Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía y Física

Trabajo Especial

Licenciatura en Ciencias de la Computación

Watershed Topológico Concurrente

Autor: Damián Montenegro
Director: Dr. Oscar H. Bustos

Córdoba, Argentina, Mayo 2015

Resumen

La *Transformación Watershed Topológico* es un procedimiento de la morfología matemática, basado en la popular *Transformación Watershed*, cuyo fin es la segmentación de una imagen. Si bien existen algoritmos secuenciales poderosos para llevarla a cabo, en materia de concurrencia hay mucho por hacer. La paralelización toma especial importancia si se tiene en cuenta que las computadoras de uso diario poseen múltiples núcleos, lo que posibilita notables mejoras de rendimiento. Por tal motivo, este trabajo estará enfocado en el desarrollo de un algoritmo paralelo para el cálculo de la *Transformación Watershed Topológico*.

The *Topological Watershed Transformation* is a Mathematics morphology procedure, which is based on the known *Watershed Transformation*. The main aim of this procedure is the image segmentation. There are efficient sequential algorithms to work with; but in concurrent development there is much work to do. Concurrency highly improves the performance in computers for general purposes with many cores. For the reason stated before, this job is focused in the development of a parallel algorithm to perform the *Topological Watershed Transformation*.

Clasificación: I.4.6 Segmentation (Image Processing and Computer Vision)

Palabras Clave: segmentación, morfología matemática, watershed, topología, watershed topológico

Índice general

Capítulo 1

Introducción

El análisis de imágenes comprende los procesos que se utilizan con el fin de extraer información de las mismas. Uno de ellos es el de segmentación, que tiene por propósito obtener una representación de la imagen que facilite localizar los objetos que se encuentran en ella.

Entre los métodos más populares de segmentación de imágenes, se encuentra el denominado *Transformación Watershed*, que procesa una imagen en escala de grises y genera como resultado una en blanco y negro, en la cual se resaltan los contornos de las figuras que se encuentran en la imagen original. Si bien tiene muchas aplicaciones y es de gran utilidad en ciertos casos, el resultado, al ser en blanco y negro, sufre la pérdida de la información brindada por los tonos de grises de la imagen original, lo que puede dar lugar a dificultades al momento de establecer algunas propiedades de la imagen.

Este trabajo focalizará en una variante de la mencionada Transformación Watershed, denominada *Watershed Topológico*. Naturalmente, este enfoque también resalta los contornos de los objetos de la imagen, pero tiene como principal ventaja que mantiene la información de los tonos de grises. Concretamente, se harán dos implementaciones de algoritmos que calculan el Watershed Topológico de una imagen en escala de grises. La primera será una de orden cuasi-lineal, mientras que la segunda será una concurrente, con el objetivo de mejorar el tiempo de procesamiento.

1.1. Motivación y Objetivos

Este trabajo está dirigido a personas con conocimientos en programación, que deseen introducirse al campo de la Transformación Watershed Topológico. En este informe se describen los conceptos teóricos necesarios para comprender el procedimiento, junto a la correspondiente bibliografía para aquel lector que se interese en profundizar.

Como resultado final, se ofrecerá el código fuente en lenguaje *C++* co-

rrespondiente a cada implementación, de modo que el lector pueda apreciar los detalles finos de implementación, como así también hacer modificaciones para ver los efectos en la imagen resultante. Además, se ofrecerán los ejecutables de cada implementación, lo que permitirá al lector hacer pruebas con imágenes de su interés.

Se pretende que el procesamiento de una imagen se realice en un tiempo razonable, por lo que se harán análisis de rendimiento que grafiquen el comportamiento del ejecutable. Por este motivo, también está dedicado a aquellas personas que deseen aplicar la transformación a imágenes propias con fines personales.

Para cumplir con tales fines, será muy importante que el código fuente pueda ser comprendido fácilmente por terceros, como así también que sea eficiente, constituyendo una gran motivación para todo desarrollador de software, como así también una carta de presentación profesional.

1.2. Estado del Arte

Tal como se mencionó previamente, se ha propuesto como objetivo de este trabajo lograr una implementación eficiente. En este sentido, se encuentra descripto en [5] un algoritmo secuencial de orden cuasi-lineal para llevar a cabo la transformación, por lo que será una de las bibliografías fundamentales de este trabajo. Cabe mencionar en este punto, que existe una implementación en código abierto, en lenguaje *C*, que puede ser descargada de `http://perso.esiee.fr/~info/tw/index.html`.

Sin embargo, este algoritmo no saca rédito de los procesadores multi-core de la actualidad, por lo que el foco se pondrá en su paralelización. Respecto a esto, vale la pena mencionar que existe un algoritmo paralelo descripto en [8], aunque en el presente trabajo se introducirá una implementación alternativa.

1.3. Método de medición

Se compilará cada una de las implementaciones con *g++* v4.7.3 utilizando los flags *-O3 -std=c++11 -Wl,–no-as-needed -pthread*. El entorno de ejecución será el Cluster Mendieta del CCAD-UNC, que forma parte del SNCAD-MinCyT, Argentina.

Las pruebas involucrarán imágenes aleatorias con una profundidad de 256 niveles de grises, es decir, el valor de cada pixel es un número aletorio entre 0 y 255, con las siguientes resoluciones:

- 500x500 pixels
- 1000x1000 pixels

- 1500x1500 pixels
- 2000x2000 pixels
- 2500x2500 pixels
- 3000x3000 pixels
- 3500x3500 pixels

Por cada imagen, se medirá el rendimiento de la implementación cuasilineal y la implementación paralela utilizando 1, 2, 4, 8 y 16 threads.

La prueba de rendimiento involucrará el reporte de las siguientes variables:

- Promedio de memoria principal ocupada (en MB)
- Porcentaje promedio de CPU utilizado
- Porcentaje máximo de CPU utilizado
- Tiempo tomando por la transformación sin incluir preprocesamiento
- Tiempo tomando por la creación del Árbol de Componentes
- Tiempo total

Para el cálculo de las primeras tres variables, se utilizará el comando de linux *ps* (*Process Snapshot*), verificando cada 100 milisegundos el valor correspondiente a ambas variables, para luego reportar el promedio de todos los registros. En este sentido, cabe aclarar que este procedimiento se automatizará a partir de un script en lenguaje *Python* utilizando el módulo *psutil*.

Tal como se verá en el Capítulo 3, para llevar a cabo la transformación se requiere, como paso previo, la creación de una estructura de datos llamada *Árbol de Componentes*. En este trabajo, el algoritmo de paralelización presentado no incluye la creación concurrente de esta estructura de datos, por tal motivo, se reporta el tiempo tomando por la transformación, el tomado para crear el árbol y el total.

1.4. Estructura del Trabajo

En esta sección se pretende indicar cómo está constituído este informe para facilitar la lectura y comprensión. En el mismo, se encuentran, además del actual, cuatro capítulos:

- **Capítulo 2 (Watershed Topológico)**: aquí se presenta la definición de Watershed Topológico. En base a la misma se propone un algoritmo fuerza bruta con el objeto de ayudar al lector a comprender la transformación.
- **Capítulo 3 (Watershed Topológico Cuasi-Lineal)**: en este capítulo se presenta el algoritmo de orden cuasi-lineal descripto en [5] y se exponen los resultados de las pruebas de rendimiento pertinentes.
- **Capítulo 4 (Watershed Topológico en Paralelo)**: en este apartado se propone una versión concurrente del algoritmo expuesto en el capítulo 3, analizando la complejidad que esto supone y, finalmente, detallando los resultados de las pruebas de eficiencia correspondientes.
- **Capítulo 5 (Conclusión)**: se muestra la apreciación final y los resultados alcanzados de este trabajo.

Además, en este informe se incluyen dos anexos. El primero, Transformación Watershed, muestra la definición de la transformación. A quienes no la conozcan, es recomendado empezar por aquí, porque, tal como se ha mencionado previamente, la transformación Watershed Topológica es una variante de esta. El segundo, incluye algunas nociones de grafos necesarias para comprender el contenido aquí desarrollado.

Capítulo 2

Watershed Topológico

En este capítulo, se introducirá una variante de Transformación Watershed propuesta por M. Couprie y G. Bertrand [2], denominada Watershed Topológico, que hace foco en la preservación de ciertas características topológicas de la imagen. A continuación se mencionará la definición formal junto a una breve reseña de las propiedades de esta transformación y se propondrá un algoritmo fuerza-bruta que permite calcularla.

2.1. Definiciones Preliminares

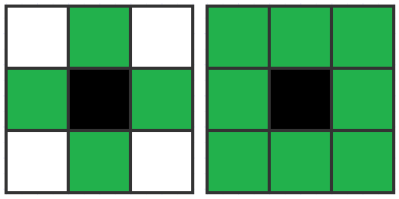
La forma más sencilla de representar una imagen digital en escala de grises, es la denominada *representación matricial*, que consiste en exhibir la imagen como una matriz numérica, donde cada celda corresponde a un pixel. Un ejemplo de ello puede observarse en la Figura 2.1.

2	3	2
2	3	2
2	3	1

Figura 2.1: Imagen digital en escala de grises

Se hace referencia a los píxeles de acuerdo a las coordenadas fila-columna que este ocupa en el arreglo. De esta manera, se habla del pixel (x, y), para referirse al ubicado en la fila x y columna y de la matriz. De este modo, se ve que el pixel $(1, 0)$ de la imagen de la Figura 2.1 tiene nivel de gris 2.

En el ambiente del procesamiento de imágenes, suele ser útil establecer relaciones entre píxeles, conocidas como *relaciones de vecindad*. Las dos más populares son las conocidas como *relación de vecindad tipo 4* y *relación de vecindad tipo 8*. Para un pixel (x, y), la primera relación está constituída



(a) (b)

Figura 2.2: Relación de vecindad entre pixeles. (a) Relación de vecindad tipo 4 (b) Relación de vecindad tipo 8

por los píxeles $(x-1,y),(x+1,y),(x,y-1),(x,y+1)$, mientras que la restante, por los píxeles $(x-1,y-1),(x-1,y),(x-1,y+1)$, $(x,y-1),(x,y+1),(x+1,y-1),(x+1,y),(x+1,y+1)$. Ambas son descriptas en la Figura 2.2

Una representación de imagen digital alternativa, adecuada para expresar las relaciones de vecindad, es la conocida como *Grilla Digital*, que consiste en exhibir la imagen como un grafo con pesos conectado y simétrico (E,Γ,F) [1], donde $E \subseteq \mathbb{Z}^2$ es el conjunto de todos los píxeles de la imagen, definidos por sus coordenadas fila - columna, Γ describe la relación de vecindad entre ellos y $F : E \to \mathbb{Z}$, de modo que para cada pixel $p \in E$, $F(p)$ indica el nivel de gris de p. En la Figura 2.3 podemos observar esta representación para la imagen de la Figura 2.1, tomando una relación de vecindad tipo 4.

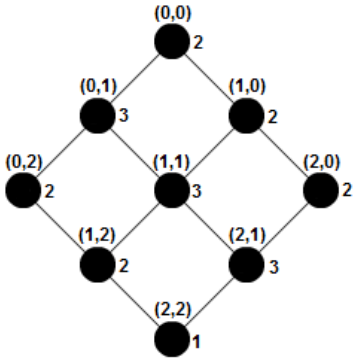


Figura 2.3: Grilla Digital de la imagen de la Figura 2.1 para relación de vecindad tipo 4. Arriba de cada nodo se observa su nombre y al costado su peso

A continuación se presentan las definiciones que serán utilizadas en el desarrollo del presente capítulo.

Definición: Sea (E,Γ,F) una grilla digital, entonces llamaremos *k-sección superior*, o *sección superior de nivel k*, al conjunto de nodos cuyo

[1] Para repasar nociones de grafos ver Anexo B. Grafos

peso es mayor o igual a k. Formalmente,

$$F_k = \{x \in E; F(x) \geq k\}$$

En la Figura 2.4 podemos observar, resaltados en verde, los nodos que conforman la F_2 de la grilla digital de la Figura 2.3.

