ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

INFORME DE MATERIA DE GRADUACIÓN

"Utilización de la plataforma Hadoop para implementar un programa distribuido que permita encontrar las paredes de células de la epidermis de plantas modificadas genéticamente."

Previa a la obtención del Título de:

INGENIERO EN CIENCIAS COMPUTACIONALES ESPECIALIZACIÓN SISTEMAS MULTIMEDIA

Presentada por:

GUSTAVO IRVING CALI MENA

ERICK ROGGER MORENO QUINDE

Guayaquil - Ecuador
2010

AGRADECIMIENTO

A Dios.

A nuestras familias y amigos que nos han apoyando siempre en el camino de nuestra vida, con consejos y su tiempo.

Al Ing. Daniel Ochoa por su tiempo y apoyo para guiarnos.

A la Ing. Cristina Abad que ha sido nuestra mentora y guía, que sin su apoyo y persistencia no hubiésemos podido lograr el presente trabajo.

DEDICATORIA

A nuestros padres, quienes con su esfuerzo y perseverancia forjaron un camino seguro para nosotros que nos conduce a un futuro prometedor lleno de oportunidades, obstáculos que vencer y esperanza, el presente trabajo es para ellos y para todas las personas que contribuyeron a nuestra formación profesional.

TRIBUNAL DE SUSTENTACIÓN

PROFESOR DE LA MATERIA DE GRADUACIÓN

MsC. Carmen Vaca

PROFESOR DELEGADO POR EL DECANO

Ing. Daniel Ochoa

DECLARACIÓN EXPRESA

"La responsabilidad del contenido de este trabajo de graduación, nos corresponden exclusivamente; y el patrimonio intelectual de la misma a la **ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL**"

(Reglamento de Graduación de la ESPOL)

Erick R. Moreno Quinde

Gustavo I. Cali Mena

RESUMEN

En este trabajo se presenta la implementación de un programa distribuido que permite encontrar las paredes de células epiteliales de plantas del tipo Arabidopsis Thaiana captadas en imágenes con interferencia diferencial de contraste mediante la implementación de un algoritmo detector de estructuras curvilíneas en un programa distribuido a partir de sus clases en C++.

El documento está dividido en 7 Capítulos que comprenden el análisis del problema, conceptual, las plataformas, solución, diseños, implementación y las pruebas y análisis de resultados utilizando tecnologías escalables de procesamiento masivo y escalable de datos en este caso imágenes para la interpretación y post procesamiento.

En el Capítulo 1 se describe la necesidad de una herramienta de gran escalabilidad. Se proponen los objetivos, el alcance, limitaciones y la justificación del presente proyecto.

En el Capítulo 2 se realiza una breve descripción del marco teórico utilizado, como los algoritmos usados y posteriormente se centra en el formato de las imágenes que se usan para el procesamiento.

En el Capítulo 3 se describe la plataforma de desarrollo, herramientas y los servicios. Más adelante se describen en detalle cada uno de estos servicios y su utilidad dentro del proyecto, enfocándose en el paradigma de MapReduce.

Se llevó a cabo un análisis del problema porque es esencial conocer el problema, el conjunto de soluciones que se dieron y cuál fue la más óptima. Este análisis se detalla en el Capítulo 4.

En el Capítulo 5 se describe la arquitectura que tiene el proyecto. Aquí se detalla cómo todos los componentes se unen y se lleva el proceso entre ellos. Además se especifica cómo se van a mostrar los resultados.

En el Capítulo 6 se realiza una descripción de los pasos seguidos para la implementación de los archivos de entrada, procesamiento de las imágenes y sus resultados, todos acoplados al entorno de Hadoop.

Finalmente, en el Capítulo 7, se detalla las pruebas de eficacia y eficiencia que se realizaron en los clústeres de Amazon, el análisis y conclusión de ellas.

ÍNDICE GENERAL

ÍNDICE DE GRÁFICOS

ABREVIATURAS

GB	Gigabyte.
TB	Terabyte.
MB	Megabyte.
FIEC	Facultad de Ingeniería en Electricidad y Computación.
AWS	Amazon Web Services.
EC2	Amazon Elastic Compute Cloud.
S3	Amazon Simple Storage Service.
HDFS	Hadoop Distributed File System.
ESPOL	Escuela Superior Politécnica Del Litoral.
OpenCV	Open Source Computer Vision.
PGM	Portable Gray Map.
AMI	Amazon Machine Image.
DIC	Differential Interference Contrast.

INTRODUCCIÓN

En la actualidad, la ciencia avanza más rápido que la tecnología, por tal motivo a veces la ciencia se estanca y muchas de las cosas que se desean demostrar se quedan pendientes hasta que se tenga el equipo adecuado.

La aplicación que se describe en este documento va a procesar una gran cantidad de imágenes de las células de plantas alteradas genéticamente, las cuales representan un gran volumen de datos.

Actualmente en el mercado existen varias herramientas que ayudan al procesamiento de estas imágenes, pero la mayoría de ellas tienen un tiempo de ejecución muy elevado. Además, a pesar de que dichas herramientas se encuentran disponibles tanto de manera gratuita como propietaria, no abastecen la demanda para el análisis de un volumen grande de información en el orden de los GB o TB. Esto introduce retraso de días o meses en los trabajos a ejecutar.

Para esto se desarrolló una algoritmo en C++, en conjunto con dos librerías útiles para el manejo de las imágenes llamadas OpenCV [1] y OpenEXR [2] usando el paradigma MapReduce [3] para integrarlo con Hadoop [4] como

plataforma distribuida para el procesamiento masivo de datos. Esta aplicación contribuye a la investigación en citología vegetal, en especial ahorrando tiempo de ejecución del programa, lo cual beneficia a los investigadores obteniendo resultados más rápidos de una cantidad considerable de información evitando largas esperas para poder continuar con la siguiente etapa de análisis.

La flexibilidad y escalabilidad que nos ofrece el paradigma MapReduce en Hadoop y el lenguaje C++ en el que está desarrollada la aplicación deja la puerta abierta para la implementación de nuevos mecanismos de procesamiento distribuido de imágenes.

CAPÍTULO 1.

1 ANÁLISIS DEL PROBLEMA

1.1 Definición del problema

En Bioinformática es común el uso de equipos de gran precisión que generan imágenes de alta resolución que son procesadas por paquetes informáticos.

Para el presente trabajo se procesará una gran cantidad de imágenes de las células de la planta denominada Arabidopsis Thaliana, las cuales fueron tomadas con un microscopio DIC [5] por sus siglas en inglés “Differential Interference Contrast”. Las imágenes generadas no son reales ya que con este microscopio se obtiene el valor de intensidad de la luz reflejada sobre la superficie de las paredes celulares con la resta de las imágenes desfasadas una pequeña distancia, lo que vuelve visible estas paredes definidas con un borde de alta intensidad y otro de baja intensidad de luz. Esta técnica es usada debido a que no se pueden teñir las muestras ya que estas mueren y se dañan.

Dado que, el microscopio DIC usa una luz direccional para capturar las imágenes, se generan un conjunto de 36 imágenes por cada muestra tomada. Este conjunto se produce como resultado de las 12 rotaciones que se utilizan hasta completar 360 grados. Por cada rotación se cambia el objetivo en 3 diferentes aumentos, con este procedimiento obtenemos en cada imagen segmentos de información de la pared celular que al combinarlas se complementan, teniendo de esta forma la pared celular completa.

Las imágenes capturadas tienen una resolución de 2560 x 1920 pixeles, que representan un gran volumen de información, tanto en tamaño en bytes, como en la cantidad de fases requeridas para su procesamiento.

