Universidad Nacional Autónoma de México

Facultad de Ciencias

PROGRAMACIÓN PARALELA DEL ALGORITMO DE SOBEL PARA LA DETECCIÓN DE BORDES EN IMÁGENES

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
FELIPE DE JESÚS NAVARRETE CÓRDOVA

DIRECTOR DE TESIS:
DR. JORGE LUIS ORTEGA ARJONA

2011

Hoja de Datos del Jurado

1. Datos del alumno
   Navarrete
   Córdova
   Felipe de Jesús
   55 3343 3345
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   302115581

2. Datos del tutor
   Dr
   Jorge Luis
   Ortega
   Arjona

3. Datos del sinodal 1
   Dr
   Sergio
   Rajsbaum
   Gorodesky

4. Datos del sinodal 2
   Dr
   Héctor
   Benitez
   Pérez

5. Datos del sinodal 3
   Mtro
   Manuel Cristobal
   López
   Michelone

6. Datos del sinodal 4
   Mat
   María Concepción Ana Luisa
   Solís
   González-Cosío

7. Datos del trabajo escrito
   Programación paralela del Algoritmo de Sobel para la detección de bordes en imágenes
   110 p
   2011

Resumen

La detección de bordes en imágenes con el filtro de Sobel implica convolucionar la imagen fuente de n pixeles, y por cada pixel p dentro de la imagen, se deben analizar sus nueve vecinos para obtener el valor final del pixel. Cuando las imágenes a tratar son de alta resolución, se requiere de un mayor tiempo de procesamiento, el cual llega a ser inaceptable. Para disminuir este tiempo, el cómputo paralelo juega un rol importante, pues posibilita la ejecución de múltiples tareas simultáneamente.

De esta manera, se realiza la tarea de convolución, con el filtro de Sobel, en diferentes subimágenes, más pequeñas que la original, simultáneamente. Al término del procesamiento en cada subimagen se recopila el resultado de cada una de ellas para formar la imagen final, donde se muestran los bordes detectados en la imagen original.

El problema a tratar dentro de un programa paralelo es la comunicación entre procesos. Para esto, se hace uso de patrones de software paralelo, que definen un modelo teórico para visualizar y entender el modo en que los procesos se ejecutan en los diferentes procesadores disponibles y cómo establecer las coordinaciones entre los diferentes procesos.

En el presente trabajo se realiza la comparación entre los patrones de software paralelo *Manager-Workers* y *Comunicating Secuential Elements* al realizar la detección de bordes con el filtro de Sobel. Para este cometido se desarrolla un programa que implementa ambos patrones y el algoritmo de Sobel dentro de un sistema distribuído. Al final de las pruebas se muestran los resultados obtenidos.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto

El procesamiento paralelo es la división de un problema, presentado como una estructura de datos o un conjunto de acciones entre múltiples procesadores que operan simultáneamente. El resultado esperado es una solución más eficiente del problema. El poder del cómputo paralelo radica en la división del problema, creando subproblemas más pequeños y fáciles de entender y que pueden ser resueltos separadamente. La partición es importante para el procesamiento paralelo porque posibilita a los componentes de software no sólo ser creados separadamente, sino también ejecutados simultáneamente.[1]

Por otro lado, en el área de procesamiento de imágenes digitales, la segmentación tiene como finalidad la obtención de objetos homogéneos de la imagen y separarlos o segmentarlos del resto de ella, para simplificar su representación en algo más manejable o simple de manipular [16]. Puede ser definida además como el proceso de agrupar pixeles con atributos similares. El mayor reto de la segmentación de imágenes es la definición de características que son únicas de cada región, y que puedan ser usadas para separar una región de otra.

Algunas técnicas de segmentación pueden ser encontradas en cualquier aplicación que involucre detección, reconocimiento y medida de objetos en una imagen [3]. Algunos ejemplos son:

- Revisión industrial.
- Reconocimiento de patrones.
- Rastreo de objetos en secuencias de imágenes.
- Clasificación de terrenos visibles en imágenes de satélites.
- Detección y medida de huesos, órganos, etc. en imágenes médicas.

1.2. Problema

La segmentación de imágenes implica un procesamiento que requiere de diversas operaciones de la computadora para obtener resultados. Al realizar estas operaciones en imágenes de alta resolución, el tiempo de procesamiento requerido aumenta considerablemente.
En diversas circunstancias es necesario tener los resultados del análisis de imágenes en un lapso de tiempo corto e incluso instantáneamente. En muchos casos, el tamaño de las imagenes que se manejan son lo suficientemente grandes para no poder cumplir este reto y su procesamiento suele tardar semanas o meses.

Este es un problema que se enfrenta al realizar el análisis automático de imágenes, pues se busca tener resultados con precisión y rapidez. El tener a nuestro alcance el procesamiento paralelo nos da una ventaja, pues ayuda a resolver problemas en menor tiempo que si se hiciera secuencialmente, esto es, con uso de un sólo procesador.

La precisión es el otro problema al cual se enfrenta. No depende únicamente del algoritmo de segmentación, pues al hacer uso de procesamiento paralelo, la imagen es dividida en varias subimágenes, y en cada una se realiza segmentación. Como esto se hace simultáneamente y por separado, al momento de combinar las subimágenes pueden existir problemas de sincronización o comunicación entre los diversos procesos, causando pérdida de información y de ahí, un análisis y resultados erróneos.

1.3. Objetivo

En esta tesis se trata la comparación de los patrones de software paralelo "Communicating Sequential Elements" (CSE)[1] y Manager-Workers[2]., utilizados para la implementación de algoritmos de detección de bordes en imágenes a escala de grises. El objetivo de esta comparación es encontrar el patrón de software que se adapte mejor a este problema, utilizando como algoritmo de detección de bordes, la convolución de imágenes con el filtro de Sobel.

Puesto que la segmentación de imágenes, incluyendo la detección de bordes, sigue siendo un tema de investigación, y el propósito de la tesis no es mejorar los algoritmos existentes o definir uno propio para mejorar la detección de bordes de una imagen, me enfocar solamente a la paralelización y comparación del algoritmo de Sobel, implementado bajo las dos organizaciones software paralelo.

1.4. Hipótesis

La hipótesis de la tesis es que *el patrón Manager-Workers permite realizar el procesamiento en menor cantidad de tiempo que CSE*, tomando en cuenta las mismas

[1] La traducción al español es *Elementos Secuenciales Comunicantes*.

[2] La traducción al español es *Maestro-Trabajadores*. Tanto para Manager-Workers como CSE, se utilizan sus términos en inglés a lo largo del trabajo ya que resulta más fácil identificar las arquitecturas en este idioma que en español.

condiciones y restricciones, tanto de hardware como de software.

1.5. Método

Para comparar los patrones mencionados es necesario que las condiciones de ejecución de ambas sean idénticas, donde lo único que difiere es la manera en que se ejecutan y comunican los procesos.

La comparación se realiza implementando ambos patrones arquitectónicos y ejecutándolos para obtener los tiempos y examinar los resultados. La manera de realizar esto se detalla en la Sección 5.

1.6. Contribuciones

Los resultados de esta tesis tienen como objetivo dar a conocer qué patrón de software paralelo resulta conveniente en detección de bordes en imágenes, específicamente utilizando el filtro de Sobel, y bajo qué condiciones su uso es mejor.

1.7. Estructura de Tesis

El Capítulo 2 da un panorama general sobre los temas que cubre esta tesis:

- Procesamiento paralelo, donde se explica qué es, los patrones de procesamiento paralelo, las ventajas de su uso, la división de un problema y cuándo conviene aplicar el procesamiento paralelo.
- Segmentación de imágenes, donde se explica qué es, sus algoritmos básicos, límites y ejemplos de uso tanto en medicina y otras reas.

El Capítulo 3 da una explicación sobre trabajos y resultados relacionados al tema de la tesis, con los cuales se puede hacer una evaluación.

En el Capítulo 4 se explica el algoritmo de segmentación utilizado en este trabajo para la detección de bordes en imágenes digitales. Además se explica la comunicación entre procesos con los patrones Manager-Workers y CSE.

En el Capítulo 5 se realizan los experimentos y se evalúan de los resultados de la tesis, los cuales son comparados entre sí. Las evaluaciones que se muestran son en cuanto al tiempo de ejecución.

En el Capítulo 6 se dan las conclusiones obtenidas en esta tesis, la reafirmación de hipótesis y contribuciones de este trabajo así como el trabajo futuro que se puede desprender de los resultados obtenidos.

Capítulo 2

Antecedentes

En este capítulo se da una breve introducción al tema de procesamiento paralelo y distribuido, así como de segmentación de imágenes. Se aclaran los puntos más relevantes para efectos posteriores de este trabajo, los cuales proveen definiciones básicas relacionadas al cómputo paralelo y a la segmentación de imágenes; clasificación, organización y construcción de un procesamiento paralelo, así como los modelos que permiten la comunicación y sincronización entre los procesos; tipos de segmentación de imágenes; y una breve descripción matemática de algunos algoritmos comúnmente utilizados en la detección de bordes.

2.1. Procesamiento Paralelo y Distribuido

Un programa paralelo consiste en un conjunto de procesos que son ejecutados simultáneamente. Un proceso define sus propias variables procedimientos y métodos iniciales. Además accede a sus propios propias variables, pero también puede acceder a procedimientos comunes definidos por éste o por otros procesos[9].

El poder del cómputo paralelo radica en la división de un problema, dividiéndolo en subproblemas más pequeños y fáciles de entender y que pueden ser resueltos separadamente. La división es importante para el procesamiento paralelo porque posibilita a los componentes de software no sólo ser creados separadamente, sino también ejecutados simultáneamente[1].

2.1.1. La Necesidad del Cómputo Paralelo

Los cálculos numéricos en los que se involucran ecuaciones diferenciales parciales como dinámica de fluidos y predicción de tiempo, entre otras, son un ejemplo de aplicaciones que requieren de una gran cantidad de cálculos y cuyos resultados necesitan ser mostrados en poco tiempo. El deseo de desarrollar problemas más complejos es medido en términos de tiempo proveyendo el desarrollo de máquinas más rápidas y posiblemente paralelas[2].

En este tipo de aplicaciones es importante el costo y velocidad: El hardware a utilizar no debe ser muy caro y el cómputo debe terminar en una cantidad de tiempo aceptable

para una aplicación en particular.

Otro campo de aplicación de procesamiento paralelo es la adquisición, extracción y manejo de información dentro de sistemas en diferentes puntos geográficos. Estos sistemas deben ser capaces de operar correctamente, en ocasiones con capacidad de comunicación limitada y en otras en ausencia de un mecanismo de control central.

2.1.2. Clasificación de Flynn

En 1966 M. J. Flynn propuso una clasificación de sistemas computacionales basada en flujo de datos e instrucciones. De acuerdo con esta clasificación, el flujo de instrucciones o de datos puede ser simple o múltiple y se aprecia en la Figura 2.1:

- SISD: single-instruction single-data streams (flujos de dato e instrucciones simples).
- SIMD: single-instruction multiple-data streams (flujos simples de instrucciones y flujos múltiples de datos).
- MISD: multiple-instruction single-data streams (flujos múltiples de instrucciones y flujos simples de datos).
- MIMD: multiple-instruction multiple-data streams (flujos múltiples de instrucciones y datos).

La clasificación SISD representa máquinas uniprocesador. SIMD y MISD representan computadoras paralelas: Cuando sólo hay una unidad de control y todos los procesos ejecutan la misma instrucción de manea sincronizada, la máquina paralela es de tipo SIMD; cuando cada proceso tiene su propia unidad de control y puede ejecutar diferentes instrucciones en diferentes datos, tenemos una clasificación MIMD[4]. La clasificación MISD se puede ver como pipelines, donde en cada etapa se realiza una operación relativamente compleja[15].

Debido a que la clasificación de Flynn es válida para sitemas de hardware y el presente trabajo utiliza un cluster de computadoras (Sección 2.1.4) para la ejecución del software desarrollado, el trabajo se encuentra dentro de la lasificación MIMD de Flynn, pues los clúster permiten la ejecución de múltiples datos e instrucciones diferentes simultáneamente. Sin embargo, extendiendo esta clasificación para poder hacer una interpretación aplicada al software desarrollado, el trabajo se encuentra dentro de la clasificación SIMD. Esto es, como se ve en el Capítulo 4, porque se tienen diferentes datos operados por la misma instrucción.

2.1.3. Organización de la Memoria

Memoria Compartida

En la memoria compartida, los procesadores comparten una memoria común (memoria global) a la cual todos ellos pueden leer y escribir, con la misma oportunidad de acceso.

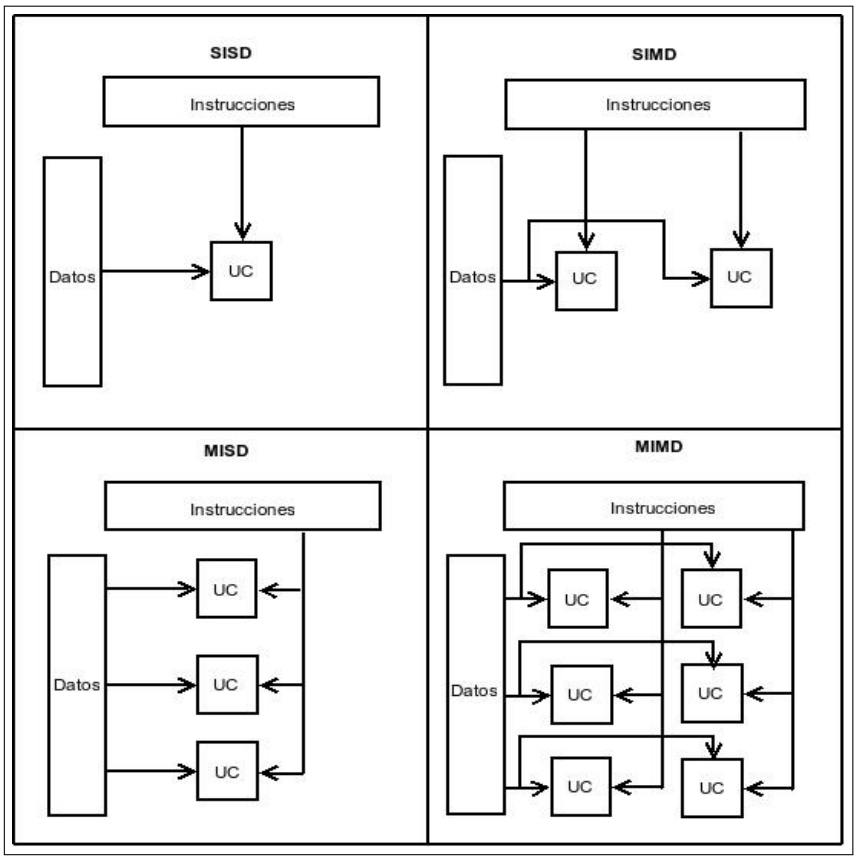


Figura 2.1: Clasificación de Flynn.

En el modelo formal de este esquema, se asume tener más de un procesador activo. En cada instrucción, cada procesador activo puede leer o escribir en la memoria y activar a otro procesador[15].

Para el diseño de sistemas de memoria compartida se deben tener en cuenta algunas consideraciones: control de acceso, sincronización, protección y seguridad. El control de acceso hace el mapeo de procesos con recursos. En este paso se hace la revisión de cada petición de acceso a un recurso por el procesador a la memoria compartida, mediante una tabla de contenidos de acceso, la cual contiene banderas que determinan la posibilidad de hacer uso del recurso. La sincronización limita el tiempo de acceso de procesos compartidos a recursos compartidos. La protección se encarga de que los procesadores no puedan tener acceso a recursos pertenecientes a otro procesador[4].

Las computadoras con arquitectura memoria compartida, pueden ser clasificadas, de acuerdo al modo en que los procesadores y la memoria están conectados y al tiempo de acceso a la memoria[4], en dos categorías[1]:

- UMA. Acceso Uniforme a la Memoria (en inglés: Uniform Memory Access). Todos los procesadores tienen acceso a la memoria global a través de un bus de datos,

[1]Clasificación propuesta por E. E. Johnson en 1988.

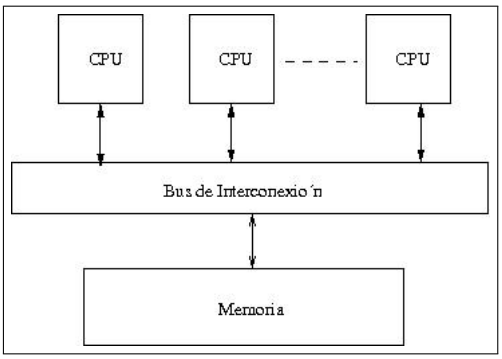


Figura 2.2: Memoria compartida (UMA).

o un switch crossbar[2]. De este modo, todos los procesadores tienen la misma velocidad de acceso a cualquier dirección de la memoria (Figura 2.2).

- NUMA. Acceso No Uniforme a la Memoria (en inglés: Non Uniform Memory Access). Cada procesador tiene su memoria local, pero además puede acceder directamente a memorias remotas. Es no uniforme, porque el tiempo de acceso de un procesador a la memoria local es menor que el tiempo de acceso a la memoria remota, el cual depende de la distancia entre el procesador y la memoria.

Memoria Distribuida

La memoria distribuida está compuesta de un conjunto de computadoras o nodos, las cuales se comunican por medio de la red. Cada nodo es capaz de ejecutar un conjunto propio de instrucciones y acceder únicamente a su memoria local. La comunicación entre los nodos, para leer y escribir en memorias remotas, se da por paso de mensajes, que a diferencia de la memoria compartida, no se puede tener acceso a la memoria de algún otro nodo directamente[6].

Un nodo es capaz de almacenar información en buffers temporales y enviar o recibir datos al mismo tiempo que procesan otros [3]. Los procesadores no comparten memoria y cada uno tiene acceso sólo a su propio espacio de direcciones. Cada nodo necesita dispositivos de entrada y salida para mandar y recibir información a otros nodos a través de la red (Figura 2.3).

2.1.4. Clúster de Computadoras

Los clúster de computadoras son un conjunto de computadoras independientes combinadas en un sistema unificado a través de software y una red de alta velocidad y se comportan como una computadora única[4, 18]. Cada sistema dentro del clúster, definido como nodo, puede ejecutarse independientemente de los demás[18].

[2]Un switch crossbar, es un switch capaz de establecer conexión de múltiples entradas con múltiples salidas.

[3]Comunicación asíncrona.

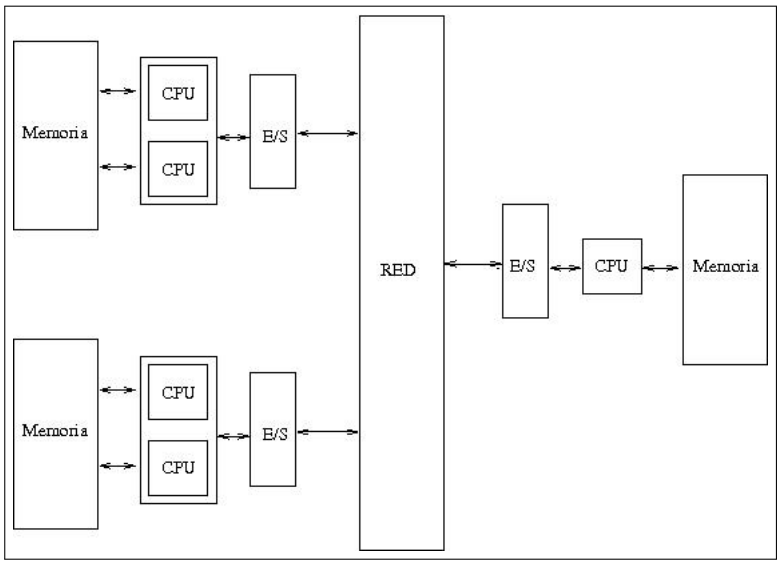


Figura 2.3: Memoria Distribuida.

Son utilizados como un sistema de memoria distribuida de gran poder, que juegan un papel importante en la solución de problemas científicos, de ingenierías y de comercio.

La construcción de un clúster es fácil debido a su flexibilidad: pueden tener todos la misma configuración de hardware y sistema operativo (clúster homogéneo), diferente rendimiento pero con arquitecturas y sistemas operativos semejantes (clúster semi-homogéneo) o diferente hardware y sistema operativo (clúster heterogéneo)[4].

El uso común de los clúster es ejemplificado en simulaciones, biotecnología, modelamientos de mercado financiero, minería de datos, data-streaming y servidores de internet para audio y juegos.

La comunicación entre los nodos del clúster se da generalmente por medio de paso de mensajes, y los programas son comúnmente escritos en C o FORTRAN[4].

2.1.5. Medidas de Desempeño y Eficiencia

La velocidad y eficiencia de un programa paralelo, son medidos acorde al número de procesadores disponibles y no al número de procesos. Para poder realizar las mediciones, se definen los siguientes términos: desempeño, speed-up y eficiencia[7].

Desempeño

El tiempo de ejecución de un programa en p procesadores, definido como T_p, es el tiempo que tarda un programa en ejecutarse. Normalmente se mide en unidades de tiempo como milisegundos.

Speed-up

El speed-up en p procesadores se define como

$$S_p = T_0/T_p$$

donde T_0 es el tiempo del algoritmo serial más rápido en un sólo procesador. Esta fórmula hace una comparación entre un programa paralelo y el algoritmo serial más rápido con el fin de encontrar los beneficios obtenidos al trasladar la aplicación serial a paralela. Aún si este programa paralelo es ejecutado en un sólo procesador (T_1), es de esperarse que el tiempo de ejecución sea mayor que la implementación serial (T_0)[7].

La ecuación

$$\bar{S}_p = T_1/T_p$$

determina el speed-up algorítmico en p procesadores, ganado por la paralelización de un algoritmo, junto con los efectos de sincronización y comunicación. Lo ideal sería que $\bar{S}_p$ creciera linealmente con p. Desafortunadamente, lo mejor que podemos encontrar es que inicialmente la velocidad es casi lineal y eventualmente decrece conforme se utilizan más procesadores y mecanismos de comunicación[7].

Eficiencia

La eficiencia en p procesadores se define como

$$E_p = 100 \times \bar{S}_p/p$$

cuya es un porcentaje y la medida ideal es 100 % de para los p procesadores. En la práctica esta medida decrece conforme aumenta p[7].

2.1.6. Granularidad y Balance de Carga

Granularidad

La granularidad expresa el número de instrucciones ejecutadas antes e utilizar algún mecanismo de sincronización[7]. La granularidad puede ser tan pequeña [4] como una simple operación aritmética o tan grande [5] como un programa. Si la granularidad es muy grande, el paralelismo es reducido, pues las tareas son agrupadas y ejecutadas secuencialmente por un procesador[4].

Balance de Carga

Para optimizar el rendimiento de un programa paralelo es necesario asegurar que todos los procesadores realizan trabajo útil la mayor parte del tiempo posible. Esto quiere decir, que deben ser evitados posibles puntos de sincronización, donde los procesadores esperen información de algún otro procesador. En este sentido, un programa secuencial es totalmente eficiente[7].

[4] Granularidad fina.
[5] Granularidad gruesa.

El *balance de carga* se refiere a la implementación de un algoritmo, el cual emplee la mayor cantidad de procesadores posibles mientras que cada uno de ellos están activos[7].

2.1.7. Comunicación

Dentro de un programa paralelo, las tareas suelen requerir datos de otras, por lo que es necesario contar con una adecuada comunicación entre ellas.

La comunicación entre dos procesos la podemos definir como un canal, donde cada proceso puede enviar y recibir mensajes de otro proceso[6]. La comunicación asociada a un proceso puede ser definida en dos fases. Primero, se define un canal que ligue al proceso con otro. Después se definen los datos que serán enviados y recibidos entre los procesos. Es importante omitir canales y operaciones de comunicación innecesarios, pues el envío y recepción de mensajes implica un costo físico el cual puede perjudicar el desempeño.

Es posible categorizar la comunicación en varias formas: local/global, estructurada/no-estructurada, estática/dinámica y síncrona/asíncrona[6].

Local: Cada proceso se comunica con un conjunto pequeño de procesos ("vecinos").

Global: Cada proceso requiere tener comunicación con todos los procesos.

Estructurado: Un proceso y sus vecinos forman una estructura regular como un árbol o una malla.

No estructurado: La red de comunicación forma una gráfica arbitraria.

Estática: La identidad de la estructura de comunicación no cambia con el tiempo.

Dinámica: La identidad de la estructura de comunicación cambia y es determinada por los datos durante el tiempo de ejecución.

Síncrona: Procesos productores y consumidores ejecutan y cooperan en la transferencia de datos coordinadamente.

Asíncrona: Los procesos consumidores solicitan y requieren datos sin cooperación del productor.

Durante la comunicación asíncrona, los procesos que poseen datos (productores) no son capaces de determinar cuando otros procesos (consumidores) requieren los datos, por lo que los consumidores deben pedir los datos explícitamente[6].

Paso de Mensajes

El paso de mensajes es utilizado para el envío de un objeto (mensaje) de un proceso a otro. Cuando el cliente requiere un recurso o servicio envía un mensaje, que contiene la petición, al servidor. El servidor cumple con la petición y envía de regreso el mensaje con la respuesta[18].

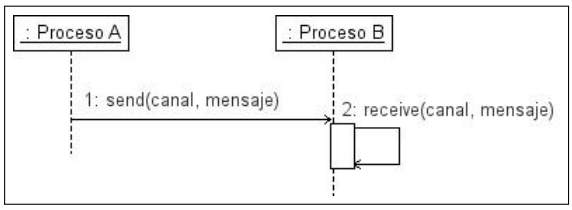


Figura 2.4: Diagrama de secuencias de *paso de mensajes* entre dos procesos.

Un mensaje es considerado como una colección de información y puede ser una instrucción, datos o señales de interrupción[4].

Los mensajes son enviados a través de un canal de comunicación mediante una operación como `send(canal, mensaje)` y recibidos mediante ese canal con una operación como `receive(canal, mensaje)`. Los mensajes pueden ser pasados síncrona o asíncronamente[10].

El paso de mensajes permite obtener datos u objetos entre procesos en computadoras diferentes o incluso en la misma. Supongamos que se tienen dos procesos: A y B (Figura 2.4). El proceso A le envía un mensaje al proceso B mediante la operación `send()`. Este mensaje contiene información sobre lo que solicita A o alguna subrutina que debe ser ejecutada por B. Por su parte el proceso B utiliza la opración `receive()` para obtener le mensaje proveniente de A y poder procesarlo.

Mediante el canal de comunicación, al que se hace referencia en las operaciones `send(canal, mensaje)` y `receive(canal, mensaje)`, los procesos receptor y remitente están ligados entre sí. En lugar de utilizar una dirección de memoria para la comunicación de ambos procesos se utiliza éste canal[10].

El paso de mensajes puede realizarse de tres combinaciones posibles[18]:

Bloqueo de remitente y receptor. El proceso remitente se bloquea hasta que el receptor lo recibe y el receptor se bloquea hasta que el remitente realiza el envío. El paso de mensajes síncrono es llamado también *rendezvous simple*[10, 18].

Bloqueo del receptor sin bloqueo del remitenete. El receptor permanece bloqueado hasta que un mensaje llegue mientras que el remitenete no lo hace. De esta manera un proceso puede enviar mensajes a diferentes destinatarios rápidamente. El proceso que recibe el mensaje tiene que estar bloqueado hasta que recibe el mensaje a procesar[18].

Sin bloqueo de remitente ni receptor. En este caso, ni el receptor ni el remitente se bloquean debido a las operaciones `send` o `receive`.

Los dos últimos tipos de paso de mensajes son asíncronos. En estos casos, puesto que el remitente no se bloquea, el mensaje enviado es encolado en un buffer, de donde el receptor recibe los mensajes.

Llamadas a Procedimientos Remotos

Las llamadas a procedimientos remotos (en inglés Remote Procedure Calls o RPC) permiten a los programas realizar llamadas a procedimientos que se encuentran en otras máquinas. Cuando un proceso en la máquina A realiza una llamada a un procedimiento en la máquina B, el proceso en A es interrumpido para dar ejecución al proceso en B. La información es transportada entre los procesos por medio de los parámetros y obtener información procesada por el resultado del procedimiento[19].

Este tipo de comunicación, es una variante del paso de mensajes[18], y la idea general, es que el llamado de un procedimiento remoto parezca una llamada a un procedimiento local[18, 19].

Un procedimiento *stub*[6] P debe ser incluido en el espacio de direcciones del proceso que realiza la llamada. Este procedimiento crea un mensaje que identifica el procedimiento a ser llamado e incluye los parámetros dentro del mensaje[18]. Enseguida, el proceso cliente solicita el envío del mensaje al servidor bloqueandose hasta obtener la respuesta[19].