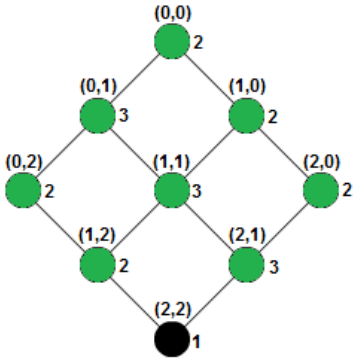


Figura 2.4: F_2 de la grilla digital de la Figura 2.3

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos *k-sección inferior*, o *sección inferior de nivel k*, al conjunto de nodos cuyo peso sea menor a k. Formalmente,

$$\overline{F_k} = \{x \in E; F(x) < k\}$$

En la Figura 2.5 podemos observar, resaltados en verde, los nodos que constituyen la $\overline{F_3}$ de la grilla digital de la Figura 2.3.

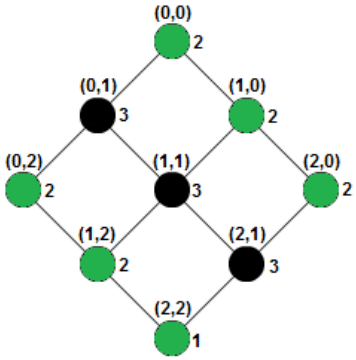


Figura 2.5: $\overline{F_3}$ de la grilla digital

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos *k-componente superior*, o *componente superior de nivel k*, a una componente conectada de F_k.

Por ejemplo, a partir de F_2, descripta en la Figura 2.4, podemos concluir que esta grilla digital contiene una sola 2-componente superior, compuesta por todos los nodos de F_2.

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos *k-componente inferior*, o *componente inferior de nivel k*, a una componente conectada de $\overline{F_k}$.

Para ilustrar esta definición, podemos tomar $\overline{F_3}$ de la Figura 2.5, veremos que hay dos 3-componente inferior, una compuesta por los nodos $(0,0),(1,0),(2,0)$ y la otra por $(0,2),(1,2),(2,2)$.

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos *mínimo regional de F* a una k-componente inferior que no contenga una $(k-1)$-componente inferior.

Para el caso de la grilla de la Figura 2.3, si calculáramos todas las componentes inferiores, concluiríamos que existen dos mínimos regionales, uno compuesto por los nodos $(0,0),(1,0),(2,0)$ y la otra por el nodo $(2,2)$, lo cual concuerda con la noción intuitiva de mínimos regionales dada en el Anexo A.

2.2. Definición de Watershed Topológico

La definición formal de Watershed Topológico de una imagen, se basa en el concepto de nodo W-simple. Sea (E, Γ, F) una grilla digital y $X \subset E$ no vacío, entonces un nodo $x \in X$ es *W-simple* para X si es adyacente a exactamente una componente conectada de $\overline{X}$.

Como ejemplo, se puede tomar la grilla de la Figura 2.3 y tomar $X = F_3 = \{(0,1),(1,1),(2,1)\}$. Tal como se ha mencionado previamente y como puede verse fácilmente en la Figura 2.5, $\overline{F_3}$ contiene dos componentes conectadas, una compuesta por los nodos $(0,0),(1,0),(2,0)$ y la otra por $(0,2),(1,2),(2,2)$, por lo tanto, ninguno de los nodos de F_3 es W-simple, pues todos ellos son adyancentes a las dos componentes conectadas de $\overline{F_3}$.

Ahora bien, sea (E, Γ, F) una grilla digital, $p \in E$ y $k = F(p)$, se dice que el nodo p es *W-destructible* para F si es W-simple para F_k. Decimos que $G : E \to \mathbb{Z}$ es *Watershed Topológico* de F, si G se obtiene a partir de F disminuyendo sucesivamente el valor de puntos W-destructible en uno hasta lograr estabilidad, es decir, que no queden puntos W-destructible. En la nomenclatura de Transformaciones Watershed, las *Catchment Basins* de G serán los mínimos regionales de G y el resto de los puntos serán las *Watershed Lines*.

Continuando con la grilla de la Figura 2.3, es sencillo ver que el nodo $(1,2)$ es W-destructible, pues $F((1,2)) = 2$ y $F_2 = E - (2,2)$, al advertir que F_2 constituye una componente conectada, queda claro que $\overline{F_2} = (2,2)$

es adyacente a tal componente conectada y, por ende, es W-destructible. Un análisis similar muestra que el nodo $(0, 2)$ también es W-destructible. El Watershed Topológico de la Figura 2.1 puede verse entonces en la Figura 2.6.

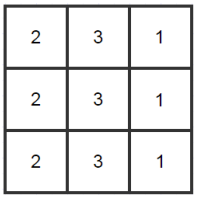


Figura 2.6: Watershed Topologico de la Imagen de la Figura 2.1

2.3. Bondades

Si bien la Transformación Watershed de imágenes en escala de grises constituye una herramienta de gran utilidad y un paso muy importante en la segmentación de imágenes, la mayoría de los enfoques existentes, tal como se menciona en [4], tienen algunas desventajas. En primera instancia, el resultado es una imagen binaria, lo que implica que se pierde la información brindada por los tonos de grises de la imagen original, lo que limita su utilización para procesamiento adicional. Además, muchos de ellos, como el basado en el paradigma de inundación, producen Watershed Lines que no necesariamente se encuentran en los contornos más significativos de la imagen original. Esto significa que pueden presentarse serias dificultades al intentar obtener conclusiones a partir de la imagen resultante.

En contrapartida, la transformación presentada en este capítulo preserva la conectividad de cada $\overline{F_k}$, es decir, la cantidad de componentes conectadas es la misma tanto en la imagen original como en su transformación, en cada $\overline{F_k}$. Esto constituye una gran virtud, y las implicaciones de la misma se encuentran detalladas en [3].

2.4. Implementación Fuerza Bruta

En este apartado se presenta un algoritmo para llevar a cabo la Transformación Watershed Topológico de una imagen. La idea es recorrer la grilla nodo por nodo, verificando si cada uno es W-destructible o no. En caso afirmativo se disminuye su nivel de gris en 1, caso contrario su valor no cambia. Una vez escaneada la imagen en su totalidad, si se encontró algún nodo W-destructible, se repite todo el proceso. Caso contrario, el algoritmo termina.

Es sencillo ver que la dificultad del algoritmo se encuentra en cómo determinar si un nodo es W-destructible o no, asi que se comienza por estudiar

esta situación. Sea (E, Γ, F) una grilla y sea $p \in E$ y $k = F(p)$, una observación clave es que p será W-destructible si y sólo si cada uno de los $q \in \Gamma^-(p)$, donde $\Gamma^-(p) = \{n \in \Gamma(p) : F(n) < F(p)\}$, pertenecen a la misma componente conectada en $\overline{F_k}$. A partir de esto, se pueden distinguir dos casos triviales, uno es aquel en el que $\Gamma^-(p) = \emptyset$, en este caso p no será W-destructible puesto que no será adyacente a ninguna componente conectada en $\overline{F_k}$. El segundo, es la situación en la que $\Gamma^-(p)$ está compuesto por sólo un $q \in E$, en este caso p será W-destructible puesto que es adyacente sólo a la componente conectada a la que pertenece q en $\overline{F_k}$.

Con esto en mente, sólo queda por resolver el caso en el que p tenga más de un vecino con nivel de gris inferior a k. La idea es calcular cada uno de los nodos alcanzables a partir de algún $q \in \Gamma^-(p)$ en $\overline{F_k}$, y luego chequear si $\Gamma^-(p)$ está contenido en dicho conjunto.

A continuación se presenta el pseudo-código para determinar si un $p \in E$ es W-destructible o no.

Algorithm 1

```
1:  function isWdestructible(E, Γ, F, p)
2:      if Γ⁻(p) = ∅ then return false
3:      if Γ⁻(p).size = 1 then return true
4:      reachableNodes ← ∅
5:      index ← 0
6:      reachableNodes.insert(Γ⁻(p)[0])
7:      while ¬(reachableNodes.containAll(Γ⁻(p))) ∧
    index<reachableNodes.size do
8:          q ← reachableNodes[index]
9:          for all n ∈ Γ(q) do
10:             if F(n) < F(p) then
11:                 reachableNodes.insert(n)
12:         index ← index + 1
        return index<reachableNodes.size
```

Se recibe por parámetro un grafo con pesos y un $p \in E$, el resultado será un valor booleano indicando si p es W-destructible o no. Las líneas 2 y 3 reflejan los casos triviales mencionados anteriormente. De la línea 4 a la 6 se inicializan dos variables locales, entre ellas reachableNodes, un set (por definición no admite valores duplicados) con sólo un elemento, algún $q \in \Gamma^-(p)$, que se llamará *semilla*. A partir de la línea 7 comienza un loop que insertará en reachableNodes aquellos nodos alcanzables a partir de la semilla en $\overline{F_k}$. En la primera iteración se insertarán aquellos $n \in \Gamma(q)$ tales que $F(n) < F(p)$, es decir, $n \in \overline{F_k}$. La siguiente iteración tomará el siguiente elemento del set y se repetirá el proceso. Se iterará hasta que, o bien $\Gamma^-(p) \subseteq reachableNodes$, esto significa que todos los $q \in \Gamma^-(p)$ pertencen

a la misma componente conectada de $\overline{F_k}$, o bien se haya calculado todos los $t \in E$ alcanzables a partir de la semilla en $\overline{F_k}$, con lo que concluimos los contrario, y por lo tanto p no es W-destructible.

Es interesante analizar la complejidad del algoritmo. Para ello, hay que tener en cuenta que el peor de los casos es aquel en el que todos los $n \in E \ /p$ sean alcanzables a partir de la semilla y el último en incorporarse a reachableNodes sea algún $q \in \Gamma^-(p)$. Sea N el tamaño de E y k, el de la vecindad. En cada iteración, se ejecutará k veces la función $insert()$, aunque se insertarán $k-1$ nodos puesto que algún vecino ya pertenecerá a reachableNodes. En virtud de esta observación, tendremos que el bucle se ejecutará $N/(k-1)$ veces, involucrando en cada una de ellas la llamada a $containsAll()$. A su vez, se ejecutará el método $insert()$ k veces en cada iteración, con lo cual se concluye que la complejidad del algoritmo está dada por $(N/(k-1)) * \Theta(containsAll()) + (N/(k-1)) * k * \Theta(insert())$. Naturalmente, k será muy pequeño en relación a N, por lo cual se puede reducir el resultado anterior a $N * \Theta(containsAll()) + N * \Theta(insert())$. Si se asume que las operaciones $containsAll()$ y $insert()$ son lineales respecto al tamaño del conjunto, se puede concluir que el orden del algoritmo es $\Theta(n^2)$

Para llevar a cabo la transformación, sólo es necesaria una rutina que itere cada uno de los $p \in E$ en busca de puntos W-destructubles, disminuya su valor si corresponde, y repita el proceso hasta lograr estabilidad.

A continuación se presenta el pseudo-código del algoritmo que lleva a cabo la transformación.

Algorithm 2

```
1: procedure doTopologicalWatershed(E, Γ, F)
2:     repeat
3:         someChange ← false
4:         for all p ∈ E do
5:             if isWdestructible(E, Γ, F, p) then
6:                 F(p) ← F(p) − 1
7:                 someChange ← true
8:     until ¬someChange
```

Nuevamente, vale la pena analizar la complejidad de este algoritmo. El peor caso, seria aquel en el que hay sólo un mínimo regional constituido por sólo un nodo, cuyo nivel de gris es el mínimo, denotado min, y todos los demas nodos poseen el máximo nivel de gris, denotado max. Es sencillo ver que el orden bucle interno es $\Theta(n^3)$. El bucle externo se ejecutará $max - min$ veces. En la práctica, la cantidad de nodos supera ampliamente el valor $max - min$ por lo tanto, se puede concluir que el orden del algoritmo es $\Theta(n^3)$.