El algoritmo inicialmente estaba hecho en MATLAB [6]. MATLAB permite un desarrollo rápido si se conoce el modo de programación y se aprovecha su capacidad de aplicar diferentes operaciones sobre matrices, pero así mismo, su interfaz gráfica y funciones desarrolladas en Java lo hace lento en respuesta, el manejo de los índices de las matrices no es el mejor computacionalmente hablando. Por ejemplo el uso de MATLAB en un PC podría requerir alrededor de 17 minutos para el procesamiento de 12 imágenes.

El problema a resolver consiste básicamente en la reducción del tiempo de procesamiento de los algoritmos al procesar un volumen grande de imágenes.

Como parte de las actividades de este Capítulo, se realizó un estudio de herramientas de programación que implementen los pasos del algoritmo para el procesamiento de las imágenes.

Tomando como base en las necesidades de mejora se optó por implementar el algoritmo en C++ con las librerías de procesamiento de gráficos OpenCV [1] y OpenEXR [2], las cuales aparte de permitir mejorar los tiempos permiten utilizar una plataforma que realice procesamiento masivo y escalable de datos como lo es Hadoop.

1.2 Objetivos generales

La presente tesis tiene como objetivo reducir el tiempo de ejecución de un algoritmo de detección de paredes celulares, que genera como resultado una imagen binaria que pueda ser usada para detectar las células.

1.3 Objetivos específicos

Para llegar al objetivo general se definieron los siguientes objetivos específicos:

- Re codificar el algoritmo de Matlab a una aplicación C++.

- Adaptar el código C++ que obedezca el paradigma MapReduce para poder ser ejecutado en Hadoop.
- Adaptar las clases de entrada y salida de Hadoop para el manejo de imágenes.
- Comparar las gráficas que representen a los tiempos de ejecución del algoritmo en los clústeres de Amazon con un número variable de nodos.

1.4 Justificación

Las investigaciones en el campo de genética cada vez empiezan a avanzar más y más requiriendo procesar una inmensa cantidad de información.

Esto hace que nos veamos en la necesidad de tener una herramienta, fácil de extender y de fácil aprendizaje, que permita el procesamiento de una gran cantidad de imágenes en tiempos cortos, y que además genere como salida nuevas imágenes que se puedan procesar y ayuden a obtener más resultados.

1.5 Alcances y limitaciones

Este proyecto básicamente procesa imágenes de células en escala de grises como entrada y generan el resultado en imágenes con valores binarios, de la misma resolución que las imágenes de entrada.

Para este proyecto se usan imágenes de entrada en formato PGM [7] con una resolución de 2560 x 1920 pixeles, e imágenes de salida en formato EXR de la misma resolución.

CAPÍTULO 2.

2 ANÁLISIS CONCEPTUAL

2.1 El Algoritmo

El Algoritmo de procesamiento para la detección de las paredes celulares que vamos a implementar se divide en 5 pasos, los cuales están comprendidos en:

1.- Filtros de coincidencias.

2.- Detección de las estructuras curvilíneas.

3.- Image Registration.

4.- Combinación de imágenes.

5.- Binarización de la imagen.

En la sección siguiente se describirán más a detalle cada uno de los pasos aquí mencionados.

2.2 Descripción del algoritmo

2.2.1 Filtros de coincidencia.

En la primera fase se aplican los filtros de coincidencia (Matching Filter), son máscaras de tamaño variable que modelan la distribución de valores de intensidad de acuerdo a la orientación de la pared celular. Los filtros son de diferentes tamaños y rotaciones que aplicados a las imágenes DIC de las células unen las dos líneas de intensidades opuestas que definen la pared celular en una sola de intensidad máxima.

2.2.2 Detección de estructuras curvilíneas.

En la segunda fase se extraen de las imágenes las líneas que la conforman, precisamente estas líneas representan las paredes celulares que son de nuestro interés.

Para la detección de las estructuras curvilíneas se usa un algoritmo propuesto por Steger [8]. La mayoría de los algoritmos utilizan un simple modelo para extraer las líneas, que no toma en cuenta los alrededores, Steger toma en cuenta la información de los alrededores de línea dándonos información de su grosor.

Se modificó el algoritmo de Steger y sé los dividió en 2 operaciones:

- Calcular los valores de las derivadas.
- Calcular la distancia de puntos sobre las líneas.

Con los valores de las derivadas en direcciones diferentes, se calcula el valor de intensidad de pixel en cada posición, esta fase da como resultado una imagen en la que las paredes de las células quedan definidas por valores de intensidad cercanos a uno y el fondo con valores de intensidad cercanos a cero creando una diferencia marcada de contrastes dejando visibles las líneas que corresponden a las paredes de las células.

2.2.3 Image Registration.

En la sección 1.1 se explicó que se toman 36 imágenes por muestra, y que en cada imagen se capturan detalles de las paredes celulares que son de nuestro interés, para obtener la información completa, se deben combinar estás imágenes en una.

La tercera fase del proceso se dividió en dos subprocesos:

- Rotación de las imágenes, el nombre de cada imagen contiene el ángulo a las que fueron tomadas, y que se deben rotar para que coincidan.
- Alineación de las imágenes, debido a que la captura de las imágenes se realizó de forma manual, existe cierto desfase entre ellas que se debe corregir.

Con la rotación y alineación de las imágenes estas quedan listas para que se puedan combinar en una, de forma que las paredes celulares

coincidan de forma correcta.

2.2.4 Combinación de Imágenes.

En la cuarta fase del procesamiento se deben combinar todas las imágenes alineadas y así obtener la información de las paredes celulares completa en una sola imagen. Para combinar todas las imágenes en una solo se compara para una misma posición los valores de intensidad en todas las imágenes y se selecciona el valor máximo.

2.2.5 Binarización de la imagen.

Para definir correctamente los bordes se necesita crear una diferencia de contraste notoria.

En la quinta fase, para que las paredes celulares queden bien definidas en la imagen se usa una función de umbral (Threshold). El valor seleccionado como umbral va a depender la calidad del resultado. Este parámetro fue conseguido experimentalmente.

En esta fase se obtiene la imagen resultante en la que las paredes celulares estarán definidas por un valor de uno mientras que el fondo y la información no relevante quedarán definidos con un valor de cero.

CAPÍTULO 3.

3 PLATAFORMA DE COMPUTACIÓN EN LA NUBE

El procesamiento de una gran cantidad de imágenes representa costos, computacionalmente hablando, considerando el tamaño de las imágenes y la cantidad de filtros que se necesita aplicar para obtener el resultado deseado.

La computación en la nube (cloud computing) es un modelo de prestación de servicios orientada hacia la escalabilidad. En la actualidad existen muchos proveedores de servicios de computación en la nube. Para el procesamiento de imágenes se ha seleccionado al proveedor Amazon, a través de sus servicios Web (Amazon Web Services) [9], ya que se obtuvo acceso gratuito a sus servicios a través de su programa de fondos Servicios Web de Amazon en la educación (AWS in Education).

Amazon presta varios servicios de computación en la nube, entre los cuales encontramos Amazon Elastic Compute Cloud (Amazon EC2) y el Amazon Simple Storage Service (Amazon S3).

3.1 Amazon Elastic Compute Cloud (Amazon EC2)

Es un servicio Web que proporciona la capacidad de cálculo de tamaño variable en la nube. Está diseñado para hacer que el procesamiento escalable en la Web sea más fácil para los desarrolladores.

