Trabajo de grado

Procesamiento de imágenes

Paralelización de algoritmos
de reconocimiento y clasificación
automática de objetos

UNLP - 1997

Alumnos

Fernando Ruscitti
Rodrigo Felice

Director

Ing. Armando De Giusti

Co-director

Lic. Marcelo Naiouf

Agradecimientos:

Agradecemos al Ing. Armando De Giusti y al Lic. Marcelo Naiouf por la colaboración prestada y por la paciencia que nos tuvieron.

A Silvia Dalinkevicius por la colaboración brindada en la edición del informe final, a Alejandra Reinoso por facilitarnos su máquina y soportarnos en su departamento, a Octavio Duré por prestarnos su máquina (a pesar de lo mucho que la utiliza).

A nuestras familias, por seguir considerándonos estudiantes, a pesar del tiempo transcurrido.

Indice:

Capitulo I

Problema a resolver

Estudios realizados demuestran un creciente interes en la automatización de procesos hasta ahora delegados casi exclusivamente a la mano y el discernimiento del hombre. Una gama muy amplia de estos procesos se basan en la percepción visual del mundo que nos rodea, por lo que su automatización tiene relación directa con el área de la informática que se ocupa del Procesamiento de Imágenes.

1.1 - Planteo del problema

En particular nos interesamos por la automatización de los procesos de clasificación y reconocimiento de objetos que provienen de alguna clase de producción serializada. La incorporación de computadores tiende a incrementar tanto la cantidad como la calidad de los productos obtenidos y a la vez reducir los costos.

Se tomó como modelo para la clasificación una cinta transportadora sobre la cual viajan los objetos a estudiar (Figura 1.1). En algún punto del recorrido de los objetos se encuentra una video cámara que toma imágenes de los mismos y las envía a una computadora. A partir de estas imágenes la aplicación clasifica al objeto y toma decisiones que eventualmente se traducen en señales que derivan al objeto a distintos sectores según sus características (calidad, color, tamaño, etc.) o lo descartan.

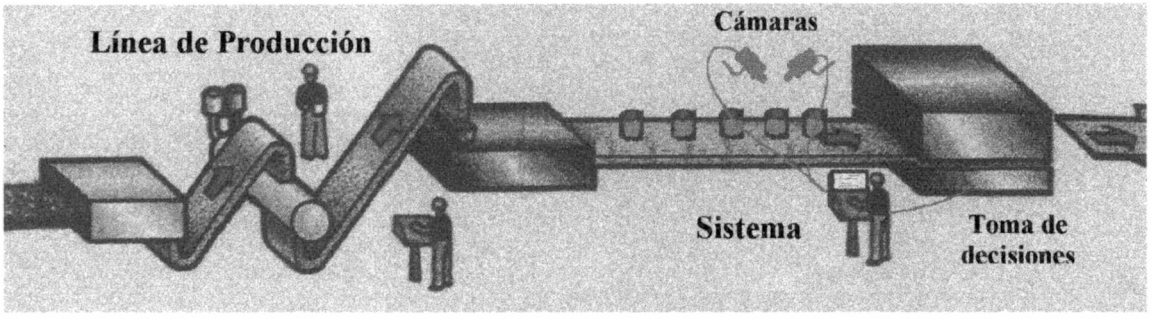


Figura 1.1 - Cinta transportadora, con el sistema clasificando los productos y decidiendo el destino de los mismos.

Comenzamos tomando como ejemplo de estos procesos industriales el caso de cintas transportadoras de huevos ya que la regularidad de estos objetos facilita los procesos de clasificación. Luego consideramos casos más complejos como frutas, botellas, cerámicas, etc. Para la clasificación se toman en cuenta diversas características de los objetos, como su área, color, diámetro máximo, perímetro, centroide y la presencia de defectos sobre su superficie (manchas, decoloraciones, etc.).

Cuando los procesos de clasificación no se encuentran automatizados son llevados a cabo mediante diferentes métodos, a saber:

- ***mecánicos:*** generalmente se utilizan zarandas con perforaciones de distintos tamaños que separan a los objetos basándose en el tamaño de los mismos. La principal característica de estos mecanismos es que solo son utilizables si los objetos en cuestión son geométricamente regulares, pues en caso contrario se dificulta el pasaje de los objetos por las perforaciones de las zarandas. Además, hay casos en que son imprecisos, y solo permiten diferenciar a los objetos por sus tamaños;

- ***electrónicos:*** están orientados a la detección de colores y son muy específicos para una aplicación en particular. Se utilizan fotocélulas que determinan el color de los objetos; esta información es recogida por Controladores Lógicos Programables (PCL) que la procesan y toman las determinaciones correspondientes;

- ***manuales:*** personal entrenado realiza la clasificación visualmente, por lo que sufren de las limitaciones de los humanos: son inexactos, dependen del entrenamiento, están sujetos al criterio personal, y además existen limitaciones que dificultan la realización de tareas rutinarias durante largos periodos. La eficiencia de los operadores se va degradando con el tiempo, y es necesario reemplazar el personal a intervalos relativamente cortos, lo que incrementa los costos de la clasificación.

1.2 - Objetivos del trabajo

Una solución informática carecería de los problemas presentados por los métodos mecánicos, electrónicos o manuales, ya que permite analizar un amplio espectro de características, es fácilmente adaptable a distintas clases de objetos y no cuenta con las limitaciones biológicas de los humanos.

Sin embargo, las aplicaciones de reconocimiento de objetos existentes cuentan con limitaciones de performance que restringen su uso a aplicaciones muy específicas donde los tiempos de respuesta no son determinantes o a proyectos de investigación donde se cuenta con máquinas especializadas de gran velocidad de computo.

Intentando superar estas limitaciones nuestro trabajo tendrá como objetivo principal el desarrollo de ***algoritmos de clasificación distribuidos*** que permitan llegar a resultados en tiempos aceptables utilizando ***redes heterogéneas de máquinas convencionales***.

Capitulo II

Procesamiento

El *procesamiento de imágenes* involucra todos aquellos procesos que sirvan para mejorar, corregir, analizar o manipular de alguna manera una imágen visual bidimensional.

Aunque el *procesamiento de imágenes* esta presente en muchas de las acciones que realizamos a diario, desde el comienzo de los tiempos, en el ambiente informático se empezó a estudiar en la década del 50 para llegar a ser hoy día una de las áreas de mayor desarrollo, aplicabilidad y con más futuro de las ciencias de la computación.

2.1- Aplicaciones del procesamiento de imágenes

2.1.1 - Sensado remoto e imágenes aéreas

Existe un gran número de áreas en las que se aplica el sensado remoto de imágenes, como la agricultura (para la predicción de la erosión de la tierra y la desertificación), la geología (búsqueda de recursos naturales y estudio de placas tectónicas), la hidrología (monitoreo de cambios en los glaciares y en el nivel de nieve), oceanografía (medición de la temperatura, nivel de plancton, detección de polución, etc.), meteorología (mediciones de la velocidad de viento, seguimiento de tormentas, etc.), cartografía (generación de mapas topográficos).

Para poder llevar a cabo las tareas antes descriptas se hace necesaria la utilización de la informática dado que el volumen de datos a procesar es considerablemente grande y las imágenes obtenidas necesitan ser transformadas para eliminar distorsiones ópticas. Además, muchas veces se requiere comparar imágenes de un mismo sitio obtenidas en distintos momentos y posiblemente desde distintos ángulos y alturas.

2.1.2 - Sistemas de visión computada

El objetivo principal del desarrollo de sistemas de visión computada es imitar el proceso de visión humano para poder automatizar tareas que tienen un alto grado de dificultad en cuanto a exactitud y tiempo, o son excesivamente rutinarias.

La utilización de este tipo de sistemas es muy amplia ya que los avances tecnológicos actuales están provocando la automatización de muchos de los procesos hasta ahora desarrollados por el hombre.

2.1.3 - Inspección industrial

Se utilizan para la clasificación y el control de calidad de manufacturas industriales, como por ejemplo en la inspección de la superficie del acero fabricado, en el control de la producción de circuitos integrados, etc.

Estos sistemas llevan a cabo tareas simples como procesar imágenes sencillas que han sido obtenidas bajo condiciones de iluminación controladas y desde un punto de vista fijo. Los sistemas deben operar en tiempo real, lo que implica el uso de hardware especializado, y sus algoritmos deben ser robustos de manera de reducir la posibilidad de fallas. Por último, los sistemas deben ser flexibles de modo que puedan adaptarse rápidamente a cambios en el proceso de producción y a los requisitos de inspección.

2.1.4 - Diagnóstico médico

Se procesan imágenes tales como radiografías, tomografías computadas, resonancias magnéticas nucleares, angiografías e imágenes microscópicas para asistir en el diagnóstico médico. En muchos procesos se hace necesario el mejorado de las imágenes obtenidas para poder así obtener resultados más fidedignos.

2.1.5 - Reconocimiento óptico de caracteres

Para automatizar el procesamiento de formularios se utilizan técnicas que permiten transformar información analógica contenida en documentos previamente escaneados a un formato digital que permita su procesamiento computacional. Ejemplos de estos documentos son los cheques, declaraciones juradas, etc.

Otra aplicación del reconocimiento de caracteres es la interpretación de texto manuscrito, con la posibilidad de su implementación en tiempo real.

2.2 - Computer vision

2.2.1 - Definición

> *Computer Vision: Ciencia que desarrolla teorías y algoritmos básicos por los cuales la información útil acerca del mundo puede ser automáticamente extraída y analizada a partir de imágenes.*

Para entender mejor los problemas y las metas que constituyen el propósito del *procesamiento de imágenes* es necesario comprender la complejidad que involucra nuestro propio sistema humano de visión.

Aunque la captación visual, el reconocimiento de objetos, y el análisis de la información recogida parece un proceso instantáneo para el ser humano, nuestro sistema de visión aún hoy no ha sido comprendido en su totalidad. Involucra sensores físicos que captan la luz reflejada por el objeto en cuestión (creando la imagen original), filtros que limpian la imagen (perfeccionándola), sensores que captan y analizan particularidades de la imagen para que, luego de una clasificación y reconocimiento de la misma, se puedan tomar decisiones sobre la base de la información acumulada. Así planteado, se puede apreciar la complejidad del aparentemente sencillo proceso de ver, el cuál conlleva una cantidad numerosa de subprocesos, todos realizados en unos pocos segundos.

El propósito y la estructura lógica de un sistema de visión computada (*Computer Vision*) es esencialmente el mismo que el de un sistema de visión humano. A partir de una imagen captada por un sensor, se realizarán todos los procesos y análisis necesarios para llegar al reconocimiento de la misma, y de los objetos que la componen.

En el diseño de un sistema de ***computer vision*** existen varias consideraciones a tener en cuenta:

¿ Qué tipo de información intentaremos extraer desde la imagen ?;
¿ Cuál es la estructura de esta información en la imagen ?;
¿ Qué conocimiento a priori necesitamos extraer de esa información ?;
¿ Qué tipos de procesos computacionales requeriremos ?;
¿ Cuáles son las estructuras para conocimiento y la representación de datos requerida ?.

2.2.2 - Etapas en el procesamiento del computer vision system

En el procesamiento requerido por computer vision encontramos cuatro aspectos principales, a saber:

i. Pre-procesamiento de los datos de la imagen;
ii. Detección de rasgos del objeto;
iii. Transformación de datos icónicos en datos simbólicos;
iv. Interpretación de la escena.

Cada una de estas tareas requiere una representación de datos distinta, como así también requerimientos computacionales diferentes. Surge entonces un interrogante:

¿ Qué arquitectura se requerirá para llevar a cabo las operaciones ?.

Muchas arquitecturas tienen desarrollados computadores en pipeline, lo que provee una limitada concurrencia de operaciones pero con menor transferencia (throughput) de datos. Otras cuentan con una malla conectada de arreglos porque su mapa de imágenes es eficiente, o aumentan esa arquitectura en malla con arquitecturas de árboles o pirámides porque proveen la jerarquía de operaciones que se cree involucra el sistema de visión biológico. Algunas arquitecturas son desarrolladas con máquinas paralelas más generales basadas en memoria compartida o hipercubos conectados.

Los problemas que se presentan en un computer vision system ocurren porque las unidades de observación no son unidades de análisis. La unidad de una imagen digital observada es el ***pixel***, abreviatura de *picture element*, el cuál tiene propiedades de posición y valor; pero el hecho de que se conozca la posición y el valor de cualquier pixel no aporta información sobre el reconocimiento de un objeto, la descripción de su forma, su orientación, la medida de cualquier distancia en el objeto, su grado de defectuosidad o simplemente información sobre qué pixels son parte del objeto en cuestión.

2.2.3 - Reconocimiento de objetos

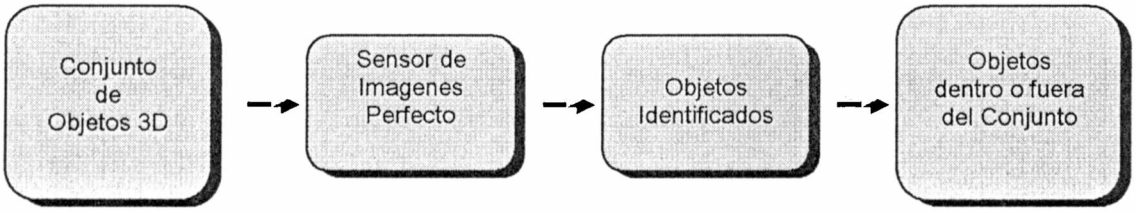
El proceso que nos permite clasificar los objetos de una imagen es llamado reconocimiento e inspección computacional de objetos, e involucra por su complejidad una variedad de pasos que sucesivamente transforman datos icónicos para reconocer información.

La detección de objetos, en tiempo real, contando solamente con la información procesada de la imagen, es una tarea problemática, pues la capacidad de detección es en función de numerosos factores. Entre ellos se encuentra:

- La similitud entre objetos del conjunto que se analiza con los del que se quiere detectar;
- Los diferentes ángulos posibles usados en la visualización de los objetos;
- El tipo de sensores involucrados;
- La cantidad de distorsión y ruido;
- Los algoritmos y procesos empleados;
- La arquitectura de los computadores utilizados;
- Las restricciones específicas de tiempo real;
- Los niveles de confidencialidad requeridos para tomar la decisión final.

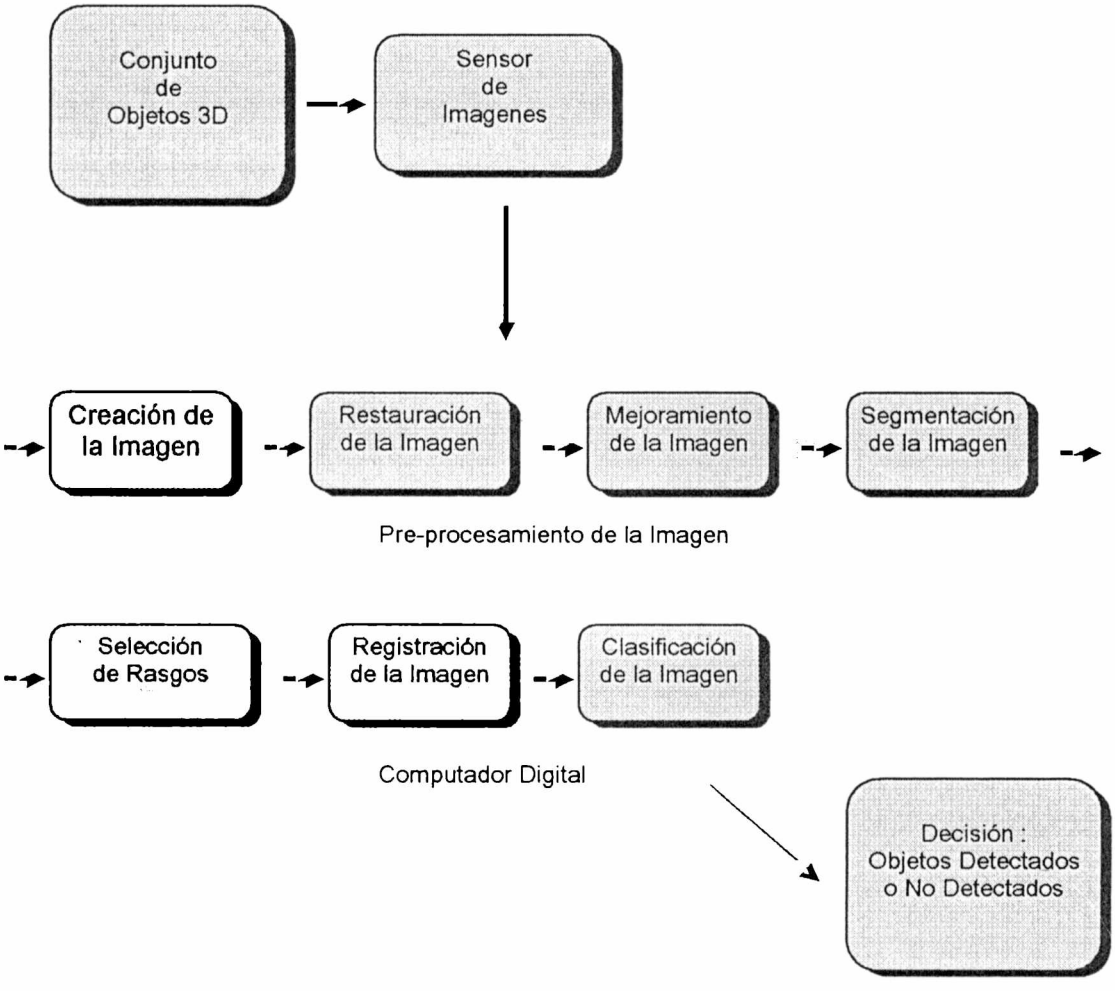
2.2.3.1 - Sistemas reconocedores de objetos

Un **sistema reconocedor de objetos *'ideal'*** puede diagramarse como muestra la siguiente figura:



Supone que el conjunto de objetos es observado por un sensor perfecto, sin transmisión de ruido ni distorsión, la cuál no es una suposición realista; los sensores no son perfectos.

Una visualización mas realista para el funcionamiento de un sistema de reconocimiento sería la siguiente:



Aquí la imagen es ***creada*** por un proceso matemático aplicado a los datos sensados, luego se la limpia o ***restaura*** para clasificar los rasgos que serán necesarios para posteriores procesos.

La ***segmentación*** la divide en piezas mutuamente exclusivas para facilitar los procesos de reconocimiento. Se ***seleccionan*** datos útiles y se ***registran*** aspectos relevantes aportados por los sensores (altitud, velocidad, etc.) para poder llevar a cabo la ***clasificación***.

Los objetos segmentados en la imagen son chequeados contra una clase predeterminada para ver si los objetos buscados están presentes, con lo cual se completa el proceso decisivo de detección.

2.2.3.2 - Metodologías de reconocimiento de objetos

Las metodologías de reconocimiento mas difundidas se guían por los siguientes seis pasos:

1) Formación de la Imagen
2) Conditioning (Acondicionamiento de la imagen)
3) Labeling (Etiquetamiento de eventos)
4) Grouping (Agrupamiento de eventos)
5) Extracting (Extracción de propiedades)
6) Matching (Igualación de formas)

Excepto la formación de la imagen, el resto de los pasos pueden ser aplicados varias veces, en distintos niveles de procesamiento.

El manejo de ambientes no restringidos es una de las dificultades de hoy día pues los algoritmos de computer vision y reconocimiento existentes son especializados y no desarrollan mas que algunos de los pasos de transformación necesarios, a un alto grado de dificultad.

2.2.3.2.1 - Formación de la imagen

Una Imagen es una representación espacial de un objeto, en una escena bidimensional o tridimensional. Para poder trabajarla computacionalmente es necesario encontrarle una representación digital que permita organizar los datos captados. Entonces, representamos la imagen como una matriz de valores numéricos, cada uno de los cuales representa un valor de intensidad, llamado 'gray level' (nivel de gris) o 'color level' (nivel de color), que equivale al pixel antes descripto.

Definición: sea I una imagen bidimensional representada por una función 'f' tal que $I = f(x, y)$ *donde la posición en el plano es dada por las coordenadas (x,y) y la intensidad del nivel de gris es un número real.*

La digitalización de las coordenadas espaciales (x,y) se denomina muestreo de la imagen (image sampling) y la determinación de los valores de brillo digitales se denomina cuantización de niveles de gris (grey level cuantization).

Estos dos valores determinan la resolución de la imagen; a medida que los aumentamos, se incrementa la resolución, aproximando la representación a la imagen original, aunque también se incrementa el almacenamiento requerido.

2.2.3.2.2 - Acondicionamiento de la imagen (Conditioning)

Esta basado en la hipótesis que la imagen observada está compuesta de un patrón informativo, el cuál es agregado o multiplicado hasta obtener la imagen. Bajo este modelo, la operación de conditioning estima el patrón base de la imagen observada, eliminando el ruido.

El propósito del acondicionamiento es realzar y restaurar la imagen original para obtener una 'mejor', entendiéndose por 'mejor' a lo que el observador considere como calidad en la imagen.

Se pueden utilizar técnicas de realzado del contraste, smoothing y sharpening. La restauración consiste en la reconstrucción de una imagen degradada conociendo a priori la causa de la degradación (movimiento, foco, ruido, etc.).

2.2.3.2.3 - Etiquetamiento de eventos (Labeling)

Se basa en el modelo que define al patrón informativo con una estructura de arreglo de eventos (o regiones), donde cada evento es por ahora un conjunto conectado de pixels. El Labeling determina a qué clase de evento pertenece cada pixel, colocando igual etiqueta a los pixels de un mismo evento.

Como ejemplos de Labeling podemos mencionar al Thresholding (con valores de pixels muy altos o muy bajos), la detección de bordes, búsqueda de córners, etc.

2.2.3.2.4 - Agrupamiento de eventos (Grouping)

Identifica los eventos, para lo cuál puede colectar simultáneamente o identificar conjuntos de pixels muy conectados que estén en una misma clase de evento. El ***grouping*** pude ser definido como una operación de componentes conectadas (si las etiquetas utilizadas son simbólicas), una operación de segmentación (si las etiquetas son en niveles de grises) o un enlazamiento de bordes (si las etiquetas son pasos de bordes).

Hasta ahora, los procesos aplicados a la imagen observada mantenían la estructura de imagen digital. Dependiendo de la implementación, el grouping puede generar una estructura de datos de la imagen en la cuál cada pixel es dado como un índice asociado al evento al cuál pertenece, o una estructura de datos que es una colección de conjuntos donde cada uno corresponde a un evento y contiene los pares de (fila,columna) - posiciones - que participan en el evento.

Debemos destacar que las entidades de interés antes de grouping son los pixels, mientras que después son los conjuntos de pixels.

2.2.3.2.5 - Extracción de propiedades (Extracting)

Asigna nuevas propiedades al conjunto de entidades que genera el grouping (recordando que dicho conjunto solo contiene la identidad de los pixels que pertenecen a

cada evento). Alguna de estas propiedades para cada grupo de pixels son el centro, el área, la orientación y los momentos. Si el conjunto de pixels es una región entonces podríamos obtener el número de agujeros de cada región. Si es un arco, el promedio de curvatura sería una propiedad interesante.

Además de propiedades para cada conjunto de pixels, el ***extracting*** puede determinar relaciones entre grupos de pixels, como ser si éstos se tocan, si son cerrados, si un grupo está dentro de otro, etc.