Capítulo 3

Watershed Topológico Cuasi-Lineal

En [5] se propone un algoritmo de orden cuasi-lineal para el cálculo del Watershed Topológio de una imagen, más precisamente, si n denota la cantidad de vértices de una grilla digital y m, la de arcos, entonces la complejidad es $\Theta(n + m)$. Se trata de una notable mejora respecto del algoritmo fuerza-bruta descripto en el capítulo anterior, lograda a partir de dos factores. Por un lado, los pixeles se procesan siguiendo un orden de modo de asegurar que su valor se reduzca, a lo sumo, sólo una vez durante toda la ejecución. Por el otro, se disminuye el valor de un pixel W-destructible en el mínimo valor posible, en lugar de disminuirlo sólo en 1. Esto se logra gracias a la utilización de una estructura de datos llamada "Árbol de Componentes", que será presentada a continuación.

En este capítulo, se repasará en primer lugar algunas definiciones básicas, luego se introducirá el concepto de Árbol de Componentes de una imagen, para después explicar el rol del mismo en el cálculo del Watershed Topológico. Además, se dará un pseudo-código junto a una explicación línea a línea, a fin de describir el procedimiento detalladamente. Finalmente, se expondrán los resultados de las pruebas de eficiencia aplicadas sobre la implementación del procedimiento presentado.

3.1. Definiciones preliminares

Sea (E, Γ, F) una grilla digital, se denota con $C_k(F)$ el conjunto de todas las k-componentes superiores de F y con $C(F)$, el conjunto de todas las componentes superiores de F, es decir

$$C(F) = \bigcup_{k=0}^{\infty} C_k(F)$$

Análogamente se define $C_k(\overline{F})$ y $C(\overline{F})$.

La imagen que se utilizará en este capítulo para ejemplificar es la que se muestra en la Figura 3.1. A lo largo de este capítulo, se llamará (E_1, Γ_1, F_1) a la grilla digital de esta imagen, donde Γ_1 se corresponde con una relación de vecindad tipo 4. Su gráfico no se presenta porque sería engorroso y poco práctico al presentar 20 nodos.

4	3	4	1	4
4	2	4	2	4
4	1	4	2	4
4	3	4	2	4

Figura 3.1: Imagen en escala de grises

A continuación se presenta el cálculo de $C(F_1)$ y $C(\overline{F_1})$, que servirá para ejemplificar los conceptos presentados en el actual capítulo.

- $C_4(F) = \{CS_a, CS_b, CS_c\}$ donde
  - $CS_a = \{(0,0), (1,0), (2,0), (3,0)\}$
  - $CS_b = \{(0,2), (1,2), (2,2), (3,2)\}$
  - $CS_c = \{(0,4), (1,4), (2,4), (3,4)\}$
- $C_3(F) = \{CS_d, CS_e\}$ donde
  - $CS_d = CS_a \bigcup CS_b \bigcup \{(0,1), (3,1)\}$
  - $CS_e = CS_c$
- $C_2(F) = \{CS_f\}$ donde
  - $CS_f = CS_d \bigcup CS_e \bigcup \{(1,1), (1,3), (2,3), (3,3)\}$
- $C_1(F) = \{CS_g\}$ donde
  - $CS_g = E$
- $C_2(\overline{F}) = \{CI_a, CI_b\}$ donde
  - $CI_a = \{(0,3)$
  - $CI_b = \{(2,1)\}$
- $C_3(\overline{F}) = \{CI_c, CI_d\}$ donde
  - $CI_c = CI_a \bigcup \{(1,3), (2,3), (3,3)\}$
  - $CI_d = CI_b \bigcup \{(1,1)\}$

- $C_4(\overline{F}) = \{CI_e, CI_f\}$ donde
  - $CI_e = CI_c$
  - $CI_d = CI_d \bigcup \{(0,1),(3,1)\}$
- $C_5(\overline{F}) = \{CI_g\}$ donde
  - $CI_g = E$

Finalmente, $C(F) = C_1(F) \bigcup C_2(F) \bigcup C_3(F) \bigcup C_4(F)$ y
$C(\overline{F}) = C_2(\overline{F}) \bigcup C_3(\overline{F}) \bigcup C_4(\overline{F}) \bigcup C_5(\overline{F})$

3.2. Árbol de Componentes

Se piensa una imagen como un relieve topográfico completamente sumergido en agua, al que se le hacen perforaciones en sus mínimos. El nivel de agua disminuirá lentamente, con lo que se comenzará a observar islas (correspondientes a los máximos de la imagen), estas conformarán las hojas del árbol de componentes de la imagen. A medida que el nivel de agua continue descendiendo, las islas crecerán en tamaño, conformando lo que serán las ramas del árbol. Luego, a partir de algún nivel, algunas islas se fusionarán conformando una sola pieza, estas piezas serán las bifurcaciones del árbol. El proceso terminará cuando toda el agua haya desaparecido. En la Figura 3.2 se puede observar el proceso descripto.

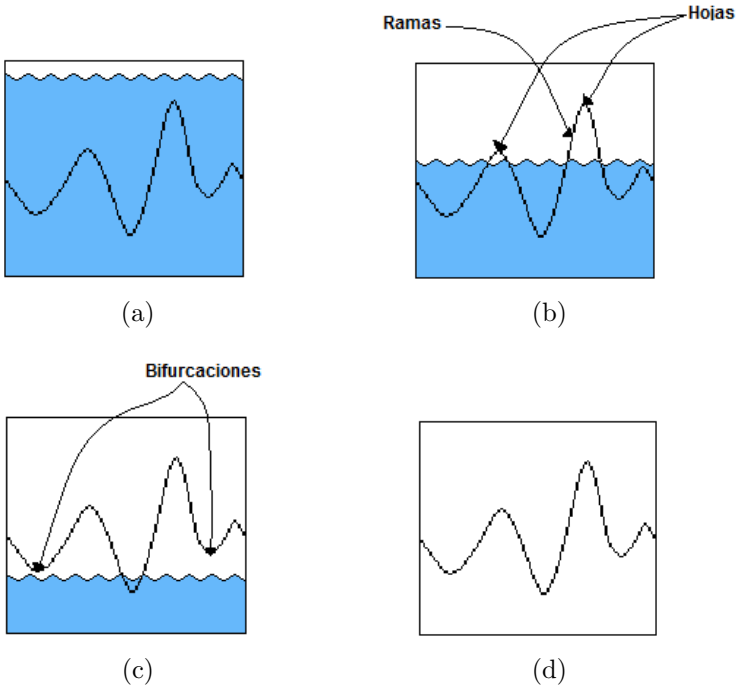


Figura 3.2: Árbol de componentes. (a) Inicio (b) y (c) Imagen parcialmente inundada (d) Fin

Lo que se acaba de ver es una definición intuitiva de Árbol de Componentes de una imagen, de modo de facilitar la comprensión del lector. A continuación la definición formal.

Sea (E, Γ, F) una grilla digital, entonces definimos el *Árbol de Componentes* de F, $T(F)$, como un árbol cuyos nodos serán elementos de $C(F)$ y habrá una arista de $c' \in C_k(F)$ hacia $c \in C_j(F)$ si $j = k+1$ y $c \subseteq c'$. En este caso, decimos que c' es *padre* de c, y también que c es $hijo$ de c'.

El *mapa de componentes* será un mapa $\Psi : E \to C(F)$, que relaciona cada $p \in E$ con el elemento de $C(F)$ que lo contiene.

Algo muy importante acerca de esta estructura de datos, es que existe un algoritmo de orden cuasi-lineal para su cálculo, se puede consultar los detalles de implementación y análisis de complejidad en [6].

En la Figura 3.3 vemos el Árbol de componentes correspondiente a la grilla digital de la imagen de la Figura 3.1

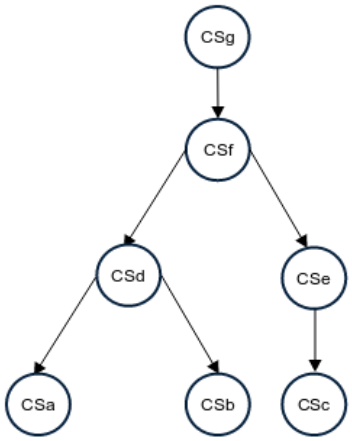


Figura 3.3: $T(F_1)$

A continuación se presenta la definición de $highestfork$ del árbol, el cual, como se verá más adelante, es fundamental para el cálculo de Watershed Topológico de una imagen, pero antes será necesario introducir algunas definiciones. Para hacer más sencilla la lectura, vamos a denotar un nodo de $T(F)$ como un par $[k, c]$ cuando $c \in C_k(F)$ y llamaremos nivel del nodo al número k.

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k_0, c_0], [k_n, c_n] \in C(F)$, entonces $[k_0, c_0]$ será un *ancestro* de $[k_n, c_n]$ si existe una sucesión nodos $[k_1, c_1], \ldots, [k_{n-1}, c_{n-1}]$ tales que $\forall i \in [0, n]$, $[k_i, c_i]$ es padre de $[k_{i+1}, c_{i+1}]$. En tal caso, se dice también que $[k_0, c_0]$ está *encima* de $[k_n, c_n]$, y que $[k_n, c_n]$ está $abajo$ de $[k_0, c_0]$. Por ejemplo, podemos ver en la Figura 3.3 que CS_f es ancestro de CS_a

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k, c], [k_1, c_1], [k_2, c_2] \in C(F)$, entonces se dice que $[k, c]$ es *ancestro común* de $[k_1, c_1]$,$[k_2, c_2]$, si $[k, c]$ es ancestro de $[k_1, c_1]$ y $[k, c]$ es ancestro de $[k_2, c_2]$. Se puede ver en la Figura 3.3 que CS_f es ancestro común de CS_a y CS_b.

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k, c], [k_1, c_1], [k_2, c_2] \in C(F)$, entonces se dice que $[k, c]$ es *menor ancestro común* de $[k_1, c_1], [k_2, c_2]$ si $[k, c]$ es ancestro común de ambos y no hay ningún otro ancestro común abajo de $[k, c]$. En la Figura 3.3 se aprecia que CS_f es el menor ancestro común de CS_a y CS_c.

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k, c], [k_1, c_1], [k_2, c_2] \in C(F)$, entonces se dice que $[k, c]$ es *menor ancestro común propio* de $[k_1, c_1]$, $[k_2, c_2]$, si $[k, c]$ es ancestro común de ambos y $[k, c]$ es distinto de $[k_1, c_1]$

y $[k, c]$ es distinto de $[k_2, c_2]$. Es sencillo ver que CS_f es el menor ancestro común propio de CS_a y CS_c.

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k_1, c_1], [k_2, c_2] \in C(F)$, entonces se dice que $[k_1, c_1]$,$[k_2, c_2]$ están *separados* si tienen menor ancestro común propio, caso contrario decimos que están *linkeados*. Por ejemplo, en la Figura 3.3 se puede ver que CS_a y CS_b están separados, mientras que CS_a y CS_f están linkeados.

Definición: Sea (E, Γ, F) una grilla digital, y sea $[k, n][k_1, c_1], [k_2, c_2], \ldots, [k_n, c_n] \in C(F)$, entonces se dice que $[k, c]$ es *highest fork* de $\{[k_1, c_1], [k_2, c_2], \ldots, [k_n, c_n]\}$ si las siguientes condiciones se satisfacen:

- Si dos nodos $[k_i, c_i]$,$[k_j, c_j]$ están separados, entonces el menor ancestro común tiene altura $\leq k$.
- Existen nodos $[k_i, c_i]$,$[k_j, c_j]$ separados, tal que $[k, c]$ es el menor ancestro común propio de $[k_i, c_i]$,$[k_j, c_j]$.

En la Figura 3.3, se puede ver que el conjunto $\{CS_a, CS_d, CS_f\}$ no tiene highest fork, pues no existen nodos separados. Sin embargo CS_d es el highest fork de $\{CS_a, CS_b, CS_d\}$, pues CS_a y CS_b están separados. Es importante notar que el highest fork de un conjunto puede no pertenecer a ese conjunto, por ejemplo si se toma $\{CS_a, CS_b\}$ su highest fork es CS_d.