Del lado del servidor, cuando un mensaje llega, se realzia una llamada local al procedimiento P[18]. De este modo el servidor realiza el procesamiento y envía el resultado al cliente, desbloqueandose y continuando con su trabajo[19].

De igual manera que en el paso de mensajes tradicional, las llamadas a procedimientos remotosse pueden realziar de manera síncrona o asíncrona. Tradicionalmente, se realiza de manera síncrona, donde el proceso cliente se bloquea hasta que el procedimiento llamado regrese un valor[18].

El esquema tradicional petición-respuesta es innecesario cuando no hay resultado que regresar y que no hay trabajo importante que mostrar al cliente, creando así llamadas a procedimientos remotos asíncronos. Con las llamadas asíncronas, el servidor inmediatamente regresa respuesta al cliente al momento de invocar la llamada por el cliente. La respuesta le sirve al cliente para saber que el servidor procede a ejecutar el procedimiento llamado[19].

2.1.8. Partición del Problema

El propósito de la partición de un problema es crear la mayor cantidad de pequeñas tareas del problema. Una partición óptima divide el problema tanto en los cálculos asociados al problema como en los datos a operar. En esta subsección se revisan tres tipos de particiones: partición de dominio, funcional y de actividad[1, 6].

[6]Este tipo de procedimiento es llamado, en inglés, como *stub*[19] o *procedure stub*[18]. Significa una pequeña rutina que sustituye a un programa más grande. En otras palabras, puede simular el comportamiento de código existente.

Partición de Dominio

En este tipo de partición, la atención es puesta principalmente sobre el dato asociado al problema, dividiéndolo en pequeñas piezas de igual tamaño[6] donde cada pieza es asociada a un proceso diferente. De esta manera, cada proceso trabaja únicamente con el dato asignado. Al realizar estas tareas separadamente, cada proceso puede requerir información de otro proceso, por lo que la comunicación entre los procesos es necesaria para mover datos e información entre ellos.

Como este tipo de partición está enfocado únicamente a dividir los datos del problema, podemos definirlo como un sistema SIMD, extendiendo la clasificación de Flynn al contexto del problema a resolver.

Partición Funcional

El enfoque de este tipo de partición es la división de las operaciones a realizar en lugar de los datos del problema, generando un número de operaciones independientes más pequeñas[6].

Como ejemplo, este tipo de partición es utilizado para modelar el clima terrestre, donde se comprenden factores como océano, atmósfera hidrología entre otros. Cada componente puede ser paralelizado mediante partición de dominio, aunque resulta más simple si, primero se descompone el problema mediante técnicas de partición funcional. En este ejemplo, cada tarea realiza diversos cálculos, como , presión atmosférica, movimiento marítimo, velocidad de viento, temperatura de la superficie marítima, etc., y mediante comunicación entre las tareas, se transfieren e interactúan entre sí los datos resultantes de cada una de las tareas computadas: la presión atmosférica genera velocidad del viento, dato que es utilizado para obtener el movimiento del océano, que a su vez, sirve para generar la temperatura de la superficie marítima, etc[6].

El hecho que la división del problema esté enfocado en las operaciones que se realizan, este tipo de partición se puede definir como un sistema MISD, extendiendo la clasificación de Flynn al contexto del problema a resolver.

Partición de Actividad

Se realiza partición tanto de los datos a operar como los algoritmos a utilizar. De esta manera se realizan diferentes operaciones en diferentes datos simultáneamente[1].

Extendiendo la clasificación de Flynn al contexto del problema a resolver, se pude definir este tipo de partición como un sistema MIMD debido a que se utilizan diferentes instrucciones y datos a operar.

2.1.9. Mapeo de Procesos

En el mapeo de procesos, se define dónde se ejecuta cada proceso. La meta principal es minimizar el tiempo de ejecución total del programa paralelo. Para lograr esto se siguen dos estrategias[6]:

1. Asegurar que procesos capaces de ejecutarse independientemente sean ejecutados en diferentes procesadores.

2. Disponer a los procesos que se comunican frecuentemente en el mismo procesador o en procesadores con conexión directa entre sí.

Las limitantes de los recursos pueden restringir el número de procesos que pueden tomar lugar en un sólo procesador.

El problema de mapeo es NP-Completo. Aunque como alternativa se han utilizado diferentes estrategias y heurísticas efectivas. Varios algoritmos de mapeo utilizan descomposición de dominio además de un número fijo de procesos iguales y comunicación estructurada, local y global[6].

Para obtener un mejor balance de carga en algoritmos basados en descomposición de dominio más complejos, es necesario contar con algoritmos de balance de carga los cuales se encargan de mapear y aglomerar los procesos con los procesadores, comúnmente usando heurísticas que ayuden a determinar el periodo de ejecución de estos algoritmos. Los métodos probabilísticos de balance de carga tienden a ser menos costosos que los que explotan una estructura en una aplicación[6].

En los problemas en los que la cantidad de procesos o la cantidad de cálculos cambian dinámicamente, es conveniente utilizar balance de carga dinámica, en la que se determina un nuevo mapeo y aglomeración, de acuerdo a las necesidades del problema[6].

Las implementaciones basadas en descomposición funcional generalmente se coordinan entre procesos sólo al inicio y al final de la ejecución de la aplicación. Para estos casos es posible usar algoritmos de planificación de tareas, que asignan procesos a procesadores inactivos[6].

2.2. Diseño de Software Paralelo

Para la solución de un problema mediante programación paralela se tienen en cuenta dos factores:

- *Problema de software* que se refiere a la creación de software que contiene algoritmos aplicados a datos para la solución de algún problema[1].

- *Problema de paralelización* que se refiere a la descripción de la solución de la ejecución simultáneamente de múltiples procesos secuenciales[[1] et al.] para obtener una ejecución más eficiente del programa por medio del procesamiento y la comunicación paralela entre los procesos. Para este caso, la programación paralela es viable cuando la solución al problema es conocida, por lo que es necesario resolver primero el problema de software[1].

El diseño de software paralelo "*propone técnicas para tratar con el problema de paralelización*"[1] proveyendo formas de organizar el software de tal manera que se pueda

hacer uso de múltiples procesadores eficientemente[1].

Diversas propuestas de descripciones organizacionales han sido propuestas para el diseño de software paralelo que proveen estructuras globales y tratan de organizar los algoritmos y los datos a operar. Estas propuestas presentan las siguientes vetajas[1]:

- Complejidad de programación no muy grande.
- Simplifica la comunicación entre los componentes de software reduciendo las sobrecargas por sincronización.
- Permite entender la estructura general de un programa paralelo.

Una de estas propuestas de descripciones son patrones de software paralelo[1].

2.2.1. Patrones de Software Paralelo

Los patrones de software "*son descripciones de organizaciones regulares de los componentes del software*"[1] que permiten entender cómo está diseñado y organizado un sistema de software y proveen la estructura fundamental del programa, especificando propiedades y responsabilidades de los subsitemas además de la forma en que están unidos[1].

Mediante los patrones de software paralelos se proveen técnicas para la implementación de software paralelo que permitan la ejecución del programa en el menor tiempo posible. También permiten identificar cómo realizar las operaciones de los datos simultáneamente[1].

Un patrón de software paralelo reprsenta el punto de partida dentro del diseño de un programa paralelo. No obstante, un patrón de software no produce un programa paralelo ni información sobre el desempeño[1].

La implementación de un patrón de software está organizada en cuatro etapas[1, 6]:

1. Partición
2. Comunicación
3. Aglomeración
4. Mapeo

Durante las primeras dos etapas, se centra la atención en la concurencia y escalabilidad. Durante las últimas dos, se centra la atención en cuestiones de rendimiento[1, 6].

A continuación son mencionados algunos patrones de software paralelos.

Pipes and Filters

Cada componente paralelo efectúa una operación diferente simultáneamente siguiendo un orden estricto de las operaciones y datos.

El procedimiento inicia cuando el primer conjunto de datos está disponible y es procesado en la primera etapa del `pipe`. Cuando ha terminado el procesamiento, el resultado es pasado a otra etapa del pipe mientras que la primera es capáz de recibir datos nuevos[1].

El canal de comunicación, en este patrón de software paralelo, es el `pipe`, que se encarga de transmitir los datos y efectuar la sincronización entre procesos. Cada proceso que se encarga de generar datos nuevos o recibir datos del pipe, efectuar alguna operación y enviar el resultado, es llamado `filter`[1].

Este patrón sigue el ejemplo de una línea de ensamblaje de autos, por ejemplo, donde la línea de ensamblaje puede ser vista como el pipeline y cada estación de ensamblaje como un filter. En este ejemplo cada línea de ensamblaje efectúa una operación (armado de motor, carrocería, tapicería, pintura) al mismo tiempo[1].

Como el pipeline requiere de datos y procedimientos ordenados, el filter n requiere de procesamiento previo por parte del filter $n-1$. La clave del paralelismo en este patrón es que el filter n puede procesar datos al mismo tiempo que el filter $n+1$ y que eel filter $n-1$, claro está, que los datos que operan cada filter, son distintos[1].

La figura 2.5 muestra el esquema general del patrón de software *Pipes and Filters*

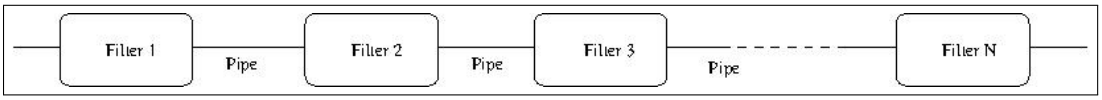


Figura 2.5: Patrón de Software *Pipe and Filters*

Communicating Sequential Elements (CSE)

Cada componente realiza las mismas operaciones con distintos datos, donde las operaciones en cada uno de los componentes dependen de los resultados de otros componentes. Este patrón se puede visualizar como una malla rectangular.

En este patrón hay dos elementos[1]: el elemento secuencial, donde se realiza el cómputo de los datos y proveen un mecanismo para el envío y recepción de mensajes; y los canales de comunicación, que son el medio de envío y recepción de mensajes así como la sincronización de ellos.

Cada uno de los elementos secuenciales realiza la misma operación y envían sus resultados parciales a los vecinos cercanos a través de un canal de comunicación. Todos los componentes reciben los resultados de los vecinos. Una vez terminada la recepción y sincronización, cada componente continúa con los siguientes cálculos y continúa este ciclo hasta que cada uno de los componentes haya terminado[1].

La Figura 2.6 muestra el esquema general de este patrón de software paralelo.

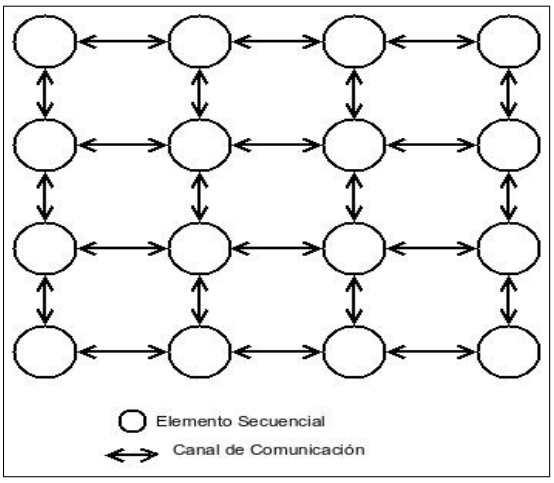


Figura 2.6: Patrón de Software *CSE*.

Manager-Workers

En este patrón de software paralelo se conceptualizan dos elementos principales[1]: el director (manager, por su término en inglés, que será utilizado en futuras referencias) y los trabajadores (workers, por su término en inglés, que será utilizado en futuras referencias). Este esquema permite que cada proceso pueda realizar la misma operación.

La tarea del manager es crear el número de workers, realizar la partición del problema, iniciar la ejecución y recopilar los resultados para unirlos y obtener el resultado final. A los workers, por otro lado, les corresponde pedir las tareas al manager, realizar los cálculos y mandarle los resultados, solicitando más datos. Todos los workers realizan la misma operación simultáneamente con sus datos propios y no existe comunicación entre ellos, por lo que el manager es el encargado de la comunicación y sincronización de procesos y datos.

La Figura 2.7 muestra un esquema del funcionamiento de este patrón de software paralelo.

Shared Resource

Los componentes que actúan en este patrón son capaces de efectuar diferentes operaciones en diferentes conjuntos de datos simultáneamente[1].

Los diferentes componentes ejecutan una operación secuencial diferente con diferentes datos. El recurso compartido (shared resource) mantiene la integridad de los datos, mediante operaciones de sincronización, cuando cada componente requiere leer/escribir datos almacenados en él. Cada componente se comunica únicamente con el recurso compartido pidiendo algún dato. El recurso compartido puede proveer de este dato si no hay algún otro componente utilizándolo. Desde luego, ningún dato puede ser utilizado al mismo tiempo por varios componentes para evitar incoherencias en los resultados. La utilización de este patrón no requiere la existencia de algún orden específico ni de datos ni de funciones a realizar[1].

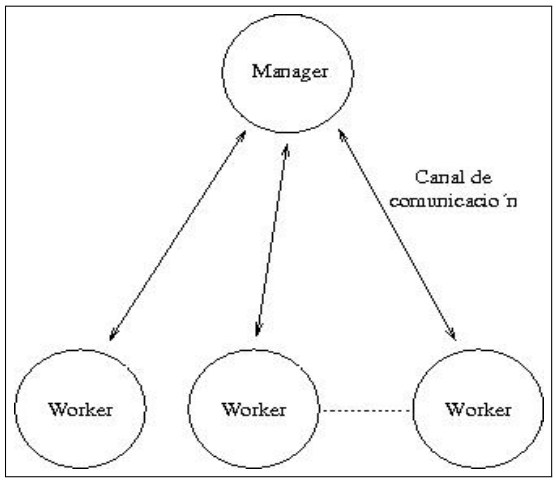


Figura 2.7: Patrón de software *Manager Workers*

En términos generales, este patrón opera de manera similar a Manager-Workers (2.2.1). La diferencia es que en Manager-Workers, cada worker realiza la misma operación sobre diferentes datos, mientras que en este patrón cada componente es capáz de realizar diferentes operaciones sobre diferentes datos.

2.3. Imágenes Digitales

La representación de una imagen digital está dada por una matriz de M filas por N columnas, y puede ser descrita como una función[8]:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

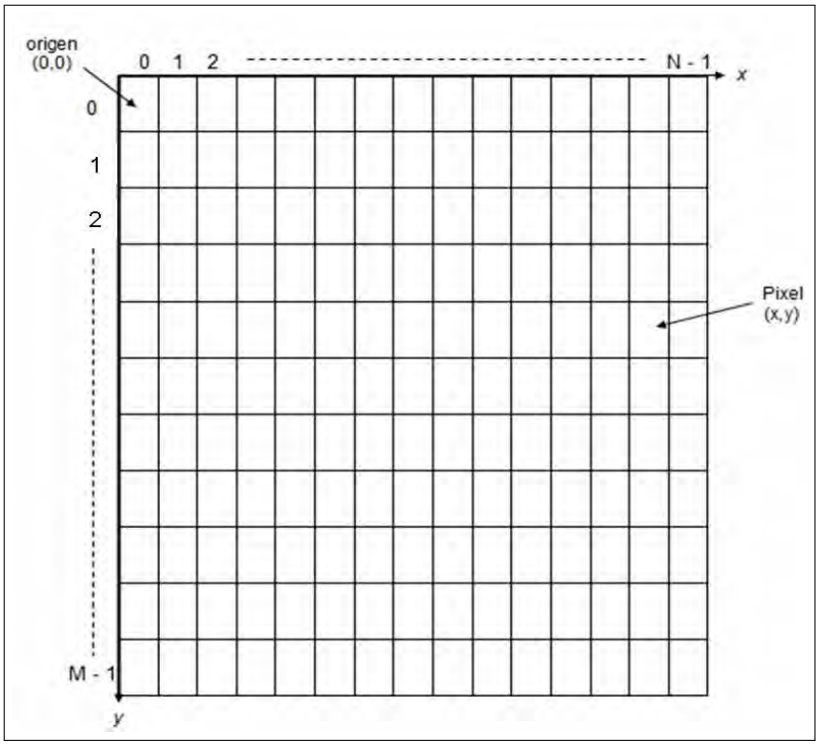


Figura 2.8: Representación discreta de una imagen.

A cada elemento de la matriz se le denomina pixel [7] cuyo valor es proporcional al brillo del punto correspondiente en la escena[13]. Cada pixel está formado por m bits que permite tener 2^m niveles de brillo (desde 0 hasta $2^m - 1$). Si m es igual a 8 se obtienen 256 niveles de brillo (de 0 a 255) obteniendo así una imagen en *escala de grises* de 8 bits, como se muestra en la Figura 2.9. Este valor de m es considerado como valor ideal, el cual esta relacionado con la señal de la amplitud de banda de la cámara[13] con la que se toma la imagen.
Las imágenes a color están representadas con 3 componentes distintos de intensidad, a diferencia de las imágenes en escala de grises, que están representadas sólo con un componente de intensidad.

[7] Abreviatura del inglés de "*picture element*".

Figura 2.9: Imagen en escala de grises de 300×300 pixeles, donde cada pixel está compuesto de 8 bits.

2.3.1. Histogramas

Los histogramas muestran cómo se comporta la intensidad de brillo de una imagen. Este comportamiento es modelado por una gráfica donde el eje x muestra los niveles de brillo de la imagen y el eje y muestra el nivel de brillo[13]. Matemáticamente, el histograma para una imagen a escala de grises es representado por una función $h : [0, L-1] \to \mathbb{N}$ $h(r_k) = n_k$ donde L [8] representa la totalidad de tonos de grises, r_k es el k-ésimo nivel de gris y n_k el número de pixeles en la imagen con esa tonalidad[8].

La manipulación de los histogramas es usada para mejorar la imagen además da información estadística sopbre ella[8]. Intuitivamente, una imagen cuyo histograma tiende a ocupar todo el rango de posibles tonos de grises exhibe una gran variedad de tonos de grises. Un histograma cuyos valores de tonos de gris están cargados a valores cercanos a cero, revela una imagen oscura. Análogamente, un histograma cuyos valores de tonos de gris están cargados a valores cercanos a L, revela una imagen muy clara o con alta luminosidad.

2.3.2. Convolución de Imágenes

La convolución de imágenes es el proceso en el que se interpreta un pixel por la forma que toma de acuerdo a sus vecinos[5]. Los pasos a seguir para realzar la convolución son los siguientes:

1. Por cada pixel en la imagen original, el pixel resultante es el resultado de una función de los valores de los vecinos del pixel original.

2. Continuar el primer paso hasta que la imagen haya sido completada en la nueva imagen.

Diversos operadores utilizan como función la suma ponderada incluyendo el pixel central, en la que los factores están dados por una matriz cuadrada que se le denomina

[8] En imágenes a escala de grises de 8 bits por pixel, L tiene un valor de 256.

máscara [9]. La ecuación matemática[3, 8] para la convolución para una máscara de $n \times m$ se muestra en la ecuación 2.1.

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.1}$$

La Figura 2.10 muestra la disposición de las coordenadas de una máscara. El centro de esta máscara está en $w(0,0)$. El valor de a de la ecuación 2.1 es $(m-1)/2$, donde m es número de columnas de la máscara. De manera análoga, el valor de b de la ecuación 2.1 es $(n-1)/2$, donde n es el número de filas de la máscara. Esta ecuación realiza la suma ponderada, donde el nuevo valor del pixel central es la suma de cada producto de los pixeles vecinos por el coeficiente respectivo de la máscara dada, incluyendo el valor antiguo del pixel central[8].

w(-1, 1)	w(-1, 0)	w(-1, 1)
w(0, -1)	w(0, 0)	w(0, 1)
w(1, -1)	w(1, 0)	w(1,1)

Figura 2.10: Máscara de 3×3, donde se muestra la disposición de las coordenadas.

Los valores de estas máscaras son elegidos en base a estas reglas[5]:

- La suma de los valores de la matriz es positiva o igual a cero. Si la suma es cero, se eliminan pixeles ajenos (generados por ruido). Una suma negativa expone dichos pixeles.
- Usar ceros donde el color es ambivalente.
- Usar valores negativos donde no se desea tener color.
- Usar valores positivos donde no se desea un valor distinto de cero.

Por ejemplo, en La Figura 2.11(a) se detectan sólo los bordes verticales que se encuentran en una transición de oscuro a claro de la imagen. Intercambiando 1 por -1 y viceversa, se detectan los bordes verticales que se encuentran en una transición de claro a oscuro. De igual manera, en la Figura 2.11(b), se detectan los bordes horizontales que se encuentran en una transición de oscuro a claro en la imagen. Intercambiando 1 por -1 y viceversa, se detectan los bordes horizontales que se encuentran en una transición de claro a oscuro[5].

[9] Este término puede ser referido por algunos autores como *operador*, *plantilla* o *kernel*.

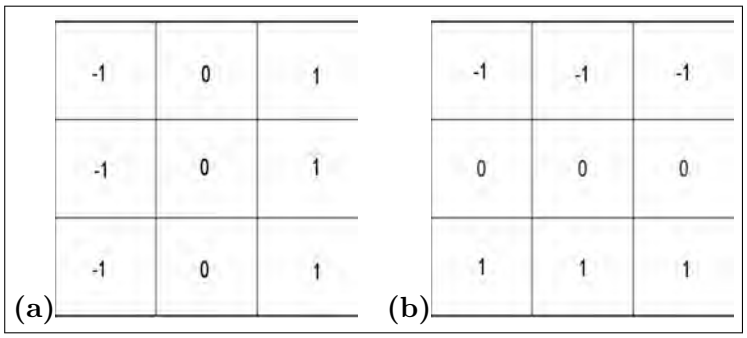


Figura 2.11: Matrices que muestran detección de bordes: **(a)** bordes verticales que se encuentran en una transición oscuro-claro; **(b)** bordes horizontales que se encuentran en una transición oscuro-claro.

2.3.3. Segmentación de Imágenes

La segmentación tiene como finalidad la obtención de objetos homogéneos de la imagen y separarlos o segmentarlos del resto de ella, para simplificar su representación en algo más manejable o simple de manipular[17]. Puede ser definida además como el proceso de agrupar pixeles con atributos similares. El mayor reto de la segmentación o participación de imágenes es la definición de características que son únicas de cada región, y que puedan ser usadas para separar una región de otra.
Algunas técnicas de segmentación pueden ser encontradas en cualquier aplicación que involucre detección, reconocimiento y medida de objetos en una imagen[3]. Algunos ejemplos son:

- Revisión industrial.
- Reconocimiento de patrones.
- Rastreo de objetos en secuencias de imágenes.
- Clasificación de terrenos visibles en imágenes de satélites.
- Detección y medida de huesos, órganos, etc. en imágenes médicas.

El primer reto en la segmentación de imágenes[17], es definir características únicas para separarlas de otras. Esto puede ser realizado de diferentes maneras incluyendo intensidad de la imagen, color, luminosidad y textura. Una vez determinado, el siguiente reto es encontrar el mejor método para capturar éstas características y manejarlas para partir la imagen. Cualquier problema con la imagen como ruido o datos perdidos o erróneos añade problemas adicionales a la segmentación.

2.3.4. Detección de Bordes

La detección de bordes en una imagen permite encontrar los objetos, su figura, tamaño y algo de su textura. Estos bordes son un conjunto de pixeles conectados y delimitan dos regiones. Como se observa en la parte superior de la Figura 2.12, un borde

se encuentra donde el nivel de gris de la imagen cambia de un valor bajo a uno alto y viceversa. Este borde se encuentra al centro de esta transición[16]. Matemáticamente, el borde encontrado es la derivada esta función. El borde encontrado muestra un punto resaltante en el borde, mientras el resto se muestra más oscuro, como se muestra en la parte inferior de la Figura 2.12.

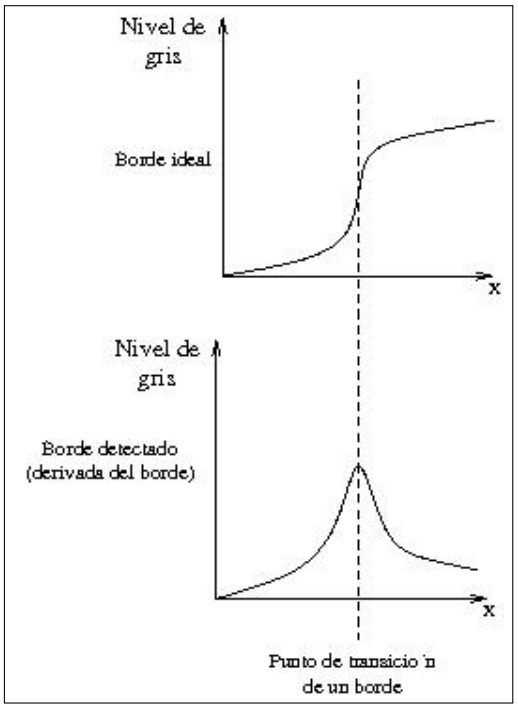


Figura 2.12: Valores de grises en un borde.

Derivadas de Primer Orden

El interés de las derivadas se centra en el comportamiento que tienen en áreas donde el nivel del tono gris es constante al inicio o final de las discontinuidades (desniveles)[8], y a lo largo de los desniveles entre los tonos de gris. Las derivadas de una función digital, en este caso, la imagen, son definidas en términos de diferencias[8]. En el caso unidimensional, la derivada de la función $f(x)$ es

$$\frac{\partial f}{\partial x} = f(x+1) - f(x). \tag{2.2}$$

Para el caso bidimensional, el cual presenta la función de una imagen, la primera derivada está basada en aproximaciones del gradiente 2-D[8]. El gradiente de una imagen $f(x,y)$ en el punto (x,y) es el vector:

$$\nabla \mathbf{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.3}$$

que nos permite obtener la máxima tasa de cambio de f en el punto x, y, donde

$$\frac{\partial f}{\partial x} = f(x+1, y) - f(x, y) \tag{2.4}$$

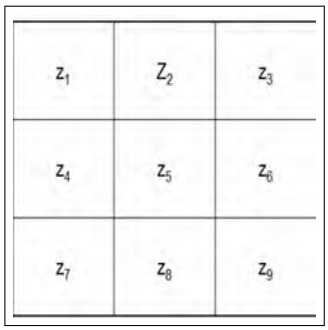


Figura 2.13: Máscara de 3×3.

y

$$\frac{\partial f}{\partial y} = f(x, y+1) - f(x, y). \tag{2.5}$$

La magnitud del vector ∇f da la máxima tasa de crecimiento de $f(x, y)$ por unidad de distancia en dirección $\nabla \mathbf{f}$[8]:

$$\nabla f = mag(\nabla \mathbf{f}) = \sqrt{G_x^2 + G_y^2}. \tag{2.6}$$

Esta magnitud es referida frecuentemente como el gradiente[8]. También es común realizar una aproximación de la magnitud utilizando valores absolutos en lugar de raíces cuadradas, para obtener cálculos más sencillos:

$$\nabla f \approx |G_x| + |G_y|. \tag{2.7}$$

Las aproximaciones al gradiente son dadas por máscaras y las cuales se convolucionan en la imagen entera. Sustituyendo las ecuaciones 2.4 y 2.5 en la ecuación 2.3, se sugiere la utilización de una máscara de 2×2 para la aproximación de la derivada de primer orden, aunque no son comúnmente utilizadas pues carecen de un centro claro. En lugar de ellas se utilizan máscaras de 3×3, como se muestra en la Figura 2.13, que muestra el centro etiquetado con z_5. De ahí es posible calcular G_x y G_y:

$$G_x = z_8 - z_5 \tag{2.8}$$

y

$$G_y = z_6 - z_5 \tag{2.9}$$

y por lo tanto, la ecuación 2.6 queda como sigue:

$$\nabla f = [(z_8 - z_5)^2 + (z_6 - z_5)^2]^{\frac{1}{2}}. \tag{2.10}$$

La aproximación más adecuada con la máscara de 3×3 está dada por las siguientes ecuaciones:

$$G_x = (z_7 + z_8 + z_9) - (z_1 + z_2 + z_3) \tag{2.11}$$

$$G_y = (z_3 + z_6 + z_9) - (z_1 + z_4 + z_7) \tag{2.12}$$

La diferencía entre las tercera y primer fila (Ecuación 2.11) aproxima a derivada respecto al eje x y la diferencia entre la tercera y primer columna (Ecuación 2.12) aproxima a la derivada respecto al eje y. Los valores y tamaños de estas máscaras varían, aunque los tamaños son generalmente de $N \times N$ con N impar. Las máscaras más conocidas son de 3×3 y corresponden a los operadores de *Prewitt* y *Sobel*.

