UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

FACULTAD DE INGENIERIA

PROCESAMIENTO PARALELO DE IMAGENES ULTRASONICAS PARA DIAGNOSTICO MEDICO

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

MANLIO FABIO VALDIVIESO CASIQUE

DIRECTOR: DR. JULIO SOLANO GONZALEZ

MEXICO, D. F. AGOSTO DE 1996

DEDICATORIA:

A mis padres: Ricardo y Quintina, Jose Luis y Minerva, porque me han brindado todo lo que un ser humano puede necesitar: comprensión, apoyo, ejemplo, pero, sobre todo, cariño.

A mis hermanos y primos, por inyectarme el esfuerzo y las ganas de seguir adelante.

A Alicia, por estar siempre a mi lado.

A todos mis amigos.

Manlio Fabio Valdivieso Casique
México, D.F. a 23 de agosto de 1996

AGRADECIMIENTOS:

Al Dr. Julio Solano González por la dedicación y empeño que mostró en la realización de este trabajo.

A la Facultad de Ingeniería por haberme albergado durante estos cinco años y proporcionado conocimiento y altos valores.

Al Departamento de Electrónica y Automatización del IIMAS por todo el apoyo otorgado.

A DGAPA (Proyecto PAPIID-D0303593) y CONACYT (Proyeco 2146P-A9507 y PACIME-F284-A9209) por haber proporcionado los recursos para la elaboración de este trabajo.

ÍNDICE

1

INTRODUCCIÓN

1.1 GENERALIDADES

Los sistemas convencionales de cómputo operan en forma secuencial donde las instrucciones de un programa son ejecutadas una a la vez (arquitecturas Von Neumann). Este tipo de arquitecturas están compuestas por un procesador central conectado a un banco de memoria por medio de un bus [5], lo que provoca que el rendimiento de los sistemas dependa de la velocidad del mismo.

Muchas de las aplicaciones desarrolladas hoy en día requieren de una mayor cantidad de procesamiento, tal es el caso del procesamiento de imágenes utilizado en tomografía ultrasónica, reconocimiento de patrones, control, etc. Es por ello que han surgido ideas que consideran nuevas arquitecturas de computadoras para satisfacer esa demanda. Una de esas alternativas es la de utilizar más de un procesador para la realización de una aplicación específica, nace entonces la idea del procesamiento paralelo, que no es otra cosa que un grupo de procesadores trabajando en conjunto para realizar una tarea especial de forma más rápida y eficiente [1].

El procesamiento paralelo es una alternativa para la solución de aplicaciones que requieren de una respuesta en tiempo real, éstas exigen un alto grado de esfuerzo computacional que no podrían ser resueltas por un sistema uniprocesador. Un ejemplo claro de este tipo de aplicaciones es la que se está llevando a cabo en el Departamento de Electrónica y Automatización (DEA) del Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS) en coordinación con el Instituto de Cibernética, Matemática y Física de Cuba, y que consiste en el desarrollo de un tomógrafo ultrasónico computarizado de alto desempeño y bajo costo.

El proyecto está conformado propiamente por tres fases: adquisición, procesamiento del tomograma y despliegue del mismo. Las tres fases anteriormente mencionadas se encuentran muy entrelazadas. En la primera fase se adquieren los datos de cada una de las imágenes mediante la reflexión de una señal que excita los sensores que conforman el sistema de adquisición, estos datos son amplificados y finalmente transformados a un nivel digital [3]. Se reciben en total 56 vectores cada uno con 256 elementos, lo que nos lleva a conformar un tomograma que

consta de 56 columnas por 256 renglones. Cada elemento o pixel de la imagen tiene asociado un byte para brindar imágenes con 256 tonos de gris, desde el negro (nivel 0) al blanco (nivel 255).

La segunda fase, la del procesamiento, se omitirá por un momento para comentar la última etapa, el despliegue. El despliegue de información debe realizarse en un monitor con una resolución estándar, tal es el caso de la resolución de 640*480 pixeles, para crear las condiciones de visualización a las cuales se encuentran habituados los especialistas [3].

De acuerdo a la resolución del dispositivo de despliegue y al tamaño de la imagen, es necesario un agrandamiento de la misma para que sea más perceptible al observador, aquí encontramos la fase de procesamiento. Interpolando la imagen por un factor de 3 sobre las columnas se puede lograr que la imagen se ajuste a la resolución del monitor. Se tendrá entonces un tomograma con una cantidad de información triplicada de la original, es decir, de 168 columnas por 256 renglones.

Ahora bien, se pretende además que el sistema opere en tiempo real, es decir, que se capturen y desplieguen al menos 25 cuadros por segundo [4]. Cada cuadro consta de 168*256*1 = 43008 bytes lo que nos arroja un total de 25*43008*1 = 1.075200 Mbytes que deben ser procesados por segundo. Como se puede ver es una cantidad enorme de información a ser procesada y, de acuerdo al algoritmo de interpolación utilizado, se incrementará también el número de operaciones por cada nuevo dato interpolado. Por tanto, en este trabajo se utiliza una red de procesadores trabajando en forma paralela para resolver el problema.

La implementación original del sistema fue realizada utilizando un algoritmo de orden cero o replicación de pixeles, este algoritmo es claramente simple y requiere de poco procesamiento, pero tiene la desventaja de que la imagen interpolada es un poco burda [2]. Es por ello necesaria la implementación de algoritmos de orden mayor que, aún utilizando más datos y aumentando su complejidad computacional, proporcionen una mejor calidad de la imagen.