3.3. Algoritmo

En este apartado se explicará cómo utilizar el Árbol de Componentes en el cálculo de la transformación.

Sea (E, Γ, F) una grilla digital, para calcular el Watershed Topológico de F vamos a calcular $T(\overline{F})$. En este caso, habrá una arista de $c' \in C_k(\overline{F})$ hacia la componente $c \in C_j(\overline{F})$ si $j = k + 1$ y $c' \subseteq c$.

En la Figura 3.4, se puede ver $T(\overline{F_1})$.

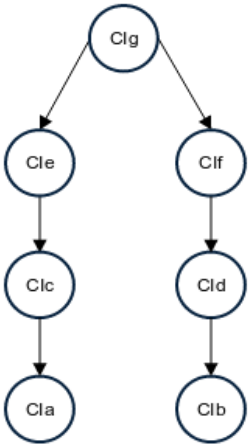


Figura 3.4: $T(\overline{F_1})$

A continuación se menciona un teorema que explica el rol del Árbol de Componentes en el cálculo de la transformación, cuya demostración se encuentra en [5].

Teorema: Sea (E, Γ, F) una grilla digital, $p \in E$. Denotamos como $V(p) = \{[k, C(q)] : q \in \Gamma^-(p)\}$, donde $k = F(q) + 1$ y $C(q)$ será la componente $c \in C_k(\overline{F})$ que contiene a q.

- Si V(p) $\neq \emptyset$ y $V(p)$ no tiene highest fork en $T(\overline{F})$, entonces p será W-destructible para F para todos los valores $k : w \leq k \leq F(p)$, y no será W-destructible para F para valor $w - 1$, donde w es el mínimo nivel de $V(p)$.

- Si $V(p) \neq \emptyset$ y $V(p)$ tiene highest fork en $T(\overline{F})$ cuyo nivel es $w \leq F(p)$, entonces p será W-destructible para F para todos los valores $k : w \leq k \leq F(p)$, y no será W-destructible para F para $w - 1$.

Este resultado es de gran importancia, pues permite reducir las iteraciones del algoritmo fuerza bruta en gran medida. Claro, en lugar de disminuir cada punto W-destructible en 1 y luego iterar toda la imagen nuevamente, en cada iteración se logran avances sustanciales para eliminar los puntos W-destructibles de la imagen.

Tal como se mencionó en la reseña del presente capítulo, existe una forma de asegurar que el valor de cada punto se disminuya, a lo sumo, sólo una vez. Es una técnica muy sencilla, y consiste disminuir en primero aquellos puntos W-destructibles cuyo valor resultante (determinado por el resultado anterior) sea menor.

3.4. Implementación

Una observación clave, es que al disminuir el valor de un $p \in E$ de acuerdo al resultado visto anteriormente, $\Psi(p)$ pasará a ser o bien el mínimo de V(p), o bien el highest fork de V(p), de acuerdo a si existe el highest fork o no. Esto nos permitirá mantener actualizado el mapa de componentes a medida que se disminuyen los valores de cada uno de los puntos W-destructibles.

En base a esta observación, se define la función *isWdestructible* de modo que retorne un nodo. De manera que, si la función retorna $[k, c]$ para un $p \in E$, entonces el valor de p debe disminuirse a $k - 1$, y una vez hecho esto $\Psi(p) = [k, c]$.

A continuación se muestra el pseudo-código de esta función.

Algorithm 3

```
1: function isWdestructible(E, F, Γ, p, T(F̄), Ψ)
2:     V ← ∅
3:     for all q ∈ Γ⁻(p) do
4:         V.insert(Ψ(q))
5:     if V = ∅ then return ∅
6:     [k, c] ← highestFork(T(F̄), V)
7:     if [k, c] = ∅ then return minLevel(V)
8:     if k ≤ F(p) then return [k, c]
       return ∅
```

Es importante mencionar que en [5], se establece cómo calcular el highest fork de un set $s \subseteq C(\overline{F})$ a partir del cálculo del menor ancestro común de dos elementos, para lo cual existe un algoritmo sencillo y eficiente detallado en [7].

Ahora se detallará el algoritmo que lleva a cabo la transformación. Se utilizará un mapa : $E \to C(\overline{F})$, que asocia cada $p \in E$ con el nodo retornado por la función *isWdestructible*. Para procesar cada uno de los $p \in E$ en el orden conveniente, lo único que se necesitará es que ese mapa este ordenado de modo creciente con respecto al nivel del nodo.

A continuación, se explica la idea del algoritmo. Lo primero que debe hacer es llenar el mapa con aquellos $p \in E$ W-Destructibles. Luego, debe iterarlo y extraer cada uno de los elementos. En las sucesivas iteraciones, se obtiene algún $p \in E$, se disminuye su valor y se actualiza el mapa de componentes. Puesto que el valor de p ha cambiado, hay que re-calcular el estado de cada uno de los vecinos de p, para lo que se volverá a utilizar la función *isWdestructible*, y se actualizará el mapa en consecuencia.

A continuación se da el pseudo-código que lleva a cabo la transformación de acuerdo al proceso descripto.

Algorithm 4

```
1: procedure doTopologicalWatershed(E, Γ, F, T(F̄), Ψ)
2:     componentMap ← ∅
3:     for all p ∈ E do
4:         [i, c] ← isWdestructible(E, F, Γ, p, T(F̄), Ψ)
5:         if ¬[i, c] = ∅ then
6:             componentMap.insert(p, [i, c])                ▷ sorted by i
7:     while ¬componentMap.empty() do
8:         p, [i, c] ← componentMap.extractFirst()
9:         F(p) ← i − 1
10:        Ψ(p) ← [i, c]
11:        for all q ∈ Γ(p), F(q) ≥ i do
12:            [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
13:            if [j, d] = ∅ then
14:                componentMap.removeIfExists(q)
15:            else
16:                componentMap.addOrUpdate(q, [j, d])
```

3.5. Medición y Desempeño

Se llevó a cabo la implementación del algoritmo descripto y se realizaron las pruebas de rendimiento mencionadas en la introducción de este trabajo. En las siguientes tablas, se muestran los resultados obtenidos, donde la primera columna muestra el tiempo total de procesamiento, la segunda y la tercera, el porcentaje del mismo tomado por la transformación y la creación del árbol de componentes resp., la cuarta y la quinta, el consumo promedio y máximo de CPU durante toda la ejecución resp. y la última, la cantidad de memoria principal promedio ocupada, expresada en MB.

Ejecución con imágenes aleatorias tomando una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.8	54.5	34.1	91.3	109.5	73.6
random1000x1000	7.9	57.5	36.9	97.8	109.5	298.7
random1500x1500	19.2	56.6	36.9	97.3	109.5	673.4
random2000x2000	34.8	57.4	38.0	99.2	109.5	1200.2
random2500x2500	56.4	58.1	37.9	99.4	109.6	1902.4
random3000x3000	87.1	60.6	35.6	99.5	109.5	2729.9
random3500x3500	129.1	61.0	34.2	98.3	109.5	3788.1

Ejecución con imágenes aleatorias tomando una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	3.0	67.2	22.5	91.9	109.5	68.5
random1000x1000	11.8	65.6	31.0	98.5	109.5	270.3
random1500x1500	26.9	63.7	32.8	98.7	109.5	603.1
random2000x2000	47.6	62.3	34.7	99.4	109.5	1069.8
random2500x2500	76.1	61.9	35.2	99.6	109.5	1681.0
random3000x3000	110.8	62.2	35.0	99.4	109.7	2413.1
random3500x3500	156.3	62.4	35.0	99.6	109.5	3317.3

Es interesante hacer las pruebas de rendimiento sobre imágenes reales, sin embargo, la tarea de seleccionar imágenes no se trivial, puesto que, el comportamiento del algoritmo depende de la cantidad de píxeles W-destructibles que posea, con lo cual la comparación entre imágenes distintas puede ser engañoza. Más aún, debido al inmenso campo de acción de la segmentación de imágenes, sería muy complejo escoger un conjunto de imágenes representativo que provean resultados confiables. Más allá de esto, a modo ilustrativo, se decidió hacer las pruebas sobre una imgen satelital ontenida por el *Landsat 8*, correspondiente a la región cuya latitud es -27.4°y longitud, -69.2°. Tal ubicación corresponde a la Cordillera de los Andes en el norte Argentino. Concretamente, se cortó, en torno a la region central, de acuerdo a las resoluciones requeridas (notar que la imagen de 500x500 píxeles estará contenida en la de 1000x1000 píxeles, y esta última, en la de 1500x1500 píxeles y así sucesivamente). Los resultados se muestran a continuación.

Ejecución con imagen satelital tomando una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
real500x500	1.9	61.7	20.9	87.0	109.5	51.3
real1000x1000	9.4	69.0	23.6	94.8	109.5	213.6
real1500x1500	23.6	71.9	22.3	96.9	109.5	477.7
real2000x2000	49.1	76.2	20.3	98.5	109.5	870.4
real2500x2500	98.9	79.3	18.4	99.6	109.5	1371.8
real3000x3000	177.4	80.9	16.8	99.4	109.5	1962.0
real3500x3500	318.9	83.5	14.7	99.6	109.8	2694.9

Ejecución con imagen satelital tomando una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
real500x500	3.1	79.1	14.1	94.7	109.5	54.5
real1000x1000	14.0	77.1	19.9	98.3	109.5	214.1
real1500x1500	29.3	73.6	24.0	99.4	109.5	486.2
real2000x2000	56.3	74.5	23.3	99.6	109.5	862.6
real2500x2500	101.4	74.2	23.5	99.6	109.5	1346.0
real3000x3000	164.2	72.6	25.3	99.6	109.5	1927.5
real3500x3500	272.9	73.2	25.0	99.8	109.5	2651.7

El siguiente gráfico muestra el tiempo transcurrido para llevar a cabo la transformación, sin incluir el armado del árbol de componentes, en función del tamaño de la imagen.

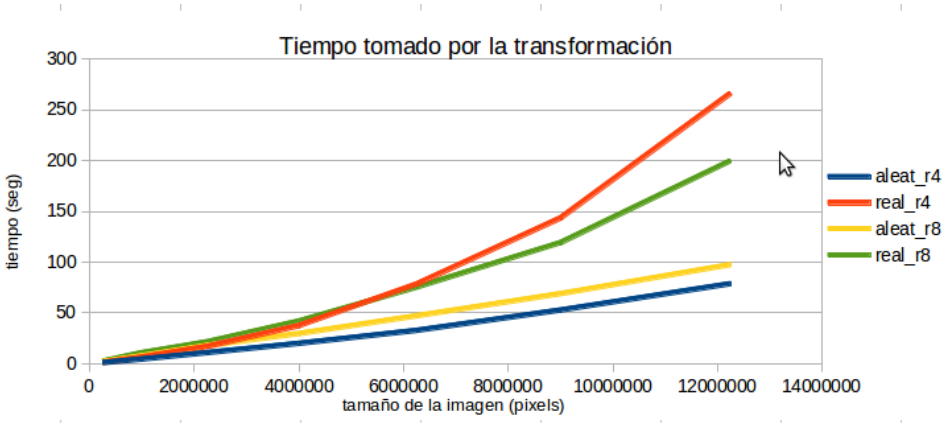


Figura 3.5: Tiempo tomado para llevar a cabo la transformación en función del tamaño de imagen

Se puede observar que los resultados obtenidos con imágenes aleatorias confirman que el orden del algoritmo es cuasi-lineal. Sin embargo, no se puede decir lo mismo para el caso de las imágenes satelitales, lo que refleja lo establecido previamente respecto de la influencia de la cantidad de píxeles W-destructibles que posea la imagen.

El siguiente gráfico muestra el consumo de memoria principal promedio en función del tamaño de la imagen.