Amazon EC2 tiene una sencilla interfaz de servicio Web que nos permite obtener la capacidad de configurar con mínimo esfuerzo. Nos proporciona un control completo de sus recursos de computación y además permite ejecutar el entorno de computación demostrado de Amazon. [10]

En Amazon EC2 es donde se sube el Amazon Machine Image (AMI) el cual contiene un conjunto de librerías y los algoritmos del proyecto.

3.2 Amazon Simple Storage Service (Amazon S3)

Proporciona una sencilla interfaz de servicios Web que pueden ser utilizados para almacenar y recuperar cualquier cantidad de datos, en cualquier momento, desde cualquier lugar de la Web. [11]

AWS ha integrado S3 con EC2, de tal manera que transferir datos entre ambos servicios sea eficiente y no represente un costo alto para el cliente.

3.3 Procesamiento masivo y escalable de datos

El procesamiento digital de imágenes resulta costoso computacionalmente, si a esto le sumamos que se van a procesar un volumen de imágenes del orden de los GB, para un solo computador esto resultaría poco práctico por restricciones de memoria y tiempo. En la actualidad, existen algunas plataformas para el procesamiento masivo y escalable de datos, pero de todas ellas, Hadoop es la que más acogida a tenido, debido a que está basada en principios desarrollados por Google.

3.4 Hadoop y Hadoop Pipes.

Como plataforma distribuida se escogió a Hadoop y su módulo Hadoop Pipes [12] como una solución a la ejecución de aplicaciones desarrolladas en el lenguaje C++, además que es soportada por Amazon EC2.

Hadoop es un framework open-source desarrollado en Java que nació como proyecto de Apache para el procesamiento y consulta de grandes cantidades de datos sobre un grupo de computadores [4], se distribuye bajo licencia “Apache License 2.0”. La programación en Hadoop implementa el paradigma MapReduce el cual sobrescribe clases en Java de las que se hablará en detalle más adelante.

Hadoop Pipes, es una alternativa que se ofrece para ejecutar programas desarrollados en C++ con el paradigma MapReduce, a diferencia de Streaming [13], que utiliza la entrada y salida estándar para comunicarse con las clases de Java de Hadoop, Pipes usa sockets como el canal por el que se comunica con el proceso que hace seguimiento de las tareas en ejecución denominado tasktracker de Hadoop con el proceso de ejecución de C++.

3.5 Paradigma MapReduce

MapReduce es un modelo de programación que facilita la implementación de programas orientados al procesamiento y generación de grandes volúmenes de datos. En la **Figura 3.5.1** observamos un esquema simplificado del flujo de ejecución del paradigma.

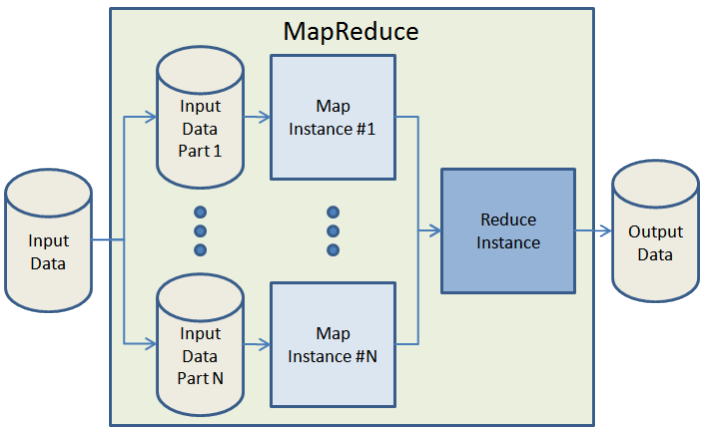

Figura 3.5.1 Paradigma MapReduce

Los usuarios especifican o sobrescriben una clase llamada Mapper, que se ejecuta en una etapa inicial denominada fase de mapeo. El Mapper procesa la información de entrada recibida como un par clave/valor para generar un conjunto intermediario de pares clave/valor, y una clase Reducer que combina todos los valores intermedios asociados con la misma clave intermediaria en una fase denominada de reducción o combinatoria. Los datos no son compartidos entre mappers ni entre reducers, lo que permite que cada instancia de esta función pueda ser procesada de manera distribuida en diferentes computadores, Hadoop es el encargado de distribuir los datos a los diferentes Mapper y Reducer de manera escalable y tolerante a fallos. De esta manera, se esconden los complejos detalles de la implementación distribuida al desarrollador, quien puede concentrarse únicamente en la lógica del procesamiento vía el paradigma descrito. [3]

3.6 Diferentes Alternativas

Existen diferentes alternativas a Hadoop que no permiten implementar aplicaciones distribuidas entre ellas podemos mencionar a Windows Azure.

Windows Azure [14] es una plataforma propietaria de Microsoft que permite la ejecución de aplicaciones distribuidas implementadas y

probadas median un SDK desarrollado para Visual Studio.

Sin embargo, esta solución presenta varios desventajas frente a Hadoop, de las cuales podemos mecionar que es propietaria, brinda mucho menos soporte, los nodos estan instalados únicamente con Windows y el desarrollo se limita al SDK propocionado, compatible con la linea de desarrollo de Visual Studio.

CAPÍTULO 4.

4 ANÁLISIS DE LA SOLUCIÓN.

4.1 ¿Qué estamos resolviendo?

El estudio realizado sobre las paredes de las células genéticamente alteradas dejó como resultado imágenes en escala de grises aproximadamente 4MB de información. Por cada muestra se tiene un total de 36 imágenes de las cuales se procesaran solamente 12 correspondientes a un zoom en particular, en este caso el conjunto que visualmente tiene mejor definidos las paredes celulares.

Con la integración del algoritmo con Hadoop, las imágenes son procesadas en paralelo permitiendo detectar las paredes de las células en conjuntos de imágenes que están en el orden de los GB. En el presente trabajo se usaron 156 imágenes que dan un volumen total de aproximadamente 700 MB.

Con la finalización del programa se ahorra el tiempo que se llevaba en el procesamiento de las imágenes con el algoritmo implementado en MATLAB, y se obtiene como resultado una imagen que se puede volver a procesar.

4.2 Procesamiento de imágenes en una plataforma distribuida

Al analizar una aplicación que se implementa en una plataforma distribuida, una de las primeras tareas consiste en determinar las tareas que se asignaran al Mapper y Reducer.

Para resolver el problema hemos definido un algoritmo con una serie de fases que están descritos en el capítulo 2, y que hemos resumido en la **Figura 4.2.1**:

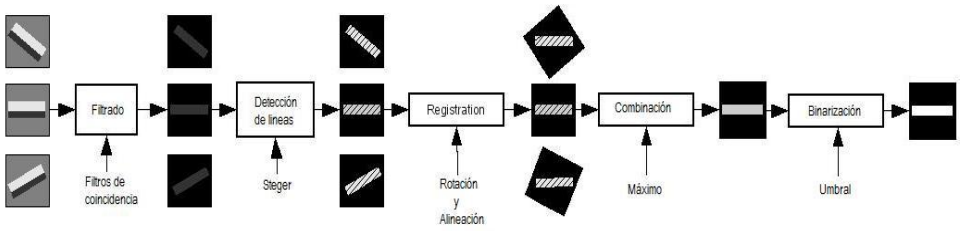

Figura 4.2.1 El algoritmo

Con el flujo del procesamiento mostrado en la **Figura 4.2.1** fue fácil identificar las fases que se pueden implementar en el Mapper las cuales se van a ejecutar en paralelo. Para el presente trabajo se seleccionó hacer el filtrado y la detección de líneas en la fase de mapeo debido a que las operaciones antes mencionadas son independientes de las imágenes a procesar e independientes entre ellas también.