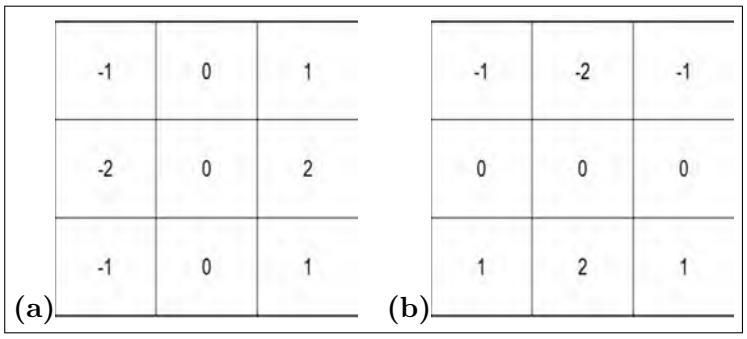


Figura 2.14: Máscaras de Sobel. En **(a)** se muestra la máscara para calcular G_x. En **(b)** la máscara para calcular G_y.

2.3.5. Detección de Bordes con el Operador de Sobel

La máscara propuesta por Sobel se muestra en la Figura 2.14.

Con estas dos máscaras y aplicando la ecuación de aproximación de primera derivada 2.6 se tiene el pseudo código para la detección de bordes, mostrado en el listado 2.2. La función de convolución recibe los siguientes parámetros:

- La imagen fuente, que es donde obtiene la información de los pixeles.
- La matriz de convolución, que puede ser alguna de las mostradas en la Figura 2.14.
- La coordenada (x, y) del pixel central,sobre el que se realiza la suma ponderada. En otras palabras, la coordenada del pixel al que se calcula el gradiente.

y regresa el gradiente o suma ponderada del pixel central.

La función Sobel recibe como parámetro la imagen a la que se desea detectar bordes y regresa la imagen resultante, que contiene únicamente los bordes.

```
1  function convolucion(I, K, posX, posY)
2     sum = 0;
3     kwidth = width(K)
4     kheight = height(K)
5     for x = 0 to kwidth do
6          for y = 0 to kheight do
7               coordX = posX - 1 + x
8               coordY = posY - 1 + y
9               if coordX in (0, width(I)) and coordY in (0, height(I)) then
10                   sum = sum + I(coordX, coordY) * K(x, y)
11              end
12         end
13    end
14    return sum
15 end
```

Programa 2.1: Algoritmo de convolución.

```
function Sobel(I)
   w = width(I)
   h = height(I)
   for x = 0 to w - 1 do
       for y = 0 to h - 1 do
           gradX = convolucion(KX, x, y, I)
           gradY = convolucion(KY, x, y, I)
           Inueva(x, y) = sqrt(gradX * gradX + gradY * gradY)
       end
   end
   return Inueva
 end
```

Programa 2.2: Algoritmo para detección de bordes con el operador de Sobel.

Generalmente se necesita de un valor denominado *umbral* el cual permite decidir si un pixel pertenece a un borde o no. Si la magnitud del gradiente es mayor al umbral, entonces se ha encontrado un borde, de lo contrario, pertenece al fondo de la imagen. Este valor es elegido casi siempre por algún usuario, quien variando el valor del umbral (de 0 a 255) puede determinar cuál es el resultado óptimo. En otras ocasiones el umbral puede ser tomado automáticamente explorando el histograma de la imagen o con diferentes técnicas. Ninguna de estas técnicas de obtención automática del umbral es abarcada en este trabajo, pues no está en el objetivo.

Pero también es posible no utilizar ningún umbral y utilizar únicamente el valor dado por la magnitud del gradiente, asegurándose que ese valor no sea menor a 0 ni mayor a 255. Para este trabajo es utilizado este método, es decir, no se toma en cuenta ningún umbral que discrimine.

Las bases para la elección del operador de Sobel para la detección de bordes en imágenes, en este trabajo, son que ofrece un mejor desempeño que otros detectores de bordes contemporáneos como el de Prewitt[13] y tiene una mayor omisión de ruido que otros filtros, incluyendo los de derivadas de segundo orden, que son aún más sensibles al ruido que los demás filtros basados en primera derivada.

2.4. Resumen

En este capítulo se introdujeron conceptos básicos de cómputo paralelo y distribuido y de procesamiento de imágenes.

Para el desarrollo de un programa paralelo es importante conocer la manera en que se puede organizar la memoria, ya sea de forma distribuida o compartida. La memoria compartida generalmente implica que todos los procesadores se encuentran dentro de una misma computadora y tienen la misma oportunidad de acceder a la memoria global (*UMA*). En el esquema *NUMA*, cada procesador tiene su propia memoria local, pero además, puede acceder directamente a la memoria local de otros procesadores. El hecho de tener memoria local por procesador permite reducir el número de accesos a memoria y el tiempo de espera debido al conflicto por obtener un dato de un recurso compartido, que en el caso de UMA, es la memoria global, permitiendo en NUMA que cada procesador acceda a memoria paralelamente.

En los sistemas de memoria distribuida (como los clúster), cada nodo cuenta con su conjunto de procesadores y su memoria local.y el acceso a las memorias remotas se da por *paso de mensajes*, el cual puede ser *síncrono* si el receptor se bloquea hasta que el remitente recibe el mensaje y viceversa o *asíncrono* en caso contrario. El paso de mensajes es un mecanismo de comunicación que permite la sincronización entre procesos.

Otro paso importante es la participación del problema. Si sólo se divide el problema en los datos a operar es *partición de dominio*. Si sólo se divide el problema en las operaciones o funciones que se realizarán la división es *partición funcional*. También es posible tener partición funcional y de dominio al mismo tiempo. La partición de dominio permite establecer qué patrón de software es posible utilizar para la resolución de un problema. Por ejemplo, si se decide utilizar partición de dominio, *Manager Workers* es una buena elección, en cambio para partición funcional y de dominio a la vez, es mejor utilizar *CSE*.

En Manager Workers existen dos participantes: en *manager* que se encarga de realizar la partición de datos y enviar una porción a cada *worker* mediante un canal de comunicación, donde cada uno realiza la misma tarea pero con distintos datos. Al final de este proceso, el manager se encarga de organizar el resultado final, con los resultados parciales de cada worker.

En CSE, cada nodo realiza una operación secuencial posiblemente distinta a la de otro nodo, pero cada uno de los datos a procesar depende del resultado del procesamiento de sus nodos vecinos.

Por otro lado, se habló de segmentación de imágenes digitales. La segmentación de imágenes permite conocer más a detalle objetos dentro de una imagen digital. Por la cantidad de cálculos requeridos en este tipo de procesos, la utilización de programas paralelos es favorable, dividiendo así la carga de trabajo entre varios procesadores.

Uno de los métodos de segmentación más utilizados es la *detección de bordes*, La base de la detección de bordes se encuentra en el gradiente de una imagen, que puede ser interpretada como una función $f(x, y) = A$, donde A es una matriz de $n \times m$. El gradiente permite conocer la máxima tasa de cambio del color en un punto específico. Las imágenes digitales tienen que ser tratadas discretamente, por lo que la derivada está definida en términos de diferencias. Las aproximaciones a este gradiente están dadas por máscaras o filtros, que son matrices de 3×3. Las más conocidas son las de Sobel y Prewitt, pero la de Sobel s menos sensible al ruido por lo que permite obtener mejores resultados.
Además de las máscaras, la detección de bordes necesita un algoritmo para aplicarlas: convolución de imágenes, que es la suma ponderada donde el pixel central y cada uno de los vecinos son multiplicados por coeficientes dados por las máscaras. En otras palabras, el valor de un pixel esta relacionado con sus vecinos y la máscara que se le aplica.

Capítulo 3

Trabajo Relacionado

En este capítulo se presenta el artículo *Parallel Multi-scale Feature Extraction and Region Growing: Application in Retinal Blod Vessel Detection* [14] cuyo panorama está relacionado con el cómputo paralelo y segmentación de imágenes, con el fin de obtener bordes de vasos sanguíneos en imágenes retinales. El algoritmo de segmentación utilizado proviene de los artículos *Segmentation of blood vessels from red-free and fluorescein retinal images* [12] e *Improvement of a retinal blood vessel segmentation method using the insight segmentation and registration toolkit (ITK)* [11].

El objetivo consiste en automatizar y minimizar el procesamiento de las imágenes retinales mediante técnicas de segmentación de imágenes y programación paralela, para obtener los pixeles con la información más importante de la imagen.

3.1. Segmentación de Imágenes Retinales

Para la detección de bordes que representan vasos sanguíneos retinales en la imagen, se utiliza convoluciona la imagen utilizando un filtro. En el artículo referido [14], para la convolución de la imagen retinal se utiliza como filtro la función gaussiana (la cual no se aborda, pues esta fuera del objetivo del trabajo) que permite discriminar la clase de un pixel.

La segmentación de la imagen es realizada en dos etapas: extracción de características y crecimiento de región.

3.2. Implementación Serial

3.2.1. Extracción de Características

En este paso se definen los pixeles de la imagen, que representan las clases de pixel, mediate discriminación efectuada en dos operaciones[11, 12, 14]:

- Se obtiene la magnitud del gradiente resultante al convolucionar cada pixel de la imagen con la función gaussian. La magnitud representa la pendiente de la inten-

sidad de la imagen para un pixel en particular.

- Después se calcula el máximo eigenvalor de la matriz Hessiana, dada por la segunda derivada del pixel convolucionado con la función gaussiana. Si el máximo eigenvalor es mucho mayor a uno, el pixel pertenece a una región de vaso sanguíneo.

Teniendo esta información se puede proceder al siguiente paso del algoritmo.

3.2.2. Crecimiento de Región

En esta etapa del algoritmo se definen dos clases de pixel: aquellos que pertenecen al fondo de la imagen y aquellos que representan un vaso sanguíneo. Estas clases son determinadas mediante el máximo eigenvalor[11, 12, 14].

El crecimiento de región inicia detectando los pixeles cuya magnitud del gradiente es baja. Estos pixeles, denominados semillas, pueden representar el fondo de la imagen o un vaso sanguíneo. Las semillas crecen basándose en el valor del máximo eigenvalor que indica a que clase pertenecen los pixeles y en la información dada por la clasificación de sus ocho vecinos[11, 12, 14].

En este paso se definen los pixeles que delimitan a la representación de un vaso sanguíneo con los que representan el fondo de la imagen.

3.3. Implementación Paralela

La implementación paralela del algoritmo utiliza división de dominio: la imagen a segmentar es dividida en subimágenes y cada una es procesada con el mismo algoritmo, pero por diferentes procesadores. Esta división se aplica para las dos etapas del algoritmo[14].

3.3.1. Extracción de Características

El cálculo de la magnitud del gradiente y del máximo eigenvalor no involucra dependencia de otros pixeles sino sólo del procesado. Esto permite tener una división de la imagen en subimágenes y que cada procesador extraiga las características sin depender de comunicación con otros procesadores.

La imagen final es la combinación de todas las subimágenes combinadas, pero se pueden obtener falsos bordes en los límites de las subimágenes debido a la falta de información que está contenida en las demás subimágenes.

Para evitar esto, cada subimagen contiene cierta cantidad de pixeles de las subimágenes vecinas. Esta cantidad de pixeles se empalman al combinar las subimágenes en la imagen final, por lo que esta región debe ser eliminada en la imagen final. [14]

3.3.2. Crecimiento de Región

En esta etapa, el procesamiento de cada pixel requiere de la información de sus pixeles vecinos. Si la división de la imagen es horizontal, el crecimiento de aquellas regiones que crecen horizontalmente no tienen problema, aunque no sucede lo mismo con los vasos que crecen verticalmente, pues al truncar la imagen se pierde información de los pixeles vecinos inferiores o superiores, ya que se encuentran en otra subimagen. Análogamente se tiene esta situación si la imagen es dividida verticalmente.

Para evitar este problema y la saturación de canales de comunicación, la imagen original es dividida horizontalmente y verticalmente. Primero se divide en una dirección y se realiza el procesamiento. Después en la otra dirección y se realiza el procesamiento. De esta manera se obtienen dos imágenes, las cuales pueden contener pixeles representantes de vasos sanguíneos, que se encuentren en la imagen dividida horizontalmente, pero no en la dividida verticalmente y viceversa. Para obtener el resultado correcto, se aplica la operación lógica OR pixel con pixel en ambas subimágenes. [14]

Patrón de Software

El patrón que se utiliza es Manager-Workers. El nodo manager se encarga de realizar la partición de la imagen en distintas subimágenes y de repartir cada una de ellas a los nodos workers, quienes realizan el procesamiento de imágenes.

En ambas etapas del algoritmo, cada uno de los nodos worker realiza el procesamiento, ya sea extracción de características o crecimiento de región, sobre la subimagen que le corresponde. Este procesamiento se realiza sin comunicación con algún otro nodo worker, por lo que cada nodo es independiente entre sí.

Cuando un nodo worker ha terminado su tarea, envía la subimagen procesada al manager. El nodo manager espera tener todas las subimágenes procesadas para poder unirlas y dar por terminada la etapa del algoritmo (extracción de características o crecimiento de región).

3.3.3. Speed-up

El artículo referido [14] indica que obtiene el speed-up incrementando gradualmente el número de procesadores de uno a catorce. El cálculo fue realizado con una imagen de 2890×2308 pixeles.

Para la etapa de extracción de características se muestra que la gráfica del speed-up es casi lineal. Para la etapa de crecimiento de región, el speed-up se muestra sin cambios grandes cuando el número de nodos es superior a 9. El tiempo total del procesamiento de la imagen es menor a dos minutos.

3.4. Resumen

El artículo citado [14] realiza la convolución de la imagen utilizando la segunda derivada de la función gaussiana como filtro. Primero obtiene pixeles, denominados semillas.

Después determina la clase de dichas semillas: pixeles que representan el fondo de la imagen o los que representan un vaso sanguíneo. Una vez determinada la clase de las semillas se procede a crecer dichas semillas para concluir la segmentación de la imagen.

Para paralelizar este algoritmo el artículo utiliza partición de dominio [14] y se aplica para las dos etapas del algoritmo. De esta manera la imagen se divide en subimágenes.

En la primera etapa, cada subimagen es enviada (por el nodo maestro o manager) a un nodo diferente (worker). Cada uno de los worker, sin necesidad de comunicarse con otro, busca las semillas de la imagen y después la envía al nodo maestro. Para evitar tener información errónea en los pixeles de los bordes de las subimágenes, cada subimagen tiene cierta cantidad de pixeles de las subimágenes vecinas. Estos pixeles no son procesados, sólo usados para obtener información.

En la segunda etapa, la imagen es dividida horizontalmente y verticalmente por el nodo manager y cada subimagen es enviada a los nodos worker, quienes realizan el crecimiento de región a partir de las semillas encontradas en la etapa previa. Al terminar el procesamiento, el nodo manager une las subimágenes de los nodos worker para obtener dos imagenes resultantes: el procesamiento de la imagen que fue dividida horizontalmente y el de la imagen dividida verticalmente. Aplicando la operación lógica OR pixel a pixel entre las dos imágenes se obtiene la imagen final.

Este proceso se realiza porque al dividir la imagen horizontalmente, el crecimiento de región se ve truncado para aquellos pixeles que son bordes de vasos sanguíneos y crecen verticalmente. Lo análogo sucede si se divide la imagen verticalmente. Además, la operación OR evita que exista la comunicación entre nodos worker y permite mayor tiempo de procesamiento que de comunicación.

Capítulo 4

Detección de Bordes de Imágenes en Paralelo

En este capítulo se describe la manera en que se realiza la detección de bordes en paralelo mediante las arquitecturas Manager-Workers y CSE. Para que la detección de bordes se realice en paralelo, es necesario una partición adecuada, ya sea de dominio o funcional, para que cada nodo realice una tarea al mismo tiempo que otro, y que las comunicaciones estén implementadas adecuadamente para que no haya incoherencia de datos, deadlocks o que la eficiencia del trabajo se vea afectada por exceso de comunicaciones.

La aplicación utiliza la biblioteca de sincronización desarrolladas por Stephen J. Hartley [1].

4.1. Convolución de Imágenes con el Filtro de Sobel

La convolución de la imagen es realizado secuencialmente por los workers, en la arquitectura Manager-Workers y por los elementos secuenciales, en la arquitectura CSE. La convolución de la imagen utiliza el operador de Sobel para encontrar bordes en una imagen blanco y negro de 3 bytes por pixel, donde cada byte representa un valor de intenisdad de brillo del RGB para blanco y negro.

La implementación en Java para la detección de bordes con el operador de Sobel (mostrada en el Programa 4.2) requiere un objeto BufferedImage como entrada, el cual representa la subimagen que se procesa. El procesamiento recorre el arreglo bidimensional de la imagen representado por el objeto WritableRaster. Este objeto es el que contiene la información de cada uno de los pixeles de la subimagen. Cada pixel recorrido es un pixel central dentro de su vecindad y es necesario conocer su gradiente para lo cual se realiza la convolución vertical y horizontalmente (líneas 7 y 8 del Programa 4.2). Este procedimiento esta dado en el Programa 4.1.

La obtención de los pixeles vecinos mostrados en la línea 7 del Programa 4.1 se encuentra dentro de un bloque

```
try {
```

[1] http://elvis.rowan.edu/~hartley/ConcProgJava/

```
    ...
} catch (ArrayIndexOutOfBoundsException ex) {
    ...
}
```

pues un pixel central puede estar ubicado en el borde de la imagen y no tener vecinos. Con esto se asegura que no se intente obtener pixeles no existentes (índices no válidos dentro del arreglo bidimensional de la imagen).

Teniendo el gradiente se calcula la magnitud con la Ecuación 2.7. Es importante asegurarse que este resultado se encuentre entre cero y 255. Una vez realizado esto, el valor de la magnitud actualiza el color del pixel central en la nueva imagen.

```
1  private int convolucion(short[][] kernel, WritableRaster rasterImg,
2      int coordX, int coordY) {
3      int sum = 0;
4      for (int x = 0; x < 3; x++) {
5          for (int y = 0; y < 3; y++) {
6              try {
7                  int tmp = rasterImg.getSample(coordX -1 + x, coordY - 1 + y, 0)↩
                       ;
8                  sum += tmp * kernel[x][y];
9              } catch (ArrayIndexOutOfBoundsException ex) {
10                 continue;
11             }
12         }
13     }
14     return sum;
15 }
```

Programa 4.1: Implementación de la convolución en Java.

```
1  public BufferedImage procesa(BufferedImage src) {
2      BufferedImage nueva = new BufferedImage(src.getWidth(), src.getHeight(), ↩
           BufferedImage.TYPE_BYTE_GRAY);
3      WritableRaster wrN = nueva.getRaster();
4      WritableRaster wr = src.getRaster();
5      for (int x = 0; x < src.getWidth(); x++) {
6          for (int y = 0; y < src.getHeight();  y++) {
7              int gradX = convolucion(GX, wr, x, y);
8              int gradY = convolucion(GY, wr, x, y);
9              int magnitud = Math.abs(gradX) + Math.abs(gradY);
10             magnitud = (int) (magnitud - magnitud * 0.25);
11             if (magnitud > 255) magnitud = 255;
12             if (magnitud < 0) magnitud = 0;
13             wrN.setSample(x, y, 0, magnitud);
14         }
15     }
16     return nueva;
17 }
```

Programa 4.2: Implementación de Sobel en Java.

4.2. Diseño de la Arquitectura Manager-Workers

Mediante la arquitectura Manager-Workers la imagen es almacenada en memoria del nodo manager. Este nodo se encarga de dividir la imagen en tantas subimágenes

como sea posible y mandar cada subimagen a un nodo worker para que éste la procese. Además, cada subimagen contiene información de las subimágenes contiguas. Esta información adicional se denomina *padding*.

Los nodos worker se encargan de recibir cada subimagen enviada por el nodo manager y aplicar el filtro de Sobel. Una vez terminada esta tarea, la subimagen resultante, que es la que contiene los bordes de la imagen, es enviada al nodo manager.

Si existen más subimágenes para procesar, el nodo manager las envía a los workers que se encuentren disponibles.

Paralelamente al envío de subimágenes a los nodos worker, el nodo manager procesa la imagen final, uniendo cada subimagen procesada por los nodos worker.

La ventaja de usar esta arquitectura es que el número de canales de comunicación depende del número de nodos worker a utilizar, puesto que cada nodo worker únicamente puede comunicarse con el nodo manager.

4.2.1. Partición del Problema

Para esta arquitectura se emplea partición de dominio. La imagen se divide en varias subimágenes de $n \times m$ pixeles.

La detección de bordes de cada subimagen no depende del procesamiento previo de otras subimágenes, por lo que el procesamiento de cada subimagen puede realizarse independientemente en cada uno de los workers.

Sin embargo, se necesita agregar un padding que le añade información de pixeles de otras subimágenes.

Como el filtro de Sobel se realiza mediante una convolución de imagen, el valor final de un pixel p_i depende del valor de sus ocho vecinos. Sin el padding, los pixeles que se encuentran en los bordes de las subimágenes contendrán información errónea después de realizar la convolución. Añadiendo este padding la subimagen puede realizar la convolución de la imagen con los datos de sus vecinos y obtener una resolución correcta.

Al añadirle el padding, la subimagen de $n \times n$ pixeles es ahora de $(n+p) \times (m+p)$ pixeles, donde p indica el número de pixeles que se utilizan de padding, y los pixeles que están ahora en el borde de la imagen corresponden a pixeles de otras subimágenes.

En la Figura 4.1(a) se tiene la imagen que se va a procesar y en la Figura 4.1(b) se representa la división de la imagen en cuatro subimágenes. Cada subimagen es mandada a un worker para que se realice la segmentación y extraiga los bordes de la imagen.

La Figura 4.2 muestra una subimagen que es enviada a un nodo worker. En **(a)**, no se utiliza padding y en **(b)** se utiliza un padding de 10 pixeles, que es el área sombreada. El tamaño elegido en el padding de la Figura 4.2 **(b)** es para apreciar mejor la

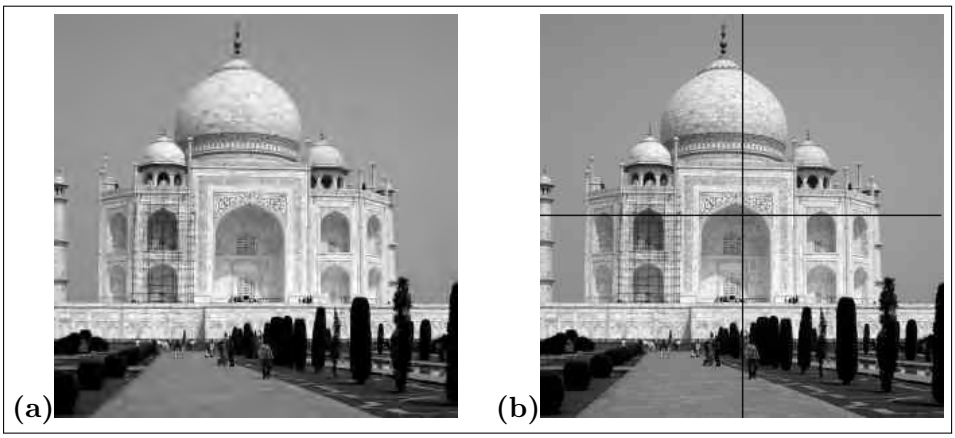


Figura 4.1: Imagen a enviar. (a) Imagen que se va a procesar. (b) Representa la división de la imagen en cuatro subimágenes.

información agregada.

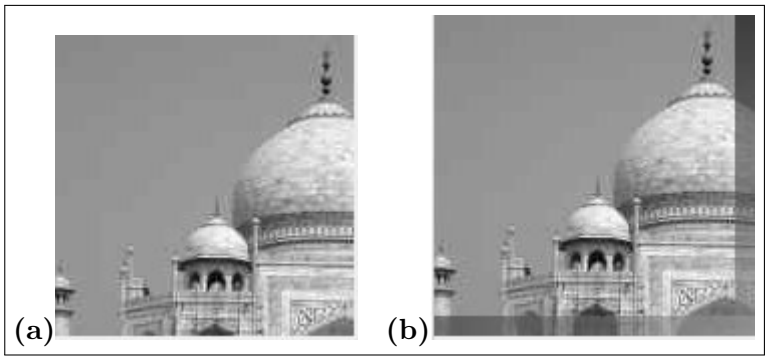


Figura 4.2: Subimagen sin padding *(a)* y con padding de 10 pixeles *(b)*.

Debido a que la convolución de la imagen mediante el filtro de Sobel se realiza verificando el valor de los vecinos más próximos del pixel en cuestión y no mayores a un pixel de distancia, el padding óptimo, para este caso en particular, es de un pixel. Si el padding es mayor, se realiza procesamiento redundante, puesto que se asignan valores a pixeles que no se tomarán en cuenta al momento de generar la imagen final (estos valores corresponden a pixeles de otra subimagen); aún así se tiene un resultado deseado como en la Figura 4.3(b). Si el padding es uno, los pixeles extra de la subimagen se toman para procesar los pixeles de los bordes y evitar que la imagen final tenga cortes indeseados, como en la Figura 4.3(b). Si el padding es cero, el resultado se muestra en la Figura 4.3(a).

4.2.2. Diseño de la Comunicación

Las imágenes se pasan entre el nodo manager y los nodos worker mediante mensajes.

El primer problema que existe al pasar una imagen por la red es que el tipo de dato utilizado para representarla en Java (BufferedImage) no es serializable, por lo que la

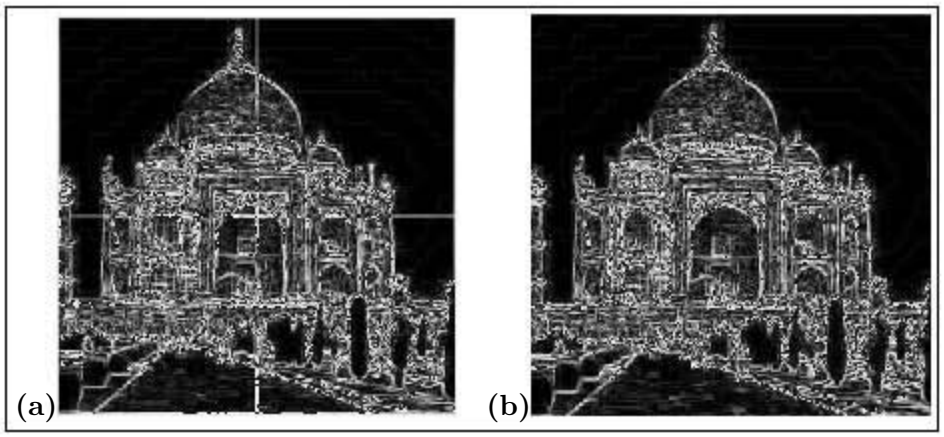


Figura 4.3: Imágenes resultantes después de la convolución con el filtro de Sobel. En (a) no se utilizó padding, en (b) sí.

imagen debe convertirse a bytes, como se muestra en el Programa 4.3.

```
1 public static byte[] toByteArray(BufferedImage imagen) {
2     ByteArrayOutputStream baos = new ByteArrayOutputStream();
3     try {
4         ImageIO.write(imagen, "jpg", baos);
5     } catch (IOException ex) {
6         ex.printStackTrace();
7     }
8     return baos.toByteArray();
9 }
```

Programa 4.3: Conversión de BufferedImage a bytes.

El mensaje necesita tener información adicional:

1. El número de pixeles que conforman el padding inferior, superior, izquierdo y derecho.
2. El ancho de la imagen.
3. El alto de la imagen
4. La coordenada (x, y) de la imagen donde empieza la subimagen.

Teniendo estos datos en el mensaje, al recibirse es necesario contar con una forma de convertir los bytes de la imagen en BufferedImage. Esto se representa en el Programa 4.4.

```
1 public static BufferedImage fromByteArray(byte[] byteArray) {
2     ByteArrayInputStream bais = new ByteArrayInputStream(byteArray);
3     BufferedImage imagen = null;
4     try {
5         imagen = ImageIO.read(bais);
6     } catch (IOException ex) {
7         ex.printStackTrace();
8     }
9     return imagen;
```

```
10 }
```

Programa 4.4: Conversión de bytes a BufferedImage.

La comunicación entre el manager y un worker se realiza de manera síncrona: el manager envía un mensaje al worker n y éste procesa la subimagen. Hasta que éste termine la tarea que está realizando, el nodo manager no le envía otra subimagen para procesar.