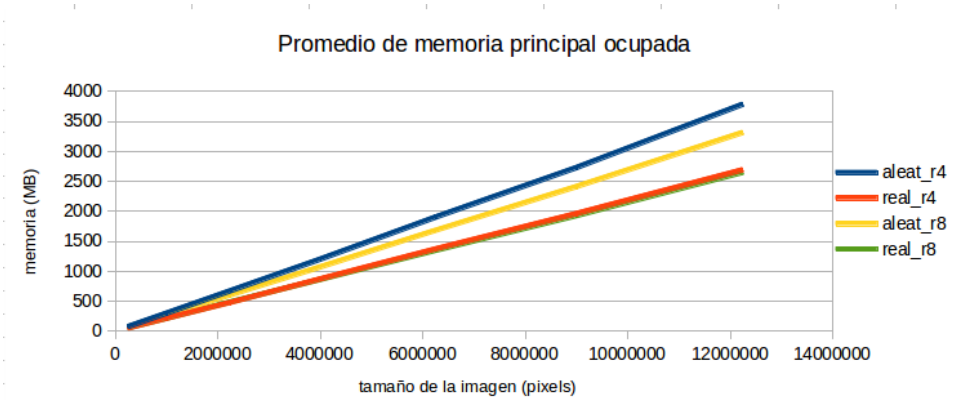


Figura 3.6: Memoria principal en función del tamaño de la imagen

Se muestra que el consumo de memoria es lineal respecto respecto a la cantidad de píxeles, tanto para el caso de imágenes aleatorias, como para el de las satelitales.

Finalmente, las mediciones muestran que, en todos los casos, el consumo de CPU involucró la utilización de un solo núcleo al 100 %, lo que es de esperar teniendo en cuenta que el procesamiento involucra principalmente cálculo.

Capítulo 4

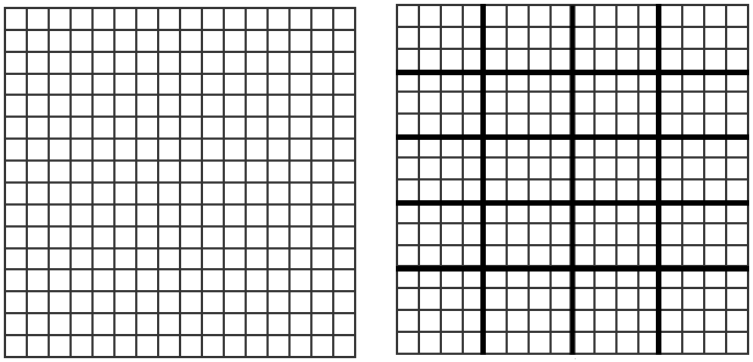
Watershed Topológico en Paralelo

En este capítulo, se presentará un algoritmo paralelo para el cálculo del Watershed Topológico de una imagen, basado en el algoritmo descripto en el capítulo anterior. Para ello, se hará un análisis de las dificultades que introduce la paralelización y se propondrá un algoritmo acorde. Finalmente, se brindará un pseudo-código que implemente la solución propuesta y se expondrán los resultados de las pruebas de eficiencia aplicadas sobre la implementación de tal solución.

4.1. Presentación del problema

Básicamente, el algoritmo cuasi-lineal descripto en el capítulo anterior, se vale del árbol de componentes de la imagen para hacer la transformación, y puede resumirse en dos pasos: el primero, scannear la imagen en su totalidad y armar un mapa ordenado que indique, para cada punto W-destructible, el valor al que debe disminuirse y la componente a la que pertenecerá cuando esto suceda, el segundo, extraer cada punto de ese mapa, disminuir su valor, actualizar el árbol de componentes y, finalmente, actualizar el mapa recalculando la W-destructibilidad de cada uno de sus vecinos. El algoritmo termina una vez que el mapa se vacía.

La idea del algoritmo en paralelo que se propone en este trabajo, se basa en dividir en fragmentos la imagen a transformar y aplicar, en cada uno, el algoritmo cuasi-lineal de manera concurrente, tal como si fuesen imágenes distintas. Notar que no hay sólo una forma de dividir la imagen, por ello, vale aclarar que se dividirá de acuerdo al divisor más cercano a la raíz cuadrada del número de fragmentos deseados, así por ejemplo si se desea dividir la imagen en 20 fragmentos, se dividirá la imagen en 4 fragmentos iguales (salvo quizás el último), y cada subfragmento en 5 subfragmentos iguales (salvo quizás el último). Esta división puede verse gráficamente en



(a) (b)

Figura 4.1: (a) Imagen original. (b) División en 20 fragmentos.

la Figura 4.1.

El primer paso mencionado previamente, puede hacerse en paralelo sin problemas puesto que la imagen sólo se lee y, por lo tanto, no hay problemas de sincronización. Respecto del segundo, surgen problemas de paralización evidentes. Una situación que lo pone de manifiesto es la siguiente: se supone un thread t corriendo en el fragmento A de la imagen, mientras, concurrentemente, otros threads corren en cada uno de los otros fragmentos. Sistemáticamente, lo que hace t es tomar algún punto W-destructible p ($\in$ fragmento A), disminuir su valor, actualizar el árbol y actualizar el mapa para cada uno de los $q \in \Gamma(p)$. ¿Qué sucede si algún q pertenece a otro fragmento? El valor de q puede haber cambiado respecto del momento en el que se armó el mapa que utiliza t, y, en consecuencia, este puede haber quedado desactualizado, con lo cual no se puede estar seguro del valor al que debe ser disminuido p. Del mismo modo, si se dismunuyera p, quedarán desactualizados los mapas de los fragmentos en los que p tenga vecinos. Debido a estos problemas, se terminará con una imagen resultante que no satisface la definición de Watershed Topológico.

De aquí en adelante, se dirá que un punto es *borde* si posee vecinos en fragmentos ajenos, desde luego esto dependerá de la relación de vecindad que se esté utilizando. En la Figura 4.2 se observan los puntos bordes de una imagen dividida en 4 fragmentos, tomando una relación de vecindad tipo 4 o tipo 8.

4.2. Estado del arte

Como se vio en el apartado anterior, el problema para hacer un algoritmo paralelo puede resumirse en contestar la pregunta ¿Cómo resolver los problemas de sincronización que presentan los puntos borde?

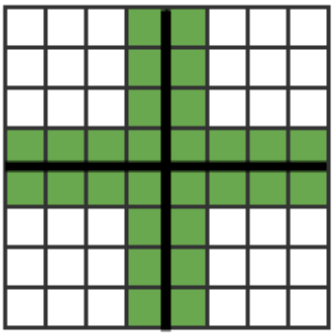


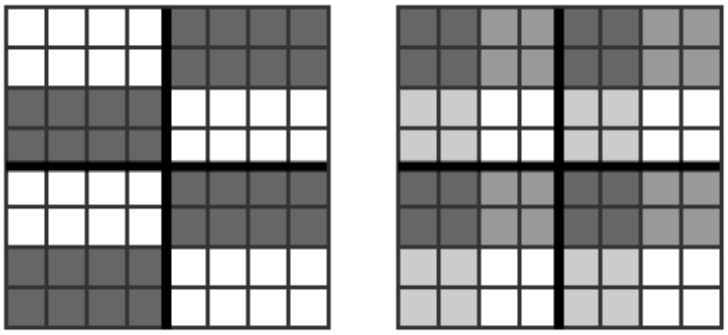
Figura 4.2: En verde se muestran los puntos borde de una imagen dividida en 4 fragmentos, tomando una relación de vecindad tipo 4 o tipo 8.

Una de las respuestas que primero viene a la cabeza, es que todos los threads utilicen un mapa global y que cada thread actualice el valor de los puntos que pertenecen a su fragmento y de los vecinos de los puntos borde pertenecientes a fragmentos ajenos. Sin embargo, el problema de sincronización sigue existiendo, puesto que puede modificarse al mismo tiempo el valor de un punto borde por un thread, y el valor de su vecino en otro fragmento por otro thread, con lo cual no se puede asegurar que la transformación sea correcta.

En [8] se propone una solución basada en particionar cada fragmento en subfragmentos, de modo de asegurar que si un thread está modificando un punto, ningún otro esté modificando algún vecino perteneciente a otro fragmento. De este modo, cada thread procesa un subfragmento i, y, cuando todos terminan, cada uno procesa el $i+1$, continuando de esta manera hasta procesar todos los subfragmentos.

Por supuesto, una sola iteración puede ser insuficiente puesto que al disminuir un punto p que pertenece a un subfragmento, puede que alguno de sus vecinos en algún otro subfragmento se vuelva W-destructible. Por este motivo, se utiliza un arreglo global para marcar los puntos que hayan sido modificados, y asi procesar nuevamente cada uno de los vecinos en su respectivo subfragmento.

La desventaja que presenta este enfoque, es que el modo de particionar los fragmentos depende del tipo de vecindad que se utilice, tal como puede observarse en la Figura 4.3, lo que supone un gran problema de adaptabiliad del algoritmo, puesto que para cada relación de vecindad que desee soportar el algoritmo, habrá que idear un subfragmentado acorde (que puede no ser trivial).



(a) (b)

Figura 4.3: Subfragmentación para una imagen dividida en 4. (a) Subfragmentación para el caso de una relación de vecindad de tipo 4. Cada fragmento se divide en 2 subfragmentos. (b) Subfragmentación para el caso de una relación de vecindad de tipo 8. Cada fragmento se divide en 4 subfragmentos.

4.3. Algoritmo

La solución que se presenta en este trabajo, consiste en tratar los puntos borde de modo secuencial, y, de este modo, superar la desventaja del algoritmo descripto anteriormente. Notar además, que este enfoque simplifica su desarrollo.

De modo concreto, se propone dividir la imagen en fragmentos y, de forma concurrente, calcular el mapa para cada uno tal como lo hace el algoritmo cuasi-lineal. Luego, utilizar el mapa de modo análogo al algoritmo cuasi-lineal, con la salvedad de, en caso de haber tomado del mapa un punto borde, este se marca en un arreglo global y no se modifica su valor.

Una vez que todos los threads terminan, un único thread repite el proceso para los puntos bordes marcados en el arreglo global, marcando en un nuevo arreglo aquellos puntos que no sean borde y que deban ser recalculados.

Una vez hecho esto, el proceso se repite nuevamente, donde cada thread se encarga de procesar aquellos puntos marcados en el arreglo global que pertenecen a su fragmento, y así hasta eliminar todos los puntos W-destructibles.

4.4. Implementación

A continuación se presentará el pseudo-código del algoritmo propuesto en el apartado anterior.

La siguiente función se encarga de inicializar el mapa de un fragmento pasado como parámetro.

Algorithm 5

```
1: function initializeMap(E, Γ, F, T(F̄), Ψ, Tile)
2:     for all p ∈ Tile do
3:         [i, c] ← isWdestructible(E, F, Γ, p, T(F̄), Ψ)
4:         if [i, c] ≠ ∅ then
5:             componentMap.insert(p, [i, c])          ▷ sorted by i
       return componentMap
```

Notar que coincide con la lógica utilizada por el algoritmo cuasi-lineal, salvo que, en lugar de recorrer toda la imagen, se restringe al set de puntos $Tile$ pasado como parámetro.

Lo que sigue es el procedimiento que se encarga de procesar un fragmento de la imagen.

Algorithm 6

```
1: procedure doTopologicalWatershedOnTile(E, Γ, F, T(F̄), Ψ, Tile)
2:     componentMap ← InitializeQueue(E, Γ, F, T(F̄), Ψ, Tile)
3:     while ¬componentMap.empty() do
4:         p, [i, c] ← componentMap · extractFirst()
5:         if Γ(p) ⊈ Tile then
6:             borders.insert(p)
7:         else
8:             F(p) ← i − 1
9:             Ψ(p) ← [i, c]
10:            for all q ∈ Γ(p), F(q) ≥ i do
11:                [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
12:                if [j, d] = ∅ then
13:                    componentMap.removeIfExists(q)
14:                else
15:                    componentMap.addOrUpdate(q,[j,d])
```

Nuevamente, notar el parecido al algoritmo cuasi-lineal, lo único que se agrega es el chequeo de si el punto es borde, y en este caso, se extrae del mapa y se marca en el arreglo global borders.