Para la fase de Image Registration es necesario tener una imagen de

referencia en común por cada muestra con la cual se van a alinear el resto de imágenes. La decisión de poner la fase Image Registration en la fase de mapeo se tomó en base a las ventajas del paralelismo que esta fase nos ofrece.

En la fase de reducción se obtienen todas las imágenes que fueron procesadas por la fase de mapeo de forma ordenada y agrupadas por la clave intermediaria, razón por la cual en esta fase es más conveniente realizar al combinación de todas las imágenes, y finalmente, la binarización de la misma para obtener la salida de la aplicación.

Hemos elegido Hadoop Pipes como herramienta para el procesamiento masivo de datos, a través de clústeres levantados bajo demanda utilizando los Servicios Web de Amazon (AWS).

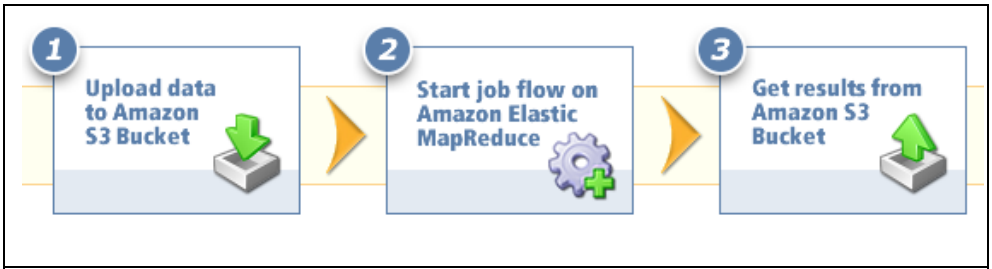

Figura 4.2.2 Flujo de procesamiento EC2

La **Figura 4.2.2** define como se desarrolla el flujo de procesamiento en los nodos de Amazon EC2 [10].

CAPÍTULO 5.

5 DISEÑO DEL MÓDULO.

5.1 Diseño General

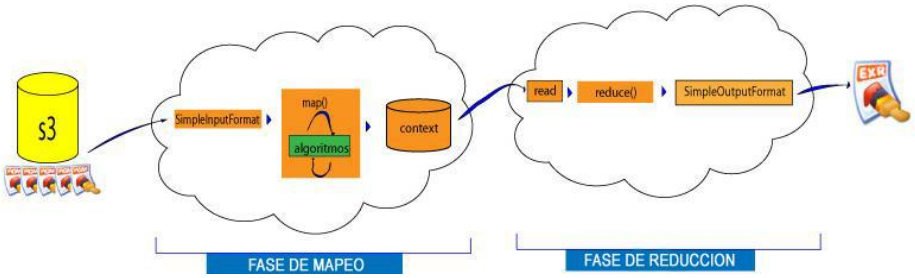

Figura 5.1.1 Diseño arquitectónico del módulo

El conjunto de imágenes se encuentra en un bucket[1] en Amazon S3, el cual es bajado para el procesamiento a los nodos de Amazon.

Empieza la fase de mapeo, todas las imágenes son leídas del bucket una vez que son procesadas por las clases de entrada se devuelve una cadena de bytes contenida dentro de una cadena de caracteres, cada carácter representa información de los pixeles de la imagen a los que se le aplica el algoritmo para detectar las paredes celulares, la información resultante es almacenada en un buffer para enviarlo a la

[1] Bucket es una carpeta de nivel superior en Amazon S3. Sirve al propósito de un contenedor de archivos.

fase de reducción o combinatoria.

En la fase de reducción se lee cada imagen correspondiente a una muestra, para combinarlas en una sola imagen, aplicando una función de umbral (threshold) guardando el resultado en un buffer para ser escrito en el Sistema de Archivos Distribuido de Hadoop (Hadoop Distributed File System HDFS) utilizando como nombre de archivo, el nombre de la muestra.

5.1.1 Archivos de entrada (PGM)

Son imágenes de un conjunto de rotaciones que va de 0 a 360 grados de la misma muestra de la célula, son en total 12 imágenes por cada muestra. Las imágenes están en formato PGM la cual es ampliamente utilizada por investigadores por su simplicidad. [15]

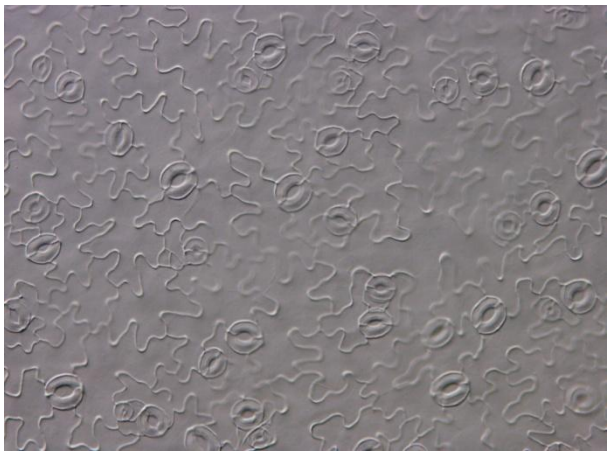

Figura 5.1.1.1 Ejemplo de imagen de entrada.

Las imágenes en formato PGM son fáciles de leer y de escribir, debido a su estructura, como se puede ver en la **Figura 5.1.1.2**.

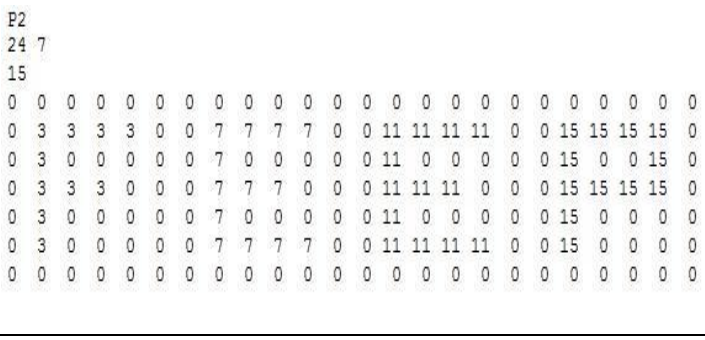

Figura 5.1.1.2 Ejemplo de Imagen PGM

Cada una de estas imágenes se almacena en Amazon EC2 para su posterior procesamiento. Hadoop no provee de métodos para leer imágenes. Sin embargo, para procesar cada una de ellas y leerlas en un formato binario, se utilizará una clase que extienda de FileInputFormat[1] y que pueda leer el archivo de entrada completo sin segmentarlo y almacenarlos como un string binario. A esta clase se la denominó SimpleFileInputFormat y permite leer imágenes almacenados en el Sistema de Archivos Distribuido de Hadoop (Hadoop Distributed File System HDFS).

5.1.2 El ambiente EC2

En el EC2 de Amazon se ejecutarán básicamente dos tareas, la

[1] El FileInputFormat es la clase base de Hadoop para leer los datos entrada en el mapper.

ejecución de los Mappers y la ejecución de los Reducers.

Cada Mapper recibe una imagen como archivo de entrada, de la cual se extrae el nombre, el ancho, el alto y los valores de la matriz de pixeles de la imagen, se procesa con el algoritmo ya explicado y se devuelve la imagen procesada.