2.2.3.2.6 - Igualación de formas (Matching)

Los eventos (o regiones) ocurridos en la imagen son identificados y medidos a través de los procesos anteriormente descriptos, pero para poder asignarles un significado es necesario realizar una organización conceptual que permita identificar un conjunto específico de eventos en la imagen observada como una instancia imaginada de algún objeto previamente conocido. Otras veces, un objeto o un conjunto de partes de objeto son reconocidos, lo que permite realizar mediciones como distancia entre dos partes de objeto, ángulo entre líneas, área de una parte, etc.

Entonces, el ***matching*** determina la interpretación de algunos conjuntos de eventos relacionados, asociando esos eventos con algún objeto tridimensional dado o figura bidimensional. Las figuras simples se corresponderán con eventos primitivos y la propiedad de medición desde un evento primitivo frecuentemente será adecuada para permitir el reconocimiento de la figura. Las figuras complejas se corresponderán con conjuntos de eventos primitivos. Aquí, el reconocimiento se obtendrá por el uso de un vector de propiedades de cada evento observado así como de las relaciones entre los eventos.

2.2.3.3 - Binary Machine Vision

En la detección y reconocimiento de objetos o de defectos en los objetos por un sistema de computer vision, frecuentemente la imagen de entrada es simplificada por la generación de una imagen de salida donde los pixels tienden a tener valores altos si forman parte del objeto de interés y valores bajos en caso contrario.

Para reconocer realmente un objeto, es necesario identificar primero las regiones que puedan llegar a ser parte del objeto. El camino más simple para identificar esas regiones de objetos es hacer una operación de **threshold_labeling** en la cuál a cada pixel que tiene un valor suficientemente alto se le asigna el valor binario 1, lo que identifica al pixel como uno con posibilidad de formar parte del objeto de interés. A cada pixel que no tiene un valor suficientemente alto se le asigna el valor binario 0, que indica que la posibilidad es pequeña.

La generación y el análisis de esta imagen binaria son llamadas **Binary Machine Vision**. El primer paso de la B.M.V. es hacer threshold de una imagen en escala de grises, etiquetando así cada pixel como 0 binario (fondo de la imagen) o 1 binario (objeto de interés). Entonces el thresholding es una operación de **Labeling**.

Dependiendo de la complejidad de los objetos y de la naturaleza de las figuras a ser identificadas y su posición relativa esperada, la siguiente etapa de procesamiento puede ser una de entre dos técnicas de agrupamiento: etiquetar componentes conectadas o segmentar signaturas.

Ambas técnicas hacen una transformación en la clase de unidades procesadas hasta ahora. Las unidades de la imagen son los pixels. Las unidades después de la

transformación son más complejas; se las llama regiones o segmentos y están compuestas de agrupamientos de pixels. Después de que las regiones son definidas, una variedad de mediciones pueden ser hechas en ellas, lo que podemos llamar **extracción de rasgos**.

Las regiones son finalmente asignadas a una clase de objeto o a una clase de defecto de objeto o categoría a través de una técnica de reconocimiento de patrones. Esto constituye los pasos de **matching e inferring** (igualación y deducción).

La secuencia de operaciones thresholding_labeling de componentes conectadas, mediciones de propiedades de regiones y reconocimiento estadístico de patrones, es llamada ***Análisis de Componentes Conectadas*** (Algoritmos SRI - Gleason y Agin, 1979; Agin 1980).

La secuencia de operaciones thresholding, segmentación, signatura, mediciones de propiedades de regiones y reconocimiento estadístico de patrones, es llamada ***Análisis de Signaturas***.

2.2.3.3.1 - Thresholding

Es una operación de labeling sobre una imagen en escala de grises o colores. Distingue pixels que tienen un valor alto de gris o color de los que tienen un valor bajo.

Es uno de los métodos mas usados para extraer una figura o un rasgo de interés particular de una imagen. El operador binario de threshold produce una imagen en blanco y negro ya que asigna un 1 a pixels con valor alto y un 0 a los que tienen un valor bajo; generalmente el objeto de interés queda negro (valor binario 1) y el fondo blanco (valor binario 0).

2.2.3.3.1.1 - Definición

Definición: sea f una imagen dada por la matriz de la fig.2.2 (bosquejada en la fig. 2.1). En la figura, el nivel de gris se mueve entre 0 y 8, donde 0 representa blanco y ocho negro. El threshold de la imagen f con el valor 7 lleva a blanco (0) todos los pixels con valor de gris menor a 7, y lleva a negro (1) todos los pixels con valor de gris igual o mayor a 7.

La imagen resultante (figura 2.3) tiene solamente dos valores de grises, 0 y 1. La letra H es claramente discernible en la imagen resultante. Aquí vemos que los caracteres con * no son parte del dominio de la imagen y por lo tanto no se ven afectados por el threshold.

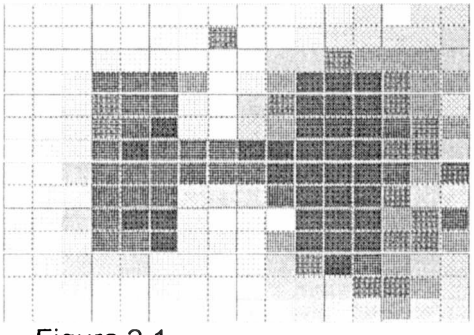


Figura 2.1

*	0	0	0	*	0	0	0	1	2	3	*	3	3
0	0	0	0	1	1	6	0	0	2	3	3	3	4
0	0	2	1	1	0	0	*	4	4	6	5	5	4
0	0	2	7	7	7	5	0	2	5	8	8	8	6	5	5
0	0	2	6	7	7	2	0	4	6	8	8	8	6	5	3
0	0	2	6	7	8	2	0	3	5	8	8	8	7	6	5
0	1	2	7	8	7	7	7	8	8	8	8	8	6	6	4
0	0	4	7	7	7	7	7	7	8	8	8	8	7	5	6
0	2	2	7	7	7	3	4	4	7	8	8	8	6	4	3
1	1	4	7	8	8	2	0	1	*	8	8	8	5	6	7
1	0	2	7	7	8	2	2	1	5	8	8	8	6	6	5
0	1	3	3	4	2	2	0	1	4	6	8	7	5	4	3
0	0	2	1	1	2	0	0	0	3	4	4	6	6	5	3
*	0	0	0	0	0	0	0	2	2	3	2	3	5	3	3

Figura 2.2

*	0	0	0	0	*	0	0	0	0	0	0	0	*	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	*	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	0	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	1	1	1	0	0	0	1	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0	*	1	1	1	0	0	1
0	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 2.3

Podemos introducir la notación para la imagen resultante definida por:

$$[\,\textbf{THRESH}\,(f,t)\,] \qquad (i,j) = \begin{cases} 1, \text{si } f(i,j) >= t \\ 0, \text{si } f(i,j) < t \\ *, \text{si } f(i,j) = * \end{cases}$$

Observando las figuras se puede notar que:

- Los asteriscos son pixels donde el sensor no ha producido información, por lo tanto esos pixels no forman parte del dominio de la imagen;
- el lado derecho de la imagen es más oscuro que el izquierdo, lo cuál puede ser producto de una distorsión causada por el sensor, luz reflejada o transmisión degradada de los datos, etc.
- En la figura 1.3 hay dos pixels de la H arriba a la izquierda que fueron blanqueados (llevados a 0 binario) por el thresholding con valor 7, pero que tenían un valor de gris 6 (bastante oscuro). Asimismo, se obtienen 6 pixels extras en la parte derecha. Podríamos mejorar la parte izquierda de la H con un valor de threshold 6, pero empeoraría la parte derecha; o podría mejorar la parte derecha de la H con valor de threshold 8 pero perdería pixels de la izquierda. Entonces llegamos a la conclusión que el mejor valor de threshold a elegir es el 7.

2.2.3.3.1.2 - Valor de threshold

Sobre la última de las observaciones del punto anterior, podemos agregar que hay métodos para elegir automáticamente este valor ideal de threshold, como por ejemplo hallar el **histograma** de la imagen. Esta definición automática es uno de los puntos mas importantes del threshold:

Qué base puede ser usada? para encontrar un valor que separe el fondo oscuro del objeto brillante (o viceversa) de una manera óptima, sería deseable conocer la distribución de los pixels oscuros y la de los brillantes.

El valor de threshold podría entonces ser determinado como ese valor de separación por el cuál la porción de pixels oscuros etiquetados 1 (mal etiquetados) iguale la porción de pixels brillantes etiquetados 0. Esto equipararía la probabilidad de error al etiquetar pixels de fondo como parte del objeto y viceversa. La dificultad radica en que la distribución independiente de pixels oscuros y brillantes generalmente no es conocida en un primer momento.

2.2.3.3.1.3 - Histograma

El histograma nos dice cuántos pixels hay en la imagen para cada valor de gris (o color). Se define como:

$$HIST(m) = \#\{(r,c) \;/\; I(r,c) = m\}$$

Con - *m : cualquier valor de gris (o color);*
- *# : operador que cuenta número de elementos en el conjunto.*

La función **histograma** puede ser implementada fácilmente usando un arreglo con MAX elementos (MAX es el valor de gris o color mas alto), como se detalla en seudocódigo a continuación:

```
Procedure Histograma ( Img, Hist )
   for x := 0 to MAX do
      HIST ( x ) := 0;
   for row := 1 to MAXROW do
      for col := 1 to MAXCOL do
         gris := I ( row , col );
         HIST ( gris ) := HIST ( gris ) + 1;
      end for;
   end for;
end Histograma;
```

2.2.3.3.2 - Labeling de componentes conectadas

2.2.3.3.2.1 - Introducción

Consiste en el etiquetamiento de las componentes conectadas con valor de pixel 1 (sobre una imagen binaria) seguido de mediciones sobre las regiones encontradas; la región es una unidad mas compleja que el pixel y tiene un conjunto de propiedades mas completo (forma, posición, estadísticas del nivel de gris, etc.), por lo que para cada región se puede construir una N_tupla o vector de esas propiedades medidas. Una forma de reconocer objetos diferentes, objetos defectuosos o características es distinguir entre las regiones en base a sus propiedades.

El etiquetamiento cambia la unidad de trabajo: el pixel es reemplazado por la región o segmento. Todos los pixels que tienen valor binario 1 y están conectados entre sí tienen la misma etiqueta identificatoria. La etiqueta es un único nombre o índice de la región a la cual el pixel pertenece; además, es el identificador para una región. El labeling de componentes conectadas es una operación de *grouping*.

Los algoritmos para labeling de componentes conectadas agrupan todos los pixels pertenecientes a la misma región y les asignan igual etiqueta [Software para realizar Labeling de Componente Conectadas puede ser encontrado en Ronce and Divijver (1984) y en Cunningham (1981)].

2.2.3.3.2.2 - Operadores de componentes conectadas

El operador de etiquetamiento se utiliza normalmente en aplicaciones donde las imágenes están formadas por un número pequeño de objetos siempre en contraste con el fondo. El operador toma la imagen resultante del Thresholding y agrupa los pixels con valor binario 1 en regiones conectadas, llamadas componentes conectadas. El resultado es una imagen simbólica donde cada pixel 1 tiene un valor entero que identifica la región a la cuál pertenece. En las siguientes figuras se puede observar el resultado de aplicar un operador de componentes conectadas.

0	1	1	0	1
0	1	1	0	1
1	1	0	0	0
0	0	0	1	1
0	1	1	1	0

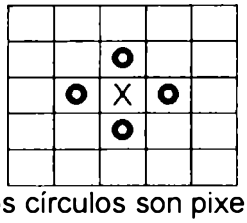
imagen binaria inicial

0	1	1	0	2
0	1	1	0	2
1	1	0	0	0
0	0	0	3	3
0	3	3	3	0

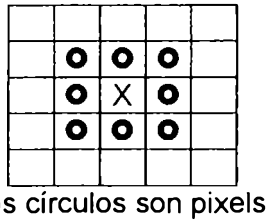
imagen simbólica result. de la aplicación del oper. de componentes conectadas

*Definición: dos pixels p y q pertenecen a una componente conectada C si existe una secuencia de pixels (p_0, p_1, ... , p_n) de C donde $p_0 = p$, $p_n = q$ y p_i es **'vecino'** de p_{i-1} para i=1, ..., n.*

Esta definición de componentes conectadas es dependiente de la definición que se haga del término 'vecino'. Se pueden tomar como vecinos de un pixel dado a los que se encuentren pegados al norte, sur, este y oeste (entonces serán regiones 4_conectadas) o se puede considerar además como 'vecino' a los pixels ubicados al sudeste, sudoeste, noreste y noroeste (con lo que las regiones se denominarán 8_conectadas) como se puede ver en la siguiente figura.



los círculos son pixels 4-conectados a la X



los círculos son pixels 8-conectados a la X

```
            them CAMBIO := true;
            LABEL( L , P ) := M;
        end
    end for
  end for;

  // Pasada Bottom-up //
  for L := NLINES to 1 by -1 do
    for P := NPIXELS to 1 by -1 do
        if LABEL( L , P ) <> 0 then
        Begin
            M := MIN ( LABELS ( NEIGHBORS ( ( L , P ) ) U ( L , P ) ) );
            if M <> LABEL( L , P )
            then  CAMBIO := true;
            LABEL( L , P ) := M;
        end
    end for
  end for;
until CAMBIO = false
end Iterador;
```

Algoritmo cíclico

Esta basado en el algoritmo clásico de Componentes Conectadas para Grafos descripto en Rosenfeld and Pfaltz (1966) [Se lo puede encontrar en la mayoría de los libros que tratan el tema, como por ejemplo en Aho, Hopcroft and Ullman (1983)].

Hace solo dos pasadas por la imagen pero requiere una extensa tabla global para almacenar equivalencias. La primera pasada ejecuta la propagación de etiquetas. Cuando se presenta una situación en la que dos etiquetas distintas pueden ser propagadas al mismo pixel, se propaga la mas pequeña y cada equivalencia encontrada es agregada a la tabla. Cada entrada en la tabla de equivalencias es un par ordenado (etiqueta, etiqueta equivalente).

Después de la primera pasada se encuentran las 'clases de equivalencias' haciendo la clausura transitiva del conjunto de equivalencias de la tabla. Cada clase de equivalencia es asignada a una única etiqueta, frecuentemente la menor (o la mas antigua) de la clase. La segunda pasada asigna a cada pixel la etiqueta de la clase de equivalencia correspondiente a la etiqueta que se le asignó al pixel en la primera pasada.

Una aproximación al algoritmo descripto, escrita en seudo-código, es la que se detalla a continuación:

```
//-------------------------//
  Procedure Clásico
//-------------------------//
// Inicialización de la Tabla de Equivalencias //
EQTABLE := CREATE();
```

```
// Primera Pasada Top-down //
for L := 1 to NLINES do
  // Inicialización de las etiquetas de la línea L en cero//
  for P := 1 to NPIXELS do
     LABEL( L , P ) := 0;
  end for;

  // Procesamiento de la línea //
  for P := 1 to NPIXELS do
     if F( L , P ) := 1 then
     Begin
        A := NEIGHBORS ( ( L , P ) );
        if ISEMPTY( A )
        them M := NEWLABEL()
        else  M := MIN( LABELS ( A ) );
        LABEL( L , P ) := M;
        for X in LABELS( A ) and X<>M
           ADD( X , M , EQTABLE )
        end for;
     end
  end for
end for;

//Búsqueda de las Clases de Equivalencias
//Resolver encuentra las componentes conectadas de la estructura //de grafo
definida por el conjunto de equivalencias creado en //la primer pasada.//

EQCLASES := Resolver( EQTABLE );

for E in EQCLASES
  EQLABEL( E ) := min ( LABELS( E ) )
end for;

// Segunda Pasada Top-down //
for L := 1 to NLINES do
  for P := 1 to NPIXELS do
     if I( L , P ) = 1
     then  LABEL( L , P ) := EQLABEL( CLASE( LABEL( L , P ) ) )
  end for
end for
end Clásico;
```

El problema principal de este algoritmo radica en la tabla de equivalencias global; en imágenes grandes con muchas regiones se torna demasiado extensa, siendo problemático su almacenamiento.

2.2.3.3.3 – Segmentación de Signaturas

2.2.3.3.3.1 – Definición

Consiste en la toma de una o mas proyecciones de una imagen binaria o de una subimagen de una imagen binaria, segmentando cada proyección para formar una nueva unidad de mas alto nivel, y tomando mediciones de las propiedades de cada segmento proyección.

2.2.3.3.3.2 – Análisis de signaturas

El análisis de signaturas, al igual que el análisis de componentes conectadas, realiza cambios en las unidades de trabajo, pasando de pixels a regiones o segmentos. Consiste en la generación de una imagen binaria mediante una técnica de thresholding, realizando proyecciones, para luego tomar decisiones sobre los objetos en base a las propiedades de dichas proyecciones. Estas pueden ser verticales, horizontales, diagonales, circulares, radiales, espirales o proyecciones generales.

El análisis de signaturas ha sido usado primero para el reconocimiento de caracteres [Sanz, Hinkle, and Dinstein (1985), y Sanz y Dinstein (1987) debaten sobre una clase de arquitectura pipeline para el cómputo de proyecciones y de rasgos geométricos basados en proyecciones]. Es importante para binary vision porque puede ser computada fácilmente en tiempo real.

Una proyección general de una imagen binaria dada puede ser obtenida al enmascarar una imagen indexada proyectada con la imagen binaria dada. Cada pixel de la imagen resultante toma el valor 0 si el correspondiente pixel de la imagen binaria es un 0. Y si el pixel de la imagen binaria es un 1, entonces el pixel tomará el valor del pixel correspondiente en la imagen indexada proyección.

La signatura, que es una proyección, es el *histograma* de los pixels distintos de cero de la imagen resultante. Para obtener una proyección vertical, la imagen indexada proyección contiene el valor *c* en todo pixel de la columna coordenada *c*. De ese modo, una proyección vertical es una función unidimensional que para cada columna tiene el valor dado por el número de pixels en la imagen binaria en la columna conteniendo valor binario 1.

En el análisis de signaturas, la imagen binaria debe ser procesada de modo tal que el desorden de todos los objetos no importantes sea eliminado antes de sacar la proyección. Esto podría ser hecho en parte por el procesamiento de escala de grises que precede al thresholding, en parte por operaciones morfológicas binarias después de thresholding, y en parte por un simple enmascaramiento.

La siguiente etapa en la segmentación de signaturas es la segmentación de proyecciones. La segmentación lleva a cabo una transformación de la unidad pixel a la unidad segmento proyectado. Segmentos sucesivos mutuamente excluyentes pueden ser determinados mediante la localización de lugares donde los valores de la proyección son relativamente pequeños.

La segmentación de proyecciones induce a la segmentación de la imagen en los siguientes pasos: suponemos que un segmento determinado desde la proyección

vertical está dado por { c | s = c = t } y que un segmento determinado desde la proyección horizontal esta dado por { r | u = r = v }. Estos segmentos verticales y horizontales definen un segmento R en la imagen dado por: R = { (r , c) | u = r = v and s = c = t }.

Si hay N_H segmentos de la proyección horizontal y N_V de la vertical, entonces se inducirán que en la imagen hay $N_H.N_V$ segmentos mutuamente excluyentes, cada uno de los cuales será por el momento una subimagen rectangular.

Esta inducción de segmentación en la imagen desde las segmentaciones de proyecciones verticales y horizontales conduce hacia un camino iterativo de refinamientos de la segmentación inicial de la imagen.

2.2.3.3.3.3 – Técnica de refinamiento iterativo

La técnica de refinamiento iterativo trabaja de la siguiente manera:

1º. Segmenta las proyecciones verticales y horizontales de la imagen.
2º. Induce la segmentación en la imagen desde esas segmentaciones.
3º. Trata cada subimagen rectangular de la misma forma que fue tratada la imagen.
4º. Determina las proyecciones verticales y horizontales, segmenta las proyecciones e induce la segmentación hacia atrás sobre la subimagen.
5º. Continúa en este camino hasta que cada subimagen rectangular resultante tiene proyección vertical y horizontal que consiste de un segmento y por lo tanto no puede ser nuevamente dividida.

A partir de las proyecciones vertical, horizontal y diagonal se puede computar fácilmente el centroide, momentos centrales secundarios y límites del rectángulo.

2.2.3.3.3.4 - Ejemplo de cálculo de proyecciones horizontales y verticales

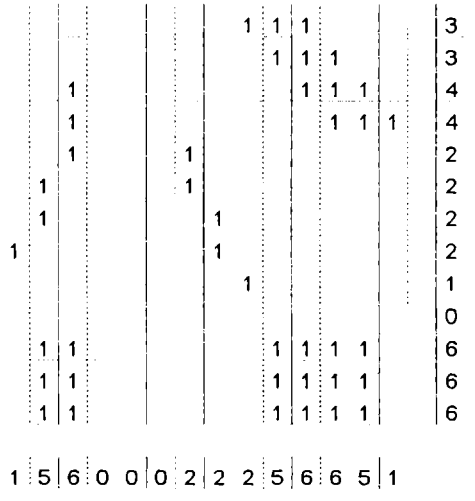


Figura 2.4 - Imagen Binaria y las proyecciones vertical y horizontal. La proyección horizontal es el histograma de la imagen binaria enmascarada por la proyección horizontal mascara de la figura 2.5.

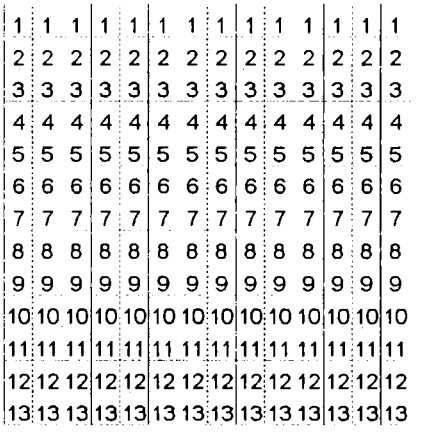


Figura 2.5 - Máscara proyección horizontal.

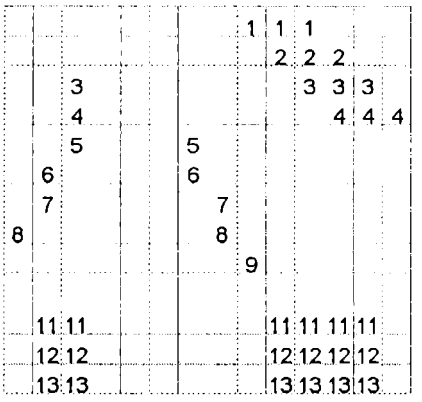


Figura 2.6 - Imagen binaria de la fig. 2.4 enmascarada por la proyección horizontal máscara de la fig. 2.5.

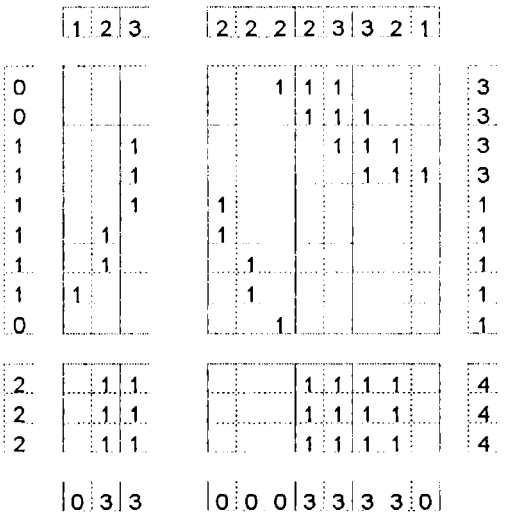


Figura 2.7 - Imagen binaria segmentada en regiones sobre la base de la segmentación inicial de las proyecciones vertical y horizontal. También se ven las proyecciones horizontal y vertical de cada región.