A continuación se muestra la lógica del procedimiento encargado de procesar los puntos borde.

Algorithm 7

```
1: procedure doTopologicalWatershedOnBorder(E, Γ, F, T(F̄), Ψ)
2:     componentMap ← InitializeQueue(E, Γ, F, T(F̄), Ψ, borders)
3:     while ¬componentMap.empty() do
4:         p, [i, c] ← componentMap · extractFirst()
5:         F(p) ← i − 1
6:         Ψ(p) ← [i, c]
7:         for all q ∈ Γ(p), F(q) ≥ i do
8:             [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
9:             if [j, d] = ∅ then
10:                componentMap.removeIfExists(q)
11:                changed.removeIfExists(q)
12:            else
13:                if q ∈ componentMap then
14:                    componentMap.update(q, [j, d])
15:                else
16:                    changed.insert(q)
```

Como puede observarse, es aquí donde se disminuye el valor de los puntos borde, motivo por el cuál esta lógica debe correr en modo secuencial. Al terminar este procedimiento, todos los puntos borde habrán dejado de ser W-destructibles, pero al disminuir su valor, puede que algunos puntos no borde se hayan vuelto W-destructibles, por lo cual se los marca en el arreglo globlal changed.

Finalmente, aquí se presenta el procedimiento encargado de hacer la transformación Watershed paralela.

Algorithm 8

1: **procedure** doTopologicalWatershed(E, Γ, F, $T(\overline{F})$, Ψ, n)
2: $tiles \leftarrow divideImageInTiles(n)$
3: $threadPool \leftarrow createThreadPool(n)$
4: $borders \leftarrow \emptyset$ ▷ global for all threads
5: $changed \leftarrow \emptyset$ ▷ global for all threads
6: **for** $i = 1 \rightarrow n$ **do**
7: $threadPool[i].run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i]))$
8: $waitForAllThreadsToFinish()$
9: $doTopologicalWatershedOnBorder(E, \Gamma, F, T(\overline{F}), \Psi)$
10: **while** $changed \neq \emptyset$ **do**
11: **for** $i = 1 \rightarrow n$ **do**
12: $threadPool[i].run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i] \cap changed))$
13: $waitForAllThreadsToFinish()$
14: $doTopologicalWatershedOnBorder(E, \Gamma, F, T(\overline{F}), \Psi)$

Notar que se recibe por parámetro la cantidad de threads que se utilizarán en el proceso. En el for de las líneas 6 y 7, cada thread se encarga de procesar su respectivo fragmento, y una vez que todos terminan, se procesan los puntos borde. Luego, mientras haya puntos no bordes por revisar, cada thread procesa los que se encuentran en su fragmento, marcando en el arreglo los puntos borde si los hubiere, y nuevamente se procesan los mismos, repitiendo este proceso hasta lograr estabilidad.

4.5. Medición y Desempeño

En esta sección, se muestran primero los resultados de las pruebas de rendimiento descriptas en la introducción de este trabajo, luego se analiza la correctitud del algoritmo, más adelante se estudia la mejora en la velocidad de procesamiento y ,finalmente, se hace una comparación cuantitativa respecto del algoritmo descripto en [8].

En las siguientes tablas, se muestran los resultados de las prubas, donde la primera columna muestra el tiempo total de procesamiento, la segunda y la tercera, el porcentaje del mismo tomado por la transformación y la creación del árbol de componentes resp., la cuarta y la quinta, el consumo promedio y máximo de CPU durante toda la ejecución resp. y la última, la cantidad de memoria principal promedio ocupada, expresada en MB.

Ejecución con 1 threads y una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	2.1	63.9	25.7	93.9	109.5	82.1
random1000x1000	9.8	65.3	30.0	98.2	109.6	321.5
random1500x1500	23.7	65.9	29.8	99.1	109.6	721.6
random2000x2000	44.5	66.1	29.5	99.0	109.6	1290.5
random2500x2500	74.0	67.5	28.9	99.6	109.5	2035.5
random3000x3000	115.8	69.8	26.5	99.5	109.7	2934.7
random3500x3500	174.1	71.2	25.5	99.7	109.5	4041.9

Ejecución con 2 threads y una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.4	49.5	39.1	122.3	199.3	75.7
random1000x1000	6.8	49.4	43.3	129.6	209.2	301.5
random1500x1500	16.3	50.6	43.3	130.9	209.2	677.7
random2000x2000	30.4	51.0	43.4	131.1	209.2	1218.0
random2500x2500	49.4	51.9	43.2	131.9	209.2	1929.6
random3000x3000	75.6	54.1	41.0	134.1	219.1	2765.0
random3500x3500	110.9	55.5	40.0	135.1	209.2	3835.6

Ejecución con 4 threads y una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.1	41.8	48.3	153.9	398.7	72.7
random1000x1000	5.6	37.5	52.0	154.3	408.4	289.2
random1500x1500	13.6	39.5	52.3	158.1	408.4	649.7
random2000x2000	25.7	41.4	51.3	158.1	408.4	1176.9
random2500x2500	42.8	41.9	50.0	156.9	408.4	1877.7
random3000x3000	62.2	43.9	50.0	167.8	408.4	2663.1
random3500x3500	89.6	44.9	49.3	171.6	408.5	3698.3

Ejecución con 8 threads y una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.2	31.0	44.4	161.8	797.2	73.9
random1000x1000	5.0	31.9	58.7	184.0	806.5	281.0
random1500x1500	12.1	33.1	58.6	184.6	806.8	635.9
random2000x2000	22.2	32.7	59.4	188.1	806.8	1133.6
random2500x2500	36.9	35.2	58.0	189.0	816.5	1815.6
random3000x3000	52.8	34.3	58.7	203.2	806.8	2557.7
random3500x3500	76.4	35.2	57.8	208.1	816.7	3563.7

Ejecución con 16 threads y una relación de vecindad tipo 4

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.2	25.4	43.9	160.9	806.3	75.3
random1000x1000	4.8	27.3	61.1	200.6	1563.4	277.5
random1500x1500	11.3	27.9	62.8	206.6	1594.1	617.3
random2000x2000	20.6	27.1	64.0	208.2	1613.6	1110.1
random2500x2500	33.2	28.1	64.5	214.7	1603.8	1764.0
random3000x3000	49.0	29.4	63.1	229.1	1614.1	2508.1
random3500x3500	72.2	31.6	61.2	235.0	1605.2	3517.2

Ejecución con 1 threads y una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	4.4	57.9	15.6	75.8	109.5	75.6
random1000x1000	13.7	70.7	26.3	98.6	109.7	286.0
random1500x1500	32.3	69.8	27.4	99.3	109.7	635.1
random2000x2000	57.1	68.3	28.9	99.5	109.7	1123.5
random2500x2500	92.6	68.4	28.9	99.6	109.6	1760.3
random3000x3000	134.2	68.4	28.9	99.7	109.6	2522.4
random3500x3500	191.9	68.8	28.6	99.7	109.6	3465.2

Ejecución con 2 threads y una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	2.3	56.6	29.2	132.1	209.4	69.6
random1000x1000	9.2	56.7	39.1	137.7	209.4	274.6
random1500x1500	21.7	55.6	40.8	136.8	209.4	608.3
random2000x2000	39.2	54.0	42.2	136.0	209.4	1080.8
random2500x2500	62.8	53.7	42.5	135.4	209.3	1694.5
random3000x3000	90.8	53.8	42.6	135.7	209.3	2425.8
random3500x3500	130.5	54.4	42.0	135.9	219.2	3341.5

Ejecución con 4 threads y una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.7	53.5	39.3	176.9	398.5	69.1
random1000x1000	7.3	45.0	49.7	176.8	408.4	264.0
random1500x1500	17.5	44.4	50.6	171.4	408.5	590.9
random2000x2000	31.4	42.7	52.6	169.6	408.6	1047.5
random2500x2500	52.2	44.1	51.0	166.4	408.4	1653.6
random3000x3000	74.1	43.2	52.3	168.7	408.4	2351.1
random3500x3500	107.2	44.9	50.7	167.9	408.4	3263.6

Ejecución con 8 threads y una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.7	37.9	39.5	191.0	797.3	68.8
random1000x1000	6.4	36.1	56.3	205.6	807.1	258.4
random1500x1500	15.2	36.1	58.0	203.6	806.8	571.4
random2000x2000	27.6	34.9	59.8	199.8	806.8	1024.3
random2500x2500	44.9	35.0	59.5	197.0	806.9	1614.0
random3000x3000	67.8	37.5	57.5	195.2	806.8	2313.4
random3500x3500	94.6	37.0	58.0	198.5	806.8	3201.8

Ejecución con 16 threads y una relación de vecindad tipo 8

	Tiempo Total (seg)	Transf. (%)	Árbol de comp. (%)	Consumo de CPU prom. (%)	Consumo de CPU Máx. (%)	Consumo de Mem. prom. (MB)
random500x500	1.5	35.6	44.9	227.8	1545.0	66.9
random1000x1000	5.9	31.7	60.9	232.9	1593.7	254.4
random1500x1500	14.1	31.0	63.1	226.4	1603.8	566.4
random2000x2000	25.7	29.8	64.2	219.4	1603.8	1004.8
random2500x2500	41.7	29.8	64.4	218.7	1603.9	1591.3
random3000x3000	60.1	30.3	64.4	221.1	1613.6	2262.9
random3500x3500	85.3	29.6	64.1	224.2	1613.7	3140.9

Es importante analizar la correctitud del algoritmo presentado, lo cual no es trivial puesto que el resultado final depende del orden en que se hayan tomado los nodos W-destructibles, algo que cambia entre ejecución y ejecución debido a la concurrencia. Tal como se desprende del pseudo-código mostrado en la sección anterior, la implementación paralela consiste dividir la imagen a procesar en fragmentos y ejecutar, concurrentemente, el algoritmo secuencial en cada uno de ellos, con la particularidad de posponer el procesamiento de los puntos borde. Si bien desde un punto de vista intuitivo es correcto, es importante hacer pruebas sobre los resultados obtenidos. En este sentido, se decidió calcular la distancia euclídea la imagen obtenida de la ejecución secuencial y las de la paralela, donde dicha distancia se define de la siguiente manera:

$$EU = \frac{\sum_{i=1}^{N} |p_i - q_i|}{N}$$

donde p_i y q_i denotan el i-ésimo píxel de cada imagen y N, el tamaño en píxels de cada imagen. A continuación, se muestran los resultados obtenidos:

	1 thread	2 threads	4 threads	8 threads	16 threads
random500x500_r4	0	0.0127	0.0244	0.0587	0.0879
random500x500_r8	0	0.0281	0.05	0.1031	0.1472
random1000x1000_r4	0	0.0042	0.0084	0.0167	0.0223
random1000x1000_r8	0	0.0103	0.016	0.0264	0.0421
random1500x1500_r4	0	0.0015	0.0031	0.005	0.0076
random1500x1500_r8	0	0.0037	0.0062	0.0131	0.0185
random2000x2000_r4	0	0.0006	0.0011	0.0024	0.0039
random2000x2000_r8	0	0.0015	0.003	0.0072	0.0106
random2500x2500_r4	0	0.0002	0.0007	0.0014	0.002
random2500x2500_r8	0	0.001	0.0017	0.0044	0.0068
random3000x3000_r4	0	0.0002	0.0005	0.0011	0.0015
random3000x3000_r8	0	0.0009	0.0014	0.0027	0.0041
random3500x3500_r4	0	0.0003	0.0004	0.0009	0.0011
random3500x3500_r8	0	0.0005	0.0009	0.002	0.0029