Cada resultado de los Mappers es enviado al Reducer como se muestra en la **Figura 5.1.2.1**:

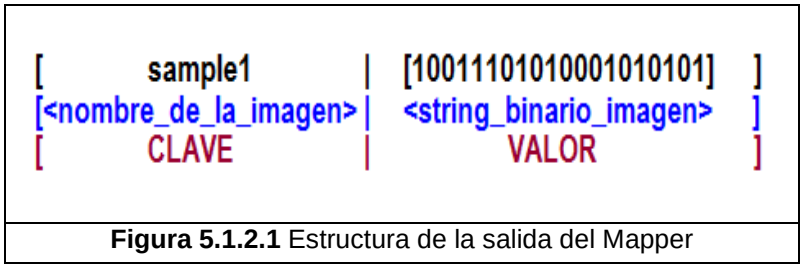

Figura 5.1.2.1 Estructura de la salida del Mapper

Durante el procesamiento Hadoop requiere una par clave/valor para el procesamiento la clave será el nombre de la muestra y el valor la información de la matriz de pixeles contenida en una cadena de caracteres de formato binario.

Esta estructura permite guardar la información en un HashMap para poder ser agrupados y ordenados antes de ser enviados al Reducer.

Luego, se crea un Reducer por cada clave que se haya generado en el Mapper, este procesará todas las imágenes guardadas bajo el

valor de esa clave, y se combinan todas las imágenes en una, y luego se la guarda en el Sistema de Archivos Distribuido de Hadoop con la estructura mostrada en la **Figura 5.1.2.2**.

```
[    sample1    |    [010011101001010100]    ]
[    sample2    |    [011001101001010100]    ]
[    sample3    |    [010011100001010101]    ]
```

Figura 5.1.2.2 Estructura de la salida del Reducer

5.1.3 Archivos de salida (EXR)

La salida del Reducer son imágenes en formato OpenEXR, las cuales tienen un alto rango dinámico de valores para almacenar la información de los píxeles, proporcionando mayores detalles como se puede observar en la **Figura 5.1.3.1**.

Figura 5.1.3.1 Imagen OpenEXR de ejemplo

Para poder realizar el correspondiente almacenamiento, se creará una

clase que extienda de FileOutputFormat[1] y que sobrescriba los métodos necesarios para que el resultado final se almacene como archivo EXR en el nodo Master.

[1] El FileOutputFormat es la clase base de Hadoop para la escritura de datos de salida en archivos (de texto o binario, dependiendo de la subclase).

CAPÍTULO 6.

6 IMPLEMENTACIÓN DE LA SOLUCIÓN

6.1 Algoritmo y codificación.

En el presente capítulo se explicará las implementaciones que fueron necesarias realizar para llevar a cabo el desarrollo de la aplicación distribuida que cumpla con los objetivos antes descritos.

Para la implementación del programa se compilaron los archivos fuentes SimpleFileInputFormat.java la cual lee del Hadoop Distributed File System las imágenes completas sin dividirlas como comúnmente se hace con las cadenas de caracteres que procesa nativamente Hadoop. SimpleRecordReader.java es una clase que, por cada imagen leída pone el nombre de la imagen como clave, y convierte la imagen en un arreglo de bytes el cual es puesto como valor para la clave correspondiente. SimpleFileOutputFormat.java inicializa una variable de tipo Path de Hadoop donde se configura la ruta en la que se va a guardar el archivo de salida y llama a SimpleRecordWriter.java, aquí es donde se crea un archivo con el nombre indicado y guarda la imagen. Con estas clases se

generó un archivo con extensión jar llamado hadoopimageformat, este archivo se debe colocar en la Cache distribuida[1] de Hadoop para su posterior invocación.

En la porción resultante del capítulo se muestra las imágenes resultantes de cada una de las etapas del procesamiento con su respectiva descripción.

Primero se hizo el código en C++ sin implementar MapReduce de Hadoop con el cual se identificaron los siguientes pasos para el procesamiento:

La primera parte del algoritmo consiste en convertir el tipo de dato de entrada de tener valores enteros a valores de tipo float de 32 bit.

Luego se leen los archivos de texto que contienen los valores de los filtros de coincidencia que utiliza el algoritmo, con estos valores se forma una matriz cuadrada que se aplica como máscara de filtrado a las imágenes.

La imagen resultante del filtrado se normaliza entre valores de 0 a 1 para verificar que los valores de intensidad de pixel de la imagen estén dentro de ese rango de valores.

Los resultados de esta primera fase se muestran en la **Figura 6.1.1**.

[1] Cache Distribuida es una porción del HDFS la cual puede ser accedida desde cualquier nodo como si fuera un directorio local.

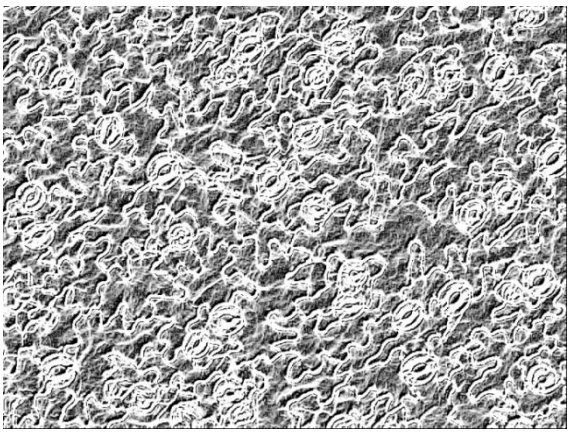

Figura 6.1.1 Resultado de los filtros de coincidencia y la normalización.

A continuación, en la **Figura 6.1.2** se muestra el resultado de aplicar el algoritmo de Steger para detectar las líneas curvas y resaltarlas en la imagen, utilizando la librería StegerLineDetector, de Filip Rooms [16].

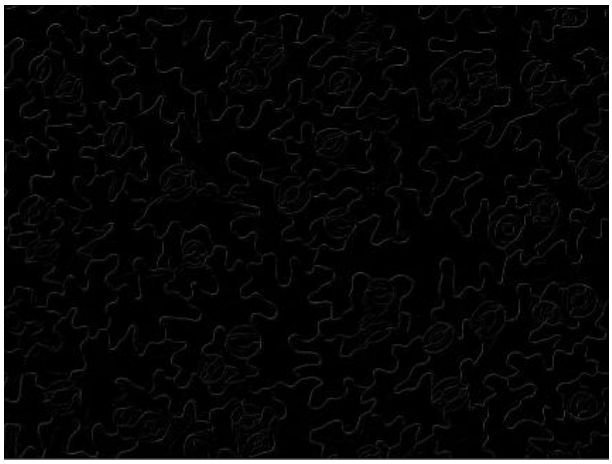

Figura 6.1.2 Resultado del algoritmo de Steger.

Como se mencionó el punto 1.1 Definición del Problema, cada imagen posee parte de la información de las paredes celulares, la cual al combinar se complementaría, para esto a cada imagen resultante del algoritmo de Steger correspondiente a la misma muestra se aplica una rotación en grados como se indica en el nombre del archivo para luego alinearla teniendo como referencia la imagen de cero grados, ya que existe un margen de error al momento de la captura de las imágenes.

Posterior a esto se combinan todas las imágenes en una obteniendo el resultado mostrado en la **Figura 6.1.3**:

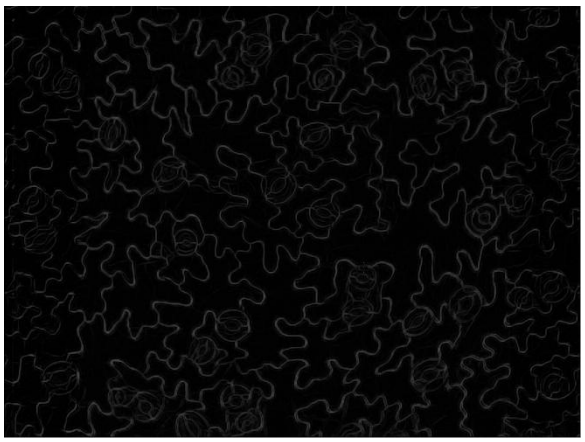

Figura 6.1.3 Resultado de la combinación de imágenes.