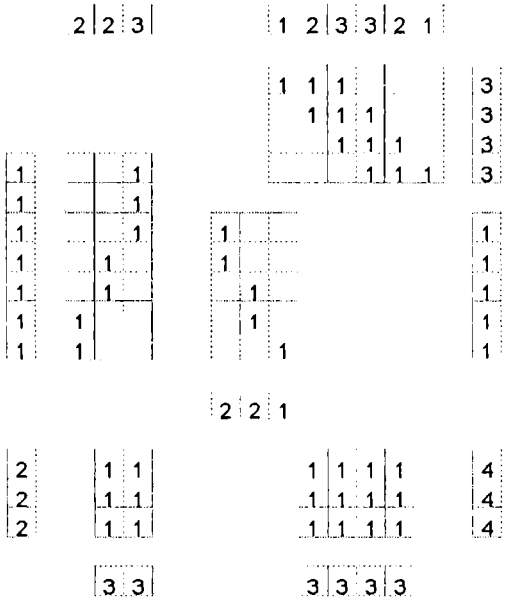


Figura 2.8 Se observa la imagen binaria segmentada en regiones sobre la base de la segmentación de la fig. 2.7. También se ven las proyecciones horizontal y vertical de cada región.

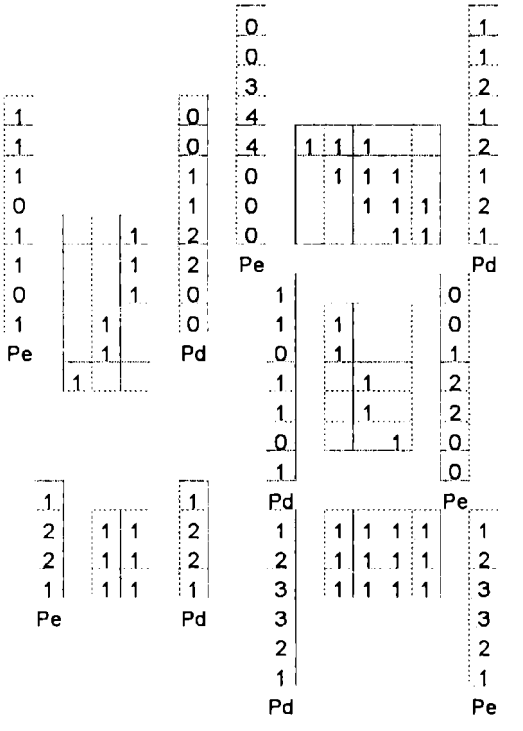


Figura 2.9 – Proys. diagonales Pd y Pe para c/u de las 5 regiones de la fig. 2.8. Pd es la proy. diag. a 45° en el sentido del reloj desde la horizontal.

La fase final del análisis de signaturas son las mediciones de rasgos de cada uno de los segmentos proyección. Un rasgo puede consistir de la suma de todos los valores proyección en el segmento, con lo que se obtiene el área del objeto. Otro es la suma pesada de todas las posiciones de la proyección en el segmento, pesada por el valor de proyección, con lo que se produce una posición proyectada central.

Se pueden calcular otros rasgos que incluyan números y alturas de picos (convexidades) y número y profundidades de cuencas (concavidades). Si la forma de la proyección es suficientemente simple, es posible aproximar a una forma funcional los valores de los segmentos proyección. El vector de rasgos puede ser construido desde los parámetros computados de las coincidencias. Finalmente, el segmento proyección puede ser normalizado teniendo una longitud pre-especificada, y los valores de la proyección normalizada pueden constituir un vector de medidas proyectadas.

Así, después del análisis de signaturas, para cada segmento proyección tendremos una N-tupla o vector de las propiedades medidas. Para reconocer diferentes objetos, defectos de los objetos, o características se distingue entre los segmentos proyectados en base a las propiedades medidas. Esta es la función del *reconocimiento estadístico de patrones*.

2.2.3.3.3.5 - Conclusiones

Cuando la segmentación de componentes es realizada sobre una situación simple (simple significa que los componentes están separados entre sí y que hay pocos componentes) entonces la segmentación de signaturas es la técnica a elegir debido a que posee la más rápida de las implementaciones entre todas las técnicas de análisis de componentes, tanto utilizando hardware de pipeline especial como hardware de computadoras comunes.

Sin embargo, cuando hay muchas componentes y las mismas están cerca unas de otras con protuberancias que encajan en huecos de las otras la segmentación de signaturas no funcionará. En este caso se debe realizar un análisis de componentes conectadas.

2.2.3.3.4 - Análisis de regiones

Consiste en el cálculo de las propiedades globales para cada región producida por el algoritmo del labeling de componentes conectadas, o cada segmento producido por la segmentación de signaturas.

Las propiedades de cada sector o segmento son almacenadas en un vector de mediciones que es la entrada para un clasificador.

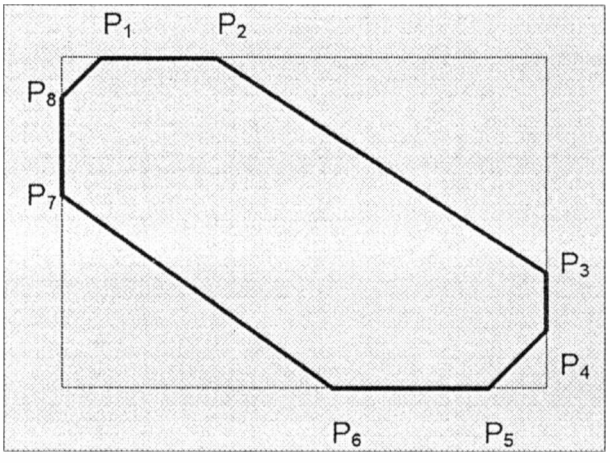
2.2.3.3.4.1 - Propiedades de las regiones

El operador de componentes conectadas produce regiones. Hay una gran variedad de propiedades medibles que pueden ser hechas en cada región en base a la forma de la misma y de los valores del nivel de gris (VNG) para aquellos pixels que la forman.

Algunas de estas propiedades son:

- El histograma de los VNG de la región: como ya se explicó, todos los pixels en una imagen dan lugar al histograma de la imagen; de aquí se desprende que los VNG para todos los pixels en una región dan origen al histograma de los VNG de la región. Esto significa que el VNG es solamente una estadística sumatoria del histograma.
- La varianza, la asimetría y la kurtosis son otras estadísticas del nivel de gris de las regiones.
- La medida de co-ocurrencia del nivel de gris de la distribución espacial en las regiones constituye una estadística sumatoria sobre la microtextura de las regiones.
- El nivel de gris espacial de segundo momento: puede medir el grado con el cuál la región es sombreada, con un lado ligeramente mas brillante que el otro.
- El rectángulo frontera de la región es el rectángulo más pequeño (con los lados orientados paralelos a los ejes fila y columna de la imagen) que la contiene o circunscribe.
- Los puntos extremos; la región tiene ocho puntos extremos: el más arriba a la izquierda (P_1), el más arriba a la derecha (P_2), el más a la derecha de arriba (P_3), el más a la derecha de abajo (P_4), el más abajo a la derecha (P_5), el más abajo a la izquierda (P_6), el más a la izquierda de abajo (P_7), el más a la izquierda de arriba (P_8).

Cada punto extremo se apoya en el rectángulo límite de la región, normalmente orientado, como se puede apreciar en la siguiente figura:



Los puntos extremos se presentan en pares opuestos. Cada par de puntos extremos opuestos definen un eje, el cual tiene propiedades muy útiles como la longitud del eje y su orientación.

- el área: si denotamos el conjunto de pixels de la región como R, se define:

$$\text{Área}: A = \sum_{(r,c) \in R} 1$$

- la longitud del perímetro P de una región sin agujeros es la secuencia de sus pixels del borde interior. Un pixel forma parte del borde si tiene un pixel vecino que no pertenece a la región.

$$\text{Perímetro:} \quad P_4 = \{ (r,c) \in R \mid \text{Vecinos}_4 (r,c) - R \neq \emptyset \} \quad \text{(con 4-conectividad)}$$

$$P_8 = \{ (r,c) \in R \mid \text{Vecinos}_8 (r,c) - R \neq \emptyset \} \quad \text{(con 8-conectividad)}$$

- centroide de una región:

$$\text{Centroide } (\bar{r}\ \bar{c}) \quad \bar{r} = 1/A \cdot \sum_{(r,c) \in R} r$$

$$\bar{c} = 1/A \cdot \sum_{(r,c) \in R} c$$

- el número de agujeros, la longitud del perímetro, la longitud del mayor y menor eje de la elipse que mejor encaje y la orientación de los ejes mayores.

- el segundo momento central, el radio del círculo que la circunscribe, el radio del máximo círculo contenido, la medida de distancia μ_R desde el centroide hasta el límite de la figura y la desviación estándar s_R de las distancias desde el centroide hasta el límite de la figura, son otras propiedad medible para cada región. Las propiedades μ_R y s_R pueden ser definidas en términos de los pixels (r_k,c_k), con K = 0,...,K-1 en el perímetro P:

$$\mu_R = 1/K \cdot \sum_{k=0}^{k-1} \| (r_k, c_k) - (\bar{r}, \bar{c}) \|$$

$$s_R^2 = 1/K \cdot \sum_{k=0}^{k-1} [\| (r_k, c_k) - (\bar{r}, \bar{c}) \| - \mu_R]^2$$

Haralick (1974) muestra que μ_R / s_R tiene las siguientes propiedades:

- Cuanto mas circular es la figura digital, la medida μ_R / s_R se incrementa monótonamente.
- Los valores de μ_R/s_R para figuras digitales similares y continuas es similar.
- La orientación y el área son independientes.

Además, el número de lados N de un polígono regular digital puede ser estimado a partir de la medida μ_R/s_R :

$$N = 1.4111(\mu_R/s_R)^{.4724}$$

Podemos determinar para cada región R la medida del nivel de gris μ y la varianza del nivel de gris s^2 . μ es una medida de primer orden que determina el promedio del nivel de gris, y se define como:

$$\mu = 1 / A \cdot \sum_{(r,c)\ e\ R} I(r,c)$$

y s^2 es una propiedad de segundo orden que se calcula como:

$$s^2 = 1 / A \sum_{(r,c)\ e\ R} [I(r,c) - \mu]^2 = [1 / A \cdot \sum_{(r,c)\ e\ R} I(r,c)^2] - \mu^2$$

2.2.3.3.4.2 - Propiedades de los segmentos

Las propiedades que se obtienen desde las proyecciones vertical, horizontal y diagonal incluyen área, centroide de la región, segundo momento y rectángulo límite. Estas propiedades, que se calculan mediante el empleo del análisis de signatura, son muy utilizadas. para determinar la orientación y posición de un rectángulo y la posición de un círculo.

La proyección vertical P_V es definida por:

$$P_V(c) = \#\{r \mid (r,c)\ e\ R\}$$

La proyección horizontal P_H es definida por:

$$P_H(r) = \#\{c \mid (r,c) \in R\}$$

La proyección diagonal P_D que va desde abajo a la izquierda hacia arriba a la derecha es definida por:

$$P_D(d) = \#\{(r,c) \in R \mid r + c = d\}$$

La proyección diagonal P_E que va desde arriba a la izquierda hacia abajo a la derecha es definida por:

$$P_E(e) = \#\{(r,c) \in R \mid r - c = e\}$$

El área A puede ser obtenida desde cualquier proyección, como por ejemplo:

$$A = \sum_{(r,c) \in R} 1 = \sum_{r} \sum_{\{c \mid (r,c) \in R\}} 1 = \sum_{r} P_H(r)$$

La fila superior (rmin) del rectángulo límite está dada por:

$$rmin = \min\{r \mid (r,c) \in R\} = \min\{r \mid P_H(r) <> 0\}$$

La fila inferior (rmax) del rectángulo límite está dada por:

$$rmax = \max\{r \mid (r,c) \in R\} = \max\{r \mid P_H(r) <> 0\}$$

La columna de mas a la izquierda (cmin) del rectángulo límite está dada por:

$$cmin = \min\{c \mid (r,c) \in R\} = \min\{c \mid P_V(c) <> 0\}$$

La columna de mas a la derecha (cmax) del rectángulo límite está dada por:

$$cmax = \max\{ c \mid (r,c) \in R \} = \max\{ c \mid P_V(c) <> 0 \}$$

La fila centroide $\bar{r}$ puede ser obtenida desde la proyección horizontal P_H realizando el siguiente cálculo:

$$\bar{r} = 1/A \cdot \sum_{(r,c) \in R} r = 1/A \cdot \sum_{r} \sum_{\{c \mid (r,c) \in R\}} r =$$

$$= 1/A \cdot \sum_{r} r \sum_{\{c \mid (r,c) \in R\}} 1 = 1/A \sum_{r} r P_H(r)$$

La columna centroide $\bar{c}$ puede ser obtenida desde la proyección vertical P_V realizando el siguiente cálculo:

$$\bar{c} = 1/A \cdot \sum_{(r,c) \in R} c = 1/A \cdot \sum_{c} \sum_{\{r \mid (r,c) \in R\}} c =$$

$$= 1/A \cdot \sum_{c} c \sum_{\{r \mid (r,c) \in R\}} 1 = 1/A \sum_{c} c P_V(c)$$

El diagonal centroide $\bar{d}$ puede ser obtenida desde la proyección diagonal P_D:

$$\bar{d} = 1/A \cdot \sum_{d} d P_D(d)$$

El diagonal centroide $\bar{e}$ puede ser obtenida desde la proyección diagonal P_E:

$$\bar{e} = 1/A \cdot \sum_{e} e P_E(e)$$

El diagonal centroide $\bar{d}$ está relacionada con la fila y columna centroide:

$$\bar{d} = 1/A \cdot \sum_{d} d \sum_{\{(r,c)\in R \mid r+c=d\}} 1 =$$

$$= 1/A \cdot \sum_{d} \sum_{\{(r,c)\in R \mid r+c=d\}} (r+c) =$$

$$= 1/A \cdot \sum_{d} \sum_{\{(r,c)\in R \mid r+c=d\}} r + 1/A \cdot \sum_{d} \sum_{\{(r,c)\in R \mid r+c=d\}} c =$$

$$= 1/A \cdot \sum_{(r,c)\in R} r + 1/A \cdot \sum_{(r,c)\in R} c = \bar{r} + \bar{c}$$

De manera similar, el diagonal centroide $\bar{e}$ está relacionada a la fila y columna centroide:

$$\bar{e} = \bar{r} - \bar{c}$$

Capitulo III

paralelos

Los factores que aumentan la pérformance de un algoritmo en una arquitectura particular dependen del grado de paralelismo y de la sobrecarga por scheduling y sincronización de tareas. La elección de un algoritmo para solucionar un problema particular es fuertemente influenciada por la arquitectura de hardware y las herramientas de software disponibles.

3.1 - Introducción

Para el paralelismo a nivel de tareas, cuando más fino es el grano de paralelismo mayor es el costo de scheduling y sincronización [Según el criterio presentado en Dennis 80, Arvind 83 y Gurd 85, los algoritmos pueden ser descriptos como evaluación de expresiones en un sistema *Data Flow*, los que tienen poca sobrecarga de scheduling y sincronización. Por otro lado, Gottieb 83 y Gajski 84 creen que lo importante es tener datos estructurados (Computación Orientada a Objetos)]. Esto significa que usualmente no se puede conseguir un incremento lineal en la velocidad por un incremento del número de procesadores.

Para encontrar el algoritmo más eficiente en una arquitectura dada, el programador deberá tener en cuenta los siguientes principios:

- Debe desarrollar circuitos rápidos y aumentarlos con metodologías para sincronización de procesos paralelos. Esto permite utilizar arquitecturas de computadores comunes y por lo tanto conservar la gran cantidad de algoritmos y programas que han sido desarrollados.
- Tiene que proveer mejores optimizaciones y compiladores vectoriales, para permitir que los procesadores paralelos sean mejor explotados.
- Debe desarrollar nuevos algoritmos mejor soportados por los nuevos lenguajes.
- Debe desarrollar nuevos modelos de computación que permitan un paralelismo masivo, lo que puede ser explotado por la construcción de grandes arquitecturas multiprocesador.

3.2- Paradigmas paralelos

Al desarrollar un algoritmo el paralelismo se puede implementar de tres maneras diferentes, a saber:

1. *Paralelismo de datos (SIMD).*
2. *Paralelismo de tareas (MIMD).*
3. *Modelos sistólicos o en pipeline.*

Frecuentemente, los algoritmos son desarrollados sin referencias a una arquitectura particular, y usan mas de un modelo de paralelismo, por lo que son difíciles de implementar en la práctica.

Para la elección del modelo de paralelismo a utilizar, se debe tener en cuenta que no cualquier tipo de paralelismo resolverá un problema dado en forma eficiente.

Aunque existe un número considerable de computadores paralelos que han sido desarrollados tanto comercialmente como en universidades, todavía hay poco software diseñado para ellos.

La mayoría de éstas maquinas son entregadas solamente con un compilador para un lenguaje convencional que ha sido extendido para posibilitar el modelo de computación paralela (pasaje de mensajes y sincronización entre procesadores).

3.2.1 - Paralelismo de datos

El paradigma de paralelismo de datos es explícitamente sincronizado y se mapea al modelo de programación SIMD, donde cada uno de los procesadores ejecuta el mismo código sobre sus datos locales en forma simultánea.

Las ventajas de este modelo radican en que el flujo de control es muy simple y que el conjunto de datos cambia de un estado a otro. Además los resultados son determinísticos e independientes del número de procesadores físicos del sistema, lo que facilita la depuración y la portabilidad del software.

Aunque el modelo es óptimo por ejemplo para el procesamiento de imágenes a bajo nivel, no es obvio como representar o implementar tareas de razonamiento de alto nivel. Hay que tener en cuenta que el particionamiento de la imagen entre el conjunto de procesadores disponibles depende de varios factores, como por ejemplo la facilidad para particionarla según el modo en que este almacenada y según la operación que se desee realizar sobre la misma.

3.2.2 - Paralelismo de tareas

Programar en una computadora multiprocesador es muy distinto a programar en una computadora con un solo procesador. Una computadora multiprocesador requiere:

- El algoritmo necesita ser particionado en subtareas;
- Las subtareas y los datos deben ser distribuidos entre los procesadores;
- El sistema debe estar configurado para permitir comunicación y sincronización entre procesadores.

El orden de las pautas antes descriptas debe ser mantenido pues no se pueden determinar las necesidades de comunicación antes de decidir la división de tareas y datos entre los procesadores.

3.2.3 - Modelos sistólicos o en pipeline

El algoritmo es particionado temporalmente y cada etapa computa un resultado parcial y se lo pasa a la siguiente etapa.

3.3 - Algoritmos de propósito general

A continuación se detallan dos algoritmos de uso general en el desarrollo de aplicaciones paralelas:

```
PROCESO MAESTRO;
Begin
  for cada imagen do
```

```
    PAR
      SEQ
        Obtener y digitalizar la Imagen
        Hacer un Threshold de la Imagen
        Eliminar el ruido
        SEQ for cada objeto
          encontrar objetos separados
          enviar objetos a proc. disponibles
        colectar resultados de Proc. Clientes
    END PAR
  end for
End
```

```
PROCESO CLIENTE;
Begin
  SEQ
    Recibir los Datos del Objeto
    Determinar el Borde del Objeto
    Analizar el Borde para reconocer el Objeto
    Enviar los resultados al Proceso Maestro
End
```

3.4 - Lenguajes paralelos

Diseñar un *'lenguaje para el procesamiento de imágenes'* no difiere demasiado del diseño de cualquier otro tipo de lenguaje de programación. Debemos considerar:

- Las estructuras de datos, los tipos de datos y especialmente el manejo de vectores y matrices, su descripción y representación.
- Las operaciones sobre los datos y la manipulación de los mismo, especialmente de las matrices.
- La estructura de programa, el flujo de control y la sintaxis.

Para que un lenguaje sea considerado paralelo debe soportar alguno de los conceptos de paralelismo y comunicación. El paralelismo puede ser a nivel de procesos (ej. ADA), objetos (ej. EMERAL), sentencias (ej. OCCAM), expresiones (lenguajes funcionales) o cláusulas (ej. PARLOG, Concurrent PROLOG). La comunicación puede ser punto a punto (ej. OCCAM) o broadcast.

Pocos de los algoritmos existentes han sido escritos en un lenguaje paralelo, ni siquiera los ejecutados en supercomputadores. Esto se debe principalmente a que los continuos incrementos en el poder de procesamiento de los procesadores seriales convencionales y las mejoras logradas en la tecnología de compiladores desalientan el cambio a una implementación paralela. Una alternativa de solución que posibilite la utilización de lenguajes paralelos sería la traducción automática del código convencional.

3.5 - Conclusiones

Para hacer efectivo el uso de multicomputadores los algoritmos tienen que ser particionados en subtareas y mapeados sobre los procesadores disponibles. También es necesario configurar la red de comunicación entre los procesadores para una topología de red específica.

Existen pocas herramientas que permitan realizar estas tareas automáticamente. A causa de ésto, la eficiencia de los programas creados es proporcional a la habilidad y al 'craft' del programador.

Un algoritmo escrito para una máquina particular en un lenguaje dado puede necesitar ser re-implementado si la topología de la red de procesadores cambia, por ejemplo agregando mas procesadores.

Escribir programas para datos paralelos en un arreglo de procesadores SIMD generalmente es simple debido a que involucra control secuencial. En cambio, es más difícil escribir programas para procesadores MIMD porque uno debe darse cuenta de la necesidad de particionar tanto los datos como el código a través de los procesadores, y también es necesario configurar los canales de comunicación entre los mismos. Esto lleva a una estructura de control compleja, complicando la tarea de debugging.

3.6 - Parallel Virtual Machine (P.V.M.)

Es un conjunto integrado de herramientas de software y librerías que emulan en forma flexible un framework para computaciones concurrentes heterogéneas de propósito general, sobre computadoras interconectadas de diversas arquitecturas. El objetivo principal del sistema PVM es permitir que computaciones concurrentes o paralelas sean ejecutadas sobre una colección de computadores de éste tipo.

3.6.1 - Principios del PVM

Los principios en los que se basa PVM son los siguientes:

- ***Pool de computadoras configurado por el usuario:*** las tareas computacionales de la aplicación se ejecutan sobre un conjunto de máquinas seleccionadas por el usuario para una instancia en particular del programa PVM. Tanto máquinas con una sola CPU como hardware multiprocesador (incluyendo computadoras con memoria compartida y memoria distribuída) pueden ser parte del pool mencionado. Este puede ser alterado agregando o quitando máquinas durante la operación (lo que una característica importante para la tolerancia a fallas).

- ***Acceso al hardware transparente:*** los programas de aplicación o bien pueden ver el medio ambiente de hardware como una colección de elementos de procesamiento virtuales o bien pueden elegir explotar las capacidades de máquinas específicas en el pool de computadoras posicionando ciertas tareas computacionales en las computadoras más apropiadas.