4.2.3. Mapeo y Balance de Carga

Lo ideal es que cada nodo worker contenga la misma cantidad de subimágenes a procesar y del mismo tamaño para que cada uno de ellos realice la misma cantidad de trabajo. Para que sea posible esto, en el archivo de configuración de la aplicación distribuida (`config.xml) que se explica con mayor detalle en la Sección 4.2.3` se puede definir el máximo número de threads que se pueden atender simultáneamente por worker. De esta manera, el número de subimágenes creadas son definidas por el número de workers (nw) y el número de threads por procesador (nt), dando un total de $nw \times nt$ subimágenes.

Esta división, sin embargo, ocasiona que el número de pixeles contenidos en las subimágenes que contienen los bordes de la imagen original, varíen unos cuantos pixeles respecto a las subimágenes que contienen la información del centro de la imagen original, ya que al tratar con pixeles, la división de n pixeles (ya sea a lo ancho o alto de la imagen) no permite un resultado con valores decimales, sino sólo enteros.

Por cada worker disponible, se envían nw subimágenes, hasta que no haya más por enviar. Con este método de mapeo, todos los workers contienen el mismo número de subimágenes (cada subimagen en cada worker es procesado paralelamente) y con un tamaño prácticamente igual, que varía de acuerdo al tamaño de la imagen original y al padding agregado a las subimágenes.

4.2.4. Implementación

El archivo `config.xml` contiene la configuración de la aplicación. El archivo es leído al inicio del programa y de él se pueden obtener los siguientes datos:

- Los hosts y puertos a utilizar por la aplicación.
- Número de threads que debe utilizar cada nodo worker.
- La ruta donde se localiza la imagen que se desea procesar.
- El padding.

Una vez obtenidos los datos especificados en la configuración, el nodo manager puede saber con cuáles nodos establecer comunicación para enviar la información a procesar, el archivo de la imagen que debe cargar en memoria y el número de pixeles a utilizar como padding. Una vez cargada la imagen en memoria, puede dividirla en diferentes subimágenes y agregarles el padding especificado. Estas subimágenes son convertidas a bytes y encapsuladas dentro de una clase del tipo *Mensaje* (Apéndice D.2.4) para

enviarlas a los workers.

Cada worker recibe el mensaje y convierte los bytes a BufferedImage para convolucionar la imagen con el filtro de Sobel. Teniendo la subimagen procesada, la convierte nuevamente en bytes y se la envía de regreso al nodo manager. Es importante mencionar que cada worker puede realizar diferentes procesamientos paralelamente, cada uno en un thread diferente.

El manager, por otro lado, espera los resultados de los procesos en los nodos worker, y une cada una de las subimágenes procesadas en una sola para mostrar la imagen final. Como a cada nodo worker le corresponde una única subimagen, ésta es colocada en la misma posición dentro de la resultante, de manera que ningún otro proceso puede tocar esa área y causar errores.

El Programa 4.5 muestra cómo se establece la conexión con cada uno de los nodos worker. En las líneas 3 y 4 se muestran dos ciclos que permiten realizar el mapeo de procesos a los diferentes worker. Se usan dos ciclos porque cada worker puede procesar diferentes subimágenes paralelamente, cada una en un thread diferente (ver Sección 4.2.3). Las líneas 5 y 6 son las que establecen la conexión con cada nodo. La línea 9 es la que envía el mensaje e indica que después de haber sido procesado el mensaje, el objeto `gen: GenImgFinal` se debe encargar de actualizar la imagen final dada por la variable `imagen: BufferedImage`.

```
1  imgFinal = new BufferedImage(imagen.getWidth(), imagen.getHeight(), ↩
       BufferedImage.TYPE_BYTE_GRAY);
2  ArrayList<Thread> threads = new ArrayList<Thread>();
3  for (String host : hosts.keySet()) {
4      for (int x = 0; x < maxThreads; x++) {
5          Conexion con = new Conexion(host, hosts.get(host));
6          con.conectar();
7          Mensaje msj = mensajes.remove(0);
8          GeneradorImgFinal gen = new GeneradorImgFinal(imgFinal);
9          threads.add(con.enviaYRecibe(msj, gen));
10     }
11 }
12 for (Thread t : threads) {
13     t.start();
14 }
```

Programa 4.5: Envío de Mensajes a un worker

El Programa 4.6 muestra la lógica para la recepción de mensajes del manager y el procesamiento de ellos. La línea uno crea un objeto `EstablishRendezvouz` que se encarga de abrir la conexión de un nodo por un puerto dado. En la línea tres, el objeto `rendezvouz: Rendezvouz` creado permite que el nodo manager se conecte al nodo worker y le mande los mensajes. La línea 6 obtiene el mensaje del Manager con la subimagen a procesar. Si el mensaje es nulo, el Worker da por terminado el procesamiento, en caso contrario, convierte el arreglo de bytes contenido en el mensaje a imagen y procede a convolucionar la imagen con el operador de Sobel. Una vez concluida esta operación , envía al nodo Manager (línea 14) la imagen procesada, en forma de arreglo de bytes.

```
1  EstablishRendezvouz er = new EstablishRendezvous(puerto);
```

```
2   ...
3   Rendezvouz rendezvous = er.serverToClient();
4   ...
5   Tarea tarea = operacionImagenClase;
6   Mensaje msg = (Mensaje) ren.serverGetRequest();
7   if (msg == null) {
8     rendezvouz.serverMakeReply(null);
9     rendezvouz.close();
10    System.exit(0);
11  }
12  BufferedImage img = ImgOp.fromByteArray(msg.getMensaje());
13  msg.setMensaje(ImgOp.toByteArray(tarea.procesa(img)));
14  rendezvouz.serverMakeReply(msg);
15  rendezvouz.close();
```

Programa 4.6: Recepción de Mensajes por un worker.

4.3. Diseño de la Arquitectura CSE

Mediante la arquitectura CSE la imagen es almacenada en memoria del nodo maestro. Este nodo se encarga de dividir la imagen en tantas subimágenes como sea posible y mandar cada subimagen a un nodo diferente para que éste la procese.

Los demás nodos denominados *elementos secuenciales*, se encargan de recibir cada subimagen enviada por el nodo maestro y aplicar el filtro de Sobel. La disposición de estos nodos secuenciales tiene la forma de malla o cuadrícula. Cada elemento secuencial necesita obtener información de sus vecinos para que el resultado sea correcto. Una vez terminado el procesamiento, la subimagen resultante es enviada al nodo maestro quien se encarga de recopilar la información y generar la imagen final.

4.3.1. Partición del Problema

Esta arquitectura presenta similitudes con Manager-Workers en la partición del problema:

- Se emplea partición de dominio.
- La imagen original se divide en varias subimágenes de $n \times m$ pixeles.
- La detección de bordes de cada subimagen no depende del procesamiento previo de otras subimágenes, por lo que el procesamiento de cada una de ellas puede realizarse casi independientemente en cada uno de los elementos secuenciales.
- Es necesario contar con pixeles adicionales (de otras subimágenes) para obtener el resultado correcto.

A diferencia de Manager-Workers, que esa información adicional o padding es agregada a cada subimagen antes de enviar los mensajes por el nodo maestro, en CSE sólo se envía la subimagen sin la información extra. Cada elemento secuencial al que le fue asignada una subimagen tiene la tarea de pedir información a los elementos vecinos. Si no existe esta comunicación, el resultado que se obtiene es igual a realizar el procesamiento con la arquitectura Manager-Workers sin utilización de padding, como se muestra en la Figura 4.3.

4.3.2. Diseño de la Comunicación

La comunicación en CSE se realiza entre los elementos secuenciales. Cada uno de ellos pide y recibe información de sus vecinos mediante mensajes. Este paso de mensajes se da de manera asíncrona, por lo que se requiere un buffer para almacenar los mensajes procesados.

Los mensajes contienen la porción de imagen a procesar en un arreglo de bytes, por lo que de igual manera que en Manager-Workers se necesita contar con el método de transformación de `BufferedImage-byte[]-BufferedImage`.

Cada nodo secuencial necesita conocer dónde se ubican cada uno los elementos secuenciales vecinos. La información de la ubicación de los vecinos es almacenada en el mensaje que se envía y consta del host, puerto, identificador del elemento secuencial vecino y posición de éste respecto al elemento secuencial en cuestión.

Dependiendo del número de procesadores y threads por procesador, la cantidad máxima de vecinos de cada elemento secuencial varía:

Procesadores	**Threads**	**Max. No. Vecinos**
1	1	0
1	2	1
1	≥ 3	2
2	1	1
2	2	2
2	≥ 3	3
≥ 3	1	2
≥ 3	2	3
≥ 3	≥ 3	4

Cuadro 4.1: Máximo número de vecinos por elemento secuencial.

En el Programa 4.7 se muestra parte de la lógica del envío de peticiones a cada vecino del elemento secuencial en ejecución. Para esto se crea un buffer dónde la información de cada elemento vecino es almacenada. Como se tienen a lo más cuatro vecinos, cada uno tiene asignado (y reservado) una dirección de memoria dentro del buffer. Así al vecino izquierdo le corresponde el índice cero del arreglo, al de arriba el índice uno, etc. Por cada nodo vecino es necesario crear el mensaje que contiene la petición y la conexión a éste (líneas 4 - 31). Como cada petición realizada es un thread, éstas se realizan paralelamente y asíncronamente, pues se cuenta con un buffer de recepción (líneas 29 - 31). Pese a esto, cada thread se bloquea hasta haber recibido su respuesta.

```
1  ArrayList<Thread> peticiones = new ArrayList<Thread>();
2  GridNeighbord[] vecinos = msj.getVecinos();
3  Mensaje[] buffer = new Mensaje[4];
4  for (short x = 0; x < 4; x++) {
5     GridNeighbord vecino = vecinos[x];
6     if (vecino != null) {
7          ...
8          MensajeCSE m = new MensajeCSE();
9          m.setPadding(msj.getPadding());
```

```
10          m.setVecino(vecino);
11          m.setPeticion(new Peticion(id, vecino.getSeqElemId(), Peticion.↩
                PETICION_DATOS));
12          switch (vecino.getPosicion()) {
13              case IZQUIERDA:
14                  m.setPosicionOrigen(GridNeighbord.Posicion.DERECHA);
15                  break;
16              case ARRIBA:
17                  m.setPosicionOrigen(GridNeighbord.Posicion.ABAJO);
18                  break;
19              case DERECHA:
20                  m.setPosicionOrigen(GridNeighbord.Posicion.IZQUIERDA);
21                  break;
22              case ABAJO:
23                  m.setPosicionOrigen(GridNeighbord.Posicion.ARRIBA);
24                  break;
25          }
26          m.setId(id);
27          Conexion con = new Conexion(vecino.getHost(), vecino.getPuerto());
28          con.conectar();
29          Thread peticion = con.enviaYRecibe(m, buffer);
30          peticiones.add(peticion);
31          peticion.start();
32      }
33  }
34  ...
```

Programa 4.7: Paso de mensajes asíncrono entre un nodo secuencial y sus vecinos.

4.3.3. Mapeo y Balance de Carga

De forma análoga que en Manager-Workers (Sección 4.2.3), el número de subimágenes creadas, son definidas por el número de procesadores disponibles (np) y el número de threads por procesador (nt), dando un total de $np \times nt$ subimágenes. Las subimágenes que contienen los bordes de la imagen original, varían unos cuantos pixeles respecto a las subimágenes que contienen la información del centro de la imagen original.

El envío de cada subimagen a cada elemento secuencial se realiza también de manera análoga a Manager-Workers, aunque en este caso, ya que se usa arquitectura CSE, las subimágenes no contienen ningún padding ni información extra como en Manager-Workers.

4.3.4. Implementación

Los nodos que fungen como elementos secuenciales, el nodo maestro, los puertos que se utilizan, la imagen que se procesa, el padding y el número de threads por nodo que se desean tener son descritos en un archivo de configuración XML el cual es leído al inicio del programa.

Con estos datos, el nodo maestro puede saber a qué nodos les va mandar información para que sea procesada.

Así, el nodo maestro procede a realizar la división de la imagen en diferentes subimágenes que son convertidas a bytes y encapsuladas dentro de una clase del ti-

po *Mensaje* para enviarlas a los elementos secuenciales. El Programa 4.8 muestra este proceso. Las líneas 6 - 34 indican quiénes son los vecinos de un nodo secuencial. Las líneas 36 - 40 manejan la conexión entre el nodo maestro y el elemento secuencial. La línea 44 permite que cada thread inicie su ejecución.

```
1  imgFinal = new BufferedImage(imagen.getWidth(), imagen.getHeight(),
2  BufferedImage.TYPE_BYTE_GRAY);
3  ArrayList<Thread> threads = new ArrayList<Thread>();
4  for (int x = 0; x < gridMensajes.length; x++) {
5      for (int y = 0; y < gridMensajes[0].length; y++) {
6          MensajeCSE msj = gridMensajes[x][y];
7          try {
8              msj.setVecino(
9                  gridMensajes[x - 1][y].getHost(),
10                 gridMensajes[x - 1][y].getPuerto(),
11                 gridMensajes[x - 1][y].getId(),
12                 GridNeighbord.Posicion.IZQUIERDA);
13         } catch (ArrayIndexOutOfBoundsException e) { }
14         try {
15             msj.setVecino(
16                 gridMensajes[x][y - 1].getHost(),
17                 gridMensajes[x][y - 1].getPuerto(),
18                 gridMensajes[x][y - 1].getId(),
19                 GridNeighbord.Posicion.ARRIBA);
20         } catch (ArrayIndexOutOfBoundsException e) { }
21         try {
22             msj.setVecino(
23                 gridMensajes[x + 1][y].getHost(),
24                 gridMensajes[x + 1][y].getPuerto(),
25                 gridMensajes[x + 1][y].getId(),
26                 GridNeighbord.Posicion.DERECHA);
27         } catch (ArrayIndexOutOfBoundsException e) { }
28         try {
29             msj.setVecino(
30                 gridMensajes[x][y + 1].getHost(),
31                 gridMensajes[x][y + 1].getPuerto(),
32                 gridMensajes[x][y + 1].getId(),
33                 GridNeighbord.Posicion.ABAJO);
34         } catch (ArrayIndexOutOfBoundsException e) { }
35         msj.setPeticion(new Peticion(msj.getId()));
36         Conexion con = new Conexion(gridMensajes[x][y].getHost(),
37         gridMensajes[x][y].getPuerto());
38         con.conectar();
39         GeneradorImgFinal gen = new GeneradorImgFinal(imgFinal);
40         threads.add(con.enviaYRecibe(msj, gen));
41     }
42 }
43 for (Thread t : threads) {
44     t.start();
45 }
```

Programa 4.8: Envío de mensajes del nodo maestro a los elementos secuenciales.

Por otro lado, cada elemento secuencial recibe las peticiones, similar a un nodo Worker. En el Programa 4.9 se muestra: La línea uno y tres permiten que el nodo en el cual se ejecuta el elemento secuencial espere peticiones de otro elemento secuencial o el maestro, las acepte y obtenga el mensaje.
Teniendo el mensaje, lo primero que se hace es verificar el tipo de petición: petición de procesamiento o de datos. Si el tipo de petición es de datos se busca el elemento secuencial deseado para que mande la información a su vecino, como se muestra en el Programa 4.11. De lo contrario se genera un nuevo elemento secuencial y se procede

al procesamiento de la imagen como se muestra en el Programa 4.10. Como cada nodo permite la ejecución de varios elementos secuenciales paralelamente, es necesario contar con un hash map concurrente que permita tener acceso a cada uno de los elementos creados en el nodo. La manera de acceder a ellos es por el id del nodo secuencial, el cual fue asignado por el maestro antes de realizar el mapeo del trabajo. El hash map concurrente está definido por la variable `secElems: ConcurrentHashMap`.

```
1  EstablishRendezvouz er = new EstablishRendezvous(puerto);
2  ...
3  Rendezvouz rendez = er.serverToClient();
4  ...
5  MensajeCSE msg = (MensajeCSE) rendez.serverGetRequest();
6  if (msg.getPeticion().getTipoPeticion() == Peticion.PETICION_DATOS) {
7      SequentialElementA se = secElems.get(msg.getPeticion().getSeqElemIdDest());
8      // Busca que el elemento secuencial se encuentre definido en el Hash Map.
9      while (se == null) {
10         se = secElems.get(msg.getPeticion().getSeqElemIdDest());
11     }
12     se.enviaDatos(msg);
13     rendez.serverMakeReply(msg);
14 } else if (msg.getPeticion().getTipoPeticion() == Peticion.PETICION_PROCESA) {
15     SequentialElementA seq = null;
16     seq = new SequentialElementA(msg, operacionImagen);
17     secElems.put(msg.getId(), seq);
18     seq.procesaMensaje();
19     rendez.serverMakeReply(seq.getMensaje());
20 }
```

Programa 4.9: Recepción de mensajes de los elementos secuenciales.

Para que un nodo secuencial efectúe el procesamiento de su subimagen (Programa 4.10), el primer paso es pedirle a sus vecinos que le manden información de sus pixeles. Esta información se almacena en un buffer y las peticiones se realizan asíncronamente (la parte del código que se encuentra dentro del ciclo `for`. Líneas 7 - 21). Es importante que la recepción de información de los vecinos haya terminado para poder continuar con el procesamiento. El método `join()` de la clase Thread permite esperar que un thread termine su ejecución. Éste método se usa en la línea 24 para esperar que todos los vecinos hayan enviado su información.

Al término de esto, se agregan los pixeles vecinos a la subimagen y se procede a utilizar el filtro de Sobel, que es una instacia del objeto `operacionImg: Tarea`. El resultado es guardado en el mensaje como un arreglo de bytes.

```
1  public void procesaMensaje() {
2      int nW = imagen.getWidth();
3      int nH = imagen.getHeight();
4      ArrayList<Thread> peticiones = new ArrayList<Thread>();
5      GridNeighbord[] vecinos = msj.getVecinos();
6      Mensaje[] buffer = new Mensaje[4];
7      for (short x = 0; x < vecinos.length; x++) {
8          GridNeighbord vecino = vecinos[x];
9          if (vecino != null) {
10             ...
11             MensajeCSE m = new MensajeCSE();
12             ...
13             // Agrega de datos al mensaje para pedir al vecino
14             ...
```

```
15              Conexion con = new Conexion(vecino.getHost(), vecino.getPuerto());
16              con.conectar();
17              Thread peticion = con.enviaYRecibe(m, buffer);
18              peticiones.add(peticion);
19              peticion.start();
20          }
21      }
22      for (Thread t : peticiones) {
23          try {
24              t.join();
25          } catch (InterruptedException ex) {
26              log.log(Level.SEVERE, null, ex);
27          }
28      }
29      ...
30      // Agrega los pixeles de los vecinos a img
31      ...
32      msj.setMensaje(ImgOp.toByteArray(operacionImg.procesa(img)));
33  }
```

Programa 4.10: Procesamiento de una imagen por un elemento secuencial.

Por otro lado, el envío de información de cierta cantidad de pixeles de un elemento secuencial al elemento que lo solicitó se muestra en el Programa 4.11. Para esto es necesario conocer la posición del elemento secuencial que pide los datos con respecto al que envía. Dependiendo de la posición se obtienen los pixeles que se necesitan. Se actualiza el mensaje y se envía.

```
1   public void enviaDatos(MensajeCSE mensaje) {
2       GridNeighbord.Posicion pos = null;
3       switch (mensaje.getPosicionOrigen()) {
4           case IZQUIERDA:
5               pos = GridNeighbord.Posicion.DERECHA;
6               break;
7           case ARRIBA:
8               pos = GridNeighbord.Posicion.ABAJO;
9               break;
10          case DERECHA:
11              pos = GridNeighbord.Posicion.IZQUIERDA;
12              break;
13          case ABAJO:
14              pos = GridNeighbord.Posicion.ARRIBA;
15              break;
16      }
17      int padding = mensaje.getPadding();
18      // Si no hay padding definido
19      if (padding <= 0) {
20          mensaje.setPosicionOrigen(pos);
21          mensaje.setId(id);
22          return;
23      }
24      BufferedImage imgPadding = null;
25      switch (mensaje.getPosicionOrigen()) {
26          case ABAJO:
27              imgPadding = imagen.getSubimage(0, imagen.getHeight() - padding, ↩
                    imagen.getWidth(), padding);
28              break;
29          case ARRIBA:
30              imgPadding = imagen.getSubimage(0, 0, imagen.getWidth(), padding);
31              break;
32          case DERECHA:
33              imgPadding = imagen.getSubimage(imagen.getWidth() - padding, 0, ↩
                    padding, imagen.getHeight());
34              break;
```

```
35              case IZQUIERDA:
36                  imgPadding = imagen.getSubimage(0, 0, padding, imagen.getHeight());
37                  break;
38          }
39          // Actualizo el mensaje para su regreso.
40          mensaje.setPosicionOrigen(pos);
41          mensaje.setWidth(imgPadding.getWidth());
42          mensaje.setHeight(imgPadding.getHeight());
43          mensaje.setMensaje(ImgOp.toByteArray(imgPadding));
44          mensaje.setId(id);
45      }
```

Programa 4.11: Recepción de mensajes de los elementos secuenciales.

El nodo maestro espera el término del procesamiento de cada subimagen. Cuando cada elemento secuencial termina su cómputo, el nodo maestro recupera la subimagen procesada y actualiza la imagen final.

4.4. Resumen

La partición del problema es la parte más importante para elegir adecuadamente qué arquitectura implementar. Para la detección de bordes, se utiliza partición de dominio, lo que quiere decir que, la imagen se subdivide en varias subimágenes. Este tipo de dominio permite que la detección de bordes en paralelo se realice con la arquitectura Manager-Workers o CSE.

Para la implementación de ambas arquitecturas se necesita contar con un archivo de configuración que permite indicarle al nodo manager qué imagen se desea procesar, el padding (en el caso de Manager-Workers) o pixeles extra que se concatenan a una subimagen en particular (en el caso de CSE), el número de procesos por procesador y los nodos a utilizar.

En Manager-Workers, el nodo manager lee la configuración y genera el número de mensajes requeridos para mapearlos a los diversos workers, manteniendo la mayor equidad en carga de trabajo que sea posible. La única comunicación existente en esta arquitectura se efectúa entre el nodo manager y los nodos workers. Los nodos worker no se comunican entre ellos. El nodo manager se conecta con cada uno de los nodos workers enviándoles tantas subimágenes como procesos por procesador hayan sido definidos en el archivo de configuración. La comunicación que se efectúa entre un nodo manager y un worker es síncrona. El nodo maestro tiene que esperar a que el worker termine su procesamiento para continuar con alguna otra instrucción.

En CSE, el nodo maestro lee la configuración y genera el número de mensajes requeridos para mapearlos a los diversos elementos secuenciales manteniendo la mayor equidad en carga de trabajo que sea posible. Los nodos secuenciales tienen forma de malla cuadrangular, lo que determina que cada uno tenga 2 elementos vecinos (si el nodo está en una esquina de la malla), 3 (si está en las orillas) ó 4. Al tener un elemento secuencial un mensaje para procesar, la primer operación que realiza éste es pedirle a sus vecinos pixeles de su subimagen para que el proceso no contenga pérdida de información. Cada nodo secuencial establece comunicación asíncrona con sus vecinos, de modo que puede pedirles los datos simultáneamente y no tiene que esperar a que un elemento

vecino haya enviado la información para realizar peticiones a otro. No obstante, para que cada elemento secuencial continue con el trabajo que realiza, es indispensable tener los datos de todos los elementos vecinos. La subimagen procesada por un elemento secuencial es enviada al nodo maestro, quien la almacena y genera la imagen final.

Tanto los elementos secuenciales como los workers utilizan los códigos descritos en los Programas 4.2 y 4.1 para obtener bordes en imágenes con el operador Sobel. El código mostrados en esos listados son ligeras variaciones del código mostrado en los Programas 2.2 y 2.1, acoplados para el lenguaje Java.

Las implementaciones de Manager-Workers y CSE, desarrolladas para este trabajo, permiten que las subimágenes procesadas sean recopiladas conforme se vayan recibiendo en el nodo maestro, sin importar que haya aún procesos activos de los nodos que realizan un procesamiento secuencial. Esta decisión está tomada, basándose en el hecho que, aunque el nodo i haya empezado a realizar el cómputo en el tiempo t y el nodo j en el tiempo $t + 1$, el nodo i puede terminar su cómputo tiempo después que el nodo j. Así, se ahorra la mayor cantidad de tiempo posible en la recopilación de subimágenes.

La manera en que están implementadas las dos arquitecturas permiten no sólo utilizar el operador de Sobel o simplemente detección de bordes o convolución de imágenes, sino que se permite crear códigos de segmentación de imágenes más complejos para ser ejecutados en un clúster. Al crear un objeto que implemente la interface `Tarea`, se consigue aplicar los algoritmos deseados.

Capítulo 5

Experimentación

En este capítulo se describen las características y la experimentación de la aplicación paralela con el fin de conocer los resultados y posteriormente obtener conclusiones en base a ellos. En la experimentación se buscan resultados de tiempo y precisión de ambas arquitecturas: Manager-Workers y CSE.

5.1. Características

La evaluación de Manager-Workers y CSE debe asegurar características iguales en ambas arquitecturas:

Plataforma de Ejecución. Se utiliza un clúster con las siguientes características[1]:

- **Nodo maestro**
  - Dos procesadores Intel Xeon, 2.6GHz
  - Motherboard SE7501BR2 Intel dual Xeon
  - 1GB de memoria RAM
  - Disco duro SCSI Cheeta, 80GB
- **Seis nodos esclavos**
  - Procesador Intel Pentium 4, 2.6GHz
  - Motherboard Intel Pentium 4 BOSC845GBSRL
  - 512 MB de memoria RAM
  - Disco duro Seagate de 40 GB
- **Un switch 3com Superstack 3 4226T**
  - 24 puertos 10/100
  - 2 puertos 10/100/1000
- **Cuatro No break Omnismart 500VA con regulador**
- **Sistema operativo Linux**
  - Versión de kernel 2.4.27-3-686

[1]Los procesadores de cada nodo tienen su propia memoria y no pueden acceder a la memoria de alguno otro.

Figura 5.1: Imagen utilizada para ejemplificar la segmentación de imágenes (3808×3027 pixeles, 4.2 MB).

Lenguaje de Programación. Ambas arquitecturas están desarrolladas en el lenguaje de programación Java en la versión 1.6.

División del Problema. Ambas arquitecturas emplean división de dominio, donde la imagen que se desea procesar es el objeto que se divide. La imagen debe ser dividida de igual forma en ambas arquitecturas, lo que implica que la subimagen $simg_j$ en Manager-Workers debe ser igual (sin considerar padding) que la subimagen $simg_j$ en CSE. El resultado de esta división da $n \times m$ subimágenes, donde n está dado por el número de procesadores a utilizar y m el número de threads por procesador.

Imagen a procesar. La imagen que se procesa se muestra en la Figura 5.1. Es una imagen en escala de grises (8 bytes por pixel) cuyo tamaño original es de 3808×3027 pixeles y pesa 4.2 MB.

Definidas las características anteriores, la experimentación se realiza variando el número de procesadores (np) (o nodos del clúster) y threads por procesador (nt) a utilizar, con el fin de observar el tiempo obtenido en cada arquitectura y tener más datos de comparación. Por cada variación de procesadores y threads se realizan diez ejecuciones para obtener una media adecuada.

Para no probar con un número grande de posibles combinaciones de np y nt, se hace la restricción que $np * nt$ tenga raíz cuadrada exacta y además $np * nt \leq 49$. La única excepción válida de esta restricción es cuando $np > 1$ y $nt = 1$. Estas restricciones permiten ahorrar tiempo en las pruebas, pues por cada combinación se realizan diez ejecuciones.

Al variar np se tiene un incremento gradual de los nodos utilizados: de uno a seis nodos. Con esta variación podemos comparar los tiempos de procesamiento de la imagen en base al número de nodos utilizados. Con nt se define el número de threads por procesador lo que permite que en cada nodo se realicen más operaciones simultáneamente y además comparar los tiempos en base al número de procesadores y al número de threads por procesador.

Para poder medir el tiempo de ejecución, el código toma dos muestras de tiempo:

- La primera muestra indicador se obtiene después de leer la configuración y antes de crear los mensajes a enviar y la ejecución de la segmentación de la imagen.
- La segunda y última muestra de tiempo se inicializa al término de la recopilación de todas las subimágenes.

En este momento es posible obtener una medida aproximada del tiempo de ejecución de las aplicaciones distribuidas.