Finalmente, se aplica una función de umbral (threshold) para obtener una imagen binaria con las paredes de las células resaltadas, los resultados

pueden variar dependiendo del valor de threshold seleccionado, para nuestras pruebas fue 0.005. El resultado se muestra en la **Figura 6.1.4**.

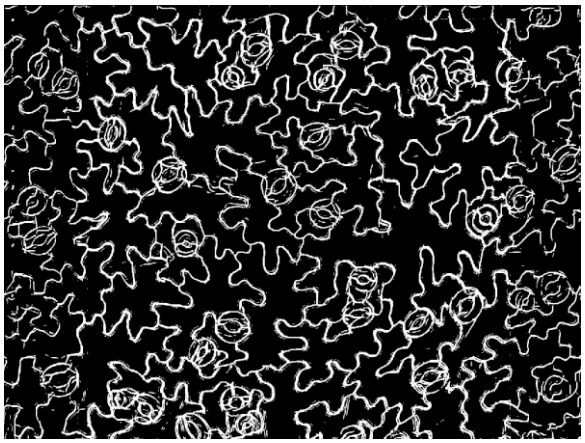

Figura 6.1.4 Salida final de programa.

Con respecto a Hadoop Pipes, en el Anexo 1 se muestra el código en C++ usando el paradigma MapReduce, en el cual se evidencia como se separó los pasos antes mencionados en las diferentes etapas de ejecución del paradigma.

Se implementaron conversiones de tipos de imágenes, una de las más útiles fue IplImageToFloat que permite convertir un tipo de dato IplImage* a un float*, utilizado por la función writeFileStream que guarda la imagen en formato EXR.

CAPÍTULO 7.

7 PRUEBAS Y ANÁLISIS DE RESULTADOS

En el presente Capítulo se detalla las pruebas realizadas para determinar la eficiencia y eficacia del programa desarrollado.

7.1 Pruebas con diferentes valores de umbral (threshold).

Como se mencionó en la sección 2.2.5 del valor de umbral dependerá la calidad de la imagen resultante, por lo que se llevaron a cabo tres pruebas con tres valores de umbral diferentes escogidos experimentalmente. Los valores de umbral fueron: 0.0007, 0.005 y 0.02.

A continuación en la **Figura 7.1.1** se muestran los resultados obtenidos variando el valor de umbral a 0.0007.

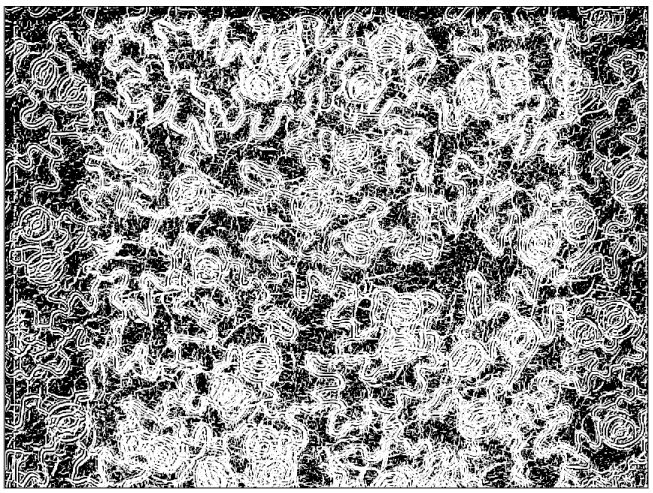

Figura 7.1.1 Resultado de usar un threshold de 0.0007.

Si se observa la **Figura 7.1.1** se puede concluir que al usar un valor de umbral muy pequeño nos da como resultado una imagen con mucho ruido e imperfecciones.

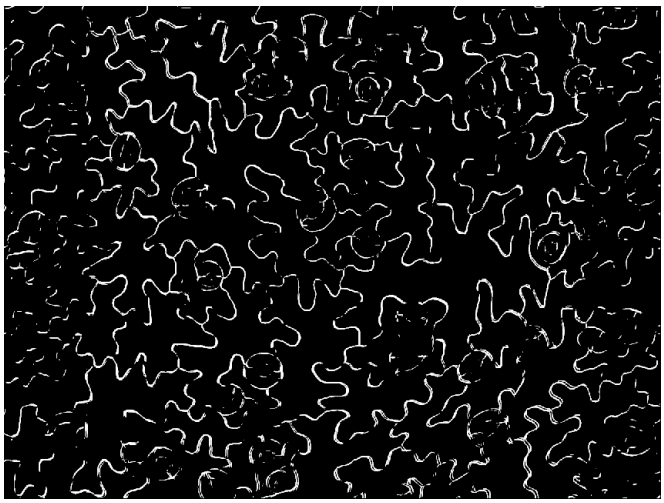

Figura 7.1.2 Resultado de usar un threshold de 0.02.

En la **Figura 7.1.2**, con un valor más alto obtenemos una imagen más

limpia pero que presenta las paredes celulares de forma discontinua, lo cual tampoco es un buen resultado para los procesamientos posteriores.

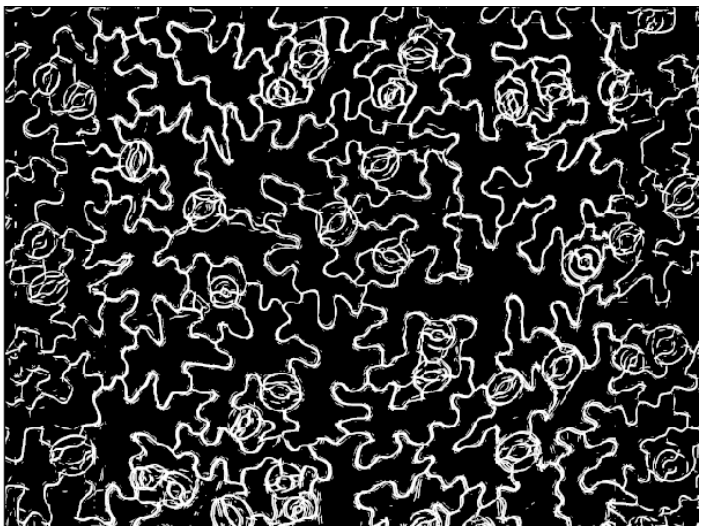

Figura 7.1.3 Resultado de usar un threshold de 0.005.

Finalmente, en la **Figura 7.1.3**, con un valor de umbral de 0.005 se obtiene un resultado limpio de ruido y con las paredes celulares bien definidas de forma continua.

Después de comparar los resultados con los 3 valores previamente definidos se determino que 0.005 es la mejor opción, se cierran los bordes entre las líneas y no presenta mucho ruido.

7.2 Pruebas de eficacia

Una de las pruebas iniciales de rendimiento que se realizaron fue obtener los tiempos de procesamiento de las diferentes fases del algoritmo hecho inicialmente en MATLAB y el algoritmo implementado en C++ con un conjunto de datos de entrada de 12 imágenes pertenecientes a un muestra, cuya comparación se puede evidenciar en la **Figura 7.2.1**.

	Matlab			C++	
Steps	Images	Time x image	Total time	Time x image	Total time
Image fusion	24	5	120	0	0
Matching	12	51	612	9	108
Line detection (2 scales)	12	13	156	17	206
Registration (300x300 patch)	11	5	55	2	22
Merging	12	1.7	20.4	1	12
Symmetry (1 radii value)	1	27	27	1	1
			990.4 seg		**351** seg
			16.5066667 min		**5.85** min

Figura 7.2.1 Comparación de resultados Matlab vs C++

Como resultado de la comparación obtuvimos una mejora de aproximadamente siete minutos, tiempo de alta importancia y mejora ya que para la investigación se requiere procesar varias muestras.