- ***Computación basada en procesos:*** la unidad de paralelismo en PVM es la tarea (a menudo, pero no siempre, un proceso UNIX), la cuál consta de una secuencia independiente de control que alternan entre comunicación y computación. PVM no emplea un mapeo de proceso a procesador. En particular, múltiples tareas pueden ejecutarse en un solo procesador.

- ***Modelo de pasaje de mensajes explícito:*** colecciones de tareas computacionales, cada una llevando a cabo una parte de la aplicación usando tanto descomposición de datos, funcional o híbrida, cooperan enviando y recibiendo mensajes explícitos la una a la otra. El tamaño de los mensajes está limitado solamente por la cantidad de memoria disponible.

- ***Soporta heterogeneidad:*** el sistema PVM soporta heterogeneidad en término de máquinas, redes y aplicaciones. Respecto del pasaje de mensajes, PVM permite que mensajes que contengan más de un tipo de datos puedan ser intercambiados entre máquinas que tengan diferentes representaciones de datos.

- ***Soporte de multiprocesadores:*** PVM usa las facilidades nativas para el pasaje de mensajes en multiprocesadores para sacar ventaja del hardware específico de los mismos. Los fabricantes frecuentemente proveen un PVM propio optimizado para sus sistemas, el cuál puede también comunicarse con la versión pública de PVM.

3.6.2 – Descripción del entorno

El sistema PVM esta compuesto de dos partes. La primera es un daemon, llamado *pvmd3* el cuál reside en todas las computadoras que componen la máquina virtual. Pvmd3 está diseñado de tal modo que cualquier usuario con un login válido puede instalar el daemon en una máquina. Cuando el usuario desea ejecutar una aplicación PVM, lo primero que debe hacer es crear una máquina virtual arrancando el PVM (ejecutando el programa 'pvm'). La aplicación PVM puede ser arrancada desde un prompt de Unix en cualquiera de los hosts. Múltiples usuarios pueden configurar máquinas virtuales superpuestas, y cada usuario puede ejecutar varias aplicaciones PVM simultáneamente.

La segunda parte del sistema es una librería de rutinas de interface PVM. Esta contiene un repertorio completamente funcional de las primitivas que son necesarias para implementar la cooperación de tareas de una aplicación. Dicha librería contiene funciones para pasaje de mensajes, ejecución de procesos, coordinación de tareas y también permiten modificaciones en la maquina virtual.

3.6.3 - Modelo computacional

El modelo computacional de PVM esta basado en la noción de que una aplicación en particular consiste de varias tareas. Cada una de estas tareas es responsable de una parte del trabajo computacional de la aplicación. En ciertas ocasiones la paralelización

es sobre las funciones de una aplicación, esto significa que cada tarea lleva a cabo funciones diferentes (ej. entrada de datos, configuracion del problema, solucion, salida de datos y muestra de los resultados). Este proceso es llamado ***paralelismo funcional***.

Mas frecuentemente, se utiliza el llamado ***paralelismo de datos***. En este método todas las tareas son iguales pero cada una solo conoce y procesa una parte pequeña de los datos. Este paradigma se denomina SPMD (single-program multiple-data). PVM soporta ambos métodos (incluso mezclados). Dependiendo de las funciones, las tareas pueden ejecutarse en paralelo y pueden necesitar sincronizare e intercambiar datos, aunque no siempre es el caso.

3.6.4 - Lenguajes soportados

El sistema PVM actualmente soporta los lenguajes C, C++ y Fortran. Estos lenguajes se han elegido como consecuencia de que la mayoria de las aplicaciones se escriben usando C o Fortran y que estan emergiendo nuevas aplicaciones basadas en lenguajes y metodologias orientados a objetos.

La interface de la libreria de usuario PVM para C y C++ esta implementada como funciones que respetan las convenciones usadas por la majoria de los sistemas en C, incluyendo a los sistemas operativos Unix-like. Para que los programas C y C++ pueden acceder a estas funciones solamente es necesario enlazarlos (linkearlos) con la libreria de pvm (livpvm3.a).

La interface con Fortran esta implementada mediante subrutinas en vez de funciones.

3.6.5 - Tareas en PVM

Todas las tareas PVM se identifican mediante un numero entero llamado ***identificador de tarea*** (TID Task Identifier). Los mensajes son enviados y recibidos desde tids. Como los tids deben ser únicos en toda la máquina virtual, son provistos por el pvmd local y no elegidos por el usuario. Aunque PVM codifica información en cada TID, se espera que el usuario trate a cada tid como un identificador entero sin ningún significado en especial. PVM provee varias rutinas que devuelven tids de modo que una aplicación pueda identificar a otras tareas en el sistema.

En ciertas aplicaciones es natural pensar en un grupo de tareas. Y también hay casos en los cuales un usuario puede desear identificar a sus tareas usando números consecutivos, **0**, **1**, ..., **p-1** donde **p** es el número de tareas. PVM incluye el concepto de grupos etiquetados por el usuario. En el momento en que una tarea se une a un grupo, se le asigna un único número de instancia dicho grupo. Estos números de instancia comienzan en 0 y se van incrementando. Manteniendo la consistencia de PVM, las funciones para manejo de grupos fueron diseñadas de modo que sean generales y transparentes para el usuario. Por ejemplo, cualquier tarea PVM puede unirse o separarse de cualquier grupo en cualquier momento sin tener necesidad de informar a las demás tareas del grupo. También es posible que los grupos se solapen y que las tareas envíen mensajes en broadcast a grupos a los cuales no pertenecen.

3.6.6 - Paradigmas de programación

El paradigma general de programación de una aplicación con PVM es el siguiente: Un usuario escribe uno o más programas secuenciales en C, C++ o Fortran 77 que contienen llamadas a la librería PVM. Cada programa es considerado una tarea que forma parte de la aplicación. Estos programas son compilados para cada arquitectura que intervenga en el pool de hosts que compone la máquina virtual, y los archivos objeto resultantes se colocan en lugares que sean accesibles para las máquinas del pool. Para ejecutar la aplicación, por lo general el usuario ejecuta una instancia de una tarea (generalmente llamada maestra) desde una maquina incluida en el pool. Este proceso ejecutado ira ejecutando a su vez otras tareas PVM, resultando al final un conjunto de tareas que realizan cálculos locales e intercambian mensajes con otras tareas para resolver el problema dado.

3.6.7 - Ejemplo de programa PVM

Un ejemplo simple que ilustra los conceptos básicos de PVM seria el siguiente:

Tarea Maestra: (hello.c)

```
#include "pvm3.h"

main()
{
    int cc, tid, msgtag;
    char buf[100];

    printf("i'm t%x\n", pvm_mytid());
    cc = pvm_spawn("hello_other",(char**)0,0,"",1,&tid);

    if (cc == 1) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
        pvm_upkstr(buf);
        printf("from t%x: %s\n", tid, buf);
    } else
        printf("can't start hello_other\n");

    pvm_exit();
}
```

Este programa esta pensado para ser ejecutado en forma manual. Lo primero que hace es imprimir su TID (obtenido mediante la función pvm_mytid()), luego ejecuta una copia de otro programa llamado hello_other mediante la función pvm_spawn().

Si la ejecución de este otro programa resulta exitosa, el programa ejecuta una recepción bloqueante (sincrónica) de un mensaje mediante la función pvm_recv. Luego de recibir el mensaje, este se imprime y también se imprime el tid del proceso que lo envío. El mensaje se extrae del buffer de recepción mediante la función pvm_upkstr. La llamada final a pvm_exit rompe el nexo entre el programa y el sistema PVM.

Tarea Cliente: (hello_other.c)

```
#include "pvm3.h"

main()
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);

    pvm_exit();
}
```

Este es el listado del programa cliente que es ejecutado por hello.c.

Lo primero que hace es obtener el tid del programa que lo ejecuto mediante la función pvm_parent. Después obtiene el nombre del host donde se esta ejecutando y lo transmite al programa maestro (hello) mediante las llamadas a las funciones pvm_initsend, que inicializa el buffer de transmisión, pvm_pkstr para agregar al buffer un string en formato independiente de la arquitectura, y pvm_send para realizar la transmisión al proceso destino.

Métricas

Principios de la performance escalable

La pérformance de las comput-doras paralelas se basa en el uso de un diseño que logre un balance entre el hardware disponible y el software desarrollado. Lograr ésto es una responsabilidad compartida entre los arquitectos de sistemas y los programadores.

4.1 - Métricas y medidas de performance

Primero estudiaremos perfiles de paralelismo y definiremos el factor de speedup asintótico, ignorando la latencia de comunicaciones y las limitaciones de recursos. Luego introduciremos los conceptos de eficiencia de un sistema, utilización, redundancia, y la calidad de las computaciones paralelas.

4.1.1 - Perfil de paralelismo en los programas

El grado de paralelismo refleja la forma en que el paralelismo de software coincide o utiliza el paralelismo del hardware disponible.

4.1.1.1 - Grado de paralelismo

La ejecución de un programa en una computadora paralela puede utilizar un número distinto de procesadores en períodos de tiempo diferentes durante el ciclo de ejecución. Para cada período de tiempo, el número de procesadores utilizados para ejecutar un programa es definido como el *grado de paralelismo* (GDP).

La gráfica del GDP como función del tiempo es llamada el *perfil de paralelismo* de un programa dado.

Los cambios en el perfil durante la ejecución de un programa dependen de la estructura de los algoritmos usados, la optimización de los programas, la utilización de recursos, y de condiciones de run-time de las computadoras. Sin embargo, esta definición del GDP asume que se dispone de un número ilimitado de procesadores y otros recursos necesarios, por lo que en ciertas ocasiones, el GDP no puede ser utilizado en computadoras reales con recursos limitados.

4.1.1.2 - Paralelismo promedio

Consideremos una computadora paralela compuesta por un número *n* de preocesadores homogéneos. El paralelismo máximo en un perfil es *m*. En el caso ideal esperamos que *n* >> *m*. La *capacidad computacional* Δ de un procesador es aproximada según su tasa de ejecución, en unidades como MIPS o Mflops, sin considerar el tiempo perdido en accesos a memoria, comunicación o sobrecarga del sistema. Cuando *i* procesadores están ocupados durante un período de observación, tenemos GDP = *i*.
El trabajo total (W) realizado se calcula usando la siguiente sumatoria:

$$W = \Delta \sum i * t_i \text{ para } 1 <= I <= m \qquad (4.1)$$

donde t_i es el tiempo total en que GDP = i y $\sum t_i$ para $1 <= I <= m = t_2 - t_1$ es el tiempo total observado.

El **paralelismo promedio** A se calcula de la siguiente forma

$$A = (\sum i * t_i) / (\sum t_i) \text{ para } 1 <= I <= m \qquad (4.2)$$

4.1.1.3 - Paralelismo disponible

El rango de paralelismo varia según el tipo de aplicación considerado. Los programas científicos y de ingeniería poseen un alto GDP debido al paralelismo de datos. En comparación, los programas con menor procesamiento numérico exhiben un paralelismo menor.

Estudios realizados por Manoj Kumar revelaron que en ciertas aplicaciones con computaciones intensivas se pueden realizar entre 500 a 3500 operaciones aritméticas en forma concurrente por cada ciclo de reloj en un medio ideal.

Por otro lado, estudios realizados por David Wall revelaron que el límite en el paralelismo a nivel de instrucción es de aproximadamente 5, y si se quitan todas las restricciones, el GDP en programas puede llegar a 17 instrucciones por ciclo.

4.1.1.4 - Speedup asintótico

Llamemos a la cantidad de trabajo realizada cuando el GPD = i como $W_i = i \Delta t_i$, lo que se puede escribir como $\sum W_i$ para $1 <= I <= m$. El tiempo de ejecución de W_i en un solo procesador (secuencialmente) es $t_i(1) = W_i / \Delta$. El tiempo de ejecución de W_i en k procesadores es $t_i(k) = W_i / k\Delta$. Si disponemos de un número infinito de procesadores, $t_i(\infty) = W_i / k\Delta$ para $1 <= I <= m$. Por lo tanto, podemos expresar el tiempo de respuesta como

$$T(1) = \sum t_i(1) = \sum W_i / \Delta \quad \text{con } 1 <= i <= m \qquad (4.3)$$

$$T(\infty) = \sum t_i(\infty) = \sum W_i / i\Delta \quad \text{con } 1 <= i <= m \qquad (4.4)$$

El speedup asintótico S_∞ es definido como la relación entre T(1) y $T(\infty)$:

$$S_\infty = T(1) / T(\infty) = \sum W_i / \sum (W_i / i) \quad \text{con } 1 <= i <= m \qquad (4.5)$$

Comparando las ecuaciones 4.2 y 4.5, concluimos que en el caso ideal $S_\infty = A$. En general, $S_\infty <= A$ si se tienen en cuenta la latencia de comunicaciones y otras sobrecargas del sistema. Téngase en cuenta que tanto S_∞ como A son definidas asumiendo que $n = \infty$ o $n >> m$.

4.1.2 - Performance armónica media

Considérese una computadora paralela con n procesadores ejecutando m programas en varios modos distintos con distintos niveles de performance. Queremos definir la performance media en este tipo de computadoras.

Los distintos modos de ejecución pueden corresponder a procesamiento escalar, vectorial, secuencial o paralelo en partes diferentes del programa. El significado armónico de la performance nos da una medida de la performance promedio de un gran número de programas corriendo en distintos modos.

La tasa de ejecución R_i para el programa i esta medida en MIPS o Mflops, y por lo tanto también lo están las expresiones siguientes.

4.1.2.1 - Performance aritmética media

Sea { R_i } las tasas de ejecución de los programas i = 1,2, ..., m. El significado aritmético de la performance se define como:

$$R_a = \sum R_i / m \quad \text{con } 1 <= i <= m \qquad (4.6)$$

La expresión R_a asume que los m programas tienen el mismo peso (carga computacional). Si los pesos fueran distintos, con una distribución π = { f_i para i = 1, 2, ... m } definimos una tasa de ejecución aritmética media pesada de la siguiente forma:

$$R_a^* = \sum (f_i R_i)$$

La tasa de ejecución aritmética media es proporcional a la suma de las inversas del los tiempos de ejecución.

4.1.2.2 - Performance geométrica media

Se define como:

$$R_g = \prod R_i^{1/m} \quad \text{con } 1 <= I <= m$$

Si tenemos una distribución pesada π = { f_i | i para i = 1, 2, ... m } definimos una tasa de ejecución geométrica media pesada de la siguiente forma:

$$R_g^* = \prod R_i^{f_i} \quad \text{con } 1 <= I <= m$$

La tasa de ejecución geométrica media ha sido definida para compararse con números normalizados que representan esta performance obtenidos en máquinas tomadas como referencia.

4.1.2.3 -Performance armónica media

Para resolver las carencias de los procedimientos anteriores, necesitamos definir una expresión de la performance basada en el tiempo de ejecución aritmético promedio. Dado que $T_i = 1/R_i$ es el tiempo de ejecución medio por instrucción para el programa i, definimos el tiempo de ejecución medio por instrucción como:

$$T_a = (1/m) * \sum T_i = (1/m) * \sum (1/R_i) \quad \text{con } 1 <= i <= m$$

La tasa de ejecución media armónica tomada de m programas de benchmarks se define tomando en cuenta el hecho de que $R_h = 1/T_a$:

$$R_h = m / \sum (1/R_i) \quad \text{con } 1 <= i <= m$$

También podemos definir esta tasa para el caso en el que los pesos son distintos:

$$R_h^* = 1 / \sum (f_i/R_i) \quad \text{con } 1 <= i <= m$$

4.1.3 - Eficiencia, utilización y calidad

Hay diversos parámetros definidos para evaluar a las computaciones paralelas, los cuales son conceptos fundamentales en el procesamiento paralelo ya que aparecen en las aplicaciones reales.

4.1.3.1 - Eficiencia del sistema

Sea O(n) el número total de operaciones unitarias llevadas a cabo por un sistema con n procesadores y sea T(n) el tiempo de ejecución medido en unidades de tiempo. En general T(n) < O(n) si los n procesadores pueden realizar más de una operación por unidad de tiempo, siempre que n >= 2. Asumimos que T(1) = O(1) en un sistema uniprocesador.

Definimos el *factor de aceleración* (speedup):

$$S(n) = T(1) / T(n)$$

La *eficiencia del sistema* para un sistema n-procesador:

$$E(n) = S(n) / n = T(1) / (n\ T(n))$$

La eficiencia es un indicador del grado real de speedup alcanzado comparada con el mayor valor posible para la misma. Dado que 1 <= S(n) <= n, tenemos que 1/n <= E(n) <= 1.

La menor eficiencia corresponde con el caso en que el programa entero se ejecuta secuencialmente en un solo procesador. La eficiencia máxima corresponde al caso en que todos los n-procesadores se están utilizando en su máxima capacidad durante todo el periodo de ejecución.

4.1.3.1.1 - Redundancia y utilización

La redundancia de una computación paralela se define como la relación entre O(n) y O(1)

$$R(n) = O(n) / O(1)$$

Esta relación indica la concordancia entre el paralelismo de software y el paralelismo de hardware. Obviamente, 1 <= R(n) <= n. La *utilización del sistema* en una computación paralela de define como:

$$U(n) = R(n)\ E(n) = O(n) / (n * T(n))$$

La utilización del sistema indica el porcentaje de recursos (procesadores, memoria, etc.) que están ocupados durante la ejecución del programa paralelo. Es interesante notar las siguientes relaciones: 1/n <= E(n) <= U(n) <= 1 y también 1 <= R(n) <= 1/E(n) <= n.

4.1.3.1.2 - Calidad del paralelismo

La calidad de una computación paralela es directamente proporcional al speedup y a la eficiencia e inversamente proporcional a la redundancia. Por lo tanto, tenemos:

$$Q(n) = (S(n) E(n)) / R(n) = T^3(1) / (n T^2(n) O(n))$$

Dado que E(n) es siempre una fracción y R(n) es un número entre 1 y n, la calidad Q(n) tiene como límite superior al speedup.

4.2 - Conclusiones

Para resumir el significado de los índices mencionados, decimos que el speedup S(n) indica la velocidad ganada en las computaciones paralelas. La eficiencia E(n) mide la porción útil del trabajo total realizado por n procesadores. La redundancia R(n) mide el incremento de la sobrecarga total. La utilización U(n) indica es uso de recursos durante la computación paralela. Finalmente, la calidad Q(n) combina los efectos del speedup, eficiencia y la redundancia en una sola expresión.

Capitulo V

Solución

El objetivo de nuestro trabajo es automatizar la clasificación de objetos utilizando algoritmos distribuidos.

5.1 – Introducción

Con el fin de abocarnos específicamente al desarrollo de algoritmos de reconocimiento distribuidos, trabajamos sobre imágenes True Color bidimensionales de los objetos en cuestión, dejando para una futura extensión los pasos de adquisición de las mismas por medio de video cámaras.

Para obtener las imágenes con las que trabajamos, en un principio fotografiamos huevos de gallina sobre un fondo oscuro con luz natural. Una vez reveladas las fotos, las escaneamos usando un scanner Genius 9000 con una resolución de 300 DPI.

Al efectuar pruebas con las imágenes escaneadas, llegamos a la conclusión de que la iluminación natural provocaba reflejos y sombras sobre la superficie del huevo, lo que impedía la determinación exacta de su color.

En una segunda etapa construimos un set con iluminación artificial desde distintos ángulos, y las fotografías obtenidas mejoraron sensiblemente. Obtuvimos también fotografías de frutas (limones, naranjas y tomates) y de cerámicas (mosaicos y azulejos) como se pueden ver en la figura 5.1.

Las imágenes escaneadas fueron almacenadas en formato JPEG (.jpg) para reducir el espacio de almacenamiento.

5.2 – Implementación secuencial

Luego de estudiar las metodologías de reconocimiento de objetos, descriptas en el capítulo II, decidimos implementar primero los algoritmos en forma secuencial, para facilitar su desarrollo y verificación.

5.2.1 – Threshold

El primer algoritmo a implementar fue el Threshold. Este sería utilizado para diferenciar el fondo de la imagen de los objetos en cuestión.

La idea del algoritmo es muy simple y consiste en recorrer cada pixel de la imagen cambiándolo por el valor 255 (blanco) si su valor se encuentra por encima del valor preestablecido de threshold (VDT), o por el valor 0 (negro) en caso contrario. A continuación detallamos en seudocódigo este algoritmo:

```
Procedure Threshold;
   Para c/pixel de la imagen
      Si el valor del pixel es >= que VDT
         valor de pixel = 255
      Sino
         valor de pixel = 0
      Fin Si
   Fin Para
Fin Threshold;
```

izquierda resolvía las equivalencias que se generaban en esta línea solamente mirando el pixel que quedaba a la derecha.
El seudocódigo de este algoritmo es el siguiente:

```
Procedure Labeling_V1
   // La primera línea es un caso especial ( no tiene arriba )
   Para c/pixel de la línea  que no es fondo // distinto de negro
      si el pixel anterior tiene etiqueta
         etiqueta = etiqueta del pixel anterior
      si no
         etiqueta = etiqueta nueva
      fin si
   fin para

   // Ahora procesamos el resto de la imagen
   Para cada línea del resto de la imagen
      // Primero de izquierda a derecha
      Para c/pixel de la línea que no es fondo // distinto de negro
         si pixel de arriba y el pixel de la izquierda están etiquetados
            etiqueta = mín ( etiqueta de arriba, etiqueta de la izquierda)
         sino si el pixel de la izquierda tiene etiqueta
            etiqueta = etiqueta de la izquierda
         sino si el pixel de arriba tiene etiqueta
            etiqueta = etiqueta de arriba
         sino
            etiqueta = etiqueta nueva
         fin si
      fin para

      // Ahora de derecha a izquierda
      Para c/pixel de la línea que no es fondo // distinto de negro
         si el pixel de la izquierda tiene etiqueta
            etiqueta = etiqueta de la izquierda
         fin si
      fin para
   fin para
fin Labeling_V1
```

En las figuras 5.2 y 5.3 se puede apreciar el funcionamiento de este algoritmo y los resultados obtenidos.

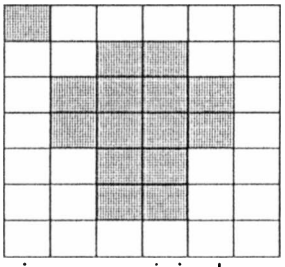


imagen original

1	0	0	0	0	0
0	0	1	1	0	0
0	1	1	1	1	0
0	1	1	1	1	0
0	0	1	1	0	0
0	0	1	1	0	0
0	0	0	0	0	0

Result. thresholding

1	0	0	0	0	0
0	0	2	2	0	0
0	2	2	2	2	0
0	2	2	2	2	0
0	0	2	2	0	0
0	0	2	2	0	0
0	0	0	0	0	0

Resultado labeling

Este algoritmo funcionaba muy bien con objetos con figuras de forma circular o poligonal pero pronto advertimos que no funcionaba con regiones del tipo de herraduras con la abertura en la parte superior, como por ejemplo una letra 'U', pues las equivalencias que se detectaban en una línea no se propagaban hacia las líneas superiores, como se muestra en las siguientes figuras:

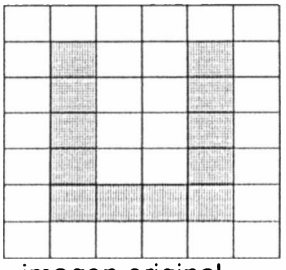


imagen original

0	0	0	0	0	0
0	1	0	0	1	0
0	1	0	0	1	0
0	1	0	0	1	0
0	1	0	0	1	0
0	1	1	1	1	0
0	0	0	0	0	0

result.del threshold

0	0	0	0	0	0
0	1	0	0	2	0
0	1	0	0	2	0
0	1	0	0	2	0
0	1	0	0	2	0
0	1	1	1	1	0
0	0	0	0	0	0

result. del labeling

5.2.3 – Labeling (segunda versión)

Para resolver los problemas que se presentaron en la primera versión del labeling decidimos realizar una nueva implementación que realizara una primera pasada sobre la imagen, línea a línea y en un único sentido.