5.2. Resultados de Tiempo

En esta sección se muestra el tiempo que tardó cada arquitectura en finalizar el procesamiento de la imagen. Los tiempos mostrados son un promedio de diez ejecuciones realizadas por cada configuración, en la que varía el número de procesadores y el número de threads por procesador con las especificaciones mencionadas en la Sección 5.1.

El tiempo obtenido en cada ejecución varía dependiendo el número de procesadores y número de threads por procesador. Cada configuración de threads y procesadores son mostradas en la Tabla 5.1. A los resultados se les agrega un intervalo de confianza dado por la fórmula de la ecuación 5.1

$$\mu = \bar{X} \pm z\frac{\sigma}{\sqrt{N}} \tag{5.1}$$

donde:

μ Es la media que se desea conocer.

$\bar{X}$ Representa la muestra, es decir, el tiempo promedio después de diez ejecuciones de la aplicación.

z La distribución normal de la muestra, que en este caso es 1.65 puesto que se quiere un nivel de confianza del 95 %.

σ La desviación estándar de la muestra.

N El tamaño de la muestra, que en este caso es diez.

En los casos donde los canales de comunicación utilizados son mucho mayores al número de procesadores utilizados (por ejemplo, un procesador con 25, 36 y 49 threads), el tiempo de ejecución es mayor al que se muestra con un sólo procesador y un sólo thread. La mayor cantidad de tiempo utilizado, en la implementación paralela, se debe a que se utilizan demasiados canales de comunicación. Varios canales de comunicación simultáneamente evitan que los datos pasen libremente entre cada worker y el

No. procesadores	Threads por procesador	Tiempo (ms)	
		M-W	CSE
1	1	14874.2 ± 59.4061	14926.7 ± 44.9246
1	4	11003 ± 83.3450	11544.3 ± 124.1407
1	9	12051.5 ± 140.9138	12583.4 ± 233.9708
1	16	13413.1 ± 172.3269	14245.4 ± 600.9284
1	25	15252.4 ± 346.5432	15951.1 ± 491.4594
1	36	17461.3 ± 309.5802	18463.9 ± 419.2079
1	49	20016.9 ± 280.8276	21577.4 ± 339.6204
2	1	9236.11 ± 1577.7575	10027.3 ± 354.5341
2	2	7462.4 ± 108.3770	7866.2 ± 54.4627
2	8	6977.4 ± 108.1318	7405 ± 103.9463
2	18	8054.6 ± 126.8117	9138.6 ± 432.7507
3	1	7672.66 ± 1085.8799	8033.4 ± 36.7364
3	3	5192.7 ± 36.8418	5621.9 ± 67.9074
3	12	5775.3 ± 105.9932	6319.7± 164.7969
4	1	6908.11 ± 1359.7192	7126.4 ± 18.2233
4	4	4492.4 ± 38.8616	4858.8 ± 50.2827
4	9	4766.4 ± 41.7443	5132.2 ± 72.2715
5	1	6461.88 ± 1418.5702	6679.88 ± 28.2376
5	5	4124.2 ± 29.3017	4448.4 ± 47.4315
6	1	6107.66 ± 1415.3397	6340.2 ± 23.6585
6	6	3962 ± 25.3855	4288.9 ± 29.7408
7	1	580255 ± 542.6831	6069.7 ± 24.3734
7	7	3826.666 ± 45.0376	4160.333 ± 77.1970

Cuadro 5.1: Tiempo utilizado para la detección de bordes.

manager. Este fenómeno pasa de igual manera si se tienen muchos nodos, aunque sólo exista un thread por cada procesador, pues genera un excesivo número de canales de comunicación [2].

En cambio, teniendo un balance entre el número de procesadores y threads por procesador se obtiene una reducción de tiempo considerable, comparándolo nuevamente con el tiempo obtenido al realizar la ejecución con un procesador y un thread. Para observa esto, en el Cuadro 5.1 se puede observar que, utilizando al arquitectura CSE con dos procesadores y 18 threads por procesador (de este modo se utilizan 36 elementos secuenciales en una malla de 2×18), se obtiene un tiempo promedio de 9138.6 milisegundos, el cual es mayor que si se usaran dos procesadores y dos o cuatro threads por procesador donde se obtienen tiempos promedio de 7866.2 y 7405 milisegundos respectivamente. De igual manera, el tiempo total al utilizar tres procesadores y 12 threads por procesador (36 elementos secuenciales) es mayor que si se utilizaran tres procesadores y tres threads por procesador (9 elementos secuenciales).

La utilización de un sólo procesador y thread, respresenta la ejecución serial de la aplicación. El Cuadro 5.1 muestra que el tiempo total para la finalización de la detección de bordes en la imagen, es cercano a los 15 segundos. En general ambas arquitecturas mantienen un tiempo total menor al tiempo utilizado por la implementación serial, exceptuando los casos en los que se utiliza un procesador y más de 25 threads por

[2]Debido a que no se dispone de un mayor número de nodos dentro del clúster, no es posible mostrar datos para más de 7 procesadores.

procesador. De hecho, cuando se utilizan dos o más procesadores, el tiempo de ejecución es menor a diez segundos.

5.3. Comparación de Resultados

Comparando las dos arquitecturas en términos de tiempo, Manager-Workers obtiene ventaja en todo momento sobre CSE.

En Manager-Workers el padding es obtenido desde que el nodo manager realiza la partición de la imagen. El manager conoce el número de pixeles que se usan de padding, por lo que al momento de generar las subimágenes a repartir, les añade los pixeles extra a cada una de ellas. Para este proceso, simplemente obtiene $n + p$ pixeles de la imagen original, donde n es el número de pixeles de cada subimagen y p el número de pixeles que se usan de padding.

En CSE, cada elemento secuencial conoce únicamente a los pixeles que le corresponden. Para que el procesamiento se realice de correctamente, cada elemento secuencial le pide a los vecinos dentro de la malla de elementos secuenciales, una porción de su subimagen. Al obtener dicha porción, es necesario que el elemento secuencial cree una nueva imagen que ahora contiene la subimagen para procesar y las porciones de subimágenes de sus vecinos.

La concatenación de los pixeles extra en CSE implica un mayor tiempo de procesamiento comparado con el tiempo que se emplea en Manager-Workers:

- Se necesita contar con fragmentos de las subimágenes de los cuatro vecinos.
- Es necesario crear una imagen nueva, cuya dimensión es el alto y ancho de la subimagen del elemento secuencial añadidos el alto y ancho de los cuatro fragmentos obtenidos de sus vecinos.
- También se debe conocer la posición en la que cada fragmento se posiciona dentro de la imagen nueva.
- Hay que copiar el contenido de cada fragmento de las subimágenes vecinas a la nueva así como el de la subimagen del elemento secuencial en cuestión.

La Figura 5.2 muestra una gráfica comparativa entre el tiempo de ejecución de Manager-Workers y CSE. La gráfica de Manager-Workers se muestra como una superficie a escala de grises, y su variación de color indica el tiempo. La gráfica para CSE se observa en una superficie cuadriculada roja. En general la gráfica de CSE se encuentra por encima de la de Manager-Workers. Ambas gráficas presentan un patrón similar.

5.4. Resultado de Procesamiento

La detección de bordes de la imagen con el filtro de Sobel se muestra en la Figura 5.3. Ambas arquitecturas, así como la implementación serial, generan el mismo resultado, difiriendo sólo en el tiempo que tarda en procesar la imagen.

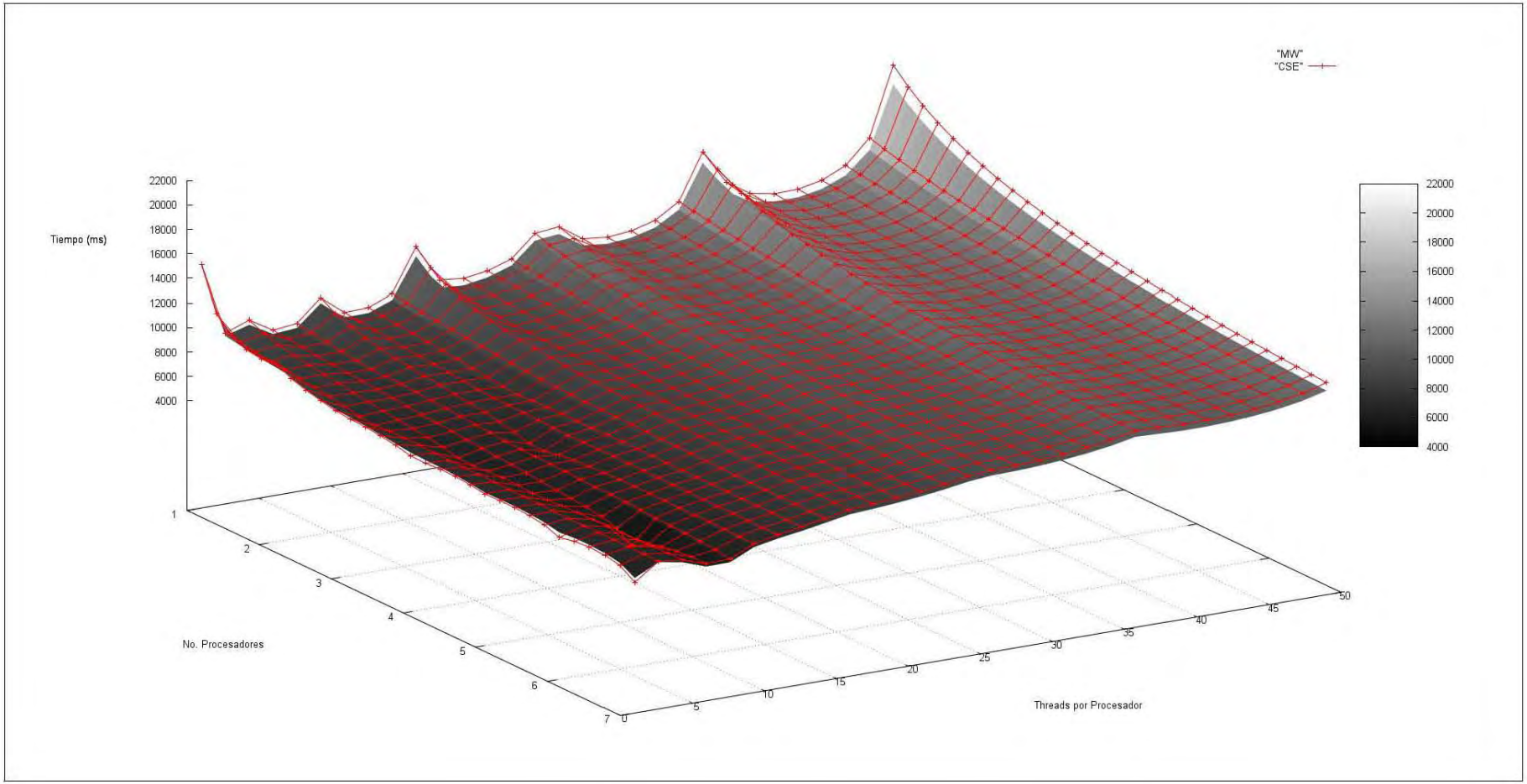


Figura 5.2: Gráfica que muestra el tiempo de ejecución de MW y CSE.

Figura 5.3: Resultado de la extracción de bordes de la imagen 5.1. La imagen original es de 3808×3027 pixeles y pesa 2.8 MB.

5.5. Resumen

La compracación entre ambas arquitecturas es posible si existen condiciones similares de ejecución. Para la compracación de ellas en este trabajo, se define lo siguiente: partición de dominio para el problema, de manera que se realizan las mismas operaciones en paralelo sobre diferentes piezas más pequeñas de datos; la ejecución de ambas arquitcturas se realiza en el mismo sistema distribuido cuyas caracterísctica son especificadas en la Sección 5.1; el lenguaje de programación utilizado para la implementación de las arquitecturas; la imagen a procesar que debe ser la misma para ambas arquitecturas.

Por cada configuración de procesadores y threads por procesador se realizaron diez ejecuciones y el tiempo promedio es el que se toma en cuenta para la comparación.

Los resultados de tiempo, mostrados en el Cuadro 5.1 muestran que Manager-Workers obtiene mejores tiempos de procesamiento que CSE. La ventaja de Manager-Workers sobre CSE es que una vez que un worker empieza a realizar el procesamiento de su subimagen correspondiente, no debe preocuparse por establecer comunicación con algún otro worker, mientres que en CSE, cada elemento secuencial al iniciar el procesamiento de su subimagen correspondiente necesita obtener información de otros elementos secuenciales, donde se pierde tiempo posiblemente por la velocidad del bus de datos y por el tiempo que tarda en obtener una respuesta.

Cada elemento secuencial necesita, además del tiempo utilizado en la comunicación entre elementos secuenciales, tiempo para concatenar a la subimagen correspondiente los pixeles de los elementos vecinos.

En Manager-Workers la cantidad de canales de comunicación está limitada al número de workers existentes y cuando cada worker obtiene su subimagen procede directamente a realizar la convolución de la subimagen correspondiente, pues dicha subimagen ya contiene pixeles de sus vecinos.

Capítulo 6

Conclusiones

En este capítulo se analizan los resultados obtenidos en el Capítulo 5 para obtener las conclusiones finales para este trabajo.

6.1. Resumen

La característica más importante del cómputo paralelo es la ejecución de subtareas realizadas simultáneamente. Para la segmentación de imágenes realizada en esta tesis, cada subimagen puede ser procesada independientemente y simultáneamente por un hilo de ejecución o thread.

Sin embargo, esta ventaja puede llevar consigo dos problemas importantes:

- Problemas de sincronización, en los que se generen *deadlocks* [1] o que haya pérdida de información generando un resultado erróneo.
- Saturación de los canales de comunicación, provocando que el tiempo de ejecución sea más lento que el tiempo de ejecución obtenido secuencialmente. El aumento de procesadores puede disminuir el tiempo utilizado para terminar una tarea, pero también aumenta el tiempo empleado en sincronización de procesos y el flujo de información que pasa a través del bus de datos [2] aumentando el tiempo total empleado para el término de la tarea.

Las subimágenes que son procesadas simultáneamente son posibles obtenerlas mediante una partición de dominio, en la que la imagen original es dividida en las diferentes subimágenes. Cada subimagen se procesa simultáneamente y este proceso es independiente de algún otro proceso que se lleve al mismo tiempo. Esto significa que al procesar una subimagen no se requiere obtener información ya procesada de alguna otra.

La aplicación desarrollada fue diseñada únicamente para detección de bordes, por lo que los resultados mostrados o generados en un trabajo futuro, basados en esta tesis,

[1] Un deadlock sucede cuando un proceso A está en espera de datos de un proceso B, pero a su vez, el proceso B está en espera de datos del proceso A. Puesto que ambos procesos están esperando obtener información, no existe ninguna ejecución y por ende, el programa queda bloqueado sin la posibilidad de terminar.

[2] Si la comunicación es de un nodo de un clúster a otro, el bus de datos se refiere a la red utilizada.

únicamente son válidos para imágenes. La forma en que fue diseñada la aplicación, permite realizar algoritmos personalizados para cualquier segmentación de imágenes, cuya comunicación entre procesos sea independiente entre sí, es decir, que el resultado de un cálculo no dependa del resultado del cálculo de otro proceso.

6.2. Confirmación de la Hipótesis

La detección de bordes con el filtro de Sobel, es realizada en menor tiempo en su versión paralela, que si fuese ejecutada secuencialmente. El valor óptimo encontrado en las pruebas realizadas y bajo las condiciones mencionadas en la Sección 5.1, muestra que el menor tiempo de ejecución es de 3962 milisegundos. Para obtener este tiempo se utiliza la arquitectura Manager-Workers, seis procesadores y seis threads por procesador. Comparando este tiempo con el tiempo obtenido, igualmente con Manager-Workers, pero haciendo uso de únicamente un procesador y un thread, la diferencia de tiempo es de poco más de 10 segundos, por lo que la paralelización resulta conveniente.

Los resultados obtenidos en la Sección 5.2 muestran el tiempo total que tardan las arquitecturas Manager-Workers y CSE en efectuar la segmentación de la imagen, que en este caso, es la detección de bordes con el filtro de Sobel. Para este caso en particular, Manager-Workers realiza la detección de bordes en menor tiempo que CSE.

Comparando los tiempos de Manager-Workers y CSE en la Tabla 5.1, la menor diferencia de tiempo es de 52.5 milisegundos, cuando se utiliza un sólo procesador y thread; mientras que la mayor diferencia de tiempo es de 1560.5 milisegundos, en el caso de un procesador y 49 threads.

El objetivo de esta tesis es conocer una configuración óptima que minimice el tiempo de ejecución de una tarea depende de la observación y pruebas y del hardware y software que se maneja así como el problema a tratar. También se pretende mostrar la comparación entre las coordinaciones Manager-Workers y CSE. Los resultados obtenidos en la Sección 5.2, que muestran lo descrito, indican que bajo cualquier configuración la detección de bordes se efectúa en menor tiempo con la aqruitectura Manager-Workers. Esto confirma la hipótesis mencionada en la Sección 1.4: ‘‘`la arquitectura Manager-Workers permite realizar el procesamiento en menor cantidad de tiempo que CSE`’’.

6.3. Contribuciones

Lo que se obtiene de este trabajo es lo siguiente:

- Manager-Workers presenta menor tiempo de ejecución que CSE, para convolución de imágenes utilizando el filtro de Sobel.
- Desarrollo de código para convolución en paralelo, utilizando las coordinaciones Manager-Workers y CSE.
- Desarrollo de código para filtro de Sobel en paralelo.

- Estudio estadístico comparativo de este trabajo para arquitecturas de software paralelas.

6.4. Trabajo Futuro

Los trabajos que pueden generarse a partir de estos resultados con el fin de observar si éstos siguen siendo semejantes son:

- Comparación de las coordinaciones empleando cualquier tipo de imágenes. Con esto se pueden evaluar no sólo imágenes a escala de grises, sino también imágenes a color con formatos RGB o CMYK, por ejemplo.
- Comparación de las coordinaciones empleando algún hardware que permita ejecuciones paralelas. La finalidad de este punto, es verificar resultados con diferente hardware (nodos de computadoras, velocidad de red, etc.) para comparar dichos resultados con los obtenidos en este trabajo y conocer si las conclusiones son iguales.
- Comparación de las coordinaciones empleando diferentes algoritmos de segmentación de imágenes. Se puede estudiar empleando los filtros de Roberts o Prewitt, por ejemplo, o utilizando algoritmos de reconocimiento de objetos que impliquen mayor cantidad de cálculos.
- Extensión de la aplicación para soporte de cualquier algoritmo de segmentación de imagenes en diferentes formatos.

Bibliografía

[1] Jorge L. Ortega Arjona. *Architectural Patterns for Parallel Programming. Models for Performance Estimation.* VDM Verlag, Saarbrücken, Alemania, 2009.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed computation: Numerical Methods.* Athena Scientific, Massachusetts, Estados Unidos de América, 1997.

[3] Nick Efford. *Digital Image Processing. A practical Introduction using Java.* Addison Wesley, Massachusetts, Estados Unidos de América, 2000.

[4] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing.* John Wiley & Sons, Inc, New Jersey, Estados Unidos de América, 2005.

[5] Wesley Faler. Image manipulation by convolution. *The C Users Journal, Vol. 8, No. 8, pp. 95 - 99*, Agosto 1990.

[6] Ian Foster. Designing and building parallel programs. http://www.mcs.anl.gov/~itf/dbpp/, 1995.

[7] T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall International, Gran Bretaña, 1992.

[8] Rafael Gonzalez. *Digital Image Processing, 2da Edición.* Prentice Hall, New Jersey, Estados Unidos de América, 2002.

[9] Per Brinch Hansen. Distributed processes: A concurrent programing concept. *Communications of the ACM, Vol. 21, No. 11, pp. 934 - 941*, 1978.

[10] Stephen J. Hartley. *Concurrent Programming Using Java.* Oxford University Press, Estados Unidos de América, 1998.

[11] M. E. Martinez-Perez, A. D. Hughes, S. A. Thom, and K. H. Parker. Improvement of a retinal blood vessel segmentation method using the insight segmentation and registration toolkit (itk). In *29th IEEE EMBS Annual International Conference, pp. 23 - 26*, Lyon, Francia, Agosto 2007.

[12] M. E. Martinez-Perez, Alun D. Hughes, Simon A. Thom, Anil A. Bharath, and Kim H. Parker. Segmentation of blood vessels from red-free and fluorescein retinal images. *Medical Image Analysis, Vol 11, No. 1, pp. 47 - 61*, Enero 2007.

[13] Mark S. Nixon and Alberto S. Aguado. *Feature Extraction and Image Processing.* Newnes, Gran Bretaña, 2002.

[14] M.A. Palomera-Pérez, M.E. Martínez-Pérez, H. Benítez-Pérez, , and Ortega-Arjona J. L. Parallel multi-scale feature extraction and region growing: Application in retinal blood vessel detection. *IEEE Engineering in Medicine and Biology Society. Transactions on Information Technology in BioMedicine, Vol. 14, No. 2, pp. 500 - 506*, Marzo 2010.

[15] Behrooz Parhami. *Introduction to Parallel Processing. Algorithms and Architectures.* Kluwer Academic Publishers, Estados Unidos de América, 2002.

[16] Dwayne Phillips. *Image Processing in C, 2da Edición.* R&D Publications, Kansas, Estados Unidos de América, 2000.

[17] Todd R. Reed. *Digital Image, Sequence Processing, Compression and Analysis.* CRC Press, Estados Unidos de América, 2005.

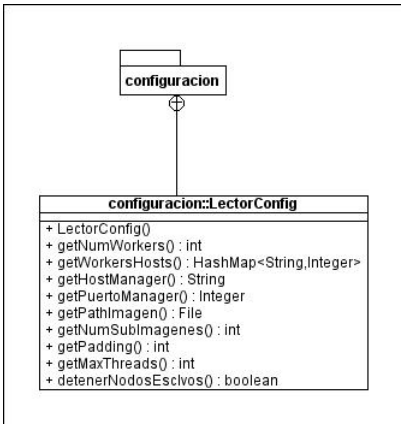
[18] William Stallings. *Operating Systems. Internals and Design Principles. 5ta Edición.* Prentice Hall, Estados Unidos de América, 2005.

[19] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms, 2da Edición.* Pearson Prentice Hall, Estados Unidos de América, 2006.