7.3 Pruebas de eficiencia

En una aplicación distribuida es necesario encontrar el tiempo de ejecución en relación al número de nodos utilizados con el cual la aplicación se estabiliza. Esto quiere decir, el umbral en el cual ya no se tienen mejoras de tiempo de ejecución cuando se aumenta el número

de nodos.

Los resultados de las pruebas de eficiencia se visualizan en la **Figura 7.3.1**:

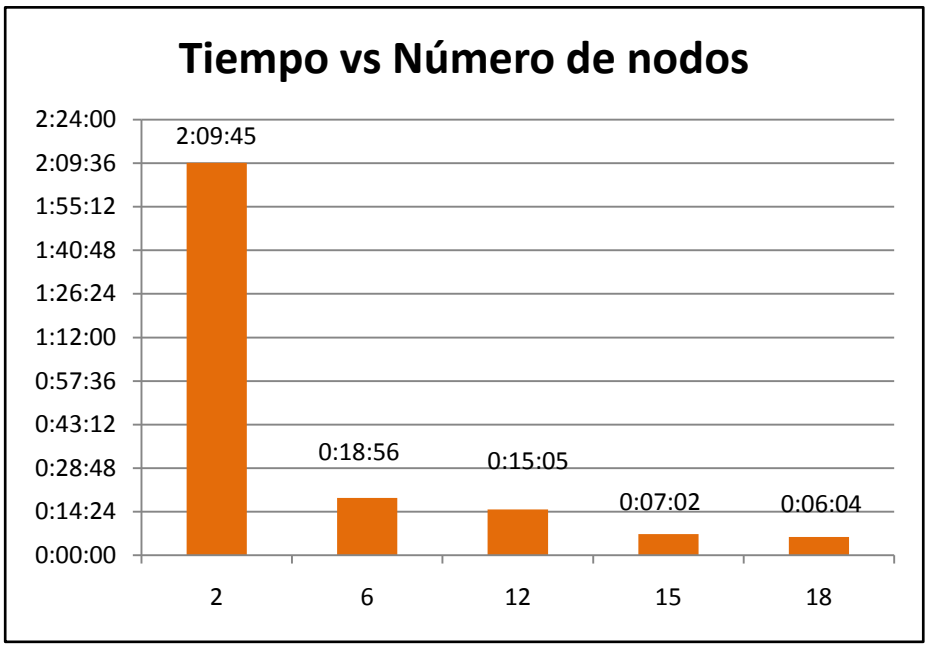

Figura 7.3.1 Reporte Gráfico del Tiempo vs Número de Nodos

Como se muestra en la **Figura 7.3.1** se realizaron cinco pruebas de procesamiento masivo de imágenes, teniendo una diferencia de tiempo muy variable, la mayor diferencia es de 1 hora 50 minutos con 49 segundos al pasar de 2 a 18 nodos y la menor diferencia es de 58 segundos al pasar de 15 a 18 nodos.

Con los siguientes resultados podemos concluir que para una variación de 2 a 6 nodos tenemos un reducción de aproximadamente 85% del tiempo de ejecución, mientras que si continuamos aumentado

el número de nodos observamos que de 6 a 12 nodos solo obtenemos un reducción del 16% de tiempo de ejecución, de 12 a 15 nodos obtenemos un 50% de reducción del tiempo y finalmente al usar 18 nodos comparados con el tiempo de usar 15 nodos se obtiene una reducción del 14%.

7.4 Análisis de los resultados

De manera general, se observa que a mayor número de nodos, se tiene un menor tiempo de procesamiento. Sin embargo, el correcto aprovechamiento del número de nodos depende del tamaño y la cantidad de imágenes a procesar, del número de Mappers a utilizar y del tamaño configurado para cada chunk[1].

Amazon solo cobra por horas de uso, dado que los valores de tiempo de ejecución del programa son menores a una hora se los redondea a su entero superior.

Finalmente se optó por la opción de 6 nodos como la mejor opción porque no consume muchos nodos, disminuye el tiempo de procesamiento de forma notable lo que se traduce en menos costo a pagar por la hora de procesamiento, además con más nodos la diferencia de tiempo es mínima y resulta más costosa.

[1] Chunk es un conjunto de datos que se envía a un procesador o cada una de las partes en que se descompone el problema para su paralelización.

CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES

1. El procesamiento de imágenes es una operación computacionalmente costosa, dependiendo de las características de la máquina o de las imágenes se podría convertir en una tarea poco práctica.

2. El número ideal de nodos para el procesamiento masivo de imágenes para el conjunto de imágenes analizados con un volumen de datos mayor a 700 MB, es de 6 nodos utilizando en los resultados de las pruebas que muestran poca diferencia de tiempo con más nodos generando un desperdicio de recursos.

3. Como nos demuestran las pruebas realizadas en el punto 7.2 el cambio de lenguaje de MATALAB que hace uso de Java a C++ produce una mejora significativa en la reducción del tiempo de procesamiento aun sin implementar el algoritmo para ser ejecutado en una plataforma distribuida.

4. La cache distribuida de Hadoop es el medio más eficiente de compartir información necesaria para el procesamiento entre todos los nodos.

5. Al analizar imágenes se debe considerar que Hadoop fue concebida como una plataforma de procesamiento masivo de texto, por lo que no posee un formato de entrada predeterminado para las imágenes, este formato de entrada se debe definir en base al formato de los datos de entrada que se van a procesar.

RECOMENDACIONES

1 Una de las posibles mejoras del proyecto es crear una interfaz para la carga de las imágenes, ejecución del algoritmo y descargar de las imágenes de salida.

2 Al levantar un nodo de Amazon, siempre se levantan con un AMI[1] por defecto con la distribución Fedora8 instalada. Es recomendable utilizar un AMI que tenga instalada la misma distribución con la que se realizaron las pruebas locales en los nodos para que sea más fácil la instalación de los paquetes y dependencias. En nuestro caso las pruebas se realizaron en la maquina virtual de Cloudera [17] con la distribución Ubuntu Intrepid.

[1] AMI (Amazon Machine Image) es un especial tipo de aplicación virtual la cual es usada para crear una maquina virtual dentro de Amazon EC2. Esto sirve como una la unidad básica de la implementación de los servicios prestados utilizando EC2.

3 Se puede obtener una mayor en la reducción del tiempo de ejecución, si se descarta procesamiento implementado para el algoritmo de Steger que no es utilizado para nuestro análisis.

4 Como posible mejora, se puede separa la fase de Image Registration dejando el proceso de rotación en el Mapper y la alineación en el Reducer para hacer la aplicación más autónoma y no dependiente de la imagen de referencia que se debe subir a la Caché distribuida por cada muestra.