Las equivalencias que se van presentando se almacenan en una tabla de equivalencias que se va generando durante la primera pasada. Luego realizamos una segunda pasada en el mismo sentido que la anterior (aunque esto no es importante) durante la cual, para cada pixel, se cambia su etiqueta por la que le corresponda a la etiqueta que tenia en la tabla de equivalencias, si es que la tuviera.

El problema que presenta este algoritmo es el mantenimiento de la tabla, ya que al ir agregando equivalencias se genera un grafo que determina las equivalencias de cada etiqueta. Para conocer estas equivalencias es necesario realizar la clausura transitiva de dicho grafo, lo que no resulta una tarea trivial y dependiendo de la forma de la imagen se puede llegar a degradar la performance del algoritmo.

```
Procedure Labeling_V2;
  // La primera línea es un caso especial ( no tiene arriba )
  Para c/pixel de la línea  que no es fondo // distinto de negro
     si el pixel anterior tiene etiqueta
        etiqueta = etiqueta del pixel anterior
     si no
        etiqueta = etiqueta nueva
     fin si
  fin para

  // Ahora procesamos el resto de la imagen
  Para cada línea del resto de la imagen
     Para c/pixel de la línea que no es fondo // distinto de negro
        si pixel de arriba y el pixel de la izquierda están etiquetados
           etiqueta = mín( etiqueta de arriba, etiqueta de la izquierda)
           si etiqueta arriba <> a la de la izquierda
             //agrego equivalencia a la tabla
             etiq. pixel equivale a max( etiqueta de arriba, etiqueta de la izquierda )
           fin si
```

```
        sino si el pixel de la izquierda tiene etiqueta
            etiqueta = etiqueta de la izquierda
        sino si el pixel de arriba tiene etiqueta
            etiqueta = etiqueta de arriba
        sino
            etiqueta = etiqueta nueva
        fin si
    fin para
fin para

// Ahora realizamos una segunda pasada resolviendo equivalencias
Para cada línea de la imagen
    Para c/pixel de la línea que no es fondo // distinto de negro
        si la etiqueta del pixel tiene equivalencia
            etiqueta = equivalencia
            // equivalencia es el resultado de la clausura transitiva
        fin si
    fin para
fin para
fin Labeling_V2
```

Sobre la base de las pruebas realizadas con este algoritmo se puede concluir que funciona correctamente para objetos de cualquier forma geométrica.

En el Apéndice B de imágenes se pueden encontrar ejemplos de la ejecución de estos algoritmos y las imágenes obtenidas.

5.2.4 - Calculo de Color Promedio

Una vez identificados los pixels que forman el objeto a clasificar (es la región de mayor tamaño resultante del labeling) se calcula el color promedio simplemente sumando los colores de estos pixels y dividiéndolo por la cantidad de pixels.

Color promedio = (Σ color de pixel) / cantidad de pixels

Cabe destacar que para llegar a un resultado satisfactorio en este cálculo es requisito indispensable que el objeto en cuestión sea mayoritariamente de un color uniforme. Además, la imagen obtenida de este objeto debe haber sido adquirida en condiciones de iluminación que no alteren las características del mismo.

5.2.5 - Identificación de Manchas

Uno de los objetivos de la aplicación es determinar la presencia de defectos en el objeto a clasificar. Para ello, desarrollamos un algoritmo que encuentra estos defectos o manchas dentro del objeto principal (identificado en el labeling antes descripto).

Este algoritmo funciona de manera similar al threshold, solo que para modificar el valor de cada pixel por 255 (blanco) o 0 (negro) no se vale de un valor de threshold sino que utiliza dos valores. Si el valor del pixel en cuestión esta dentro del rango determinado por estos dos valores, se le asigna el valor 0 (negro) ya que se trata del objeto; caso contrario de le asigna 255 (blanco) pues se trataría de una mancha.

Los dos valores mencionados se determinan sobre la base del color promedio del objeto y a un parámetro que indica la variación de color considerada como normal dentro del objeto, de manera que: valor mínimo = VMIN = color promedio - parámetro
valor mínimo = VMAX = color promedio + parámetro

```
Procedure Threshold_Especial;
  Para c/pixel de la imagen
     Si el valor del pixel es >= que VMIN y valor del pixel es <= que VMAX
        valor de pixel = 0
     Sino
        valor de pixel = 255
     Fin Si
  Fin Para
Fin Threshold;
```

Al analizar los resultados obtenidos con este algoritmo, notamos que si la imagen procesada presentaba variaciones de tono debidas a diferencias de iluminación sobre la superficie del objeto, se detectaban como manchas pixels del objeto que en realidad eran zonas poco o muy iluminadas.

Intentando solucionar estos inconvenientes, implementamos otro algoritmo que asume que el color predominante en el objeto presenta diferencias de intensidad. Por este motivo, para determinar si un pixel es o no un defecto del objeto se controla si su color es similar al color promedio encontrado, aceptando una variación lineal en las tres componentes: red, green y blue, en cuyo caso, a pesar de no ser del mismo color el pixel es considerado normal (no perteneciente a un defecto del objeto). Para lograr ésto, se realiza el siguiente cálculo (hay que tener en cuenta que estamos trabajando con imágenes en True Color (RGB)):

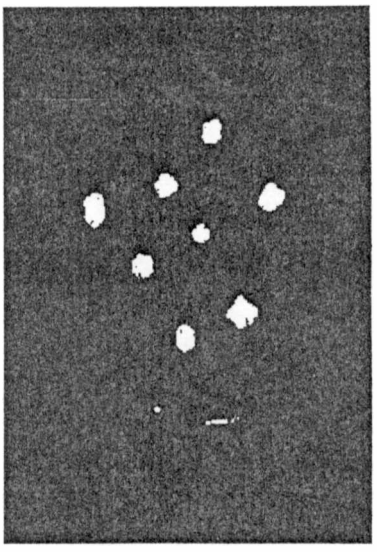
1°- Normalización de las componentes RGB del color promedio:
Red Normalizado = Red Promedio * 100 / 255
Green Normalizado = Green Promedio * 100 / 255
Blue Normalizado = Blue Promedio * 100 / 255

2°- Para cada pixel del objeto, se calcula:
Red Diferencia = (Red del pixel * 100 / 255) - Red Normalizado
Green Diferencia = (Green pixel *100 / 255) - Green Normalizado
Blue Diferencia = (Blue pixel * 100 / 255) - Blue Normalizado

3°- En el caso ideal se esperaría que los tres valores diferencia fueran equivalentes para los pixels que son del color del objeto, y distintos para los que son defectos. Como esto no se cumple en todos los casos, admitimos un margen de tolerancia en la variación entre estos valores diferencia.

Nota: Se multiplica por el factor (100/255) para llevar los colores a una escala 0-100 y poder especificar una tolerancia que signifique un porcentaje.

El resultado de este algoritmo es una imagen binaria con las manchas encontradas en el objeto en color blanco.



Al aplicarle el algoritmo de Threshold_Especial a la imagen 1 del Apéndice B se obtiene el siguiente resultado.

Sobre la imagen obtenida realizamos un labeling con el fin de calcular la cantidad de manchas o defectos y el tamaño de los mismos.

5.2.6 - Detección de bordes

Analizando el borde o perímetro de un objeto, es posible encontrar datos acerca del mismo, como por ejemplo su forma, sus diámetros, etc. Debido a esto, decidimos que sería útil incorporar en nuestra aplicación esta capacidad.

Se considera un pixel de una región como perteneciente al borde de la misma si algún pixel vecino (considerando vecinos a los pixels que están conectados al mismo por medio de conexión 4) no pertenece a la región.

A medida que se van encontrando estos pixels, sus coordenadas son guardadas en una lista para que luego puedan ser procesadas rápidamente sin necesidad de recorrer la imagen.

El algoritmo de búsqueda de bordes sería el siguiente:

```
Procedimiento Busqueda_de_borde
  Para c/pixel de la imagen
     Si el pixel pertenece a la región analizada
        Si  ( el pixel de arriba es fondo O pertenece a una región distinta
        O el pixel de abajo es fondo O pertenece a una región distinta
        O el pixel de derecho es fondo O pertenece a una región distinta
        O el pixel izquierdo es fondo O pertenece a una región distinta )
        Agrego las coordenadas del pixel a la lista
     Fin si
  Fin para
Fin Busqueda_de_borde
```

Los pixels pertenecientes a la región ubicados en el borde de la imagen son siempre considerados pertenecientes al borde.

Este algoritmo encuentra tanto bordes externos como internos en el caso de que la región presente orificios, lo que se debe tener en cuenta al utilizar las coordenadas de estos pixels para el cálculo de momentos.

5.2.7 - Determinación del centroide o centro de masa

Uno de los momentos de primer orden de una región es el *centroide* o *centro de masa* de la misma. En el caso de regiones como círculos o cuadrados, este coincide con el centro geométrico de la misma.

Nos interesamos en calcular este valor debido a que basándose en él y en el borde se pueden determinar interesantes características de los objetos y también a que si existieran en la imagen marcas para facilitar el cálculo de distancias sobre las mismas, dichas distancias están medidas desde el centro de las marcas, por lo que éste debe ser determinado. La fórmula para calcular el centroide es lo siguiente:

$$\text{Centroide de } R = C_{xy} = \underset{\text{para } (x,y) \text{ perteneciente a } R}{\text{Suma } (x, y) \,/\, \text{Area } (R)}$$

5.2.8 - Cálculo de momentos de primer orden

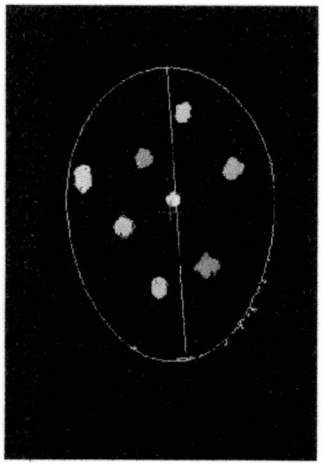
Existen dos momentos de primer orden que permiten identificar la forma general de un objeto. Estos son:

Distancia promedio desde el centro al perímetro:

$\mu_r = (1/K) * \Sigma (| (x,y) - C_{xy} |)$, para (x,y) pertenecientes al perímetro y siendo K en número de pixels del mismo.

Desviación estándar de μ_r:

$\sigma_r^2 = (1/K) * \Sigma (| (x,y) - C_{xy} | - \mu_r)^2$



Esta imagen es el resultado de la aplicación del proceso final de labeling junto con la detección del borde, centroide y perímetro de la imagen original. Puede observarse que el algoritmo pinta de distintos colores cada una de las manchas o defectos detectados.

5.3 - Paralelización de la aplicación

Para distribuir el procesamiento de las imágenes seguimos el modelo de master–slave descripto anteriormente.

Existe un proceso llamado master que es el responsable de repartir el procesamiento entre un número determinado de procesos slave o clients y de coordinar los cálculos de los mismos. Los procesos client son los que llevan a cabo el procesamiento propiamente dicho.

La paralelización es a nivel de datos, o sea que la imagen a procesar se particiona en subimágenes que son enviadas a procesos client encargados de su procesamiento. Los procesos client son idénticos entre sí, por lo que la aplicación distribuída se comporta como una máquina SIMD (Single Instruction Multiple Data).

La paralelización de algunos algoritmos no presenta complicaciones y la de otros requiere un cuidado especial que involucra la participación del master en el procesamiento para coordinar a los cients.

A continuación se analiza cada uno de los algoritmos paralelizados.

5.3.1 - Threshold

Este algoritmo no necesita ninguna consideración especial, ya que en él se analiza cada pixel de la región comparándolo con un valor límite y por lo tanto se obtienen los mismos resultados sin importar el número de divisiones que tenga la imagen procesada.

5.3.2 - Labeling

Este algoritmo presenta dificultades para paralelizarlo debido principalmente a que las regiones pueden quedar divididas en dos o más subimágenes y también debido a que las etiquetas deben ser únicas.

Para resolver estos inconvenientes decidimos dividir a las dos pasadas que se realizan en el labeling descripto anteriormente.

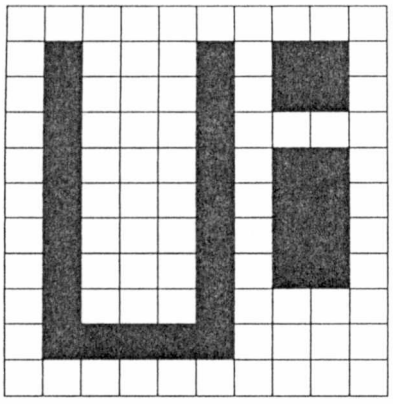
Cada client realiza la primera pasada del labeling sobre la subimagen que tiene asignada del mismo modo en que se hace en el algoritmo convencional. Luego, cada client le envía al proceso master su tabla de labeling local. Para resolver el conflicto que se genera cuando una región queda dividida en dos o más subimágenes, las líneas correspondientes al lugar en donde de divide la imagen las enviamos a dos procesos client, a uno como la última línea de una subimagen y a otro como la primera línea de esta.

Los procesos client envían también estas dos líneas especiales (que para los mismos son la primera y la última de su parte de la imagen local) con las etiquetas que les asignaron. El master concatena las tablas de labeling parciales que le envian los clients en una tabla global y simultáneamente revisa las líneas superpuestas, y en el caso en que se genere conflicto (los dos clients involucrados etiquetaron los pixels de esa línea con etiquetas distintas) agrega la equivalencia correspondiente en la tabla global.

Luego de procesar esta información, el master les envía a los clients la parte de tabla de labeling que le correspondía a cada uno, pero que ahora puede tener nuevas equivalencias. Los clients la reciben y realizan su segunda pasada del labeling basándose en esta tabla modificada.

Cuando se forma la tabla global, se soluciona también el problema de dos regiones distintas que tengan la misma etiqueta, ya que en dicha tabla se modifican las etiquetas de las tablas parciales sumándoles un número de manera que queden etiquetas distintas.

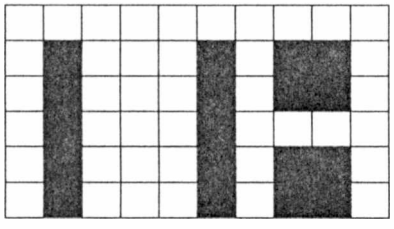
Supongamos a modo de ejemplo que contamos con la siguiente imagen:



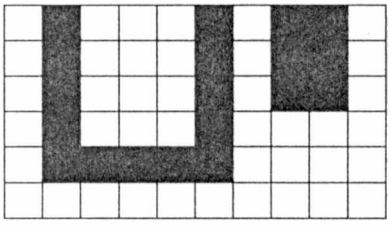
⇨ Linea que se va a procesar por dos clients.

Y la procesamos con dos procesos clients. Cada uno de ellos tendría la siguiente imagen:

Proceso I



Proceso II



Luego de realizar la primera pasada del labeling, la imagen etiquetada y la tabla de labeling local quedan de la siguiente forma:

Proceso I

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

Tabla de labeling

Etiqueta – Equivalencia

1 – 0
2 – 0
3 – 0
4 – 0

Proceso II

	1				2		3	3	
	1				2		3	3	
	1				2		3	3	
	1				2				
	1	1	1	1	1				

Tabla de labeling

Etiqueta – Equivalencia

1 – 0
2 – 1
3 – 0

Luego cada uno de estos dos procesos envía al master su tabla de labeling para que este las unifique. Recordemos que el master le suma un numero a las etiquetas de los distintos procesos para evitar que se dupliquen:

Proceso Master

Tabla de labeling unificada

1 – 0
2 – 0
3 – 0
4 – 0
} Proceden del Cliente I

5 – 0
6 – 5
7 – 0
} Proceden del Cliente II

Luego el proceso master analiza la línea conflictiva, que fue procesada por ambos clients (en este caso es la sexta) y agrega las siguientes equivalencias:

5–1 que significa que el label 5 del client II equivale al label 1 del client 1, 6–2 y 7–4. Como existía la equivalencia 6 – 5, se deducen 2 – 1 y 6 – 1. Luego se recorre la tabla y las etiquetas que no poseen equivalencias se les agrega una equivalencia a sí mismas. Esto es para que al devolver las tablas parciales a los procesos client estos sepan que número le sumo el master a sus etiquetas para hacerlas únicas.

La tabla final queda:

1 – 1 (Agregada pues no tenía equivalencia)
2 – 1
3 – 3 (Agregada pues no tenía equivalencia)
4 – 4 (Agregada pues no tenía equivalencia)
5 – 1
6 – 1
7 – 4

Luego se envían las partes de tabla a cada client, y la tabla de labeling local de los mismos queda como sigue (al recibir la parte de tabla de labeling que le corresponde, desaparece el número sumado al master a las etiquetas):

Proceso I

1 – 1
2 – 1
3 – 3
4 – 4

Proceso II

1 – 1
2 – 1
3 – 4

Con esta información, cada client realiza su segunda pasada de labeling, que consiste solamente en reemplazar cada etiqueta por su equivalente de acuerdo a la tabla de labeling recibida del master. Luego de esta segunda pasada, cada imagen parcial queda de la siguiente forma:

Proceso I

	1				1		3	3	
	1				1		3	3	
	1				1				
	1				1		4	4	
	1				1		4	4	

Proceso II

	1				1		4	4	
	1				1		4	4	
	1				1		4	4	
	1				1				
	1	1	1	1	1				

En este algoritmo, el proceso master realiza una parte del procesamiento, concretamente unifica las tablas de labeling y resuelve los conflictos en las líneas de imagen solapadas. Esto puede resultar ineficiente en casos en que el número de procesadores es muy grande (por ejemplo si hay un procesador por línea de imagen), ya que el master estaría procesando muchas líneas, con lo que se perdería las ventajas del paralelismo. Como nuestra aplicación esta pensada para ejecutarse en una red local con un número limitado de máquinas convencionales, esto no es un problema (por ejemplo si tenemos cinco máquinas convencionales y una imagen de 200 líneas el proceso master solo procesaría cinco líneas, o sea, el 2.5% de la imagen). A su vez, esta disminuye el uso de la red de comunicaciones ya que si no los clients deberían comunicarse entre sí y con otro proceso que provea etiquetas únicas.

5.3.3 - Cálculo del color promedio

Para el cálculo del color promedio distribuido usamos el mismo algoritmo descripto anteriormente, solo que ahora cada client envía al proceso master su resultado parcial y este calcula el promedio entre todos los clients. Solo hay que tener en cuenta que este promedio debe ser pesado con la cantidad de pixels que tuvo en cuenta cada proceso.

5.3.4 - Detección de bordes

Para realizar la búsqueda de los pixels que forman el perímetro del objeto en paralelo, se debe tener en cuenta que la imagen esta partida. Esto significa que no se puede tomar la convención de que los pixels de la región que estén situados sobre la primera y la última línea de la imagen pertenecen al perímetro de la región, ya que puede tratarse de una partición intermedia de la imagen.

Para resolver este inconveniente es necesario que los procesos sepan qué parte de la imagen están procesando. Estas partes pueden ser:

- La primera, lo que significa que la primera línea de su subimagen es límite de la imagen completa.
- Intermedia, lo que significa que ni la primera ni la última línea de su subimagen son límite de la imagen completa.
- La última, lo que significa que la última línea es límite de la imagen completa.

También se debe tener en cuenta que las líneas sobre las cuales se parte la imagen son procesadas dos veces y es necesario evitar en el caso que el borde de la región este sobre ellas contarlo dos veces.

Para esto desarrollamos un algoritmo similar al convencional pero que tiene en cuenta todos estos casos. El seudocódigo del mismo es el siguiente:

```
Procedimiento Busqueda_de_borde_paralelo
 // Para el primer pixel de la primera línea
 Si es el primer proceso
    Si el pixel pertenece a la región
       agrego el pixel como borde
    Fin Si
 Fin Si

 // Para el medio de la primera línea
 Para c/pixel desde el segundo al ante último de la línea
    Si el pixel pertenece a la región
       Si es el primer proceso
       O la etiqueta de abajo es distinta
       O la etiqueta de la derecha es distinta
       O la etiqueta de la izquierda es distinta
          Agrego el pixel como borde
       Fin si
    Fin si
 Fin para

 // Para el último pixel de la primera línea
 Si el pixel pertenece a la región
    Si el primer proceso
       Agrego el pixel como borde
    Fin si
 Fin si

 // El medio de la imagen (de la segunda a la ante última línea)
 // Se procesa igual que en el algoritmo convencional.


 // Primer pixel de la última línea
 Si pertenece a la región
    Agrego el pixel como borde
 Fin si

 // Para el medio de la última línea
 Para c/pixel desde el segundo al ante último de la línea
    Si el pixel pertenece a la región
       Si es el último proceso
       O la etiqueta de arriba es distinta
       O la etiqueta de la izquierda es distinta
       O la etiqueta de la derecha es distinta
          Agrego el pixel como borde
       Fin si
    Fin si
 Fin para

 // Para el último pixel de la última línea
 Si el pixel pertenece a la región
    Agrego el pixel como borde
```

```
    Fin si
Fin Busqueda_de_borde_paralelo
```

Este algoritmo soluciona los problemas mencionados anteriormente y por lo tanto permite identificar los pixels que pertenecen al perímetro de la región sin duplicados y evitando tomar como borde los pixels de los bordes de las particiones intermedias de la imagen original si es que no lo son.

5.4 - Ambiente de desarrollo

Como mencionamos anteriormente la aplicación está orientada a convertirse en un proceso tipo batch. Este proceso se aplica sobre las imágenes a medida que éstas van siendo obtenidas, y luego los resultados son enviados a dispositivos que deciden el destino de los objetos procesados de acuerdo a sus características.

Para el desarrollo, sin embargo, necesitábamos de un ambiente gráfico para poder mostrar la imagen resultante en cada paso del procesamiento para poder determinar rápidamente si los algoritmos funcionaban.

5.4.1 - Sistema X Windows

El sistema gráfico que viene con Linux es el XWindows, el cual es muy popular en el ambiente Unix. El mismo está desarrollado sobre una arquitectura client-server, que permite en forma transparente ver datos en la pantalla que en realidad se generan en cualquier máquina de una red.

Intentamos realizar aplicaciones sencillas para aprender a utilizar la API del sistema X, pero pronto advertimos que si bien dibujar elementos de interface (ventanas, diálogos, botones, etc.) es simple, mostrar imágenes True Color como las que queríamos utilizar no era para nada sencillo. Sobre todo debido a que no poseíamos mucha documentación sobre el mismo (nos basábamos en manuales on-line), y a que la API de X es muy extensa.