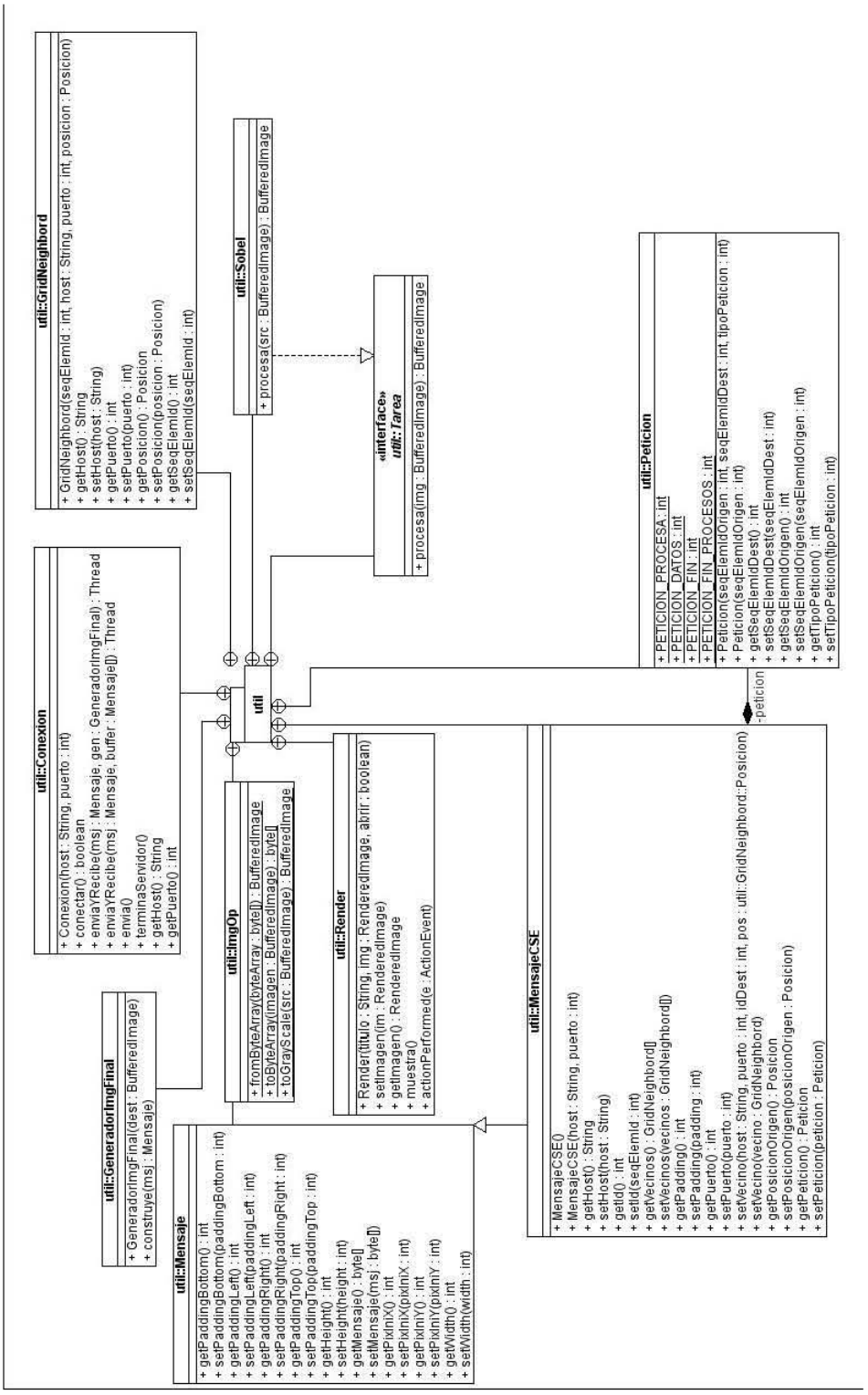
Apéndice A

Diagrama de Clases

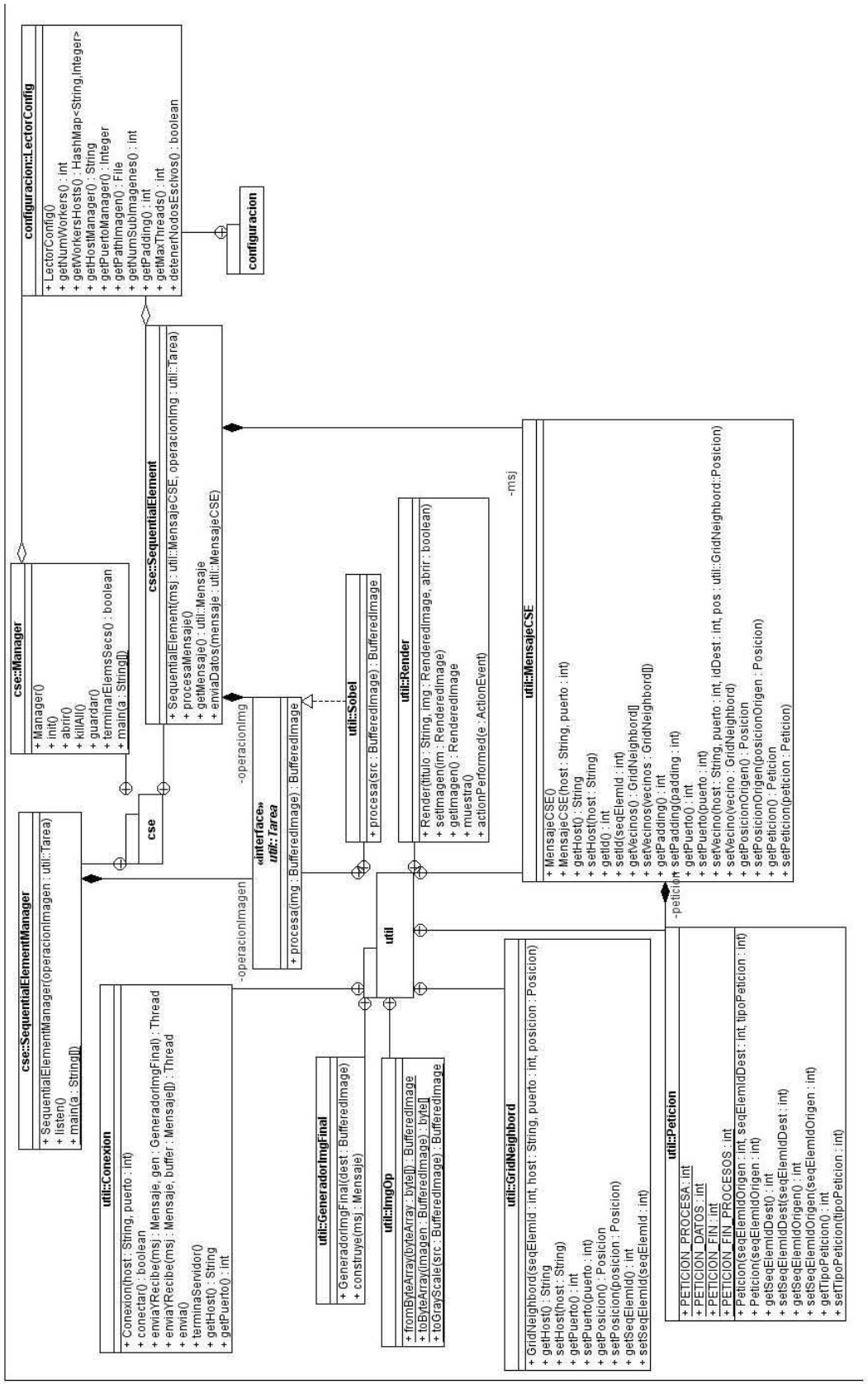
A.1. Paquete `configuracion`



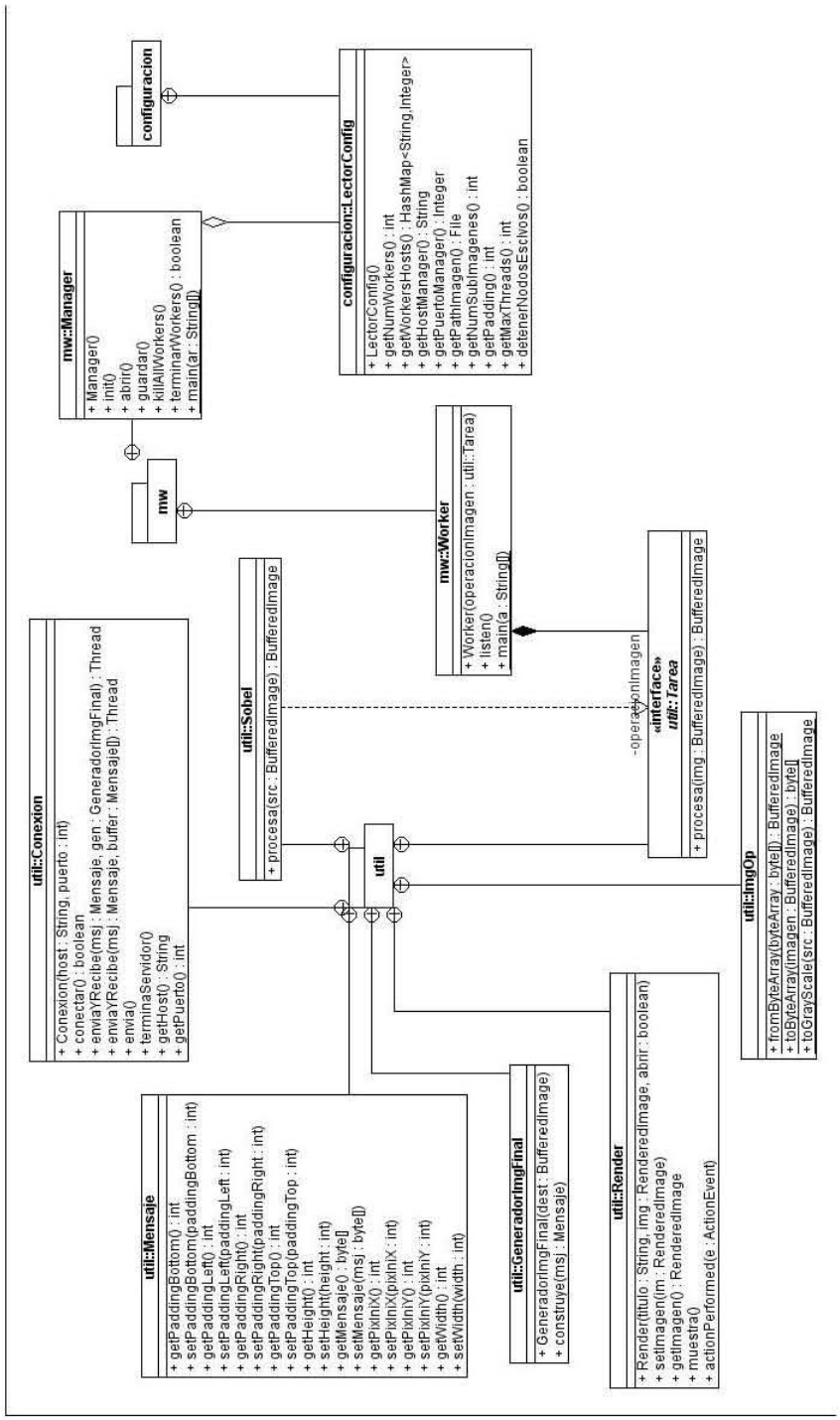
A.2. Paquete `util`



A.3. Paquete cse



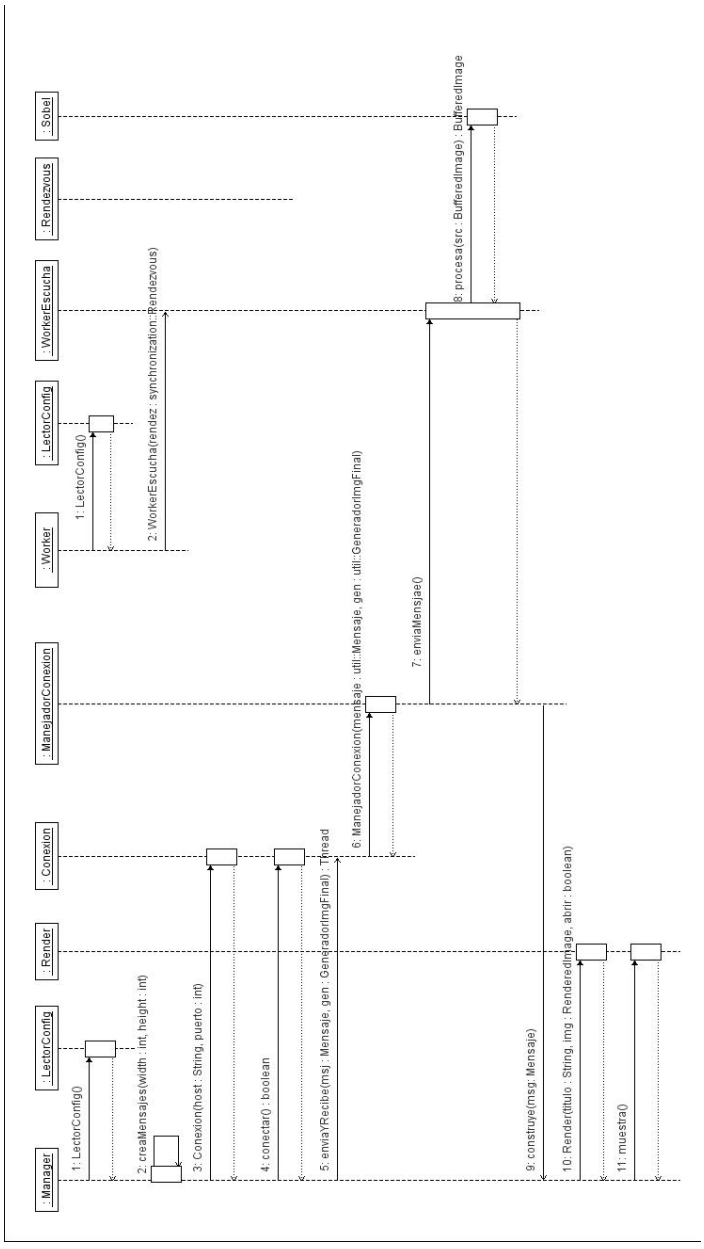
A.4. Paquete mw



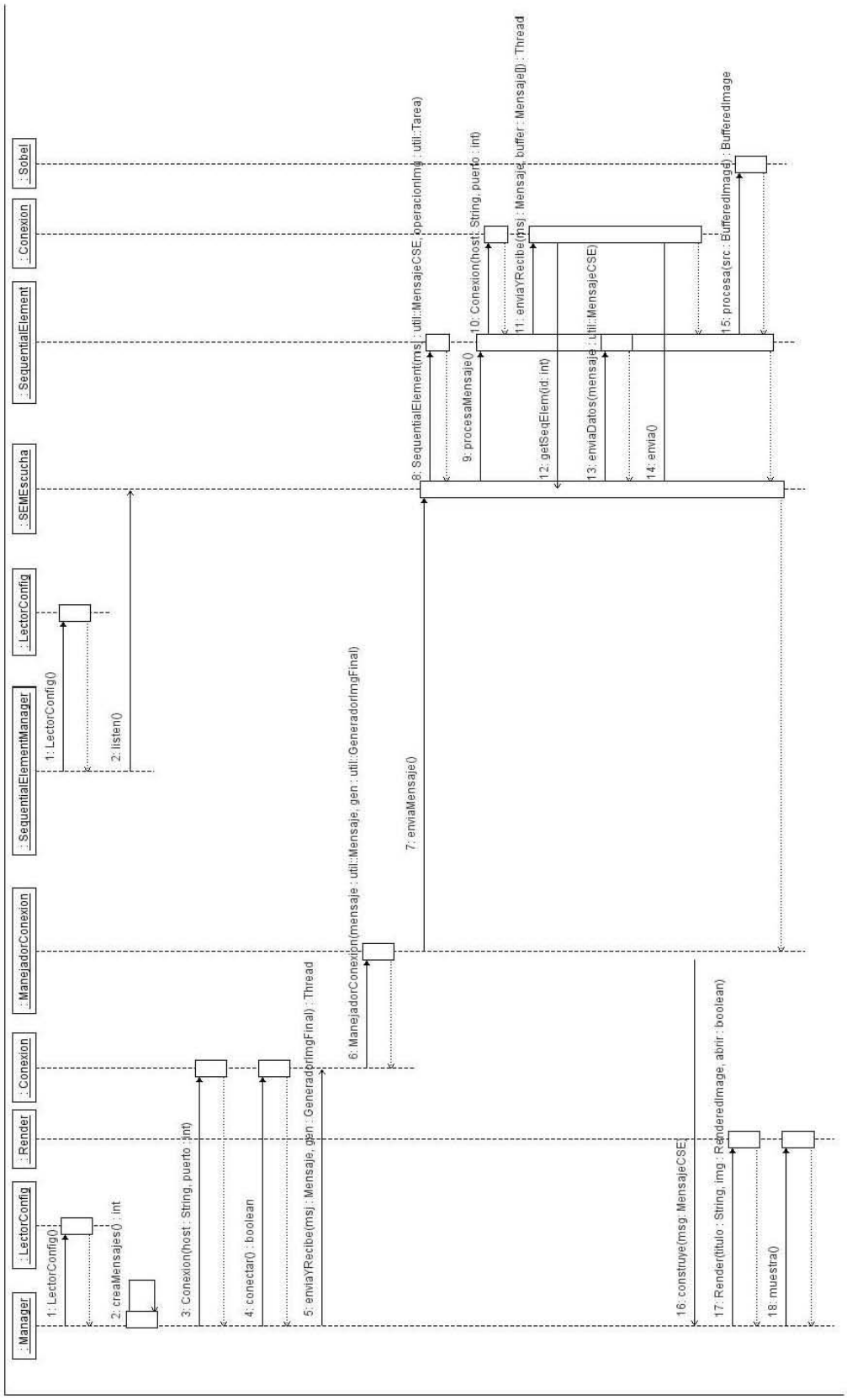
Apéndice B

Diagrama de Secuencias

B.1. Manager-Workers



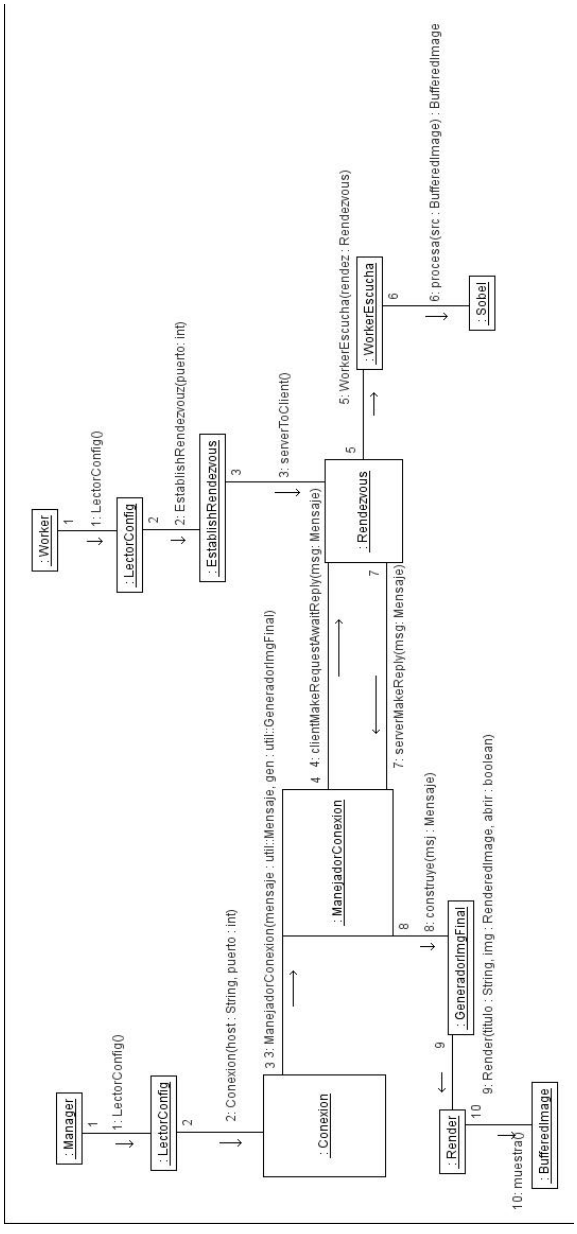
B.2. CSE


:Manager
:LectorConfig
:Render
:Conexion
:ManejadorConexion
:SequentialElementManager
:LectorConfig
:SEMEscucha
:SequentialElement
:Conexion
:Sobel
1: LectorConfig()
2: creaMensajes() : int
3: Conexion(host : String, puerto : int)
4: conectar() : boolean
5: enviaYRecibe(msj : Mensaje, gen : GeneradorImgFinal) : Thread
6: ManejadorConexion(mensaje : util::Mensaje, gen : util::GeneradorImgFinal)
1: LectorConfig()
2: listen()
7: enviaMensaje()
8: SequentialElement(msj : util::MensajeCSE, operacionImg : util::Tarea)
9: procesaMensaje()
10: Conexion(host: String, puerto : int)
11: enviaYRecibe(msj : Mensaje, buffer : Mensaje[]) : Thread
12: getSeqElem(id: int)
13: enviaDatos(mensaje : util::MensajeCSE)
14: envia()
15: procesa(src : BufferedImage) : BufferedImage
16: construye(msg: MensajeCSE)
17: Render(titulo : String, img : RenderedImage, abrir : boolean)
18: muestra()

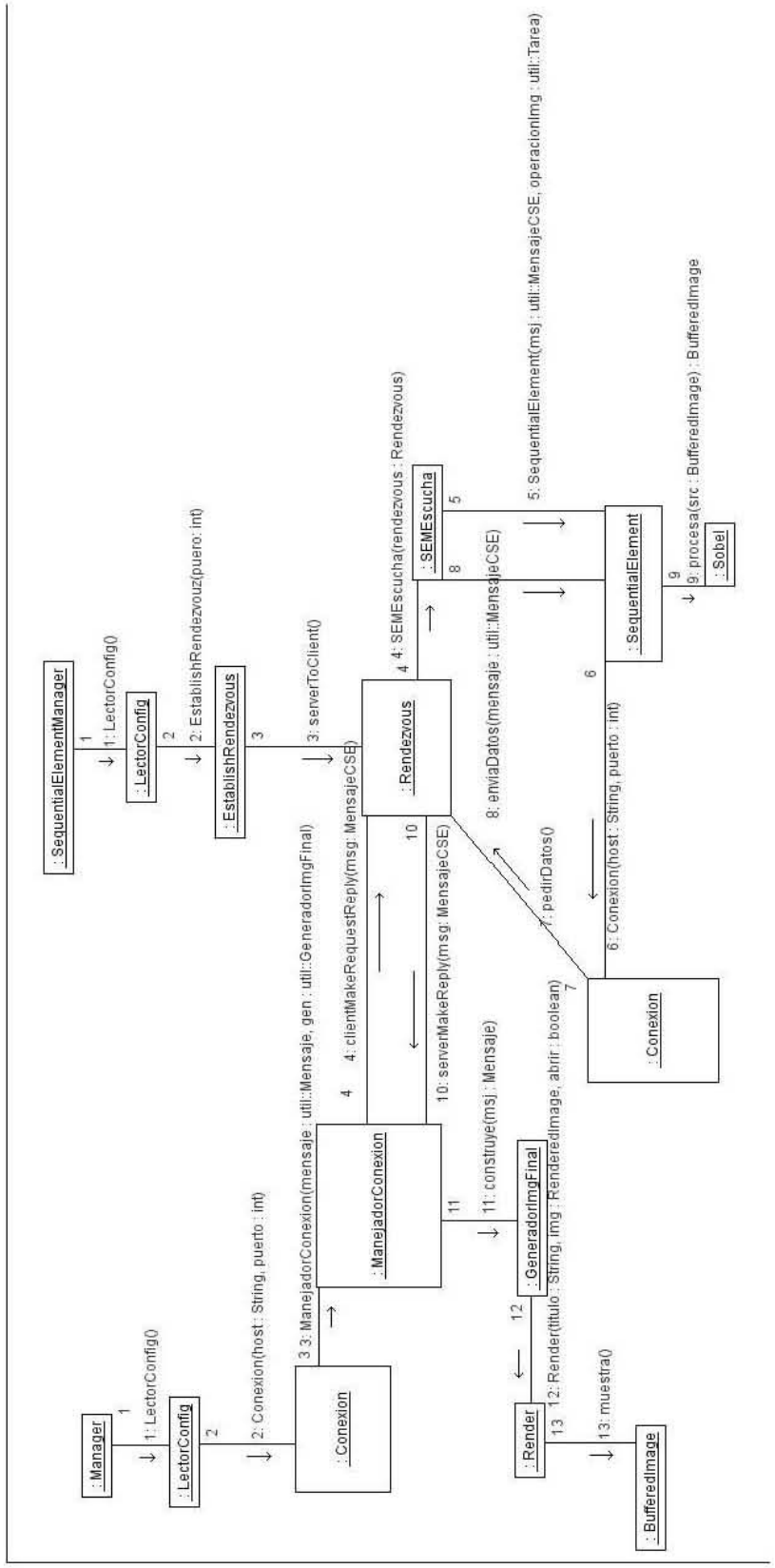
Apéndice C

Diagrama de Colaboraciones

C.1. Manager-Workers



C.2. CSE


:Manager
:LectorConfig
:Conexion
:ManejadorConexion
:Render
:BufferedImage
:SequentialElementManager
:LectorConfig
:EstablishRendezvous
:Rendezvous
:SEMEscucha
:SequentialElement
:Sobel
:Conexion
1: LectorConfig()
2: Conexion(host : String, puerto : int)
3 3: ManejadorConexion(mensaje : util::Mensaje, gen : util::GeneradorImgFinal)
4 4: clientMakeRequest\Reply(msg: MensajeCSE)
10: serverMakeReply(msg: MensajeCSE)
11 11: construye(msj : Mensaje)
12 12: Render(titulo : String, img : RenderedImage, abrir : boolean)
13 13: muestra()
1 1: LectorConfig()
2 2: EstablishRendezvouz(puero: int)
3 3: serverToClient()
4 4: SEMEscucha(rendezvous : Rendezvous)
5 5: SequentialElement(msj : util::MensajeCSE, operacionImg : util::Tarea)
6 6: Conexion(host : String, puerto : int)
7 7: pedirDatos()
8 8: enviaDatos(mensaje : util::MensajeCSE)
9 9: procesa(src : BufferedImage) : BufferedImage

Apéndice D

Código Fuente

D.1. configuracion

D.1.1. configuracion.LectorConfig.java

```
1  package configuracion;
2
3  import java.io.File;
4  import java.io.IOException;
5  import java.util.HashMap;
6  import java.util.logging.Logger;
7  import javax.xml.parsers.DocumentBuilder;
8  import javax.xml.parsers.DocumentBuilderFactory;
9  import javax.xml.parsers.ParserConfigurationException;
10 import org.w3c.dom.Document;
11 import org.w3c.dom.Element;
12 import org.w3c.dom.Node;
13 import org.w3c.dom.NodeList;
14 import org.xml.sax.ErrorHandler;
15 import org.xml.sax.SAXException;
16 import org.xml.sax.SAXParseException;
17
18 /**
19  *
20  * @author Felipe Navarrete Córdova
21  */
22 public class LectorConfig {
23
24     private static final Logger log =
25             Logger.getLogger(LectorConfig.class.getName());
26     private Document documento;
27
28     public LectorConfig() throws ParserConfigurationException, SAXException,
29             IOException {
30         DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
31         dbf.setValidating(true);
32         DocumentBuilder builder = dbf.newDocumentBuilder();
33         builder.setErrorHandler(new ErrorHandler() {
34
35             public void warning(SAXParseException exception) throws ↩
                   SAXException {
36                 log.warning(exception.getMessage());
37             }
38
39             public void error(SAXParseException exception) throws SAXException ↩
                   {
```

```
40                  log.severe("Error en la línea " + exception.getLineNumber()
41                          + ", columna " + exception.getColumnNumber());
42                  log.severe(exception.getMessage());
43                  System.exit(1);
44              }
45
46              public void fatalError(SAXParseException exception) throws ↩
                    SAXException {
47                  log.severe(exception.getMessage());
48                  System.exit(1);
49              }
50          });
51          documento = builder.parse(new File("config.xml"));
52      }
53
54      /**
55       * Obtiene el número de hosts entre los que se realizará el procesamiento
56       * paralelo.
57       * @return Número de hosts.
58       */
59      public int getNumWorkers() {
60          Node nodo = documento.getElementsByTagName("workers").item(0);
61          Element elem = (Element) nodo;
62          return elem.getElementsByTagName("host").getLength();
63      }
64
65      /**
66       * Obtiene los hosts entre los que se realizará el procesamiento paralelo.
67       * @return Los Hosts.
68       */
69      public HashMap<String, Integer> getWorkersHosts() {
70          HashMap<String, Integer> listaHosts = new HashMap<String, Integer>();
71          Node nodo = documento.getElementsByTagName("workers").item(0);
72          Element elem = (Element) nodo;
73          NodeList hosts = elem.getElementsByTagName("host");
74          for (int x = 0; x < hosts.getLength(); x++) {
75              Element host = (Element) hosts.item(x);
76              int puerto = Integer.parseInt(host.getAttribute("puerto"));
77              listaHosts.put(host.getTextContent(), puerto);
78
79          }
80          return listaHosts;
81      }
82
83      /**
84       * Obtiene el host manager.
85       * @return Host manager.
86       */
87      public String getHostManager() {
88          Element manager =
89                  (Element) documento.getElementsByTagName("manager").item(0);
90          return manager.getTextContent();
91      }
92
93      /**
94       * Obtiene el puerto por el que se establecerán las conexiones.
95       * @return El puerto.
96       */
97      public Integer getPuertoManager() {
98          Element manager =
99                  (Element) documento.getElementsByTagName("manager").item(0);
100         return Integer.parseInt(manager.getAttribute("puerto"));
101     }
102
103     /**
104      * Obtiene la imagen a procesar.
105      * @return La imagen a procesar o null si no existe.
```

```
106         */
107         public File getPathImagen() {
108             NodeList nodos = documento.getElementsByTagName("img");
109             File img = new File(nodos.item(0).getAttributes().getNamedItem("src").↩
                    getNodeValue());
110             return img.exists() ? img : null;
111         }
112 
113         /**
114          * Obtiene el número de subimágenes que se desean.
115          * @return úNmero de ásubimgenes.
116          */
117         @Deprecated
118         public int getNumSubImagenes() {
119             Element nodo = (Element) documento.getElementsByTagName("img").item(0);
120             return Integer.parseInt(nodo.getAttribute("subImgs"));
121         }
122 
123         /**
124          * Obtiene el padding o cantidad de pixeles que se le añadirán a la imagen.
125          * n pixeles de padding, significa que se le agregan n pixeles a la derecha↩
                ,
126          * izquierda, arriba y abajo (de ser posible).
127          * @return El padding.
128          */
129         public int getPadding() {
130             int valDefault = 0;
131             Element nodo = (Element) documento.getElementsByTagName("img").item(0);
132             try {
133                 int padding = Integer.parseInt(nodo.getAttribute("padding"));
134                 return padding;
135             } catch (Exception e) {
136                 return valDefault;
137             }
138         }
139 
140         /**
141          * Obtiene el máximo número de threads que se usarán por procesador.
142          * @return max threads.
143          */
144         public int getMaxThreads() {
145             int valDefault = 1;
146             Element nodo = (Element) documento.getElementsByTagName("maxThreads").↩
                    item(0);
147             try {
148                 return Integer.parseInt(nodo.getAttribute("value"));
149             } catch (Exception e) {
150                 return valDefault;
151             }
152         }
153 
154         /**
155          * Indica si es necesario detener la ejecución de cada nodo esclavo después
156          * de terminar el procesamiento de la imagen.
157          * @return true si hay que detenerlos.
158          */
159         public boolean detenerNodosEsclvos() {
160             Element nodo = (Element) documento.getElementsByTagName("pararNodosEscl↩
                    ").item(0);
161             return "si".equals(nodo.getAttribute("value"));
162         }
163 }
```

D.2. util

D.2.1. util.GeneradorImgFinal.java

```
1  package util;
2
3  import java.awt.image.BufferedImage;
4  import java.awt.image.WritableRaster;
5
6  /**
7   * Clase que permite generar la imagen final.
8   * @author Felipe Navarrete Córdova
9   */
10 public class GeneradorImgFinal {
11
12     protected BufferedImage dest;
13
14     /**
15      * Constructor.
16      * @param dest La imagen destino, donde se deplegarán las subimágenes
17      * procesadas de cada proceso participante en el procesamiento paralelo.
18      */
19     public GeneradorImgFinal(BufferedImage dest) {
20         this.dest = dest;
21     }
22
23     /**
24      * Permite que la imagen procesada y almacenada dentro de un mensaje se ↩
          dibuje
25      * en la imagen final que contiene todas las subimágenes de todos los ↩
          procesos
26      * participantes en el procesamiento paralelo.
27      * @param msj
28      */
29     public void construye(Mensaje msj) {
30         BufferedImage subImg = ImgOp.fromByteArray(msj.getMensaje());
31         WritableRaster wrSubIm = subImg.getRaster();
32         WritableRaster wrImFinal = dest.getRaster();
33
34         // óslo me interesan los datos no contenidos en el padding
35         int posIniX = msj.getPaddingLeft();
36         int posIniY = msj.getPaddingTop();
37         int posFinX = subImg.getWidth() - msj.getPaddingRight();
38         int posFinY = subImg.getHeight() - msj.getPaddingBottom();
39
40         for (int x = posIniX; x < posFinX; x++) {
41             for (int y = posIniY; y < posFinY; y++) {
42                 int pix = wrSubIm.getSample(x, y, 0);
43                 wrImFinal.setSample(msj.getPixIniX() + x - posIniX,
44                         msj.getPixIniY() + y - posIniY, 0, pix);
45             }
46         }
47     }
48 }
```

D.2.2. util.GridNeighbord.java

```
1  package util;
2
3  import java.io.Serializable;
```

```
4
5  /**
6   * Clase que representa un elemento secuencial vecino a otro elemento ↩
        secuencial
7   * dentro de la vecindad.
8   * @author Felipe Navarrete Córdova
9   */
10 public class GridNeighbord implements Serializable {
11
12     public enum Posicion {
13         IZQUIERDA, ARRIBA, DERECHA, ABAJO;
14     }
15
16     private int seqElemId;
17     private String host;
18     private int puerto;
19     private Posicion posicion;
20
21     public GridNeighbord (int seqElemId, String host, int puerto, Posicion ↩
           posicion) {
22         this.seqElemId = seqElemId;
23         this.host = host;
24         this.puerto = puerto;
25         this.posicion = posicion;
26     }
27
28     /**
29      * @return Regresa el host asociado a este vecino.
30      */
31     public String getHost() {
32         return host;
33     }
34
35     public void setHost(String host) {
36         this.host = host;
37     }
38
39     /**
40      * @return Regresa el puerto asociado a este vecino.
41      */
42     public int getPuerto() {
43         return puerto;
44     }
45
46     public void setPuerto(int puerto) {
47         this.puerto = puerto;
48     }
49
50     /**
51      * @return Regresa la posición del elemento vecino, en la malla de ↩
            elementos
52      * secuenciales, es decir, indica si se encuentra arriba, abajo, a la
53      * izquierda o derecha de un elemento secuencial.
54      */
55     public Posicion getPosicion() {
56         return posicion;
57     }
58
59     public void setPosicion(Posicion posicion) {
60         this.posicion = posicion;
61     }
62
63     /**
64      * @return Regresa el id de este vecino.
65      */
66     public int getSeqElemId() {
67         return seqElemId;
```

```
68      }
69
70      public void setSeqElemId(int seqElemId) {
71          this.seqElemId = seqElemId;
72      }
73  }
```