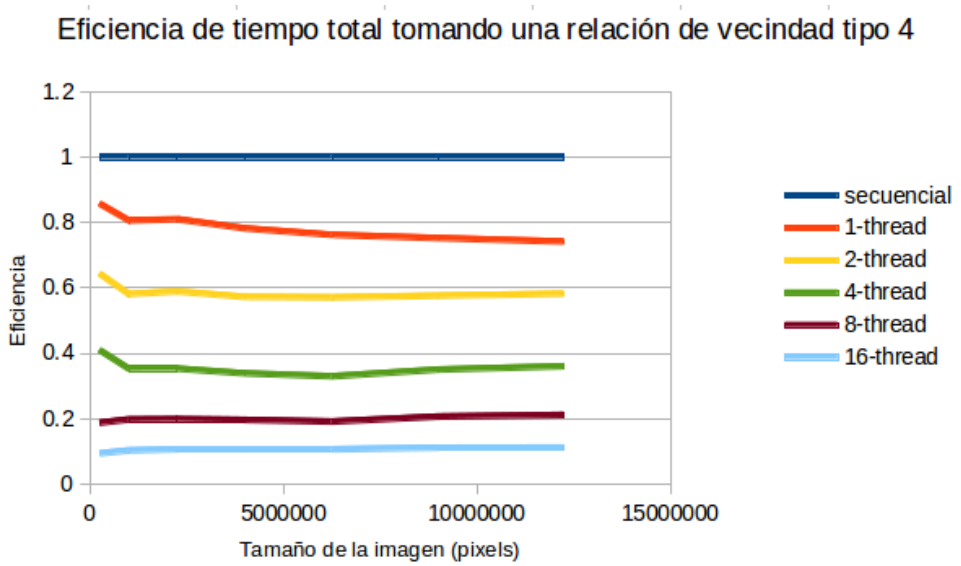
Puede observarse que, en todos los casos, las imágenes obtenidas a partir de una ejecución secuencial y la paralela utilizando sólo un thread, son idénticas, lo que confirma la observación intuitiva. Por otro lado, se puede apreciar que la distancia se incrementa a medida que crece la cantidad de hilos, lo que obviamente se debe a que, a medida que crece la cantidad de hilos, crece la cantidad de ejecuciones posibles. En base a la bajos valores obtenidos, se concluye que los resultados que produce el algoritmo concurrente son confiables.

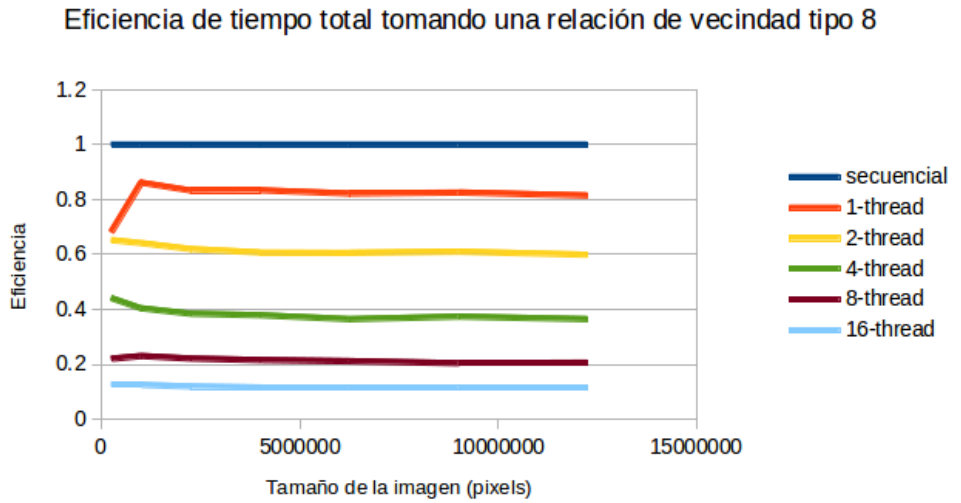
Es interesante observar las mejoras en términos de tiempo de procesamiento logradas a partir de la concurrencia. Se hará a partir de la función de eficiencia, definida de la siguiente manera:

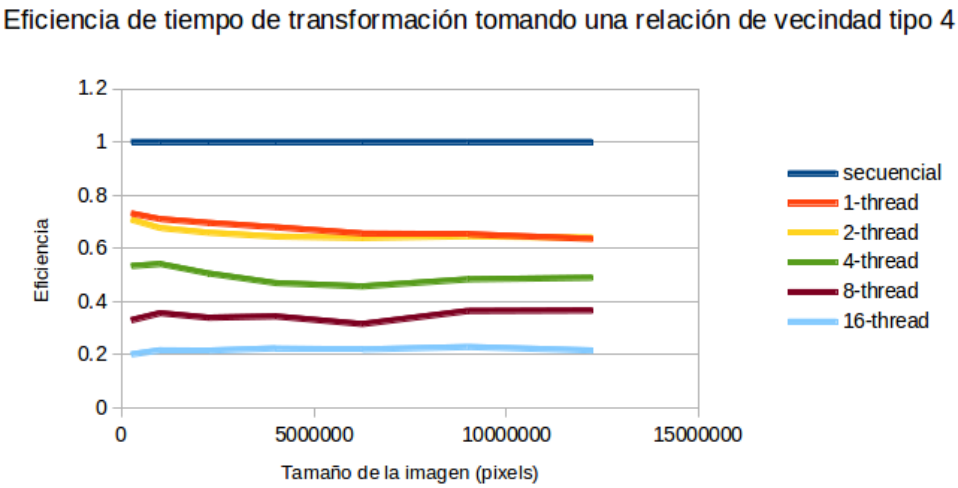
$$Eficiencia = \frac{t_s}{p * t_p}$$

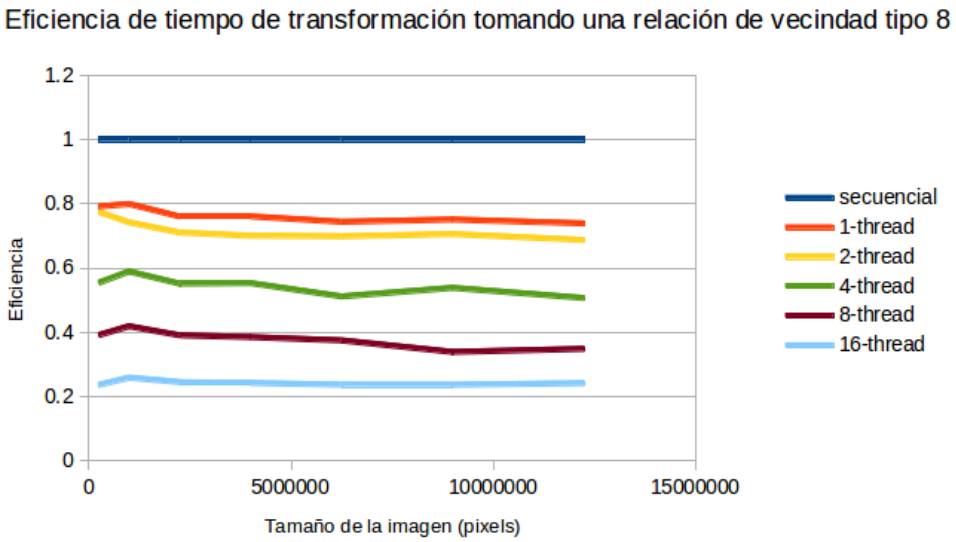
donde t_s es el tiempo de ejecución secuencial y t_p, el correspondiente al paralelo utilizando p threads, es fácil notar que el resultado es siempre 1 para el caso secuencial.

Se calculó dicha medición tanto respecto del tiempo total como del tiempo de transformación, tanto tomando una relación de vecindad tipo 4 como una tipo 8, obteniendo los resultados que se muestran a continuación.


Eficiencia de tiempo total tomando una relación de vecindad tipo 4
Eficiencia
1.2
1
0.8
0.6
0.4
0.2
0
0
5000000
10000000
15000000
Tamaño de la imagen (pixels)
secuencial
1-thread
2-thread
4-thread
8-thread
16-thread

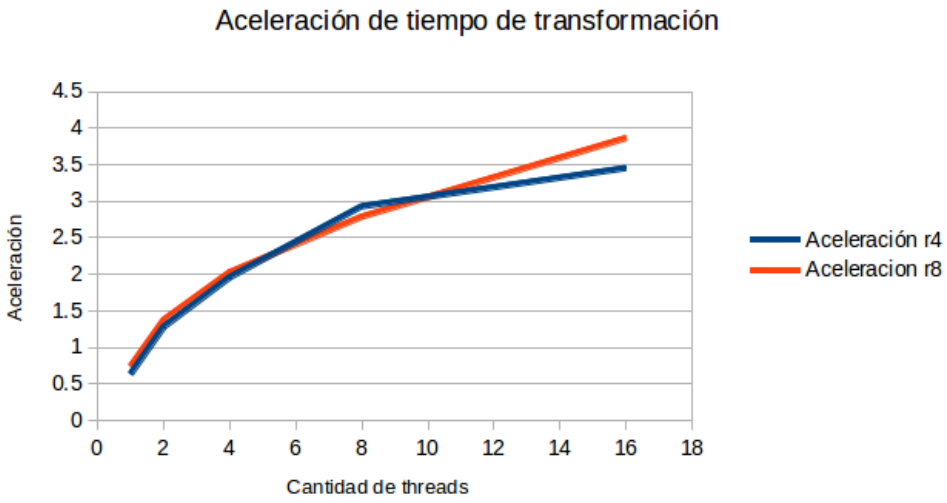

Eficiencia de tiempo total tomando una relación de vecindad tipo 8
Eficiencia
1.2
1
0.8
0.6
0.4
0.2
0
0
5000000
10000000
15000000
Tamaño de la imagen (pixels)
secuencial
1-thread
2-thread
4-thread
8-thread
16-thread





Se ve cláramente que los resultados son mejores al tomar el tiempo de transformación, lo que obviamente se debe a que la creación del árbol es secuencial.

El siguiente gráfico muestra la aceleración lograda, en el tiempo de transformación, para la imagen aleatoria de 3500x3500 pixels.



Es muy interesante constrastar estos resultados con los logrados con el enfoque desarrollado en [8]. De acuerdo a sus conclusiones, se llevaron a cabo mediciones con una imagen satelital con una resolución de 4000x4000 pixels correspondiente a un campo de aviación, tomando una relación de vecindad tipo 4, reportando una aceleración del tiempo de transformación de 0.667 para ejecuciones con 1, 2 y 4 threads, 0.508 para una con 8 threads y 0.479 para una con 16 threads. Se aprecia que a medida que aumenta la cantidad de threads, los resultados son más favorables a este enfoque.

Respecto al consumo de memoria, cabe mencionar que no se observaron grandes cambios respecto de la ejecución secuencial, por lo que se concluye que crece linealmente. No se pueden hacer comparaciones en este aspecto respecto del algoritmo descripto en [8] puesto que no se reportaron mediciones en torno a esta variable.

Respecto a la utilización de CPU, los valores máximos muestran que, en todos los casos, en algún momento se utilizaron la misma cantidad de núcleos que de threads al 100 %, mostrando la demanda de procesamiento involucrada, mientras que el consumo promedio es más pequeño en proporción al máximo a medida que aumenta la cantidad de threads, lo que se debe a que el tiempo tomado por la creación del árbol se vuelve más preponderante.

Capítulo 5

Conclusión

En este trabajo, se introdujo al lector al campo de la Transformación Watershed Topológico. En primera instancia, se abordó su definición y se propuso un algoritmo fuerza-bruta para llevarla a cabo. Luego, se presentó un algoritmo secuencial de orden cuasi-lineal descripto en [5], para después diseñar uno paralelo en base a este. Para analizar su desempeño, se llevaron a cabo pruebas de rendimiento sobre el algoritmo secuencial y el paralelo utilizando 1, 2, 4, 8 y 16 threads. Las mismas, involucraron la utilización de imágenes aletorias y el reporte del tiempo total de ejecución, discriminando el correspondiente a la creación del árbol de componententes y a la transformación, el consumo máximo y promedio de CPU y la cantidad de memoria principal utilizada.

Los resultados mostraron, en todos los casos, un alto consumo de CPU producto de la cantidad de cálculo involucrado y una ocupación de memoria principal que crece linealmente respecto del tamaño de la imagen.