5.4.2 - Librería gráfica

Para simplificar el desarrollo, decidimos usar alguna librería de las que se encuentran en las distribuciones de Linux y que son free-ware. De este modo llegamos al Gimp, que es básicamente una aplicación de procesamiento de gráficos simple, pero que permite agregar procesos plug-in. Por lo tanto, nosotros decidimos desarrollar nuestra aplicación gráfica como un plug-in de Gimp, y de esta manera podíamos utilizar la funcionalidad del mismo para mostrar imágenes en forma transparente.

El Gimp permite abrir una imagen en varios formatos (Jpeg, Tiff, Targa, Gif, etc.), la muestra, y permite aplicarle una serie de procesos que están implementados en forma de plug-ins.

Los plug-ins disponibles se configuran agregándolos a un archivo de texto llamado.gimprc, y Gimp permite al usuario seleccionar un proceso de esta lista para aplicarlo a la imagen cargada.

Una vez que es usuario eligió un proceso, Gimp lo ejecuta y le envía la imagen a procesar por medio de pipes. El proceso corre de forma completamente independiente a Gimp, y puede comunicarse con éste por medio de una API sencilla que permite mostrar nuevas imágenes, diálogos para el ingreso de datos, etc.

5.4.3 - Esqueleto de un plug-in

A continuación mostramos el seudocódigo de un plug-in de gimp:

```
Incluir la API de Gimp

Proc principal

    Inicializar la librería

    Imagen a procesar = Obtener Imagen de entrada
    Imagen resultado = Obtener imagen resultado

    Si imagen a procesar es RGB
        procesar ( imagen a procesar, imagen resultado )
        mostrar la imagen resultado
    Sino
        Mostrar mensaje de error "Formato incorrecto"
    Fin si

    Liberar memoria de imagen a procesar e imagen resultado

Fin Principal
```

En el apéndice A se detallan las funciones de la API de Gimp.

5.4.4 - Aplicación gráfica

La aplicación gráfica consiste en un plug-in de Gimp, que al ser ejecutado sobre una imagen previamente leída, presenta un diálogo que permite especificar los parámetros de procesamiento, los cuales son:

- Cantidad de procesos a utilizar
- Valor de threshold
- Tamaño de los defectos (manchas) a tener en cuenta en relación con el tamaño del objeto principal
- Diferencia de color que debe tener una región con respecto al color promedio para ser considerada defecto
- Si se desean calcular los momentos
- Si se desea mostrar los resultados parciales (solo para debug)

Luego, se realiza el procesamiento, y al final del mismo se muestra un diálogo que contiene los resultados del mismo, que son los siguientes:

- Cantidad de pixels del objeto principal (superficie)
- Color promedio del objeto principal
- Diámetro máximo del objeto principal
- Los valores (opcionales) de los momentos μ_r y σ_r^2
- Cantidad de manchas (defectos)
- Cantidad de pixels en manchas (superficie de los defectos)
- Centroide del objeto
- Centro de marcas de medición si es que se encontraron
- Tiempo del procesamiento.

5.5 - Aplicación batch

Por otro lado, realizamos la aplicación en forma batch, que como se menciona anteriormente, es el modo natural de ejecución de la misma.

Esta implementación nos fue útil para las mediciones de performance necesarias para determinar la performance de la misma.

El proceso batch es muy sencillo y permite procesar una imagen con formato JPeg, cuyo nombre es pasado como parámetro en el comando de línea, así como el resto de los parámetros de procesamiento (que son los mismos que para la aplicación gráfica). Al finalizar, los resultados obtenidos del procesamiento se imprimen en la pantalla.

Para levantar imágenes con formato JPeg, utilizamos una librería provista en la distribución de Linux llamada libjpeg, que también es free-ware y es desarrollada y mantenida por un grupo de trabajo llamado *The Independent JPEG Group's (IJG)*. Dicha librería es utilizada por varias aplicaciones estándar de procesamiento de imágenes como el XView.

Un ejemplo de utilización del proceso batch es el siguiente:

```
hbatch imagen1.jpg 4 100
```

el cual procesará la imagen imagen1.jpg con 4 procesos y utilizando un valor de threshold de 100.

Resultados *obtenidos*

Con el fin de analizar la pérformance de la aplicación, se obtuvieron los tiempos de ejecución sobre distintas configuraciones de la máquina virtual, cambiando además la imágen a procesar.

6.1 - Introducción

Al realizar las mediciones de performance de la aplicación nos encontramos con limitaciones que detallamos a continuación:

- Algunas de las fórmulas de métricas estudiadas en el capítulo V asumen un ambiente de ejecución con procesadores homogéneos, o con una cantidad ilimitada de procesadores y otros recursos. La aplicación que desarrollamos está diseñada para un ambiente heterogéneo y de recursos limitados, por lo que no se pueden aplicar éstas métricas para evaluar los resultados.

- La aplicación está desarrollada bajo el sistema operativo Unix. Este realiza un scheduling de procesos particular que distribuye el tiempo de CPU entre la aplicación que se esta ejecutando y los procesos de administración de recursos del propio S. O. (administrador de memoria, de archivos, XWindows, etc.). Esta administración del tiempo de CPU se puede modificar alterando las prioridades de los procesos que se están ejecutando. Por este motivo, al evaluar los resultados obtenidos hay que tener en cuenta que la aplicación no dispone del cien por cien del tiempo de procesamiento.

- En teoría, PVM debería distribuir los procesos teniendo en cuenta la performance y la sobrecarga de cada una de los procesadores (host) que componen la máquina virtual. Esto se consigue seteando un parámetro de Speed para cada uno de los host. Las pruebas realizadas demuestran que PVM no toma en cuenta éste parámetro, realizando una distribución uniforme de la carga de procesamiento entre todos los host.

6.2 - Hardware para pruebas

Se utilizaron dos procesadores Pentium (133 Mhz), con 32 y 48 Mb. de RAM, y un 80-486 DX4 (100 Mhz) con 12 Mb. Para realizar las mediciones se armaron dos configuraciones de red distintas:

1°. Se conectó uno de los procesadores Pentium (el de 32 Mb. de RAM) con el 80-486 DX4, utilizando placas de red NE-2000 compatibles de 10 Mbits/seg.

2°. Se conectaron los dos procesadores Pentium con el mismo tipo de placas de red.

6.3 - Resultados de las mediciones realizadas

Se realizaron sucesivas ejecuciones de la aplicación haciendo variar para cada una de ellas la cantidad de procesos clientes, la cantidad de host y la imagen a procesar. Además, cada caso fue ejecutado varias veces para disminuir los efectos del scheduling de memoria (cache) que realiza Unix.

El procesamiento realizado sobre cada una de las imágenes consistió en las siguientes mediciones:

- Area del objeto
- Color promedio
- Cantidad de defectos
- Area de los defectos
- Centroide
- Diámetro máximo
- Determinación del perímetro
- Ubicación del centro de las marcas de referencia (para las fotos que las tuviesen)

Al ir promediando las pruebas notamos que los tiempos de ejecución de la aplicación variaban sustancialmente entre corridas con cantidad de procesos clients pares e impares, cuando la cantidad de hosts de la máquina virtual era par.

Analizando detenidamente cada prueba descubrimos que PVM distribuía en forma equitativa los procesos que ejecutaba entre los host de la máquina virtual. De ésta manera, cuando la cantidad de procesos clients a ejecutar es impar, por ejemplo tres, y la cantidad de hosts es par, por ejemplo dos, PVM corre dos procesos clients en un host y un proceso client junto al proceso master en el otro host.

Esta distribución de procesos provoca un aumento de los tiempos de ejecución: el host que procesa los dos clients demora más que el otro pues el proceso master lleva poco tiempo de ejecución. En el caso de tres procesos clients, debido a que la imagen a procesar se distribuye equitativamente entre los tres, el tiempo de procesamiento final será lo que demore en procesar dos tercios de la imagen el host con dos clients. En cambio, si el total de clients es cuatro, el tiempo de procesamiento final será lo que demore en procesar la mitad de la imagen el host con dos clients y el master, siendo éste último sensiblemente inferior al de la prueba con tres clients.

Un ejemplo de la diferencia en los tiempos de ejecución entre corridas con cantidad de procesos clients pares e impares se puede apreciar en la siguiente tabla:

Procs.	1	2	3	4	5	6	7	8	9	10
Tiempo	6,18	5,85	5,48	5,1	5,77	5,55	5,95	5,76	6,43	6,28

Como consecuencia de éste análisis, decidimos realizar las pruebas con cantidad de procesos clients pares (además de la corrida con un solo proceso client), para que los resultados obtenidos muestren claramente el aumento o disminución de los tiempos de procesamiento al aumentar la cantidad de procesos clients. Aclaramos que si las pruebas se realizan con cantidad de procesos impares, los cambios en los tiempos de ejecución son porcentualmente similares.

Primer lote de pruebas

Las primeras pruebas que se realizaron consistieron en correr la aplicación sobre una máquina virtual de un solo host (en uno de los Pentium 133 con 48 Mb. de RAM). La imagen elegida para realizar éstas pruebas se puede encontrar en el Apéndice B de imágenes (imagen número 1); tiene una resolución de 264 x 384 pixels en 24 bits color.

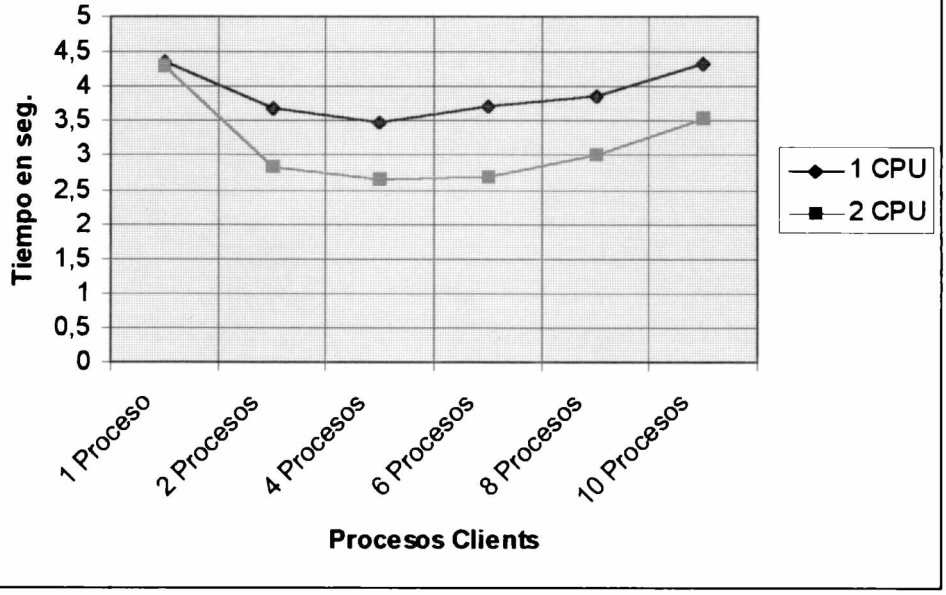
Los resultados obtenidos son los siguientes:

	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	4,36 seg.	3,695 seg.	3,47 seg.	3,71 seg.	3,865 seg.	4,33 seg.

La siguiente serie de mediciones fue realizada luego de agregar a la configuración anterior un nuevo host (el otro Pentium), procesando siempre sobre la imagen 1. Los resultados de estas mediciones son los siguientes:

	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	4,3 seg.	2,84 seg.	2,675 seg.	2,7 seg.	3,025 seg.	3,55 seg.

Fig. 6.1 Tiempos de procesamiento (imagen 1).



En la figura 6.1 pueden compararse los tiempos de ejecución de las corridas realizadas con una y dos CPUs sobre la imagen 1. El speedup obtenido, considerando los mejores tiempos para cada caso, es de 1.29.

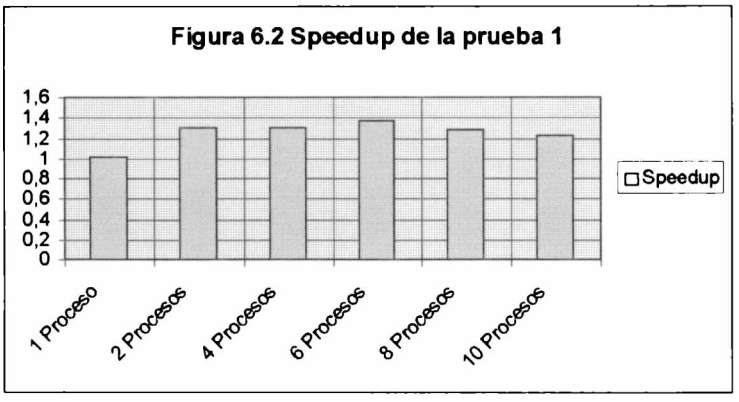


Figura 6.2 Speedup de la prueba 1

En la figura 6.2 se grafica el speedup obtenido para cada corrida (cada una con distinta cantidad de procesos clients), entre una máquina virtual de un solo host y una con dos host.

Segundo lote de pruebas

Para evaluar mejor el comportamiento de la aplicación con distintas cargas de trabajo, realizamos las mismas mediciones sobre una imagen de mayor tamaño (522 x 804 pixels en 24 bits color) que puede ser encontrada en el Apéndice B de imágenes con el número 2.

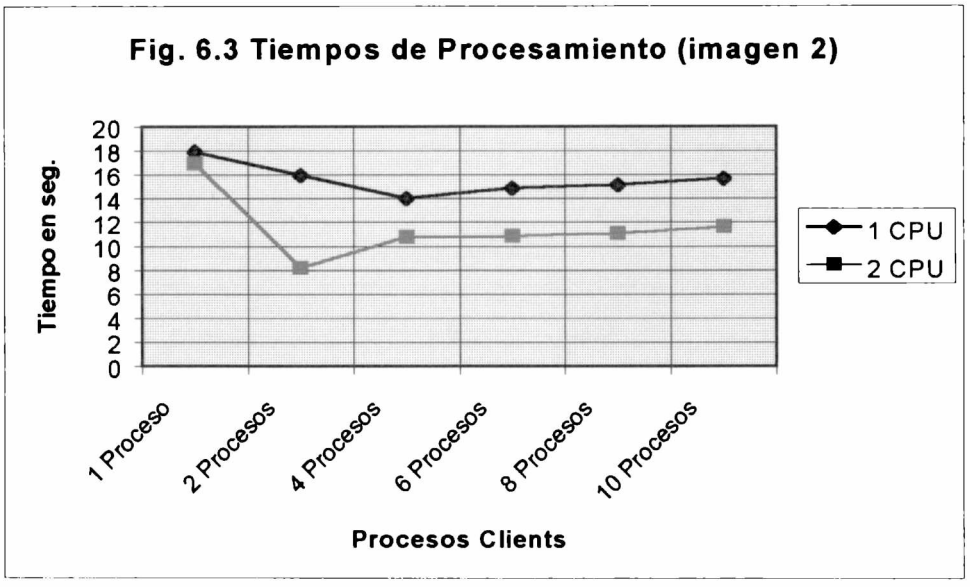
Los resultados obtenidos corriendo la aplicación sobre una máquina virtual con un solo host son los siguientes:

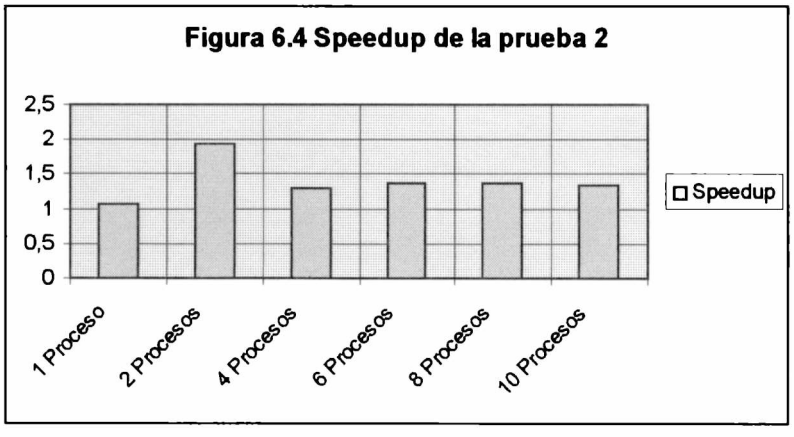
	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	17,9 seg.	15,94 seg.	14,01 seg.	14,84 seg.	15,13 seg.	15,66 seg.

Al agregar otro host a la máquina virtual, se obtuvieron los siguientes resultados:

	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	16,95 seg.	8,206 seg.	10,77 seg.	10,84 seg.	11,05 seg.	11,6 seg.

En la figura 6.3 pueden apreciarse los tiempos de ejecución de las corridas realizadas con una y dos CPUs sobre la imagen 2. El speedup obtenido considerando los mejores tiempos para cada caso, es de 1.70.





En la figura 6.4 se grafica el speedup obtenido para cada corrida (cada una con distinta cant. de procs. clients), entre una máquina virtual de un solo host y una con dos host.

Tercer lote de pruebas

Para analizar el desempeño de la aplicación en ambientes heterogéneos, configuramos una red con un computador Pentium de 133 Mhz y un 80-486 DX4 de 100 Mhz.

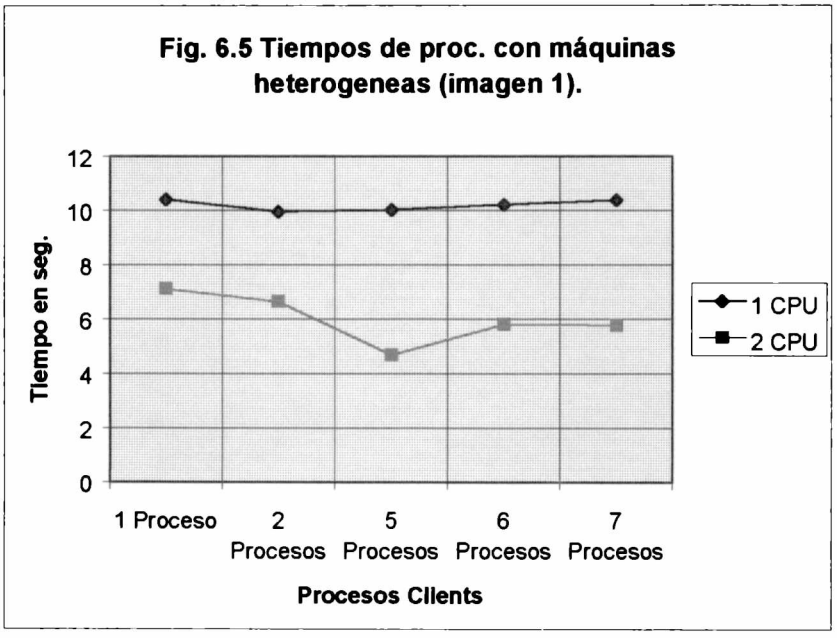
Primero corrimos la aplicación sobre una máquina virtual que tenía un único host consistente en el computador 80-486 DX4, obteniéndose los siguientes resultados:

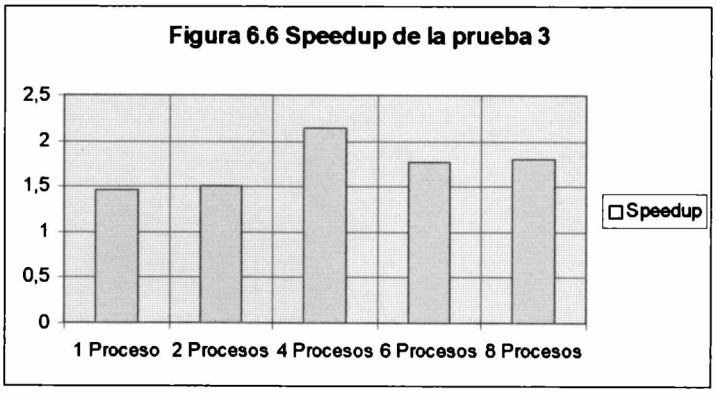
	1 Proceso	2 Procesos	5 Procesos	6 Procesos	7 Procesos
Tiempos	10,41 seg.	9,95 seg.	10,03 seg.	10,23 seg.	10,38 seg.

Agregando un host a la máquina virtual (el Pentium de 133 Mhz) se lograron bajar sustancialmente los tiempos de ejecución siempre que al haber una cantidad impar de procesos clients, la mayor cantidad posible (la mitad más uno) de ellos se ejecutara en el host más rápido. Los resultados obtenidos son los siguientes:

	1 Proceso	2 Procesos	5 Procesos	6 Procesos	7 Procesos
Tiempos	7,12 seg.	6,65 seg.	4,7 seg.	5,82 seg.	5,76 seg.

En la figura 6.5 pueden apreciarse los tiempos de ejecución de las corridas realizadas con una y dos CPUs heterogéneas sobre la imagen 1. El speedup obtenido considerando los mejores tiempos para cada caso, es de 2.11.





En la figura 6.6 se grafica el speedup obtenido para cada corrida (cada una con distinta cantidad de procesos clients), entre una máquina virtual de un solo host y una con dos host.

Cuarto lote de pruebas

La siguiente prueba la efectuamos sobre la misma configuración de red, pero ahora con una imagen de 522 x 804 pixels en 24 bits con la que se espera obtener una notoria mejora de pérformance entre una ejecución con el 80-486 DX4 como único host (CASO I) y otra con un Pentium de 133 Mhz como segundo host (CASO II).

Los tiempos obtenidos son los siguientes:

CASO I : 35,9 segundos
CASO II : 15,13 segundos

Se observa que el speedup conseguido es del orden de 2,37.

Quinto lote de pruebas

Para evaluar el comportamiento de la aplicación con imágenes de otros objetos, que a priori suponíamos debía ser similar, tomamos los tiempos de procesamiento de una foto de una naranja de tamaño (929 x 973 pixels en 24 bits color) que puede ser encontrada en el Apéndice B de imágenes con el número 3.

Los resultados obtenidos corriendo la aplicación sobre una máquina virtual con un solo host son los siguientes:

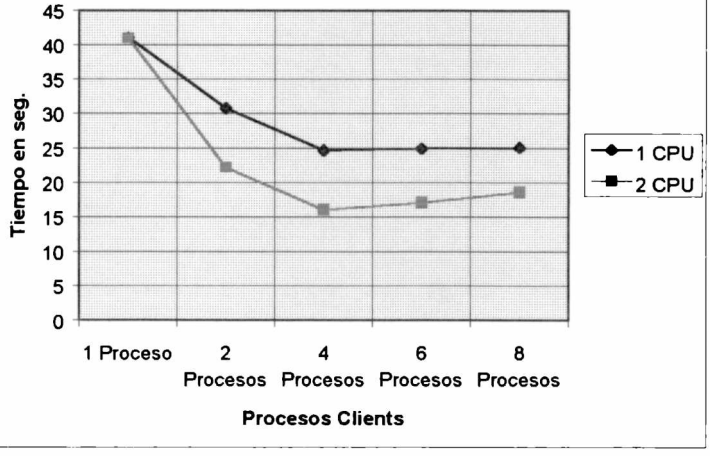
	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	41,05 seg.	30,74 seg.	24,67 seg.	24,93 seg.	25,08 seg.	26,56 seg.