D.2.3. util.ImgOp.java

```
1   package util;
2
3   import java.awt.color.ColorSpace;
4   import java.awt.image.BufferedImage;
5   import java.awt.image.ColorConvertOp;
6   import java.io.ByteArrayInputStream;
7   import java.io.ByteArrayOutputStream;
8   import java.io.IOException;
9   import javax.imageio.ImageIO;
10
11  /**
12   * Clase que permite realizar operaciones con imágenes.
13   * @author Felipe Navarrete Córdova
14   */
15  public class ImgOp {
16
17      /**
18       * Crea una imagen a partir de un arreglo de bytes.
19       * @param byteArray Arreglo de bytes.
20       * @return Imagen.
21       */
22      public static BufferedImage fromByteArray(byte[] byteArray) {
23          if (byteArray == null) {
24              return null;
25          }
26          ByteArrayInputStream bais = new ByteArrayInputStream(byteArray);
27          BufferedImage imagen = null;
28          try {
29              imagen = ImageIO.read(bais);
30          } catch (IOException ex) {
31              ex.printStackTrace();
32          }
33          return imagen;
34      }
35
36      /**
37       * Genera un arreglo de bytes de una imagen. Sólo soprta archivos jpg.
38       * @param imagen La imagen a convertir.
39       * @return óRepresentacin en bytes de la imagen.
40       */
41      public static byte[] toByteArray(BufferedImage imagen) {
42          ByteArrayOutputStream baos = new ByteArrayOutputStream();
43          try {
44              ImageIO.write(imagen, "jpg", baos);
45          } catch (IOException ex) {
46              ex.printStackTrace();
47          }
48          return baos.toByteArray();
49      }
50
51      /**
52       * Convierte una imagen a escala de grises.
53       * @param src La imagen que se desea convertir.
54       * @return La imagen a escala de grises.
```

```
55          */
56         public static BufferedImage toGrayScale(BufferedImage src) {
57             BufferedImage imagen = new BufferedImage(src.getWidth(),
58                     src.getHeight(), src.getType());
59             ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
60             ColorConvertOp op = new ColorConvertOp(cs, null);
61             return op.filter(src, imagen);
62         }
63     }
```

D.2.4. util.Mensaje.java

```
1   package util;
2
3   import java.io.Serializable;
4
5   /**
6    * Clase que contiene los datos a enviar entre nodos o procesos.
7    * @author Felipe Navarrete Córdova
8    */
9   public class Mensaje implements Serializable {
10
11      private byte[] msj;
12      private int paddingTop;
13      private int paddingLeft;
14      private int paddingRight;
15      private int paddingBottom;
16      private int pixIniX;
17      private int pixIniY;
18      private int width;
19      private int height;
20
21      /**
22       * @return La cantidad de pixeles utilizados para padding inferior.
23       */
24      public int getPaddingBottom() {
25          return paddingBottom;
26      }
27
28      public void setPaddingBottom(int paddingBottom) {
29          this.paddingBottom = paddingBottom;
30      }
31
32      /**
33       * @return La cantidad de pixeles utilizados para padding izquierdo.
34       */
35      public int getPaddingLeft() {
36          return paddingLeft;
37      }
38
39      public void setPaddingLeft(int paddingLeft) {
40          this.paddingLeft = paddingLeft;
41      }
42
43      /**
44       * @return La cantidad de pixeles utilizados para padding derecho.
45       */
46      public int getPaddingRight() {
47          return paddingRight;
48      }
49
50      public void setPaddingRight(int paddingRight) {
51          this.paddingRight = paddingRight;
```

```
52      }
53
54      /**
55       * @return La cantidad de pixeles utilizados para padding superior.
56       */
57      public int getPaddingTop() {
58          return paddingTop;
59      }
60
61      public void setPaddingTop(int paddingTop) {
62          this.paddingTop = paddingTop;
63      }
64
65      /**
66       * @return La altura en pixeles que tiene la subimagen almacenada en este ↩
             mensaje.
67       */
68      public int getHeight() {
69          return height;
70      }
71
72      public void setHeight(int height) {
73          this.height = height;
74      }
75
76      /**
77       * @return La imagen en bytes.
78       */
79      public byte[] getMensaje() {
80          return msj;
81      }
82
83      public void setMensaje(byte[] msj) {
84          this.msj = msj;
85      }
86
87      /**
88       * @return La coordenada x de la imagen original donde inicia la porción de
89       * imagen almacenada en este mensaje.
90       */
91      public int getPixIniX() {
92          return pixIniX;
93      }
94
95      public void setPixIniX(int pixIniX) {
96          this.pixIniX = pixIniX;
97      }
98
99      /**
100      * @return La coordenada y de la imagen original donde inicia la porción de
101      * imagen almacenada en este mensaje.
102      */
103     public int getPixIniY() {
104         return pixIniY;
105     }
106
107     public void setPixIniY(int pixIniY) {
108         this.pixIniY = pixIniY;
109     }
110
111     /**
112      * @return El ancho en pixeles que tiene la subimagen almacenada en este ↩
             mensaje.
113      */
114     public int getWidth() {
115         return width;
116     }
```

```
117
118     public void setWidth(int width) {
119         this.width = width;
120     }
121 }
```

D.2.5. util.MensajeCSE.java

```
1  package util;
2
3  import util.GridNeighbord.Posicion;
4
5  /**
6   * Clase que contiene los datos a enviar entre nodos o procesos.
7   * @author Felipe Navarrete Córdova
8   */
9  public class MensajeCSE extends Mensaje {
10
11     private GridNeighbord[] vecinos;
12     private String host;
13     private int puerto;
14     private int id = -1;
15     private GridNeighbord.Posicion posicionOrigen;
16     private Peticion peticion;
17     private int padding;
18
19     public MensajeCSE() {
20         vecinos = new GridNeighbord[4];
21     }
22
23     /**
24      * Crea un nuevo mensaje especificando host y puerto destino de este ↩
            mensaje.
25      * @param host
26      * @param puerto
27      */
28     public MensajeCSE(String host, int puerto) {
29         vecinos = new GridNeighbord[4];
30         this.host = host;
31         this.puerto = puerto;
32     }
33
34     /**
35      * @return El host destino.
36      */
37     public String getHost() {
38         return host;
39     }
40
41     public void setHost(String host) {
42         this.host = host;
43     }
44
45     /**
46      * @return El id del elemento secuencial remitente.
47      */
48     public int getId() {
49         return id;
50     }
51
52     public void setId(int seqElemId) {
53         this.id = seqElemId;
54     }
```

```
55
56      /**
57       * @return La vecindad del elemento secuencial al que le pertenece este ↩
             mensaje.
58       */
59      public GridNeighbord[] getVecinos() {
60          return vecinos;
61      }
62
63      public void setVecinos(GridNeighbord[] vecinos) {
64          this.vecinos = vecinos;
65      }
66
67      /**
68       * @return La cantidad de pixeles que se necesitan de los vecinos.
69       */
70      public int getPadding() {
71          return padding;
72      }
73
74      public void setPadding(int padding) {
75          this.padding = padding;
76      }
77
78      /**
79       * @return El puerto destino.
80       */
81      public int getPuerto() {
82          return puerto;
83      }
84
85      public void setPuerto(int puerto) {
86          this.puerto = puerto;
87      }
88
89      /**
90       * Actualiza un vecino de un elemento secuencial. Este método tiene uso
91       * cuando el managerCSE configura el mapeo de procesos y así, un elemento
92       * secuencial puede conocer sus vecinos.
93       * @param host
94       * @param puerto
95       * @param idDest
96       * @param pos
97       */
98      public void setVecino(String host, int puerto, int idDest, GridNeighbord.↩
             Posicion pos) {
99          vecinos[pos.ordinal()] = new GridNeighbord(idDest, host, puerto, pos);
100     }
101
102     /**
103      * Actualiza un vecino.
104      * @param vecino El vecino nuevo.
105      */
106     public void setVecino(GridNeighbord vecino) {
107         vecinos[vecino.getPosicion().ordinal()] = vecino;
108     }
109
110     /**
111      * Al mandarse este mensaje se debe conocer la posición del remitente ↩
             respecto
112      * al receptor, lo cual este método permite.
113      * @return La posición del elemento remitente respecto al receptor.
114      */
115     public Posicion getPosicionOrigen() {
116         return posicionOrigen;
117     }
118
```

```
119     /**
120      * Al mandarse este mensaje se debe actualizar la posición del elemento
121      * remitente respecto al receptor.
122      * @param posicionOrigen La posición del elemento remitente respecto al
123      * receptor.
124      */
125     public void setPosicionOrigen(Posicion posicionOrigen) {
126         this.posicionOrigen = posicionOrigen;
127     }
128 
129     /**
130      * @return El tipo de petición que solicita el elemento remitente.
131      */
132     public Peticion getPeticion() {
133         return peticion;
134     }
135 
136     /**
137      * Actualiza la petición del elemento remitente.
138      * @param peticion La petición del elemento remitente.
139      */
140     public void setPeticion(Peticion peticion) {
141         this.peticion = peticion;
142     }
143 }
```

D.2.6. util.Peticion.java

```
1  package util;
2  
3  import java.io.Serializable;
4  
5  /**
6   *
7   * @author Felipe Navarrete Córdova
8   */
9  public class Peticion implements Serializable {
10 
11     /** Indica que se debe realizar procesamiento de datos */
12     public static final int PETICION_PROCESA = 0;
13     /** Indica que se piden datos */
14     public static final int PETICION_DATOS = 1;
15     /** Indica que no hay mas por procesar */
16     public static final int PETICION_FIN = 2;
17     public static final int PETICION_FIN_PROCESOS = 3;
18 
19     private int seqElemIdDest;
20     private int seqElemIdOrigen;
21     private int tipoPeticion;
22 
23     /**
24      * Constructor.
25      * @param seqElemIdOrigen Id del elemento secuencial de origen.
26      * @param seqElemIdDest Id del elemento secuencial destino.
27      * @param tipoPeticion Tipo de petición.
28      */
29     public Peticion(int seqElemIdOrigen, int seqElemIdDest, int tipoPeticion) {
30         this.seqElemIdDest = seqElemIdDest;
31         this.seqElemIdOrigen = seqElemIdOrigen;
32         this.tipoPeticion = tipoPeticion;
33     }
34 
35     /**
```

```
36          * Constructor.
37          * @param seqElemIdOrigen Id del elemento secuencial origen.
38          */
39         public Peticion(int seqElemIdOrigen) {
40             this(seqElemIdOrigen, seqElemIdOrigen, PETICION_PROCESA);
41         }
42 
43         /**
44          * @return Regresa el id del elemento secuencial destino.
45          */
46         public int getSeqElemIdDest() {
47             return seqElemIdDest;
48         }
49 
50         public void setSeqElemIdDest(int seqElemIdDest) {
51             this.seqElemIdDest = seqElemIdDest;
52         }
53 
54         /**
55          * @return Regresa el id del elemento secuencial de origen.
56          */
57         public int getSeqElemIdOrigen() {
58             return seqElemIdOrigen;
59         }
60 
61         public void setSeqElemIdOrigen(int seqElemIdOrigen) {
62             this.seqElemIdOrigen = seqElemIdOrigen;
63         }
64 
65         /**
66          * @return Regresa el tipo de petición.
67          */
68         public int getTipoPeticion() {
69             return tipoPeticion;
70         }
71 
72         public void setTipoPeticion(int tipoPeticion) {
73             this.tipoPeticion = tipoPeticion;
74         }
75 }
```

D.2.7. util.Render.java

```
1  package util;
2 
3  import java.awt.BorderLayout;
4  import java.awt.Desktop;
5  import java.awt.Dimension;
6  import java.awt.Graphics;
7  import java.awt.Graphics2D;
8  import java.awt.event.ActionEvent;
9  import java.awt.event.ActionListener;
10 import java.awt.geom.AffineTransform;
11 import java.awt.image.RenderedImage;
12 import java.io.File;
13 import java.io.IOException;
14 import java.net.URI;
15 import java.net.URISyntaxException;
16 import java.util.logging.Level;
17 import java.util.logging.Logger;
18 import javax.imageio.ImageIO;
19 import javax.swing.BoxLayout;
20 import javax.swing.JButton;
```

```
21 import javax.swing.JComponent;
22 import javax.swing.JFrame;
23 import javax.swing.JPanel;
24 import javax.swing.JScrollPane;
25
26 /**
27  *
28  * @author Felipe Navarrete Córdova
29  */
30 public class Render extends JFrame implements ActionListener {
31
32     private static final Logger log = Logger.getLogger(Render.class.getName());
33     private RenderedImage src;
34     private PanelImagen panel;
35     private String ruta;
36     private String titulo;
37
38     /**
39      * Crea una nueva ventana con alguna imagen para mostrar. Es posible abrir
40      * las imágenes con el programa predeterminado por el sistema operativo
41      * aunque la imagen se encuentre en memoria.
42      * @param titulo El título de la ventana.
43      * @param img La imagen a mostrar.
44      * @param abrir Indica si se muestra un botón para abrir la imagen con el
45      * programa predeterminado por el sistema operativo para abrir imagenes.
46      */
47     public Render(String titulo, RenderedImage img, boolean abrir) {
48         super(titulo);
49         this.titulo = titulo;
50         this.src = img;
51         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
52         panel = new PanelImagen(src);
53         JScrollPane scroll = new JScrollPane(panel);
54         add(scroll, BorderLayout.CENTER);
55         if (abrir) {
56             JButton abrirA = new JButton("Abrir archivo");
57             abrirA.setActionCommand("abrirArchivo");
58             JPanel otro = new JPanel();
59             BoxLayout layout = new BoxLayout(otro, BoxLayout.X_AXIS);
60             otro.setLayout(layout);
61             otro.add(abrirA);
62             add(otro, BorderLayout.NORTH);
63             abrirA.addActionListener(this);
64         }
65     }
66
67     /**
68      * Actualiza la imagen a mostrar.
69      * @param im La nueva imagen.
70      */
71     public void setImagen(RenderedImage im) {
72         this.src = im;
73         panel.setImagen(im);
74         panel.repaint();
75     }
76
77     public RenderedImage getImagen() {
78         return this.src;
79     }
80
81     /**
82      * Ocaciona que se abra la ventana para mostrar la imagen.
83      */
84     public void muestra() {
85         this.pack();
86         this.setVisible(true);
87     }
```

```
88
89      public void actionPerformed(ActionEvent e) {
90          if (e.getActionCommand() != null) {
91              if (Desktop.isDesktopSupported()) {
92                  Desktop desk = Desktop.getDesktop();
93                  try {
94                      if (e.getActionCommand().equals("abrirArchivo")) {
95                          if (ruta == null) {
96                              File file = File.createTempFile("img" + titulo, ↩
                                    null);
97                              ImageIO.write(src, "jpg", file);
98                              desk.open(file);
99                          } else {
100                             desk.open(new File(ruta));
101                         }
102                     } else if (e.getActionCommand().equals("abrirCarpeta")) {
103                         try {
104                             desk.browse(new URI("file://"
105                                     + new File(ruta).getParent()));
106                         } catch (URISyntaxException ex) {
107                             log.log(Level.SEVERE, null, ex);
108                         }
109                     }
110                 } catch (IOException ex) {
111                     log.warning(ex.getMessage());
112                 }
113             }
114         }
115     }
116
117     /**
118      * Clase auxiliar en la que se pinta el componente de la imagen
119      */
120     class PanelImagen extends JComponent {
121
122         private RenderedImage img;
123
124         public PanelImagen(RenderedImage imagen) {
125             this.img = imagen;
126         }
127
128         public void setImagen(RenderedImage img) {
129             this.img = img;
130         }
131
132         @Override
133         public void paintComponent(Graphics g) {
134             Graphics2D g2 = (Graphics2D) g;
135             if (g2 != null) {
136                 g2.drawRenderedImage(img, new AffineTransform());
137             }
138         }
139
140         @Override
141         public Dimension getPreferredSize() {
142             return new Dimension(img.getWidth(), img.getHeight());
143         }
144     }
145 }
```

D.2.8. util.Tarea.java

```
1  package util;
```

```
2
3  import java.awt.image.BufferedImage;
4
5  /**
6   *
7   * @author Felipe Navarrete Córdova
8   */
9  public interface Tarea {
10
11      /**
12       * Realiza la segmentación de una imagen.
13       * @param img La imagen a segmentar.
14       * @return La imagen segmentada.
15       */
16      public BufferedImage procesa(BufferedImage img);
17
18 }
```

D.2.9. util.Sobel.java

```
1   package util;
2
3  import java.awt.image.BufferedImage;
4  import java.awt.image.WritableRaster;
5
6  /**
7   *
8   * @author Felipe Navarrete Córdova
9   */
10 public class Sobel implements Tarea {
11
12      /** Filtro sobre eje x */
13       private static final short[][] GX = {{-1, 0,  1},
14                                           {-2, 0,  2},
15                                           {-1, 0,  1}};
16      /** Filtro sobre eje y */
17      private static final short[][] GY = {{-1,  -2, -1},
18                                          {0,   0,  0},
19                                          {1,   2,  1}};
20      /**
21       * {@inheritDoc}
22       */
23      @Override
24      public BufferedImage procesa(BufferedImage src) {
25          BufferedImage nueva = new BufferedImage(src.getWidth(), src.getHeight()↩
                ,
26                  BufferedImage.TYPE_BYTE_GRAY);
27          WritableRaster wrN = nueva.getRaster();
28          WritableRaster wr = src.getRaster();
29
30          for (int x = 0; x < src.getWidth(); x++) {
31              for (int y = 0; y < src.getHeight();  y++) {
32                  int gradX = convolucion(GX, wr, x, y);
33                  int gradY = convolucion(GY, wr, x, y);
34                  int magnitud = Math.abs(gradX) + Math.abs(gradY);
35                  magnitud = (int) (magnitud - magnitud * 0.25);
36                  if (magnitud > 255) magnitud = 255;
37                  if (magnitud < 0) magnitud = 0;
38                  wrN.setSample(x, y, 0, magnitud);
39              }
40          }
41          return nueva;
42      }
```

```
43
44      /**
45       * Realiza la convolución en una imagen con un operador dado.
46       * @param kernel EL operador utilizado para la convolución.
47       * @param rasterImg La información de la imagen que se desea convolucionar.
48       * @param coordX Posición en el eje X del pixel central.
49       * @param coordY Posición en el eje Y del pixel central.
50       * @return El gradiente resultante del pixel central.
51       */
52      private int convolucion(short[][] kernel, WritableRaster rasterImg,
53              int coordX, int coordY) {
54          int sum = 0;
55          for (int x = 0; x < 3; x++) {
56              for (int y = 0; y < 3; y++) {
57                  try {
58                      int tmp = rasterImg.getSample(coordX -1 + x, coordY - 1 + y↩
                            , 0);
59                      sum += tmp * kernel[x][y];
60                  } catch (ArrayIndexOutOfBoundsException arr) {
61                      continue;
62                  }
63              }
64          }
65          return sum;
66      }
67
68  }
```

D.3. mw

D.3.1. mw.Manager.java

```
1   package mw;
2
3   import configuracion.LectorConfig;
4   import java.awt.image.BufferedImage;
5   import java.io.File;
6   import java.io.IOException;
7   import java.util.ArrayList;
8   import java.util.HashMap;
9   import java.util.logging.Level;
10  import java.util.logging.Logger;
11  import javax.imageio.ImageIO;
12  import javax.xml.parsers.ParserConfigurationException;
13  import org.xml.sax.SAXException;
14  import util.Conexion;
15  import util.GeneradorImgFinal;
16  import util.ImgOp;
17  import util.Mensaje;
18  import util.Render;
19
20  /**
21   * Esta clase representa el nodo maestro, quien distribuye y asigna las tareas.
22   * @author Felipe Navarrete Córdova
23   */
24  public class Manager {
25
26      private static final Logger log = Logger.getLogger(Manager.class.getName())↩
            ;
27      private HashMap<String, Integer> hosts;
28      private boolean terminarWorkers;
29      private BufferedImage imagen;
```

```
30      private int padding;
31      private int maxThreads;
32      private ArrayList<Mensaje> mensajes;
33      private BufferedImage imgFinal;
34
35      private long timeIni;
36
37      /**
38       * Constructor.
39       */
40      public Manager() {
41          LectorConfig lector = null;
42          try {
43              lector = new LectorConfig();
44          } catch (ParserConfigurationException ex) {
45              log.log(Level.SEVERE, null, ex);
46              System.exit(1);
47          } catch (SAXException ex) {
48              log.log(Level.SEVERE, null, ex);
49              System.exit(1);
50          } catch (IOException ex) {
51              log.log(Level.SEVERE, null, ex);
52              System.exit(1);
53          }
54
55          hosts = lector.getWorkersHosts();
56          terminarWorkers = lector.detenerNodosEsclvos();
57          try {
58              imagen = ImageIO.read(lector.getPathImagen());
59          } catch (IOException ex) {
60              log.log(Level.SEVERE, null, ex);
61              System.exit(1);
62          }
63          padding = lector.getPadding();
64          // Convierte a escala de grises de ser necesario
65          if (imagen.getType() != BufferedImage.TYPE_BYTE_GRAY) {
66              imagen = ImgOp.toGrayScale(imagen);
67          }
68          maxThreads = lector.getMaxThreads();
69
70          if (hosts.isEmpty()) {
71              log.severe("No hay hosts disponibles");
72              System.exit(1);
73          }
74
75          log.info("Hosts a usar: " + hosts.size());
76          log.info("Threads por host: " + maxThreads);
77          log.info("Mensajes a enviar: " + hosts.size() * maxThreads);
78          timeIni = System.currentTimeMillis();
79          creaMensajes(hosts.size(), maxThreads);
80      }
81
82      /**
83       * Crea Los mensajes a enviar
84       * @param width El factor de división de la imagen a lo ancho.
85       * @param height El factor de división de la imagen a lo alto.
86       * @return Los mensajes creeados.
87       */
88      private void creaMensajes(int width, int height) {
89          int[] posActual = {0, 0};
90          int w = (int) Math.ceil((double) imagen.getWidth() / (double) width);
91          int h = (int) Math.ceil((double) imagen.getHeight() / (double) height);
92          mensajes = new ArrayList<Mensaje>();
93
94          while (true) {
95              Mensaje mensaje = new Mensaje();
96              BufferedImage img = null;
```

```
97              if (posActual[0] >= imagen.getWidth()) {
98                  posActual[0] = 0;
99                  posActual[1] += h;
100             }
101             if (posActual[1] >= imagen.getHeight()) {
102                 break;
103             }
104             int ancho = w;
105             int alto = h;
106             int posIniX = posActual[0];
107             int posIniY = posActual[1];
108
109             mensaje.setPixIniX(posIniX);
110             mensaje.setPixIniY(posIniY);
111             // Verificando ancho y alto
112             if (posActual[0] + w >= imagen.getWidth()) {
113                 ancho = imagen.getWidth() - posActual[0];
114
115             }
116             if (posActual[1] + h >= imagen.getHeight()) {
117                 alto = imagen.getHeight() - posActual[1];
118             }
119             // Verificando padding
120             if (posIniX - padding >= 0) {
121                 posIniX -= padding;
122                 ancho += padding;
123                 mensaje.setPaddingLeft(padding);
124             }
125             if (posIniY - padding >= 0) {
126                 posIniY -= padding;
127                 alto += padding;
128                 mensaje.setPaddingTop(padding);
129             }
130             if (posIniX + ancho + padding < imagen.getWidth()) {
131                 ancho += padding;
132                 mensaje.setPaddingRight(padding);
133             }
134             if (posIniY + alto + padding < imagen.getHeight()) {
135                 alto += padding;
136                 mensaje.setPaddingBottom(padding);
137             }
138             // actualizo datos del mensaje
139             img = imagen.getSubimage(posIniX, posIniY, ancho, alto);
140             mensaje.setMensaje(ImgOp.toByteArray(img));
141             mensaje.setWidth(ancho);
142             mensaje.setHeight(alto);
143             posActual[0] += w;
144             mensajes.add(mensaje);
145         }
146     }
147
148     /**
149      * Inicia el procesamiento en paralelo: se generan las conexiones con los
150      * diferentes nodos y se envian los mensajes a procesar.
151      */
152     public void init() {
153         ArrayList<Thread> threads = new ArrayList<Thread>();
154         imgFinal = new BufferedImage(imagen.getWidth(),
155                 imagen.getHeight(), BufferedImage.TYPE_BYTE_GRAY);
156         for (String host : hosts.keySet()) {
157             for (int x = 0; x < maxThreads; x++) {
158                 Conexion con = new Conexion(host, hosts.get(host));
159                 con.conectar();
160                 Mensaje msj = mensajes.remove(0);
161                 GeneradorImgFinal gen = new GeneradorImgFinal(imgFinal);
162                 threads.add(con.enviaYRecibe(msj, gen));
163             }
```

```
164             }
165             for (Thread t : threads) {
166                 t.start();
167             }
168             for (Thread t : threads) {
169                 try {
170                     t.join();
171                 } catch (InterruptedException ex) {
172                     log.log(Level.SEVERE, null, ex);
173                 }
174             }
175             System.out.printf("Tiempo Final: %d\n", System.currentTimeMillis() - ↩
                    timeIni);
176         }
177 
178         /**
179          * Permite mostrar la imagen procesada.
180          */
181         public void abrir() {
182             new Render("MW", imgFinal, true).muestra();
183         }
184 
185         /**
186          * Guarda la imagen resultante.
187          */
188         public void guardar() {
189             File archivoFinal = new File("finalMW.jpg");
190             try {
191                 ImageIO.write(imgFinal, "jpg", archivoFinal);
192                 imagen = null;
193             } catch (IOException ex) {
194                 log.log(Level.SEVERE, null, ex);
195             } catch (Exception ex) {
196                 log.warning(ex.getMessage());
197             }
198         }
199 
200         /**
201          * Envía señal de apagado a los nodos especificados en la configuración.
202          */
203         public void killAllWorkers() {
204             for (String host : hosts.keySet()) {
205                 Conexion con = new Conexion(host, hosts.get(host));
206                 con.conectar();
207                 con.terminaServidor();
208             }
209         }
210 
211         /**
212          * Indica si en la configuración se pide terminar cada uno de los nodos ↩
                activos
213          * después de haber realizado el procesamiento.
214          * @return <code>true</code> si se deben terminar.
215          */
216         public boolean terminarWorkers() {
217             return terminarWorkers;
218         }
219 
220         /**
221          * Main de la clase mw.Manager.
222          */
223         public static void main(String[] ar) {
224             Manager m = new Manager();
225             m.init();
226             if (m.terminarWorkers) {
227                 m.killAllWorkers();
228             }
```

```
229             m.abrir();
230         }
231     }
```