Desde el punto de vista del tiempo de ejecución, se observó que, para una imagen aleatoria de resolución 3500x3500 píxeles, tomando una relación de vecindad tipo 4, el tiempo de transformación bajó de 78.7s para el caso secuencial, a 22.8s para el paralelo con 16 threads. Del mismo modo, se observó una mejora de 97.5s a 25.2s para el caso análogo, tomando una relación de vecindad tipo 8. Sin embargo, las comparaciones con el enfoque concurrente descripto en [8] favorecieron a este último.

Es muy importante destacar que el enfoque descripto en este trabajo es más sencillo de comprender e implementar que el desarrollado en [8], lo que es una característica deseable de cualquier algoritmo, teniendo en cuenta que facilita su abordaje y su mantenibilidad.

La implementación concurrente de la creación del árbol de componentes quedó fuera del alcance de este trabajo. Sería muy interesante lograr avances en este sentido y apreciar su efecto en el tiempo total de procesamiento.

El código fuente correspondiente a cada una de las implementaciones, la bibliografía citada y las imágenes que se utilizaron para llevar a cabo

las mediciones con su correspondiente transformación, pueden descargarse utilizando el sistema de control de versionado *git* desde el repositorio `https://github.com/d-montenegro/Topological-Watershed`.

Anexo

Anexo A

Transformación Watershed

En el área de estudio de procesamiento de imágenes, las imágenes en escala de grises comúnmente son representadas como relieves topográficos, esto es, el nivel de gris de un punto representa la altura del mismo en un eje cartesiano 3D. Esta representación favorece la apreciación del efecto de ciertas transformaciones sobre la imagen de estudio.

La segmentación de imágenes, comprende la partición de una imagen de modo de facilitar la distinción de los objetos que hay en la misma respecto del fondo, a partir del resaltado de contornos. Dentro del campo de la morfología matemática, la Transformación Watershed constituye una de las clásicas herramientas para este propósito.

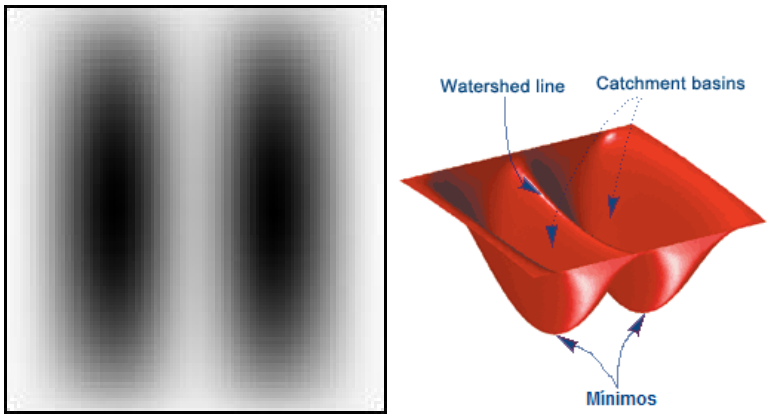
La Transformación Watershed utiliza la representación topográfica de la imagen haciendo foco en 3 tipos de puntos:

- Los mínimos regionales, son aquellos puntos que no pueden alcanzar a un punto más bajo valor sin pasar antes por uno más alto.
- Aquellos puntos en los que, si cayera una gota de agua, indefectiblemente esa gota descendería hacia un mínimo regional determinado. Técnicamente, constituyen las *Catchment Basins*.
- Aquellos puntos en los que, en caso de caer una gota de agua, no podría determinarse hacia qué mínimo descendería. Estos, son los que forman las *Watershed Lines*.

En la Figura A.1 se ejemplifican puntos pertenecientes a cada tipo.

El objetivo de la transformación es particionar la imagen de acuerdo a las Watershed Lines.

Uno de los más populares algoritmos para llevar a cabo este procesamiento es el conocido como "Watershed por Inmersión", desarrollado en [1], el cuál se basa en la idea de un relieve topográfico al que se le hacen perforaciones en sus mínimos y, lentamente, se lo sumerge en agua. El nivel de agua cubrirá primero los mínimos y, progresivamente, inundará las *cuencas* del

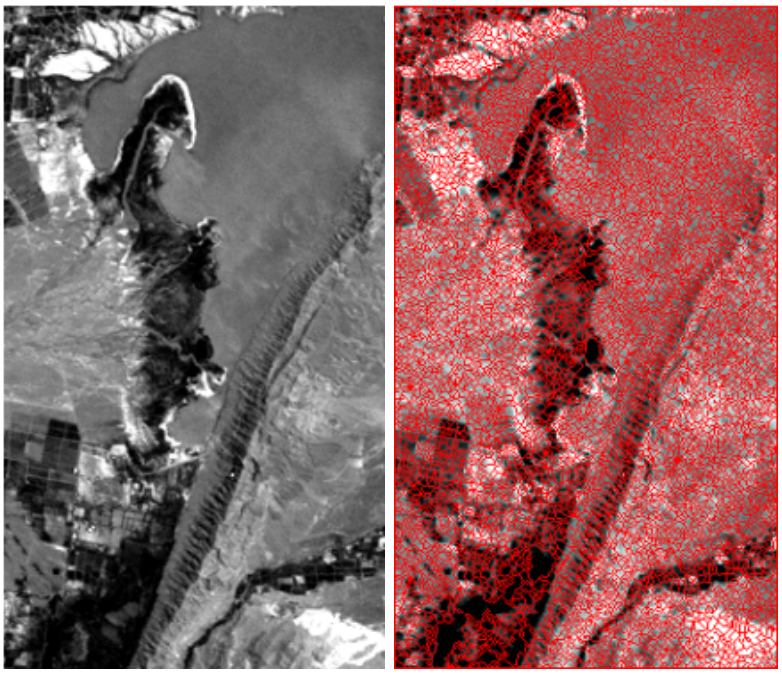


(a) (b)

Figura A.1: Watershed Lines. (a) Imagen original, (b) Se muestra el relieve topográfico, asumiendo que los puntos más oscuros son los que poseen nivel de gris inferior.

relieve. Ahora bien, en cada punto en que el agua proveniente de dos cuencas se mezcle, se construirán *represas*. Una vez que el relieve se encuentre completamente bajo el agua, quedarán las represas que dividen cada una de las regiones de la imagen. Las cuencas constituyen las Catchment Basins, mientras que las represas, las Watershed Lines. En la Figura A.2 se muestra la Transformación Watershed de una imagen.

Al observar los ejemplos, el lector puede notar claramente que se produce una sobresegmentación en la imagen. Se trata de un fenómeno muy conocido, y se debe a que cada mínimo, por más pequeño y angosto que sea, constituye su propia Catchment Basin, de aquí que el ruido y las pequeñas irregularidades presentes en la imagen produzcan este efecto tan notable. En este sentido, cabe mencionar que existen diversos métodos para apaliar este problema, pero la cobertura de dichos métodos está fuera del alcance de este trabajo.



(a) (b)

Figura A.2: Watershed Lines. (a) Imagen original, (b) Watershed Lines superpuestas a la imagen original

.

Anexo B

Grafos

Un grafo es una estructura matemática, cuya utilidad es la representación de relaciones binarias entre elementos. Formalmente, es un par (E, Γ), donde E es un conjunto de elementos, llamados vértices o nodos, y Γ es un mapa de $E \to \rho(E)$, donde $\rho(E)$ es el conjunto de todos los subconjuntos de E, conocido en el campo como *partes de* E. Por ejemplo, si $E = \{1, 2, 3\}$, entonces $\rho(E) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Para cada $e \in E$, el conjunto de elementos $\Gamma(e)$ constituye el conjunto de vértices con los que e tiene relación, y se lo denomina *vecindario de* e.

Un grafo (E, Γ) se dice simétrico si, $\forall x, y \in E : x \in \Gamma(y) \implies y \in \Gamma(x)$. Si $x \in \Gamma(y)$ entonces se dice que el conjunto $\{x, y\}$ constituye un lado del grafo y que x es vecino de y.

Gráficamente, se representa un grafo con los vértices como puntos, y las lados como flechas que unen los puntos correspondientes, o segmentos si se trata de uno simétrico. En la Figura B.1 podemos observar la representación gráfica de un grafo simétrico (V, Γ), donde $V = \{A, B, C, D, E, F, G\}$ y $\Gamma(A) = \{B, D, G\}$.

Sea (E, Γ) un grafo, $X \subseteq E$ y $x_0, x_n \in X$, entonces

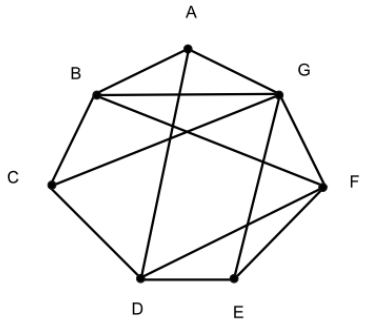


Figura B.1: Representación gráfica de un grafo.

- Un camino de x_0 hacia x_n en X será un conjunto ordenado $(x_0, x_1, ..., x_n)$ de elementos de X, donde $x_{i+1} \in \Gamma(x_i)$ para i = 0,1,...,n-1.
- Decimos que x_0, x_n están X-conectados si existe un camino de x_0 hacia x_n en X.
- Decimos que X es conectado si cada par de elementos de X están conectados en X.
- Sea $Y \subseteq E$, decimos que Y es una componente conectada de X si $Y \subseteq X$, Y está conectado y es maximal.
- (E, Γ) es conectado si E es conectado.

Podemos ejemplicar estas definiciones observando nuevamente el grafo (V, Γ) de la Figura B.1. Podemos ver que $\{A, B, C\}$ es un camino de A hacia C en V, y por lo tanto están V-conectados. Además, podemos decir que el conjunto $\{A, B, C\}$ es conectado, pero $\{A, B, C, E\}$ no lo es. Podemos mencionar que $\{A, B, C\}$ es una componente conectada de $\{A, B, C, E\}$. Finalmente, podemos decir que (V, Γ) es conectado.

Un *grafo ponderado* o *grafo con pesos* es un grafo en el que cada nodo tiene un peso asociado. Formalmente, es una estructura (E, Γ, F), donde (E, Γ) es un grafo y $F : E \to C$ es la función de peso, donde C usualmente es un conjunto numérico.

Bibliografía

[1] Vincent L. and Soille P., 1991. *Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, No. 6.

[2] Bertrand G. and Couprie M., 1997. *Topological grayscale watershed transformation*. SPIE Vision Geometry VI Proceedings, Vol. 3168.

[3] Najman L. and Couprie M., 2003. *Watershed algorithms and contrast preservation*. Discrete Geometry for Computer Imagery, Vol. 2886. Lecture Notes in Computer Science.

[4] Bertrand G., 2005. *On topological watersheds*. Journal of Mathematical Imaging and Vision, Vol. 22, No. 2-3.

[5] Couprie M., Najman L. and Bertrand G., 2005. *Quasi-linear algorithms for the topological watershed*. Journal of Mathematical Imaging and Vision, Vol. 22, No. 2-3.

[6] Couprie M. and Najman L. 2004. *Quasi-linear algorithm for the component tree*. SPIE Vision Geometry XII, Vol. 5300.

[7] Bender M.A. and Farach-Colton M., 2000. *The LCA problem revisited*. LATIN 2000: Theoretical Informatics Lecture Notes in Computer Science, Vol. 1776.

[8] van Neerbos J., Najman L. and Wilkinson M.H.F., 2012. *Towards a Parallel Topological Watershed: First Results*. 10th International Symposium on Mathematical Morphology.

[9] Roerdink J.B.T.M. and Meijster A., 2000. *The Watershed Transform: Definitions, Algorithms and Parallelization Strategies*. Fundamenta Informaticae, Vol. 41.

[10] Severance C. and Dowd K., 2010. *High Performance Computing*. Connexions, CC3.0.

[11] Eijkhout V., 2011. *Introduction to High-Performance Scientific Computing*. TACC.