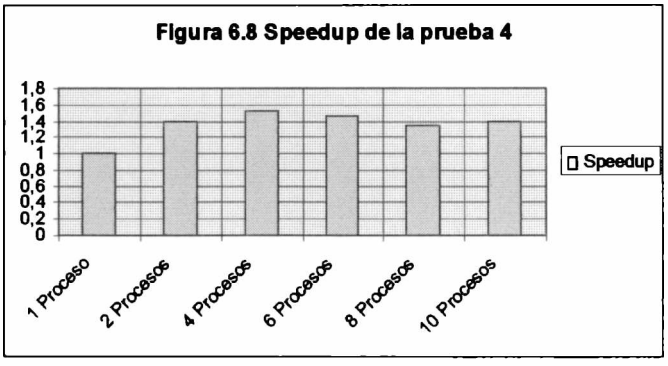
Al agregar otro host a la máquina virtual, se obtuvieron los siguientes resultados:

	1 Proceso	2 Procs.	4 Procs.	6 Procs.	8 Procs.	10 Procs.
Tiempo de ejecución	40,95 seg.	22,18 seg.	16,09 seg.	17,10 seg.	18,58 seg.	18,98 seg.

En la figura 6.7 pueden apreciarse los tiempos de ejecución de las corridas realizadas con una y dos CPUs sobre la imagen 2. El speedup obtenido considerando los mejores tiempos para cada caso, es de 1.70.

Fig. 6.7 Tiempos de procesamiento sobre la imagen 3





En la figura 6.8 se grafica el speedup obtenido para cada corrida (cada una con distinta cantidad de procesos clients), entre una máquina virtual de un solo host y una con dos host.

Conclusiones

Es conveniente realizar una detallada evaluación de las diversas herramientas utilizadas en la implementación de ésta aplicación (Linux, Xwindows, P.V.M. y C++), para poder arribar a una conclusión final del presente trabajo.

7.1 – Observaciones de las herramientas utilizadas

En el desarrollo de esta aplicación hemos utilizado diversas herramientas cuya evaluación es conveniente realizar por separado.

En primer lugar, la plataforma sobre la que se realizó el desarrollo fue Linux, que si bien es actualmente un sistema operativo bastante difundido, no hay que olvidar que es free-ware, por lo que no existe un soporte oficial al que recurrir en caso de que surjan inconvenientes.

También utilizamos la interface gráfica XWindows, en la versión Xfree, para la que caben las mismas consideraciones que para Linux.

Por otro lado, para resolver la distribución de procesos y la comunicación entre los mismos utilizamos PVM, y como del lenguaje de programación optamos por el C++, aprovechando sus características de lenguaje orientado a objetos de bajo nivel.

7.1.1 - Linux

Ventajas:

- A pesar de no ser un sistema operativo comercial, resultó ser muy estable, lo que se traduce en una eficiente administración de los recursos del hardware, proveyendo características de fault tolerance.
- Provee facilidades para la configuración de redes TCP/IP, característica muy importante para la aplicación desarrollada por su orientación de trabajo en redes heterogéneas.
- Existe una versión para el mismo del popular compilador GCC de GNU para C y C++, cuyas prestaciones resultaron ser altamente satisfactorias.
- La implementación de PVM sobre Linux funcionó muy bien.

Desventajas:

- La instalación del mismo no resulta ser muy amigable.
- Para poder usar ciertos componentes de hardware (placas de red) es necesario recompilar el kernel.

En conclusión, creemos que la elección resultó acertada y que es un sistema operativo a tener en cuenta en el caso de querer desarrollar aplicaciones de estas características (interconexión, concurrencia y manejo de volúmenes de datos considerables). También nos permite explotar al máximo la portabilidad de la aplicación.

7.1.2 - XWindows

Ventajas:

- Existen librerías free-ware que simplifican su uso.
- Permite correr aplicaciones y ver los resultados en máquinas distintas, lo que facilita la realización de pruebas.

Desventajas:

- Posee una API muy complicada y poco amigable, por lo que resulta dificultoso desarrollar aplicaciones que la usen.
- Es difícil de configurar para poder utilizar altas resoluciones gráficas.
- No resulta sencillo encontrar documentación sobre el mismo, y generalmente ésta es poco clara o está orientada a usuarios avanzados.

7.1.3 - PVM

Ventajas:

- Hace totalmente transparente la comunicación entre procesos distribuidos.
- Permite pasar también en forma transparente de configuraciones pequeñas (por ejemplo una sola máquina) a configuraciones grandes, gracia a que la configuración de la máquina virtual es independiente de la aplicación que la utiliza.
- Permite reconfigurar la máquina virtual en forma dinámica sin necesidad de modificar ni recompilar la aplicación.
- Transforma de forma automática los mensajes entre máquinas heterogéneas, para que se interpreten en forma adecuada aunque varíe el formato.

Desventajas:

- No funcionó una característica descripta del mismo consistente en la asignación de un numero que representa la velocidad de procesamiento relativa de cada host. Si bien permite asignar este número, no lo utiliza en la distribución de la carga de trabajo, llevando a cabo la misma de manera equitativa entre todos los hosts que componen la máquina virtual.
- Cuando un cliente termina, esto es, finaliza la ejecución, si había enviado datos que aún no habían sido recibidos, estos se pierden.
- No permite definir protocolos de comunicación del estilo de OCCAM, para enviar datos complejos como una estructura. En necesario realizar varias comunicaciones, y en cada una de ellas enviar un tipo distinto de datos.

7.1.4 – C++

Ventajas:

- Permite definir herencia entre clases y encapsulamiento de datos, lo que nos permitió realizar una implementación clara y fácilmente modificable y/o ampliable.
- Por su característica de bajo nivel permite una utilización óptima de los recursos disponibles.

Desventajas:

- Requiere un excesivo cuidado en el manejo de la memoria.
- El Debugger que provee la versión del compilador utilizado no es amigable (se maneja a través de una línea de comandos), dificultando la detección de errores.

7.2 – Conclusiones del trabajo realizado

Los resultados obtenidos durante la etapa de mediciones de performance nos lleva a concluir que la implementación de soluciones empleando paralelismo de grano grueso logra mejoras significativas solo cuando el procesamiento a realizar cumple las siguientes características:

- Los procesos involucrados deben ser complejos, en el sentido que requieran un alto tiempo de procesamiento.
- Durante la ejecución de los mismos, las necesidades de comunicación entre procesos no deben ser excesivas.

Esto se debe a que el costo de comunicación es significativo con relación a la velocidad de procesamiento, pues se está utilizando una red de área local (LAN), que no se caracteriza por una alta velocidad de transferencia de datos.

Si se utiliza más tiempo de CPU para comunicación que para procesamiento, la pérformance final se deteriora ya que no se obtienen las ventajas esperadas de la paralelización.

7.3 – Líneas de trabajo futuras

Si bien los resultados obtenidos satisfacen altamente los objetivos planteados antes de iniciar el trabajo, podrían seguirse líneas de estudio alternativas con la intención de obtener mejoras en ciertos aspectos, como por ejemplo:

- Para evitar la descompensación en la carga de trabajo entre procesadores con distintas velocidades, podría implementarse un administrador de procesos que dinámicamente detecte los hosts más veloces de la máquina virtual (serán aquellos que terminen más rápido de procesar) y asignarles mayor carga de procesamiento, lo que se traduce en más imagen a procesar. De esta manera, se evitaría la distribución equitativa de procesos que realiza PVM, aprovechándose al máximo el tiempo de CPU disponible.
- Podría estudiarse la manera de asignarle a los procesos de la aplicación la máxima prioridad de ejecución posible, de manera tal que Unix no dedique demasiado tiempo de CPU a sus procesos internos de scheduling.
- Con el fin de bajar los tiempos de transferencia de datos entre procesos, deberían estudiarse otras configuraciones de red y sistemas de comunicación (fibra óptica).

Además de las mejoras que se puedan introducir, existen aspectos de la aplicación real que motivó este trabajo que no fueron contemplados, como por ejemplo el módulo que debería captar la imagen desde la cinta transportadora, digitalizarla y mejorarla para su posterior procesamiento; o la etapa final de la aplicación que se encargaría de evaluar los resultados obtenidos (en el procesamiento de una imagen determinada) y tomar decisiones en base a éstos, como por ejemplo enviar el objeto en cuestión a donde el resultado de su clasificación indique.

Apéndice A

Implementación

En este apéndice se detallan las clases implementadas en la aplicación, con el fin de ayudar al lector en el análisis y seguimiento de la misma.

A.1 - Consideraciones previas

Para la implementación de la aplicación tuvimos en consideración los siguientes aspectos:

1°. Que la implementación fuera clara y fácil de extender en el caso de querer agregar nuevos algoritmos.
2°. Que PVM estaba disponible en el momento del desarrollo solo en los lenguajes C, y Fortran.

Esto nos llevó al uso del lenguaje C++, ya que provee posibilidades de programación orientada a objetos, lo que facilita el desarrollo de un código claro, modular y extendible y además es posible desde C++ acceder a las rutinas PVM que están desarrolladas en C.

Por lo tanto, definimos clases para abstraer los principales objetos que intervienen en nuestra aplicación, los cuales son *imágenes* RGB compuestas por *pixels*, *algoritmos* que se aplican sobre estas imágenes. También decidimos definir clases para encapsular las llamadas a PVM (instanciación y comunicación entre procesos) para poder en el futuro cambiar esta librería por otra similar sin tener que modificar el resto del código.

El encapsulamiento de los algoritmos en forma de clases permite agregar nuevos algoritmos en forma muy sencilla y siguiendo un esquema claro, además de permitir generar listas de algoritmos para aplicar secuencialmente a una imagen.

A.2 - Clases para abstraer las imágenes y los pixels.

La clase **HImage** encapsula una imagen RGB, la cual es una secuencia de pixels cada uno de los cuales posee tres componentes de color. Dichos pixels están encapsulados en la clase **Hpixel**.
La clase **HImage** encapsula también un cursor (**Cursor**) que permite recorrer todos los pixels de la misma en una forma ordenada.

Veamos la interface de estas dos clases:

```
class HPixel
{
public:

// Constructor
//
Hpixel  (uchar r=0, uchar g=0, uchar b=0);

// Getters && Setters
//
int     red   (); // Componerte Red del pixel
int     green(); // Componerte Green del pixel
int     blue  (); // Componerte Blue del pixel
ushort  label (); // Etiqueta del pixel
HPixel&  setRed    (int r);     // Seteo de la componente Red del pixel
HPixel&  setGreen  (int g);     // Seteo de la componente Green del pixel
```

```
HPixel&   setBlue    (int b);     // Seteo de la componente Blue del pixel
HPixel&   setLabel   (ushort l);  // Seteo de la etiqueta del pixel

uchar     peso       ();          // Indica si el pixel tiene algún valor

// Metodos
//
double    intensity  ();          // Retorna un número que representa
                                  // la intensidad del mismo

// Operadores para sumar, restar, dividir pixels y cambiar su magnitud por un
// factor.
//
friend HPixel operator *  ( HPixel& p, int f);
friend HPixel operator *  ( int f, HPixel& p);
friend HPixel operator /  ( HPixel& p, int f);
friend HPixel operator +  ( HPixel& p1, HPixel& p2);
friend HPixel operator -  ( HPixel& p1, HPixel& p2);

// Constantes
//
static HPixel black;   // Representa a un pixel negro (fondo)
static HPixel white;   // Representa a un pixel blanco (objeto)

};// End class HPixel
```

```
class HImage
{

class Cursor;
friend Cursor;

public:

// Constructores
//
HImage ();
HImage ( int w, int h );

// Destructor
//
~HImage ();

// Creación y alocación de memoria
//
void      setSize      ( int w, int h );
void      allocMemory  ();
void      freeMemory   ();
HImage&   setData      ( int w, int h, uchar * d );
```

```
// Getters && Setters
//
int   width    ();          // Retorna el ancho de la imagen
int   height   ();          // Retorna el alto de la imagen

uchar *   data     ();      // Retorna un puntero a los pixels de la imagen

int   bytesCount   ();      // Cantidad de bytes que ocupa la imagen
int   pixelCount   ();      // Cantidad de pixels que posee

HPixel    pixel    ( int x, int y );          // Retorna el pixel que de
                                              // la coordenada (X,Y)
HImage&   setPixel ( int x, int y, HPixel& p); // Modifica el pixel de la
                                              // la coordenada (X,Y)

HPixel&   pixelRef ( int x, int y ); // Retorna una referencia al pixel de
                                     // la coordenada (X,Y), que se puede modificar.

uchar *   firstLine ();     // Primera linea de la imagen
uchar *   lastLine ();      // Ultima linea de la imagen

// Metodos estáticos para encapsular coordenadas (x,y) en un
// solo número entero.
//
static int   xCord   ( int c );
static int   yCord   ( int c );
static int   asCord  ( int x, int y);


// Cursor para recorrer los pixels de una imagen.
//
  class Cursor {
  public:

  // Constructores
  //
  Cursor ( HImage& img, int inc = 3 );   // Construye un cursor para la imagen img
  Cursor ( int inc = 3 );

  // Operadores
  //
  Cursor& operator++ ();        // Se posiciona sobre el siguiente pixel
  Cursor& operator-- ();        // Se posiciona sobre el pixel anterior
  Cursor& operator++ (int);     // Se posiciona sobre el siguiente pixel
  Cursor& operator-- (int);     // Se posiciona sobre el pixel anterior
  HPixel* operator-> ();        // Retorna un puntero al pixel corriente

  // Metodos
  //
  Cursor& beg     ();     // Posiciona al cursor sobre el primer pixel de la imagen
  Cursor& next    ();     // Idem operator++
  Cursor& prev    ();     // Idem operator--
  HPixel& element ();     // Retorna una referencia al pixel corriente
```

```
  Cursor& end    ();   // Se posiciona en el último pixel de la imagen
  bool    eof    ();   // true si se llego al final de la imagen
  bool    bof    ();   // true si esta al principio de la imagen

  // Control de cuantos pixels salta el cursor en cada movimiento. Puede
  // setearse que se salte un número cualquiera de pixels en cada
  // movimiento.
  //
  int     increment    ();
  Cursor& setIncrement ( int aInc );

  Cursor& gotoXY    ( int x, int y );   // Posiciona el cursor en una
                                        // coordenada en particular
  Cursor& gotoDesp  ( int desp );       // Posiciona el cursor en un desplazamiento
                                        // en particular

  int   desp ();   // Retorna el desplazamiento actual del cursor
  int   cord ();   // Retorna la coordenada actual del cursor

  }; // End class Cursor

};// End class HImage
```

A.3 - Clases para abstraer los algoritmos

Como dijimos anteriormente, los algoritmos usados para procesar la imagen están encapsulados en clases. Para esto existe una clase virtual que es la superclase de todos los algoritmos llamada **HimageAlgo**, la cual define una interface estándar para todos los algoritmos.

Esta clase se define de la siguiente forma:

```
class HImageAlgo
{
public:

// Constructor
//
HImageAlgo ();

virtual int apply ( HImage * img ) = 0; // Metodo virtual puro que las subclases
                                        // deben implementar y significa aplicar el
                                        // algoritmo en cuestión a la imagen img.
//
// En algunos casos nos interesan los pixels reales y en otros los que
// representarian el fondo (por ejemplo para las manchas) ... por eso
// aparecen estas cosas. Por default son lo normal, pero algunos algoritmos
// los invierten.
//
HPixel&   blackPixel ();   // Representa lo que el algoritmo considera fondo
HPixel&   whitePixel ();   // Representa lo que el algoritmo considera objeto/s
```

```
HImageAlgo&   setBlackPixel ( const HPixel& pix );   // Setea lo que para el
                                                     // algoritmo es fondo
HImageAlgo&   setWhitePixel ( const HPixel& pix );   // Setea lo que para el
                                                     // algoritmo es objeto/s

HImageAlgo& negative   ();     // Invierte la consideración de lo que es fondo
                               // con lo que es objeto

};// End class HImageAlgo
```

Veamos como ejemplo del uso de todas estas clases la implementación del threshold. Primero definimos su interface basandonos en la clase HimageAlgo:

```
class HThreshold : public HImageAlgo
{
public:
// Constructor del algoritmo que recibe como parámetro el valor límite
// para realizar el threshold
//
Hthreshold   ( int aVal = 128 );
Int  value   ();    // Retorna el valor límite del threshold
virtual int  apply   ( HImage * img );   // Realiza el threshold sobre la
                                         // imagen img.
};// End class HThreshold
```

Ahora definimos el método apply, que realiza el procesamiento de la imagen:

```
int HThreshold::apply (HImage * img)
{
  // Creamos un cursor sobre los pixels de img
  //
  HImage::Cursor pixels ( *img );

  // Recorremos todos los pixels del cursor
  // Si el valor cada pixel es mayor que el valor límite, lo ponemos en blanco
  // lo que significa considerarlo como posible parte de un objeto.
  // Si el valor del pixel es menor o igual, lo ponemos en negro, lo que
  // significa considerarlo fondo en el resto del procesamiento.
  //
  for ( pixels.beg(); !pixels.eof(); pixels++ )
     if ( pixels.element().intensity() > value() )
        pixels.element() = HPixel::white;
     else
        pixels.element() = HPixel::black;
  return 0;
}
```

Observemos que la implementación del algoritmo queda muy clara y sencilla, casi como si fuera seudocódigo.

El uso de cursores para recorrer la imagen facilita de manera significativa la implementación de los algoritmos, ya que permiten acceder a la información interna de la imagen de una forma transparente e independiente del formato de la misma. También hay que resaltar que como es posible definir varios cursores al mismo tiempo sobre la misma imagen, la implementación de algoritmos mas complejos como el labeling o la detección de bordes es también simple, ya que por ejemplo se pueden definir cuatro cursores adicionales que recorran los vecinos cuatro-conectados del pixel que estamos procesando.

A.4 - Clases para abstraer a los procesos master y clientes

El modelo de paralelismo que elegimos es el master-client, o sea que hay un proceso master que coordina el procesamiento y varios procesos clients que realizan el procesamiento.

Por lo tanto, definimos dos clase para encapsular el procesamiento de estos procesos.

La idea general de la aplicación es que alguien crea una instancia de un proceso HMaster, le setea todos los parámetros necesarios (imagen a procesar, numero de procesos entre los cuales distribuir el procesamiento, valores de threshold, etc.) y lo pone a procesar mediante el llamado a un método particular del objeto (process()). Este método instancia los procesos client corespondientes según los parámetros seteados, particiona la imagen entre estos procesos y luego coordina los resultados parciales de los mismos, para llegar al resultado final.

Por otro lado, los procesos client, al ser ejecutados, crean una instancia de la clase HClient y lo ponen a procesar llamando al método process(). Este método espera recibir los datos a procesar, los procesa e intercambia resultados en forma sincrónica con el proceso master que lo instanció.

Veamos la interface de la clase HMaster:

```
class HMaster
{
public:

// Constructor y destructor
//
HMaster ();
~HMaster ();

// Constantes y definiciones
//
enum OperType { opMainObject, opManchas };

// Alocacion y seteo de la imagen a procesar
//

HMaster& allocImages();
HMaster& freeImages ();
HMaster& setImage ( HImage * img );
```

```
// Getter & Setter de la imagen a procesar
//
HImage *    srcImage     ();              // Imagen original
HMaster&    setSrcImage  ( HImage * i ) ;
HImage *    procImage    ();              // Imagen temporaria

// Procesamiento de la imagen
//

// Método principal que realiza todo el procesamiento
//
HMaster& process  ();

// Envia los parámetros de procesamiento a los clientes
//
void        sendParams          ();

// Recibe la imagen final procesada
//
void        recFinalImage       ();

// Recibe las tablas de labeling parciales, las unifica, resuelve las equivalencias
// envia la tabla modificada a los clientes
//
void        recvAndProcessTable  ( OperType op );

// Calcula el color promedio sobre la base de los resultados parciales de los clientes.
//
void        calcProm            ();

// Calcula el centroide de los objetos sobre la base de los resultados parciales de los
// clientes.
//
void        calcCenter          ();

// Cuenta la cantidad de manchas que aparecen en la tabla de labeling table
//
HMaster& countManchas           ( HLabelTable& table );

// Pinta sobre la imagen final los resultados obtenidos, solo para ponerlos en
// forma visual.
//
enum PaintOp { PRegions, PBorder, PCenter, PMarks, PDiam, PAll  };
void    paintImage          ( PaintOp op );

// Parámetros seteados antes de empezar el procesamiento por quien instancia
// a este objeto/
//
```

```
// Valor límite para el threshold
//
int          thValue      ();
HMaster&     setThValue   (int v);

// Parametro para separar las manchas. Indica cuanto tiene que diferir un
// pixel del color promedio para ser considerado parte de una mancha.
//
int          rangeMancha      ();
HMaster&     setRangeMancha   (int v);

// Numero de procesos clients paralelos a utilizar
//
int          procsNumber      ();
HMaster&     setProcsNumber ( int p );

// Tamaño de manchas a tener en cuenta. Es un porcentaje sobre el
// tamaño del objeto principal. Este parámetro sirve para decidir que tamaño
// relativo tienen que tener las manchas para ser consideradas como tales.
//
double       manchaSize       ();
HMaster&     setManchaSize    ( double aObj );

// Si se debe calcular el valor de aproximación circular y la
// desviación estandar del mismo
//
bool         calcExen         ();
HMaster&     setCalcExen      ( bool aObj = true );

// Si se deben mostrar las imagenes parciales que se van generando
//
bool         showParcialImgs      ()
Hmaster&     setShowParcialImgs   ( bool aObj = true )


//
// Resultados del procesamiento  que son consultados al final del mismo.
// Los métodos con la forma setValor son llamados desde el método process para
// setear los valores correspondientes. Quien instancia al objeto master debe
// llamar a los métodos valor para obtener los resultados.
//

// Cantidad de pixels del objeto principal
//
int          cantPixels       ();
HMaster&     setCantPixels ( int aObj );

// Cantidad de manchas encontradas
//
int          cantManchas      ();
HMaster&     setCantManchas ( int aObj );
```

```
// Cantidad de pixels que pertenecen a manchas
//
int         cantPixelsManchas   ();
HMaster&    setCantPixelsManchas( int aObj );

// Un pixel que contiene en sus tres componentes los colores promedio
// calculados para el objeto principal
//
Hpixel    pixProm       ();
HMaster& setPixProm    ( HPixel aObj );

// Tiempo total que tomo todo el proceso
//
double      totalTime     ();
HMaster&    setTotalTime  ( double aObj );

// Centro de la marca de medicion 1. En el entero que retorna estan codificadas
// la coordenada x e y, y para obtenerlas se debe usar Himage::xCord() e
// Himage::yCord(). Esta consideración vale para todos los métodos siguientes
// que retornan puntos.
//
int         mark1Center   ();
HMaster&    setMark1Center  ( int aObj );

// Centro de la marca de medicion 2
//
int         mark2Center   ();
HMaster&    setMark2Center  ( int aObj );

// Centro del objeto principal
//
int         objectCenter  ();
HMaster&    setObjectCenter  ( int aObj );

// Maximo diametro del objeto principal
//
double      objectMaxDiam     ();
HMaster&    setObjectMaxDiam  ( double aObj );

// Primer extremo del diametro maximo
//
int         maxDiamP1     ();
Hmaster&    setMaxDiamP1   ( int aObj );

// Segundo extremo del diametro maximo
int         maxDiamP2     ();
Hmaster&    setMaxDiamP2   ( int aObj );

// Momentos. Aproximación circular
//
double      uExen     ();
HMaster&    setUExen   ( double aObj );
```

```
// Desviación estandar al cuadrado de la aproximación circular
//
double    o2Exen    ();
HMaster& setO2Exen ( double aObj );

// Relacion entre los dos valores anteriores
//
double    exenRel      ();
HMaster& setExenRel    ( double aObj );


// Codigos de error de la aplicación. Permiten saber si el procesamiento
// ha sido exitoso o si se produjeron errores durante el mismo. En este caso
// se puede consultar una explicación del error en forma de mensaje.
//
enum ErrorCodes { ErrOk = 0, ErrPvmDown = 1, ... };

int          errorCode     ();
HMaster&     setErrorCode  ( int aObj );
char *       errorMsg      ( int errCode = -1 );

//
// Creacion y operaciones sobre los clientes
//
HPvmProcessArray& clients   ();          // Retorna el objeto que encapsula
                                         // al conjunto de clients.
HPvmProxi&        proxi     ( int index ); // Retorna el objeto que encapsula
                                         // al cliente index.

//
// Miscelaneas
//

// Principio de las porciones de imágenes enviadas a cada client.
//
int*         startRows     ();
HMaster&     setStartRows  ( int* aPtr );

// Funciones para dibujar cosas sobre la imagen
//
void         drawLine   ( HImage * srcImg, int p1, int p2, const HPixel& rgb );
void         drawCross  ( HImage * srcImg, int p, const HPixel& rgb );

// Función que es llamada cuando se quiere mostrar una imagen parcial
//
typedef int ( * ShowImgFn ) ( HImage *, char *, int, int );

ShowImgFn    showImgFn     ();
HMaster&     setShowImgFn  ( ShowImgFn aObj );

void         showNewImage  ( HImage * img, char * name, int xx, int yy );
```

```
// Información de las regiones a tener en cuenta (centro, borde, etc.)
//
HLabelInfo labelInfo[5];

};// End class HMaster
```

