensate for the increase in computation time for the conversion and quantized image phase.



(a) The computation time of all phases



(b) The total computation time with the sequential and parallel converting phase



(c) The speedup of the algorithm with the sequential and parallel converting phase

Figure 7: Computation time of all phases (a) and in total (b), as well as the speedup of the total computation time (c)

.

The optimal speedup of an algorithm is linear with the number of threads. In Figure 7c we see that the speedup is not really linear. For two threads it is almost linear but for more threads the speedup does not increase as much. We see that we almost get the same speedup as the original algorithm in [15] which was a speedup of 3 on 4 threads compared to our 2.6. The difference most likely lies in the quantization steps. These were not used in the max tree algorithm that was used for the tree of shapes algorithm [15].

5 Conclusion

In this research, we have focused on the parallel creation of the tree of shapes. This consists of different phases that should all be parallelizable. This includes the interpolation of the image to assign values in-between pixels and converting the image such that the tree of shapes of the input is the same as the max tree of the output. The interpolation and conversion were both proposed in [15]. In Section 4 we see the speedup of every phase. Here we see that all phases benefit greatly from parallelization, except for the converting phase. The speedup is not optimal (linear with respect to threads), but still shows a great improvement compared to the sequential algorithm. This converting phase was proposed in [15] and has some difficulties causing it to be the bottleneck of this algorithm. Section 3.2 explains the difficulties in more detail. This algorithm also has an undesirable memory cost. This is caused by the high cost of interpolation and queues. This algorithm can easily get a memory use over $1000n$ where n is the image size. And this will only get more complex when using 3D images. Since the complexity of 2D images is already high, it is not an algorithm that would work well for 3D images.

6 Future work

As mentioned in the conclusion this algorithm currently has a bottleneck in the phase where the image gets converted such that the tree of shapes of the input is the same as the max tree of the output. So the key part that needs more research is this phase. Currently, it is hard to achieve good load balancing. In future research, it might be useful to look into a parallel algorithm that identifies the flat zones in the image. Using this it is possible to find the gaps in the image. With the gaps in the image, it is possible to fire child threads that handle the gaps. It could also be used to identify the amount of memory you will need in advance, which could heavily reduce memory costs.

Due to the high memory cost, this algorithm only works on 2D images and has not been implemented to work with 3D images. In the future, especially if the load balancing can be fixed, looking into implementing 3D functionality would be worthwhile.

References

[1] Thierry Géraud, Edwin Carlinet, Sébastien Crozet, and Laurent Najman. A quasi-linear algorithm to compute the tree of shapes of n d images. In *International symposium on mathematical morphology and its applications to signal and image processing*, pages 98–110. Springer, 2013.

[2] Coloma Ballester, Vicent Caselles, and Pascal Monasse. The tree of shapes of an image. *ESAIM: Control, Optimisation and Calculus of Variations*, 9:1–18, 2003.

[3] Plamenka Borovska and Milena Lazarova. Efficiency of parallel minimax algorithm for game tree search. In *Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–6, 2007.

[4] Markus Götz, Gabriele Cavallaro, Thierry Géraud, Matthias Book, and Morris Riedel. Parallel computation of component trees on distributed memory machines. *IEEE transactions on parallel and distributed systems*, 29(11):2582–2598, 2018.

[5] Edwin Carlinet and Thierry Géraud. A comparative review of component tree computation algorithms. *IEEE Transactions on Image Processing*, 23(9):3885–3895, 2014.

[6] Ch Berger, Th Géraud, Roland Levillain, Nicolas Widynski, Anthony Baillard, and Emmanuel Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. In *2007 IEEE international conference on image processing*, volume 4, pages IV–41. IEEE, 2007.

[7] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[8] Françoise Dibos and Georges Koepfler. Total variation minimization by the fast level sets transform. In *Proceedings IEEE Workshop on Variational and Level Set Methods in Computer Vision*, pages 179–185. IEEE, 2001.

[9] Ronald I Becker, Stephen R Schach, and Yehoshua Perl. A shifting algorithm for min-max tree partitioning. *Journal of the ACM (JACM)*, 29(1):58–67, 1982.

[10] Jiří Havel, François Merciol, and Sébastien Lefèvre. Efficient schemes for computing α-tree representations. In *Mathematical Morphology and Its Applications to Signal and Image Processing: 11th International Symposium, ISMM 2013, Uppsala, Sweden, May 27-29, 2013. Proceedings 11*, pages 111–122. Springer, 2013.

[11] Roberto Souza, Luís Tavares, Letícia Rittner, and Roberto Lotufo. An overview of max-tree principles, algorithms and applications. In *2016 29th SIBGRAPI conference on graphics, patterns and images tutorials (SIBGRAPI-T)*, pages 15–23. IEEE, 2016.

[12] Pascal Monasse and Frederic Guichard. Fast computation of a contrast-invariant image representation. *IEEE transactions on image processing*, 9(5):860–872, 2000.

[13] Vicent Caselles, Enric Meinhardt, and Pascal Monasse. Constructing the tree of shapes of an image by fusion of the trees of connected components of upper and lower level sets. *Positivity*, 12(1):55–73, 2008.

[14] Enric Meinhardt Llopis et al. Morphological and statistical techniques for the analysis of 3d images. 2011.

[15] Sébastien Crozet and Thierry Géraud. A first parallel algorithm to compute the morphological tree of shapes of nd images. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 2933–2937. IEEE, 2014.

[16] Efim Khalimsky, Ralph Kopperman, and Paul R Meyer. Computer graphics and connected topologies on finite ordered sets. *Topology and its Applications*, 36(1):1–17, 1990.

[17] Fernand Meyer. Un algorithme optimal de ligne de partage des eaux. *Actes du*, 2:847–859, 1991.

[18] Michael HF Wilkinson, Hui Gao, Wim H Hesselink, Jan-Eppo Jonker, and Arnold Meijster. Concurrent computation of attribute filters on shared memory parallel machines. *Ieee transactions on pattern analysis and machine intelligence*, 30(10):1800–1813, 2008.

[19] Nimrod Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM (JACM)*, 30(4):852–865, 1983.

[20] Ugo Moschini, Arnold Meijster, and Michael HF Wilkinson. A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images. *IEEE transactions on pattern analysis and machine intelligence*, 40(3):513–526, 2017.

[21] Philippe Salembier, Albert Oliveras, and Luis Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.