ANEXOS

ANEXO A: CODIGO FUENTE

MAPPER

```
class CountMap: public HadoopPipes::Mapper {
public:
  CountMap(HadoopPipes::TaskContext& context) {

 }
  void map(HadoopPipes::MapContext& context) {
      ImageOpencv ImageCV;
      const HadoopPipes::JobConf* job = context.getJobConf();
      std::string fullname = context.getInputKey();
      std::string sample_name = ImageCV.getSampleName(fullname);
      int angle = ImageCV.getAnglefromName(fullname);
      std::string strImage = context.getInputValue();
      std::string data;
      int height;
      int width;
      IplImage* img;
      IplImage* img_32f;
      IplImage* smooth_32f;
      IplImage* smooth_normalize;
      IplImage* img_zero;
      Array2D<float> pixels_zero;
      IplImage* img_out;
      IplImage* img_rotated;
      IplImage* img_roi;
      IplImage* img_final;
      float* out;
      std::string filter_string = job->get("mapred.input.file.filter");
      std::string orientation_string = job->get("mapred.input.file.orientation");
      int filter, number_orientation;
      filter = HadoopUtils::toInt(filter_string);
      number_orientation = HadoopUtils::toInt(orientation_string);
      /*CARGA IMAGEN PGM DESDE EL DIERCTORIO DE ENTRADA*/
      data = ImageCV.getPGMImageData((char*)strImage.c_str(), &width, &height);
      img = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1 );
      img->imageData = (char*)data.c_str();



      ImageCV.readFileCache ("imagen_zero", pixels_zero, width, height);
      img_zero = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, 1 );
      img_zero = ImageCV.Array2DToImage(pixels_zero,width,height);

      img_32f = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, 1 );
      smooth_normalize = cvCreateImage(cvSize(width, height), IPL_DEPTH_32F, 1 );
      /*CONVIERTE EN IMAGEN DE FLOAT*/
      cvConvertScale( img, img_32f, (double)1.0/255.0, 0);
      cvReleaseImage(&img);
      /*SE APLICA FILTROS DE DE COINCIDENCIA*/
      smooth_32f = ImageCV.smooth_image( img_32f, filter,number_orientation);
```

```
	cvReleaseImage(&img_32f);
	/*NORMALIZACION DEL RESULTADO*/
	cvNormalize(smooth_32f,smooth_normalize, 1.0, 0.0, CV_C, NULL);
	cvReleaseImage(&smooth_32f);
	/*ALGORITMO DE STEGER*/
	img_out = ImageCV.StegerProcess(smooth_normalize);
	cvReleaseImage(&smooth_normalize);

	if(angle != 0){
		/*ROTACION*/
		img_rotated = ImageCV.rotate(img_out, angle);
		cvReleaseImage(&img_out);
		img_roi = cvCreateImage(cvSize(img_zero->width/2,img_zero->height/2),img_zero->depth,
img_zero->nChannels);
		cvSetImageROI(img_zero, cvRect(img_zero->width/4,img_zero->height/4, img_zero->width/2,
img_zero->height/2));
		cvCopy(img_zero, img_roi, NULL);
		cvResetImageROI(img_zero);

		/*ALINEACION*/
		img_final = ImageCV.image_registration( img_rotated, img_roi );
		out = ImageCV.IplImageToFloat( img_final);
	}else{
		out = ImageCV.IplImageToFloat( img_out);
	}
	std::ostringstream ostrm(ostringstream::out);
	C_OStream outStream(&ostrm);
	ImageCV.writeFileStream (&outStream,out, width, height);
	context.emit(sample_name, ostrm.str());
  }
};
```

REDUCER

```
class CountReduce: public HadoopPipes::Reducer {
public:

  CountReduce(HadoopPipes::TaskContext& context) {
  }
  void reduce(HadoopPipes::ReduceContext& context) {

	const HadoopPipes::JobConf* job = context.getJobConf();
	ImageOpencv ImageCV;
	std::string name;
	IplImage* img_32f;
	IplImage* img_out_32f;
	IplImage* img_max_32f;
	std::string width_string = job->get("mapred.input.file.width");
	std::string height_string = job->get("mapred.input.file.height");
	int height, width;
	height = HadoopUtils::toInt(height_string);
	width = HadoopUtils::toInt(width_string);
	img_32f = cvCreateImage(cvSize(width, height),  IPL_DEPTH_32F, 1 );
	img_out_32f = cvCreateImage(cvSize(width, height),  IPL_DEPTH_32F, 1 );
	img_max_32f = cvCreateImage(cvSize(width, height),  IPL_DEPTH_32F, 1 );
	/*CODIGO OPEN EXR*/
	Array2D<float> pixels;
	float* out;
	int flag = 0;
	name = context.getInputKey();
	while (context.nextValue()) {
		istringstream istrm(context.getInputValue());
		C_IStream inStream(&istrm);
		ImageCV.readFileStream(&inStream, pixels, width, height);
```

```
                img_32f = ImageCV.Array2DToImage( pixels, width, height);
                /*SE UNIFICAN TODAS LAS IMAGENES DE UNA MUESTRA*/
                if (flag = 0){
                        cvCopy(img_32f, img_max_32f, NULL);
                }else{
                        cvMax(img_32f,img_max_32f,img_out_32f);
                        cvCopy(img_out_32f, img_max_32f, NULL);
                }
                flag++;
        }
        /*SE APLICA THRESHOLD*/
        cvThreshold(img_max_32f,img_out_32f,0.005,1,CV_THRESH_BINARY);
        out = ImageCV.IplImageToFloat( img_out_32f);

        std::ostringstream ostrm(ostringstream::out);
        C_OStream outStream(&ostrm);
        ImageCV.writeFileStream (&outStream, out,img_out_32f->width, img_out_32f->height);

        cvReleaseImage(&img_out_32f);
        cvReleaseImage(&img_max_32f);
        cvReleaseImage(&img_32f);

        context.emit(name+".exr", ostrm.str());
  }
};
```

MAIN

```
int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<CountMap,
                        CountReduce>());
}
```

REFERENCIAS BIBLIOGRÁFICAS

[1] **Bradski Gary ,** OpenCV, http://opencv.willowgarage.com/wiki/, Octubre 2010.

[2] **Industrial Light & Magic and ILM,** OpenEXR, http://www.openexr.com/, Julio de 2010.

[3] **Dean Jeffrey y Ghemawat Sanjay,** *MapReduce: Simplified Data Processing on Large Clusters.* San Francisco : OSDI'04: Sixth Symposium on Operating System Design and Implementation, Diciembre de 2004.

[4] **Apache Software Foundation.** Hadoop, http://hadoop.apache.org/, Julio del 2010 .

[5] **Ochoa Daniel,** Cell Wall Detection of Arabidopsis Thaiana epithelial tissue in DIfferential Interference image, Material proporcionado en la materia de Graduación (VIB-finaldraft2.pdf), Agosto 2010.

[6] **MathWorks,** Matlab & Simulink, http://www.mathworks.com, Septiembre de 2010.

[7] **Araya Carrasco Hugo,** DESCRIPCIÓN DEL FORMATO DE IMAGEN PGM, http://www.ganimides.ucm.cl/haraya/doc/formato_pgm.doc, 2004.

[8] **Steger Carsten,** *An unbiased detector of curvilinear structure,* s.l. : IEEE

Transactions on Pattern Analysis and Machine Intelligence 20, 1998.

[9] **Amazon,** Amazon Web Services, http://aws.amazon.com/, Agosto 2010.

[10] **Amazon,** Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/, Agosto 2010.

[11] **Amazon,** Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/s3/, Agosto 2010.

[12] **White Tom**, *Hadoop: The Definitive Guide.* 2009. pág. 36, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, Junio 2009.

[13] **Apache Software Foundation,** Hadoop, http://hadoop.apache.org/common/docs/r0.15.2/streaming.html, Agosto 2010

[14] **Microsoft,** *Windows Azure,* http://www.microsoft.com/windowsazure/, Agosto 2010.

[15] **Poskanzer Jef,** PGM Format Specification, http://netpbm.sourceforge.net/doc/pgm.html, Julio de 2010.

[16] **Rooms Filip,** My software, http://www.filiprooms.be/, Febrero de 2010.

[17] **Cloudera,** Cloudera. http://www.cloudera.com, Febrero de 2010.