Ahora veamos la interface de la clase HClient, que es la que se relaciona con la clase anterior:

```
class HClient
{
public:

// Constructor y destructor
//
HClient ();
~HClient ();

// Procesamiento
//

// Este método es el que realiza todo el procesamiento
//
HClient&     process ();

// Alocación de imágenes
//
HClient&     allocImages   ();
HClient&     freeImages    ();

HImage*      srcImage   ();    // Retorna un puntero a la imagen original
HImage*      procImage  ();    // Retorna un puntero a la imagen de proceso

enum OperType { opMainObject, opManchas };

void    recParams    ();        // Recibe los parámetros de procesamiento
                                // del master
void    recImage     ();        // Recibe el pedazo de imagen a procesar
                                // del master
void    sendImage    ( HImage * img );    // Envia la imagen procesada
                                          // al master
// Envia la tabla de de labeling local al master
//
void    sendTable    ( HLabeling& lblAlgo, HImage * );

// Recibe la tabla de labeling modificada del master
//
void    recTable     ( HLabeling& lblAlgo, OperType op );

void    calcPromedio ( HCalcColor& );     // Calcula el color promedio parcial
```

```
void    calcCenter    ( HCalcCenter& );   // Calcula el centroide parcial

//Getters & Setters
//
int         thValue         ();         // Valor límite para el threshold
HClient&    setThValue      (int v);

Int         rangeMancha         (); // Diferencia de color para considerar a un pixel
                                   // como mancha
HClient&    setRangeMancha      (int v);

Int         startY          ();         // Numero de linea absoluto de la primera linea
                                        // de la imagen local. Sirve para poder calcular
                                        // las coordenadas absolutas de los pixels.
Int         flags           ();         // Parametros extra (Primer proceso, último, etc.)

bool        showParcialImgs     (); // Flag que indica si se deben enviar al master
                                    // las imágenes intermedias.


HPvmProxi& proxi            ();         // Proxi para comunicarse con el master.

}; // End class Hclient
```

El esquema de la aplicación seria como sigue:

```
int Master_Main () {

  Obtener una imagen ...
  Averiguar el valor de los parámetros de operación ...

  HMaster master;

  master  .setSrcImage          ( imagen )
          .setThValue           ( thValue    )
          .setProcsNumber       ( procNumber  )
          .setRangeMancha       ( rangeMancha )
          .setManchaSize        ( (double)manchaSize / 10.00 )
          .setCalcExen          ( (bool)calcExen )
          .setShowParcialImgs   ( (bool)debugImg )
          .setShowImgFn         ( showNewImage )
          .process              ();

  if ( master.errorCode() != HMaster::ErrOk ) {
     Mostrar mensaje de error ...
     return 0;
  }

} // End Master_Main
```

y con este código se genera un ejecutable. Por otro lado, tenemos al proceso client, que también debe ser un ejecutable ya que justamente es ejecutado por el master en el programa anterior. Este ejecutable es muy simple:

```
void Client_Main ()
{
  HClient client;
  client.process ();
} // end Client_Main
```

A.5 - Clases para abstraer los procesos PVM

Como mencionamos anteriormente, para mantener la aplicación portable y poder cambiar la librería de comunicaciones sin necesidad de modificar todo del código, decidimos encapsular las llamadas a PVM en clases.

Básicamente hay dos tipos de procesos, con requisitos de comunicación distintos. Por un lado esta el proceso **master**, que conoce a un conjunto de procesos **client** con los cuales se comunica, y por otro lado, están los procesos **client**, que solo interactúan con el master que los instanció.

Para modelar estos dos casos, creamos una clase llamada **HPvmProxi**, que representa a un proceso remoto. El master ve a cada client como una instancia de esta clase y a su vez los procesos client tienen una sola instancia de esta clase que representa al master.

Esta clase tiene los métodos necesarios para comunicarse con el proceso remoto, y en particular guarda el tid del mismo para poder así enviarle y recibir mensajes del mismo.

Como PVM tiene funciones para enviar distintos tipos de datos (ints, longs, floats, etc.), esta clase tiene métodos **send** y **recv** sobrecargados con una implementación particular para cada tipo de datos.

Por otro lado, definimos una clase llamada HPvmProcessArray que básicamente administra un número de proxis para facilitar la implementación del master.

Veamos a continuación las interfaces de estas dos clases:

```
class HPvmProxi
{
public:

// Costructores y destructor
//
HPvmProxi  ( char * n = NULL, bool ak = false );
~HPvmProxi ();


int             tid     ();             // tid del proceso remoto
HPvmProxi&      setTid  ( int aObj );

char*           name    ();             // Nombre del proceso remoto
```

```
HPvmProxi& setName    ( char* aPtr );

int               status      ();          // Estado del proceso remoto
HPvmProxi&        setStatus   ( int aObj );

//
// Operaciones de PVM
//

// Ejecutar el proceso remoto
//
HPvmProxi& spawn ( char ** argv = NULL, int flag = PvmTaskDefault,
                   char * where = NULL );

// Finalizar el proceso remoto
//
HPvmProxi& kill  ();

// Envio de mensajes (sobrecargados)
//
HPvmProxi& send ( uchar *   buff,  int count = 1, int tag = 1 );
HPvmProxi& send ( int *     buff,  int count = 1, int tag = 2 );
HPvmProxi& send ( short *   buff,  int count = 1, int tag = 3 );
HPvmProxi& send ( ushort *  buff,  int count = 1, int tag = 4 );
HPvmProxi& send ( double*   buff,  int count = 1, int tag = 5 );

// Recepción de mensajes (sobrecargados)
//
HPvmProxi& recv  ( uchar *   buff,  int count = 1, int tag = 1 );
HPvmProxi& recv  ( int *     buff,  int count = 1, int tag = 2 );
HPvmProxi& recv  ( short *   buff,  int count = 1, int tag = 3 );
HPvmProxi& recv  ( ushort *  buff,  int count = 1, int tag = 4 );
HPvmProxi& recv  ( double *  buff,  int count = 1, int tag = 5 );

}; // End class HPvmProxi
```

```
class HPvmProcessArray
{
public:

// Costructores y destructor
//
HPvmProcessArray ( int numProcs = 0, char * n = NULL,
                   bool as = false, bool ak = true );
~HPvmProcessArray ();

// Getters & Setters
//
int               numProcs     ();          // Número de procesos
HPvmProcessArray& setNumProcs  ( int aObj );
```

```
char*                name        ();              // Nombre de los procesos
HPvmProcessArray&    setName     ( char* aPtr );

//
// Operaciones de PVM
//

// Ejecutar todos los procesos del array
//
HPvmProcessArray& spawnAll (  char ** argv = NULL, int flag = PvmTaskDefault,
                              char * where = NULL );

// Terminar todos los procesos del array
//
HPvmProcessArray& killAll  ();

// Retorna el proxi número index
//
HPvmProxi& proxi ( int index ) { return proxies()[index]; }

};// End class HPvmProcessArray
```

En el caso del master, estas clases pueden utilizarse de la siguiente manera:

```
// El master tiene un HPvmProcessArray de clientes, supongamos
// que son cinco, y lo instancia de la siguiente forma:
//
HPvmProcessArray proxis ( 5, "client" );

// Luego los instancia
//
proxis.spawnAll ();

// Y luego se puede comunicar con los mismos de la siguiente forma:
//
int * buff;  // Datos a mandar ...
int buffLen;   // Cantidad de datos a mandar

// Lleno el buffer ...
// ...

// Envio a todos los procesos client
//
for ( int ii=0; ii < proxis.numProcs (); ii++)
  proxis(ii).send ( buff, buffLen );

// ....
```

Del lado del cliente, el procedimiento es como sigue:

```
// Creo un HPvmProxi
//
HPvmProxi proxi ();

// Le seteo el tid de mi parent, que es el master ...
//
proxi.setTid ( pvm_parent () );

// Y ya puedo comunicarme. Recibo un arreglo de enteros ...
//
int buff [ BUFF_LEN ];

proxi.recv ( buff, BUFF_LEN );

// ...
```

A.6 - Plug-in de Gimp

Como mencionamos en la descripción de la aplicación, para la versión gráfica de la misma utilizamos la facilidad de plug-ins dada por la librería Gimp para el desarrollo. A continuación mostramos el esqueleto de un plug-in, desde el cual, en la aplicación, se instancia un objeto del tipo HMaster para realizar el procesamiento:

```
// Inclusion de la API
//
#include "gimp.h"

int main (int argc, char **argv)
{
  Image input, output;
  prog_name = argv[0];

  // Inicializa a gimp para que se puedan recibir y mandar las imagenes
  //
  if (gimp_init (argc, argv)) {
      input = output = 0;
      input = gimp_get_input_image (0);

      // Solo procesamos si la imagen es RGB
      //
      if (input) {
          switch (gimp_image_type (input)) {
              case RGB_IMAGE:
                  output = gimp_get_output_image (0);
```

```
            // Obtenemos los parametros
            //
            int dialogID = paramDialog();
            gimp_show_dialog ( dialogID );

            // Procesamos la imagen Input y guardamos el resultado en output ..
            //.
            procesar ( input, output )

            // Mosramos el resultado final
            //
            dialogID = resultDialog ();
            gimp_show_dialog ( dialogID );

            gimp_update_image (output);
            break;

        default:
            gimp_message ("Tesis: La imagen no es RGB. ");
            break;
      }
    }

    // Liberación la memoria de las imagenes.
    //
    if (input)   gimp_free_image (input);
    if (output)  gimp_free_image (output);
  }

  // Finalizar.
  //
  gimp_quit ();
  return 0;
}
```

La función procesar se implementa de la forma en que se describe la instanciación de un proceso master en el punto A.3 en el proceso Master_Main.

Para la generación de diálogos de interface con el usuario, mostramos a continuación la porción del código usada para generar el dialogo de entrada de parámetros de procesamiento:

```
// Variables en las que se guardan las elecciones del usuario
//
static int thValue;          // Valor de threshold
static int procNumber;       // Cantidad de procesos a usar
static int rangeMancha;      // Variación del color de las manchas
static int manchaSize;       // Tamaño de las manchas a considerar
static int calcExen;         // Si hay que calcular los momentos
static int debugImg;         // Si hay que mostrar resultados parciales
```

```
int paramDialog ()
{
  int dialog_ID;
  int scale1_ID, scale2_ID, scale3_ID, scale4_ID;
  int group_ID, exen_ID, debug_ID;
  int frame1_ID, frame2_ID, frame3_ID, frame4_ID;

  dialog_ID = gimp_new_dialog ("Parametros");
  group_ID  = gimp_new_row_group (dialog_ID, DEFAULT, NORMAL, "");

  frame1_ID = gimp_new_frame (dialog_ID, group_ID, "Threshold");
  frame2_ID = gimp_new_frame (dialog_ID, group_ID, "Procesos");
  frame3_ID = gimp_new_frame (dialog_ID, group_ID, " + - Manchas");
  frame4_ID = gimp_new_frame (dialog_ID, group_ID, "Tama#o de manchas (%)");

  exen_ID = gimp_new_check_button (dialog_ID, group_ID, "Calcular
                                                  Excentricidad");
  debug_ID = gimp_new_check_button (dialog_ID, group_ID, "Mostrar parciales");

  scale1_ID = gimp_new_scale (dialog_ID, frame1_ID, 1,   255, thValue,     0 );
  scale2_ID = gimp_new_scale (dialog_ID, frame2_ID, 1,    10, procNumber,  0 );
  scale3_ID = gimp_new_scale (dialog_ID, frame3_ID, 1,   100, rangeMancha, 0 );
  scale4_ID = gimp_new_scale (dialog_ID, frame4_ID, 1,  100, (int)manchaSize,  1 );

  gimp_add_callback (dialog_ID, scale1_ID, scale_callback, &thValue    );
  gimp_add_callback (dialog_ID, scale2_ID, scale_callback, &procNumber );
  gimp_add_callback (dialog_ID, scale3_ID, scale_callback, &rangeMancha);
  gimp_add_callback (dialog_ID, scale4_ID, scale_callback, &manchaSize );
  gimp_add_callback (dialog_ID, exen_ID,  toggle_callback, &calcExen   );
  gimp_add_callback (dialog_ID, debug_ID, toggle_callback, &debugImg   );


  gimp_add_callback (dialog_ID, gimp_ok_item_id (dialog_ID), ok_callback, 0);
  gimp_add_callback (dialog_ID, gimp_cancel_item_id (dialog_ID), cancel_callback,
0);
  return dialog_ID;
}

// Estos son las funciones de callback usadas por los controles
//

// Callback para los scale
static void scale_callback (int , void * client_data, void * call_data)
{
  *((long*) client_data) = *((long*) call_data);
}

// Callback para los toggle buttons
static void toggle_callback (int, void * client_data, void * call_data)
{
  *((long*) client_data) = *((long*) call_data);
}
```

```
// Callback para el boton de OK de los dialogos
static void ok_callback (int, void *, void *)
{
  gimp_close_dialog (dialog_ID, 1);
}

// Callback para el boton cancel de los dialogos
static void cancel_callback (int, void * , void * )
{
  gimp_close_dialog (dialog_ID, 0);
}
```

Los valores elegidos por el usuario quedan almacenados en las variables definidas al principio del código.

De forma muy similar se genera el diálogo de muestra de resultados.

La compilación de un plug-in es también sencilla, solo hace falta linkear una librería llamada libgimp.lib.

Para que el plug-in aparezca en la lista de la aplicación, es necesario agregar una linea en el archivo .gimprc, que se ubica en el directorio home (~) del usuario que ejecuta la aplicación, con el siguiente formato:

plugin <Nombre_del_ejecutable> <nombre_del_menu>

Por ejemplo, para la tesis: plugin pvmapp "procesar"

Imágenes

Este apéndice contiene las fotografías utilizadas para las pruebas de pérformance de la aplicación.

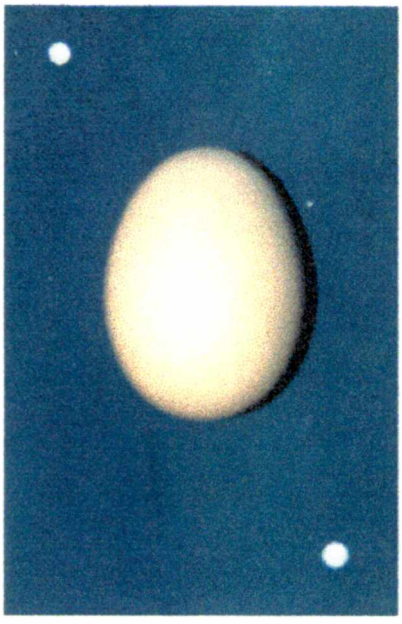


Imagen 2 - Foto de un huevo, donde se pueden observar las marcas de referencia(res. 522 x 804).

Imagen 3 - Foto de una naranja. Se utilizó en el quinto lote de prueba, para lo que se agregaron manchas artificiales similares a las de la Imagen1 (res.929 x 973).

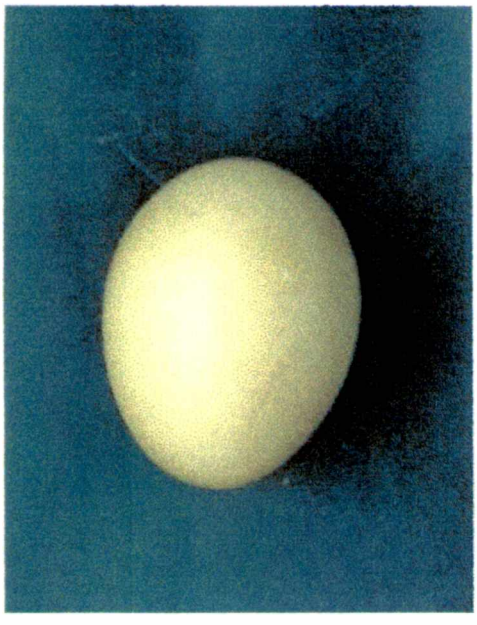


Imagen 4 - Foto de un huevo utilizada en la pruebas iniciales.

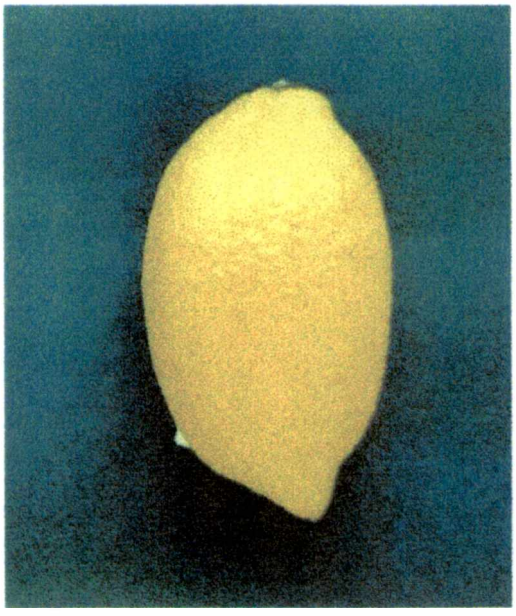


Imagen 5 - Foto de un limón utilizada en la pruebas iniciales.

Imagen 1 - Foto de un huevo manchado artificialmente (resolución 264 x 384).

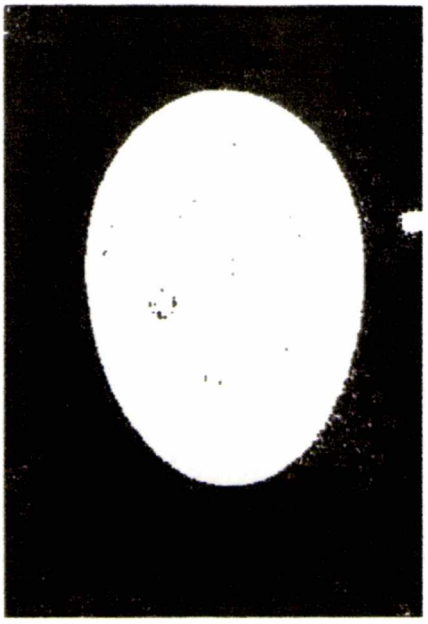


Imagen 1.1 - Resultado de la aplicación del alg. de threshold a la imagen 1 (thresh=150).

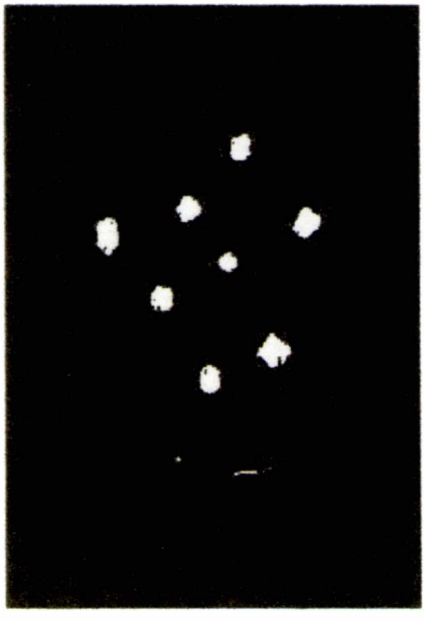


Imagen 1.2 - Resultado de la aplicación del alg. de threshold especial a la imagen 1.

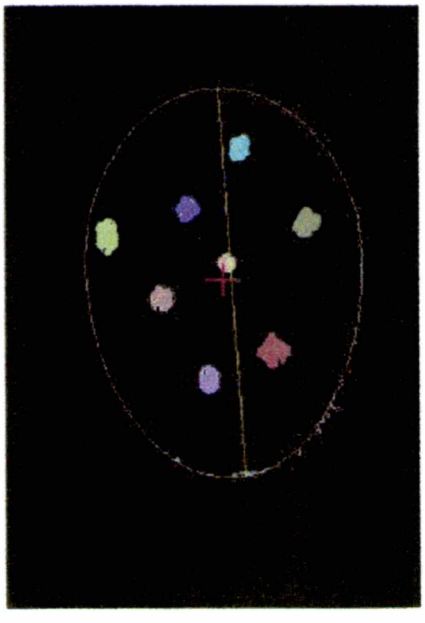


Imagen 1.3 - Resultado final donde se puede observar el borde, el centroide y el perímetro, y cada una de las manchas encontradas pintadas de un color distinto.

Bibliografía:

- Zahid Hussain: "Digital Image Processing", Ellis Horwood Limited, 1991.
- Gregory A. Baxes: "Digital Image Processing", John Wiley, 1994.
- Rafael C. Gonzáles, Richard E. Woods: "Digital Image Processing", Addison-Wesley Publishing Comp., 1992.
- M. Coffin: "Parallel programming- A new approach", Prentice Hall, Englewood Cliffs, 1992.
- Haralick Shapiro: "Computer and Robot Vision", 1994.
- Kai, Hwang: "Advaned Computer Architecture: Paralelism, Scalability, Programability", Mac Graw – Hill, 1993.
- J. Foley, A. Van Dam, S. Feiner, J. Hughes: "Computer Graphics", Addison-Wesley Publishing Comp., 1990.
- D. W. Hermann, A. N. Burkitt: "Parallel Alorihms in Computational Sience", Springer-Verlag, 1991.
- H. Lawson: "Parallel processing in industrial real time applications", Prentice - Hall 1992.
- ¨Parallel Virtual Machine¨, World Wide Web.
- M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra: ¨MPI: The Complete Reference¨, The MIT Press, 1996.
- "The Linux Network Administrator's Guide", Olaf Kirch, 1994.
- "OSF/Motif Programer's Guide", Open Software Foundation, 1994.
- "OSF/Motif Widget Writer's Guide", Open Software Foundation, 1994.
- Douggherty, Edward: "Matrix structured image processing", Prentice – Hall, 1987.
- F. Thomson Leighton: "Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes": Morgan Kaufmann Publishers, 1992.