D.3.2. mw.Worker.java

```
1   package mw;
2
3   import Synchronization.EstablishRendezvous;
4   import Synchronization.MessagePassingException;
5   import Synchronization.Rendezvous;
6   import configuracion.LectorConfig;
7   import java.awt.image.BufferedImage;
8   import java.io.IOException;
9   import java.net.Inet4Address;
10  import java.net.InetAddress;
11  import java.net.UnknownHostException;
12  import java.util.logging.Level;
13  import java.util.logging.Logger;
14  import javax.xml.parsers.ParserConfigurationException;
15  import org.xml.sax.SAXException;
16  import util.ImgOp;
17  import util.Mensaje;
18  import util.Sobel;
19  import util.Tarea;
20
21  /**
22   *
23   * @author Felipe Navarrete Córdova
24   */
25  public class Worker {
26
27      private static final Logger log = Logger.getLogger(Worker.class.getName());
28      private int puerto = 9090;
29      private Tarea operacionImagen;
30      private EstablishRendezvous er;
31
32      public Worker(Tarea operacionImagen) {
33          log.info("======= Iniciando Worker =======");
34          log.info("Leyendo óconfiguracin");
35          LectorConfig lc = null;
36          try {
37              lc = new LectorConfig();
38          } catch (ParserConfigurationException ex) {
39              log.log(Level.SEVERE, null, ex);
40              System.exit(1);
41          } catch (SAXException ex) {
42              log.log(Level.SEVERE, null, ex);
43              System.exit(1);
44          } catch (IOException ex) {
45              log.log(Level.SEVERE, null, ex);
46              System.exit(1);
47          }
48          InetAddress inet = null;
49          try {
50              inet = Inet4Address.getLocalHost();
51          } catch (UnknownHostException ex) {
52              log.log(Level.SEVERE, null, ex);
53              System.exit(1);
54          }
55          for (String host : lc.getWorkersHosts().keySet()) {
56              if (host.equals(inet.getHostAddress())
57                      host.equals(inet.getHostName())) {
```

```
58                  puerto = lc.getWorkersHosts().get(host);
59                  break;
60              }
61          }
62          this.operacionImagen = operacionImagen;
63          log.info("Se áutilizar el puerto " + puerto);
64          System.out.println("\n\n");
65          er = new EstablishRendezvous(puerto);
66      }
67
68      /**
69       * Abre la conexión en red a través del puerto especificado en el archivo ↩
             de
70       * configuración para recibir peticiones.
71       */
72      public void listen() {
73          while (true) {
74              try {
75                  Rendezvous rendezvous = er.serverToClient();
76                  new WorkerEscucha(rendezvous).start();
77              } catch (MessagePassingException ex) {
78                  log.info("óConexin interrumpida");
79                  ex.printStackTrace();
80                  break;
81              }
82          }
83      }
84
85      private class WorkerEscucha extends Thread {
86
87          private Rendezvous rendezvouz;
88
89          public WorkerEscucha(Rendezvous rendez) {
90              this.rendezvouz = rendez;
91          }
92
93          public void run() {
94              Tarea tarea = operacionImagen;
95              Mensaje msg =
96                      (Mensaje) rendezvouz.serverGetRequest();
97              if (msg == null) {
98                  log.info("Terminando Procesos");
99                  rendezvouz.serverMakeReply(null);
100                 rendezvouz.close();
101                 System.exit(0);
102             }
103             BufferedImage img =
104                     ImgOp.fromByteArray(msg.getMensaje());
105             msg.setMensaje(ImgOp.toByteArray(tarea.procesa(img)));
106             rendezvouz.serverMakeReply(msg);
107             rendezvouz.close();
108         }
109     }
110
111     /**
112      * Método main de la clase mw.Worker
113      */
114     public static void main(String[] a) {
115         Worker w = new Worker(new Sobel());
116         w.listen();
117     }
118 }
```

D.4. cse

D.4.1. cse.Manager.java

```
1  package cse;
2
3  import util.MensajeCSE;
4  import configuracion.LectorConfig;
5  import java.awt.image.BufferedImage;
6  import java.io.File;
7  import java.io.IOException;
8  import java.util.ArrayList;
9  import java.util.HashMap;
10 import java.util.logging.Level;
11 import java.util.logging.Logger;
12 import javax.imageio.ImageIO;
13 import javax.xml.parsers.ParserConfigurationException;
14 import org.xml.sax.SAXException;
15 import util.Conexion;
16 import util.GeneradorImgFinal;
17 import util.GridNeighbord;
18 import util.ImgOp;
19 import util.Peticion;
20 import util.Render;
21 /**
22  * Esta clase representa el nodo maestro, quien distribuye y asigna las tareas.
23  * @author Felipe Navarrete Córdova
24  */
25 public class Manager {
26
27     private static final Logger log = Logger.getLogger(Manager.class.getName())↩
          ;
28
29     private BufferedImage imagen;
30     private HashMap<String, Integer> hosts;
31     private boolean terminarElemsSecs;
32     private int padding;
33     private int maxThreads;
34
35     private MensajeCSE[][] gridMensajes;
36     private BufferedImage imgFinal;
37
38     private long timeIni;
39
40     public Manager() {
41         LectorConfig lector = null;
42         try {
43             lector = new LectorConfig();
44         } catch (ParserConfigurationException ex) {
45             log.log(Level.SEVERE, null, ex);
46             System.exit(1);
47         } catch (SAXException ex) {
48             log.log(Level.SEVERE, null, ex);
49             System.exit(1);
50         } catch (IOException ex) {
51             log.log(Level.SEVERE, null, ex);
52             System.exit(1);
53         }
54         hosts = lector.getWorkersHosts();
55         terminarElemsSecs = lector.detenerNodosEsclvos();
56         try {
57             imagen = ImageIO.read(lector.getPathImagen());
58         } catch (IOException ex) {
59             log.log(Level.SEVERE, null, ex);
```

```
60              System.exit(1);
61          }
62          // Convierte a escala de grises de ser necesario
63          if (imagen.getType() != BufferedImage.TYPE_BYTE_GRAY) {
64              imagen = ImgOp.toGrayScale(imagen);
65          }
66          padding = lector.getPadding();
67
68          if (hosts.isEmpty()) {
69              log.severe("No hay hosts disponibles");
70              System.exit(1);
71          }
72          maxThreads = lector.getMaxThreads();
73
74          log.info("Hosts a usar: " + hosts.size());
75          log.info("Threads por host: " + maxThreads);
76          timeIni = System.currentTimeMillis();
77          log.info("Mensajes a enviar: " + creaMensajes());
78      }
79
80      /**
81       * Crea los mensajes a enviar. Cada mensaje creado, sabe a que nodo estará
82       * dirigido.
83       * @return Número de mensajes creados.
84       */
85      private int creaMensajes() {
86          gridMensajes = new MensajeCSE[hosts.size()][maxThreads];
87          ArrayList<String> listaHosts = new ArrayList<String>(hosts.keySet());
88          int numThreads = 0;
89          int contador = 0;
90          int w = (int) Math.ceil((double) imagen.getWidth() / (double) hosts.↩
                size());
91          int h = (int) Math.ceil((double) imagen.getHeight() / (double) ↩
                maxThreads);
92          int[] posActual = {0, 0};
93          int[] posGrid = {0, 0};
94
95          while (true) {
96              MensajeCSE mensaje = new MensajeCSE();
97              if (numThreads++ < maxThreads) {
98                  String host = listaHosts.get(0);
99                  mensaje = new MensajeCSE(host, hosts.get(host));
100             } else {
101                 numThreads = 1;
102                 listaHosts.remove(0);
103                 if (listaHosts.isEmpty()) {
104                     break;
105                 }
106                 String host = listaHosts.get(0);
107                 mensaje = new MensajeCSE(host, hosts.get(host));
108             }
109             BufferedImage img = null;
110             if (posActual[0] >= imagen.getWidth()) {
111                 posActual[0] = 0;
112                 posActual[1] += h;
113                 posGrid[0] = 0;
114                 posGrid[1]++;
115             }
116
117             int ancho = w;
118             int alto = h;
119             int posIniX = posActual[0];
120             int posIniY = posActual[1];
121             // Verificando ancho y alto
122             if (posActual[0] + w >= imagen.getWidth()) {
123                 ancho = imagen.getWidth() - posActual[0];
124             }
```

```
125             if (posActual[1] + h >= imagen.getHeight()) {
126                 alto = imagen.getHeight() - posActual[1];
127             }
128             img = imagen.getSubimage(posIniX, posIniY, ancho, alto);
129             mensaje.setMensaje(ImgOp.toByteArray(img));
130             mensaje.setWidth(ancho);
131             mensaje.setHeight(alto);
132             mensaje.setPadding(padding);
133             mensaje.setId(contador);
134             mensaje.setPixIniX(posActual[0]);
135             mensaje.setPixIniY(posActual[1]);
136 
137             gridMensajes[posGrid[0]][posGrid[1]] = mensaje;
138 
139             posActual[0] += w;
140             posGrid[0]++;
141             contador++;
142         }
143         return contador;
144     }
145 
146     /**
147      * Inicia el procesamiento en paralelo. Se generan las conexiones con los
148      * diferentes nodos y se envían los mensajes a procesar.
149      */
150     public void init() {
151         imgFinal = new BufferedImage(imagen.getWidth(), imagen.getHeight(),
152                 BufferedImage.TYPE_BYTE_GRAY);
153         ArrayList<Thread> threads = new ArrayList<Thread>();
154         for (int x = 0; x < gridMensajes.length; x++) {
155             for (int y = 0; y < gridMensajes[0].length; y++) {
156                 MensajeCSE msj = gridMensajes[x][y];
157                 try {
158                     msj.setVecino(
159                             gridMensajes[x - 1][y].getHost(),
160                             gridMensajes[x - 1][y].getPuerto(),
161                             gridMensajes[x - 1][y].getId(),
162                             GridNeighbord.Posicion.IZQUIERDA);
163                 } catch (ArrayIndexOutOfBoundsException e) { }
164                 try {
165                     msj.setVecino(
166                             gridMensajes[x][y - 1].getHost(),
167                             gridMensajes[x][y - 1].getPuerto(),
168                             gridMensajes[x][y - 1].getId(),
169                             GridNeighbord.Posicion.ARRIBA);
170                 } catch (ArrayIndexOutOfBoundsException e) { }
171                 try {
172                     msj.setVecino(
173                             gridMensajes[x + 1][y].getHost(),
174                             gridMensajes[x + 1][y].getPuerto(),
175                             gridMensajes[x + 1][y].getId(),
176                             GridNeighbord.Posicion.DERECHA);
177                 } catch (ArrayIndexOutOfBoundsException e) { }
178                 try {
179                     msj.setVecino(
180                             gridMensajes[x][y + 1].getHost(),
181                             gridMensajes[x][y + 1].getPuerto(),
182                             gridMensajes[x][y + 1].getId(),
183                             GridNeighbord.Posicion.ABAJO);
184                 } catch (ArrayIndexOutOfBoundsException e) { }
185                 msj.setPeticion(new Peticion(msj.getId()));
186                 Conexion con = new Conexion(gridMensajes[x][y].getHost(),
187                         gridMensajes[x][y].getPuerto());
188                 con.conectar();
189                 GeneradorImgFinal gen = new GeneradorImgFinal(imgFinal);
190                 threads.add(con.enviaYRecibe(msj, gen));
191             }
```

```
192             }
193             for (Thread t : threads) {
194                 t.start();
195             }
196             for (Thread t : threads) {
197                 try {
198                     t.join();
199                 } catch (InterruptedException ex) {
200                     log.log(Level.SEVERE, null, ex);
201                 }
202             }
203             System.out.printf("Tiempo Final: %d\n", System.currentTimeMillis() - ↩
                    timeIni);
204         }
205 
206         /**
207          * Permite mostrar la imagen procesada.
208          */
209         public void abrir() {
210             new Render("CSE", imgFinal, true).muestra();
211         }
212 
213         /**
214          * Envía señal de apagado a los nodos especificados en la configuración.
215          */
216         public void killAll() {
217             for (String host : hosts.keySet()) {
218                 Conexion con = new Conexion(host, hosts.get(host));
219                 con.conectar();
220                 con.terminaServidor();
221             }
222         }
223 
224         /**
225          * Guarda la imagen resultante.
226          */
227         public void guardar() {
228             File archivoFinal = new File("finalCSE.jpg");
229             try {
230                 ImageIO.write(imgFinal, "jpg", archivoFinal);
231                 imagen = null;
232             } catch (IOException ex) {
233                 log.log(Level.SEVERE, null, ex);
234             } catch (Exception ex) {
235                 log.warning(ex.getMessage());
236             }
237         }
238 
239         /**
240          * Indica si en la configuración se pide terminar cada uno de los nodos ↩
                activos
241          * después de haber realizado el procesamiento.
242          * @return <code>true</code> si se deben terminar.
243          */
244         public boolean terminarElemsSecs() {
245             return terminarElemsSecs;
246         }
247 
248         /**
249          * Main de la clase cse.Manager.
250          */
251         public static void main(String[] a) {
252             Manager m = new Manager();
253             m.init();
254             if (m.terminarElemsSecs) {
255                 m.killAll();
256             }
```

```
257             m.abrir();
258         }
259     }
```

D.4.2. cse.SequentialElement.java

```
1   package cse;
2
3   import util.MensajeCSE;
4   import java.awt.image.BufferedImage;
5   import java.awt.image.WritableRaster;
6   import java.util.ArrayList;
7   import java.util.logging.Level;
8   import java.util.logging.Logger;
9   import util.Conexion;
10  import util.GridNeighbord;
11  import util.ImgOp;
12  import util.Mensaje;
13  import util.Peticion;
14  import util.Tarea;
15
16  /**
17   *
18   * @author Felipe Navarrete Córdova
19   */
20  public class SequentialElement {
21
22      private static final Logger log =
23              Logger.getLogger(SequentialElement.class.getName());
24
25      private int id;
26      private Tarea operacionImg;
27      private MensajeCSE msj;
28      private BufferedImage imagen;
29
30      /**
31       * Constructor para crear un nuevo nodo secuencial.
32       * @param msj El mensaje generado por el nodo maestro que contiene lo que ↩
             se
33       * debe operar.
34       * @param operacionImg Objeto que contiene el código de segmentación de im↩
             ágenes.
35       */
36      public SequentialElement(MensajeCSE msj, Tarea operacionImg) {
37          this.msj = msj;
38          this.operacionImg = operacionImg;
39          this.imagen = ImgOp.fromByteArray(msj.getMensaje());
40          this.id = msj.getId();
41      }
42
43      /**
44       * Procesa la imagen contenida en el mensaje enviado desde el nodo maestro.
45       */
46      public void procesaMensaje() {
47          int nW = imagen.getWidth();
48          int nH = imagen.getHeight();
49          ArrayList<Thread> peticiones = new ArrayList<Thread>();
50          GridNeighbord[] vecinos = msj.getVecinos();
51          Mensaje[] buffer = new Mensaje[4];
52
53          for (short x = 0; x < vecinos.length; x++) {
54              GridNeighbord vecino = vecinos[x];
55              if (vecino != null) {
```

```
56                  switch (vecino.getPosicion()) {
57                      case IZQUIERDA:
58                      case DERECHA:
59                          nW += msj.getPadding();
60                          break;
61                      case ARRIBA:
62                      case ABAJO:
63                          nH += msj.getPadding();
64                          break;
65                  }
66                  MensajeCSE m = new MensajeCSE();
67                  m.setPadding(msj.getPadding());
68                  m.setVecino(vecino);
69                  m.setPeticion(new Peticion(id, vecino.getSeqElemId(), Peticion.↩
                        PETICION_DATOS));
70                  switch (vecino.getPosicion()) {
71                      case IZQUIERDA:
72                          m.setPosicionOrigen(GridNeighbord.Posicion.DERECHA);
73                          break;
74                      case ARRIBA:
75                          m.setPosicionOrigen(GridNeighbord.Posicion.ABAJO);
76                          break;
77                      case DERECHA:
78                          m.setPosicionOrigen(GridNeighbord.Posicion.IZQUIERDA);
79                          break;
80                      case ABAJO:
81                          m.setPosicionOrigen(GridNeighbord.Posicion.ARRIBA);
82                          break;
83                  }
84                  m.setId(id);
85                  Conexion con = new Conexion(vecino.getHost(), vecino.getPuerto↩
                        ());
86                  con.conectar();
87                  Thread peticion = con.enviaYRecibe(m, buffer);
88                  peticiones.add(peticion);
89                  peticion.start();
90              }
91          }
92          for (Thread t : peticiones) {
93              try {
94                  t.join();
95              } catch (InterruptedException ex) {
96                  log.log(Level.SEVERE, null, ex);
97              }
98          }
99
100         int[] pixIni = {0, 0};
101         int[] pixPadL = {0, 0};
102         int[] pixPadT = {0, 0};
103         int[] pixPadR = {0, 0};
104         int[] pixPadB = {0, 0};
105
106         for (Mensaje m : buffer) {
107             MensajeCSE mensaje = (MensajeCSE) m;
108             if (mensaje == null) {
109                 continue;
110             }
111             if (mensaje.getPosicionOrigen().equals(GridNeighbord.Posicion.↩
                    IZQUIERDA)) {
112                 pixIni[0] = msj.getPadding();
113                 pixPadL[0] = 0;
114                 pixPadT[0] = msj.getPadding();
115                 pixPadB[0] = msj.getPadding();
116                 msj.setPaddingLeft(msj.getPadding());
117             }
118             if (mensaje.getPosicionOrigen().equals(GridNeighbord.Posicion.↩
                    ARRIBA)) {
```

```
119                 pixIni[1] = msj.getPadding();
120                 pixPadL[1] = msj.getPadding();
121                 pixPadR[1] = msj.getPadding();
122                 pixPadT[1] = 0;
123                 msj.setPaddingTop(msj.getPadding());
124             }
125             if (mensaje.getPosicionOrigen().equals(GridNeighbord.Posicion.↩
                    DERECHA)) {
126                 pixPadR[0] = imagen.getWidth()
127                         + (pixIni[0] > 0 ? msj.getPadding() : 0);
128                 msj.setPaddingRight(msj.getPadding());
129             }
130             if (mensaje.getPosicionOrigen().equals(GridNeighbord.Posicion.ABAJO↩
                    )) {
131                 pixPadB[1] = imagen.getHeight()
132                         + (pixIni[1] > 0 ? msj.getPadding() : 0);
133                 msj.setPaddingBottom(msj.getPadding());
134             }
135         }
136         BufferedImage img = agregaPixels(nW, nH, buffer, pixIni,
137                 pixPadL, pixPadT, pixPadR, pixPadB);
138         msj.setMensaje(ImgOp.toByteArray(operacionImg.procesa(img)));
139     }
140
141     /**
142      * Obtiene el mensaje procesado.
143      * @return El mensaje procesado.
144      */
145     public Mensaje getMensaje() {
146         return msj;
147     }
148
149     /**
150      * Envía cierta cantidad de pixeles de la imagen contenida en este elemento
151      * secuencial de acuerdo a las especificaciones contenidas en el mensaje.
152      * @param mensaje El mensaje que contiene las especificaciones de los ↩
                pixeles
153      * que se debe mandar. Este mensaje es modificado y es el que se envía de
154      * vuelta con el nuevo contenido.
155      */
156     public void enviaDatos(MensajeCSE mensaje) {
157         GridNeighbord.Posicion pos = null;
158         switch (mensaje.getPosicionOrigen()) {
159             case IZQUIERDA:
160                 pos = GridNeighbord.Posicion.DERECHA;
161                 break;
162             case ARRIBA:
163                 pos = GridNeighbord.Posicion.ABAJO;
164                 break;
165             case DERECHA:
166                 pos = GridNeighbord.Posicion.IZQUIERDA;
167                 break;
168             case ABAJO:
169                 pos = GridNeighbord.Posicion.ARRIBA;
170                 break;
171         }
172         int padding = mensaje.getPadding();
173         // Si no hay padding definido
174         if (padding <= 0) {
175             mensaje.setPosicionOrigen(pos);
176             mensaje.setId(id);
177             return;
178         }
179         BufferedImage imgPadding = null;
180         switch (mensaje.getPosicionOrigen()) {
181             case ABAJO:
```

```
182                 imgPadding = imagen.getSubimage(0, imagen.getHeight() - padding↩
                        ,
183                         imagen.getWidth(), padding);
184                 break;
185             case ARRIBA:
186                 imgPadding = imagen.getSubimage(0, 0, imagen.getWidth(), ↩
                        padding);
187                 break;
188             case DERECHA:
189                 imgPadding = imagen.getSubimage(imagen.getWidth() - padding, 0,
190                         padding, imagen.getHeight());
191                 break;
192             case IZQUIERDA:
193                 imgPadding = imagen.getSubimage(0, 0, padding, imagen.getHeight↩
                        ());
194                 break;
195         }
196         // Actualizo el mensaje para su regreso.
197         mensaje.setPosicionOrigen(pos);
198         mensaje.setWidth(imgPadding.getWidth());
199         mensaje.setHeight(imgPadding.getHeight());
200         mensaje.setMensaje(ImgOp.toByteArray(imgPadding));
201         mensaje.setId(id);
202     }
203
204     /**
205      * Agrega pixeles a la imagen de este nodo secuencial.
206      * @param wN Ancho de la nueva imagen.
207      * @param hN Altura de la nueva imagen.
208      * @param msgRecibidos Los mensajes de los vecinos. Cada mensaje contiene
209      * la subimagen extra de cada vecino.
210      * @param coordImgIni Coordenadas (x, y) de inicio de la imagen del nodo
211      * secuencial dentro de la nueva imagen.
212      * @param coordIniL Coordenada inicial de la imagen informativa del vecino
213      * izquierdo.
214      * @param coordIniT Coordenada inicial de la imagen informativa del vecino
215      * superior.
216      * @param coordIniR Coordenada inicial de la imagen informativa del vecino
217      * derecho.
218      * @param coordIniB Coordenada inicial de la imagen informativa del vecino
219      * inferior.
220      * @return La imagen que contiene la información obtenida de los vecinos.
221      */
222     private BufferedImage agregaPixels(int wN, int hN,
223             Mensaje[] msgRecibidos, int[] coordImgIni,
224             int[] coordIniL, int[] coordIniT, int[] coordIniR, int[] coordIniB)↩
                {
225         BufferedImage nueva = new BufferedImage(wN, hN,
226                 BufferedImage.TYPE_BYTE_GRAY);
227         WritableRaster imgRaster = imagen.getRaster();
228         WritableRaster wrN = nueva.getRaster();
229
230         wrN.setSamples(coordImgIni[0], coordImgIni[1], imagen.getWidth(),
231                 imagen.getHeight(), 0,
232                 imgRaster.getSamples(0, 0, imagen.getWidth(),
233                 imagen.getHeight(), 0,
234                 new int[imagen.getWidth() * imagen.getHeight()]));
235         for (short a = 0; a < msgRecibidos.length; a++) {
236             if (msgRecibidos[a] != null) {
237                 int[] coordIniTMP = null;
238                 switch (a) {
239                     case 0:
240                         coordIniTMP = coordIniL;
241                         break;
242                     case 1:
243                         coordIniTMP = coordIniT;
244                         break;
```

```
245                         case 2:
246                             coordIniTMP = coordIniR;
247                             break;
248                         case 3:
249                             coordIniTMP = coordIniB;
250                             break;
251                     }
252                     agregaPixels(nueva,
253                             ImgOp.fromByteArray(msgRecibidos[a].getMensaje()),
254                             coordIniTMP);
255                 }
256             }
257 
258             return nueva;
259         }
260 
261         /**
262          * Actualiza la imagen <code>dos</code> con la imagen <code>uno</code>
263          * iniciando en las coordenadas dadas por <code>coordDosIni</code>.
264          * @param uno La imagen que será actualizada.
265          * @param dos La imagen con información de actualización.
266          * @param coordDosIni Coordenada de los pixeles donde se inicia la
267          * actualización de la imagen <code>uno</code>.
268          */
269         private void agregaPixels(BufferedImage uno, BufferedImage dos,
270                 int[] coordDosIni) {
271             if (uno == null  dos == null) {
272                 return;
273             }
274             WritableRaster wr1 = uno.getRaster();
275             WritableRaster wr2 = dos.getRaster();
276             wr1.setSamples(coordDosIni[0], coordDosIni[1], dos.getWidth(),
277                     dos.getHeight(), 0, wr2.getSamples(0, 0, dos.getWidth(),
278                     dos.getHeight(), 0, new int[dos.getWidth() * dos.getHeight()]))↩
                        ;
279         }
280 }
```

D.4.3. cse.SequentialElementManager.java

```
1  package cse;
2 
3  import util.MensajeCSE;
4  import Synchronization.EstablishRendezvous;
5  import Synchronization.MessagePassingException;
6  import Synchronization.Rendezvous;
7  import configuracion.LectorConfig;
8  import java.io.IOException;
9  import java.net.Inet4Address;
10 import java.net.InetAddress;
11 import java.net.UnknownHostException;
12 import java.util.HashMap;
13 import java.util.concurrent.ConcurrentHashMap;
14 import java.util.logging.Level;
15 import java.util.logging.Logger;
16 import javax.xml.parsers.ParserConfigurationException;
17 import org.xml.sax.SAXException;
18 import util.Peticion;
19 import util.Sobel;
20 import util.Tarea;
21 
22 /**
23  *
```

```
24  * @author Felipe Navarrete Córdova
25  */
26 public class SequentialElementManager {
27
28     private static final Logger log =
29             Logger.getLogger(SequentialElementManager.class.getName());
30     private int puerto = 9090;
31     private Tarea operacionImagen;
32     private final ConcurrentHashMap<Integer, SequentialElement> secElems;
33     private final EstablishRendezvous er;
34
35     /**
36      * Constructor
37      * @param operacionImagen Contiene la lógica de la segmentación de imágenes.
38      */
39     public SequentialElementManager(Tarea operacionImagen) {
40         log.info("======= Iniciando Elemento Secuencial =======");
41         log.info("Leyendo óconfiguracin");
42         LectorConfig lector = null;
43         try {
44             lector = new LectorConfig();
45         } catch (ParserConfigurationException ex) {
46             log.log(Level.SEVERE, null, ex);
47             System.exit(1);
48         } catch (SAXException ex) {
49             log.log(Level.SEVERE, null, ex);
50             System.exit(1);
51         } catch (IOException ex) {
52             log.log(Level.SEVERE, null, ex);
53             System.exit(1);
54         }
55         HashMap<String, Integer> hosts = lector.getWorkersHosts();
56         InetAddress inet = null;
57         try {
58             inet = Inet4Address.getLocalHost();
59         } catch (UnknownHostException ex) {
60             log.log(Level.WARNING, null, ex);
61         }
62         for (String host : hosts.keySet()) {
63             if (host.equals(inet.getHostAddress())
64                     host.equals(inet.getHostName())) {
65                 puerto = hosts.get(host);
66                 break;
67             }
68         }
69         this.operacionImagen = operacionImagen;
70         log.info("Se utilizará el puerto " + puerto);
71         secElems = new ConcurrentHashMap<Integer, SequentialElement>();
72         er = new EstablishRendezvous(puerto);
73     }
74
75     /**
76      * Abre la conexión en red a través del puerto configurado en el archivo de
77      * configuración para recibir peticiones.
78      */
79     public void listen() {
80         while (true) {
81             try {
82                 final Rendezvous rendezvouz = er.serverToClient();
83                 new SEMEscucha(rendezvouz).start();
84             } catch (MessagePassingException ex) {
85                 log.info("óConexin interrumpida");
86                 break;
87             }
88         }
89     }
90
```

```
91      /**
92       * Objeto tipo Thread que realiza el procesamiento de las peticiones ↩
             distribuídas.
93       */
94      class SEMEscucha extends Thread {
95
96          private Rendezvous rendez;
97
98          public SEMEscucha(Rendezvous rendezvous) {
99              this.rendez = rendezvous;
100         }
101
102         @Override
103         public void run() {
104             MensajeCSE msg = (MensajeCSE) rendez.serverGetRequest();
105             if (msg == null) {
106                 log.info("Deteniendo procesos");
107                 rendez.serverMakeReply(null);
108                 rendez.close();
109                 System.exit(0);
110             }
111             if (msg.getPeticion().getTipoPeticion() == Peticion.PETICION_DATOS)↩
                    {
112                 SequentialElement se = secElems.get(msg.getPeticion().↩
                        getSeqElemIdDest());
113                 // Busca que el elemento secuencial se encuentre definido en el↩
                        Hash Map.
114                 while (se == null) {
115                     se = secElems.get(msg.getPeticion().getSeqElemIdDest());
116                 }
117                 se.enviaDatos(msg);
118                 rendez.serverMakeReply(msg);
119             } else if (msg.getPeticion().getTipoPeticion() == Peticion.↩
                    PETICION_PROCESA) {
120                 SequentialElement seq = null;
121                 seq = new SequentialElement(msg, operacionImagen);
122                 secElems.put(msg.getId(), seq);
123                 seq.procesaMensaje();
124                 rendez.serverMakeReply(seq.getMensaje());
125             } else if (msg.getPeticion().getTipoPeticion() == Peticion.↩
                    PETICION_FIN) {
126                 secElems.clear();
127                 rendez.serverMakeReply(null);
128                 rendez.close();
129             } else {
130                 rendez.serverMakeReply(null);
131                 rendez.close();
132                 System.exit(0);
133             }
134         }
135     }
136
137     /**
138      * Método main de la clase cse.SequentialElementManager
139      */
140     public static void main(String[] a) {
141         SequentialElementManager seq = new SequentialElementManager(new Sobel()↩
                );
142         seq.listen();
143     }
144 }
```