A continuación se listan los objetivos del presente trabajo de tesis:

1.2 OBJETIVOS

1.- Diseño e implementación de un sistema de procesamiento paralelo para la interpolación de imágenes ultrasónicas.

2.- Lograr la interpolación de las imágenes en tiempo real, es decir, al menos el procesamiento de 25 cuadros por segundo.

3.- Estudio del desempeño del sistema y de la calidad de las imágenes obtenidas mediante los diferentes algoritmos de interpolación utilizados.

1.3 CONTENIDO

El presente trabajo de tesis está organizado de la siguiente manera:

Capítulo 1: en este capítulo se plantea, en términos generales, las razones que motivaron el presente trabajo de tesis, los objetivos y el contenido del mismo.

Capítulo 2: se plantean las bases del procesamiento paralelo, se muestra el tipo de procesador utilizado en la aplicación y el lenguaje C paralelo utilizado para el desarrollo del sistema.

Capítulo 3: en este capítulo se describen varios de los algoritmos de interpolación de imágenes digitales más comunes y eficientes: interpolación lineal, splines e interpolación por convolución cúbica.

Capítulo 4: aquí se mencionan las diferentes técnicas para explotar el paralelismo de los algoritmos, así como su implementación mediante el proceso farming en una red de procesadores bajo diferentes topologías.

Capítulo 5: en este capítulo se analizan los resultados obtenidos para los diferentes algoritmos de interpolación, además de un análisis de la calidad de las imágenes interpoladas.

Capítulo 6: por último, en este capítulo se exponen las conclusiones generales de este trabajo y se explica, en forma breve, el trabajo que se puede realizar en un futuro para mejorar la implementación.

Se incluyen además dos apéndices con información adicional.

Apéndice A: se proporciona un diagrama a bloques de la arquitectura del transputer utilizado en esta aplicación.

Apéndice B: se muestran los programas de aplicación desarrollados para implementar los algoritmos de interpolación de imágenes en el sistema diseñado.

1.4 REFERENCIAS

[1] DE CARLINI, U Y DE VILLANO, U. **TRANSPUTERS AND PARALLEL ARCHITECTURES**. MIT PRESS, 1991.

[2] GONZÁLEZ,R. Y WOODS,R. **DIGITAL IMAGE PROCESSING**. ADDISON WESLEY, 1992.

[3] INSTITUTO DE CIBERNÉTICA, MATEMÁTICA Y FÍSICA DE LA HABANA. **MANUAL DEL SOFTWARE DE APLICACIÓN PARA TOMÓGRAFO ULTRASÓNICO**.

[4] LOW, A. **INTRODUCTORY COMPUTER VISION AND IMAGE PROCESSING.** Mc. GRAW-HILL, 1993.

[5] MORRIL, J. **PARALLEL PROCESSING**. REVISTA BYTE 1988. VOL. 13. NO. 11. P.P. 287-300. 1988.

2

PROCESAMIENTO PARALELO

2.1 INTRODUCCIÓN

El propósito de este capítulo es el de proporcionar al lector los antecedentes necesarios para una buena comprensión y apreciación de los capítulos posteriores.

Se hace alusión aquí a una nueva filosofía de cómputo -el procesamiento paralelo- que sirvió para el desarrollo del presente trabajo; se describe además el procesador utilizado en la aplicación paralela (Transputer) así como sus lenguajes de programación asociados (OCCAM y C), pasando a su vez por los conceptos fundamentales de las arquitecturas paralelas y un análisis de las métricas de desempeño que nos permiten determinar el grado de eficiencia de cada implementación.

2.2 PROCESAMIENTO PARALELO

La evolución de las arquitecturas de computadoras ha sido muy activa desde que John Von Neumann construyó su diseño basado en un solo procesador y unidades de memoria ligados por un bus de datos en 1946. En este diseño las instrucciones y datos son transportados por el procesador desde la memoria usando el bus de datos mencionado, estos datos son procesados y luego puestos de nueva cuenta en memoria por el mismo bus. Este ciclo de lectura de datos e instrucciones y de almacenamiento operan secuencialmente durante la ejecución del programa.

La mayoría de las computadoras de hoy en día están basadas en esta arquitectura, sin embargo, muchos problemas de la vida real tales como el procesamiento de imágenes y de señales, simulación y visión poseen un paralelismo inherente. Resolver estos problemas utilizando una computadora secuencial requeriría mucho tiempo de procesamiento, es por ello que estos problemas deben ser resueltos en forma paralela para aprovechar su paralelismo natural [6].

En cuanto al procesamiento de imágenes, la demanda de aplicaciones en tiempo real ha contribuido al uso y desarrollo del procesamiento paralelo, aplicaciones como interpolación, síntesis de imágenes, etc, las cuales requieren de

altos niveles de cómputo, serían irrealizables si se considerara su ejecución en un solo procesador (aún cuando este fuese muy rápido) operando en forma secuencial, mientras que combinando los recursos de varios procesadores se puede incrementar el rendimiento del sistema [8].

Por otro lado, la velocidad de los componentes de computadoras se encuentra muy cerca de los límites físicos impuestos por la velocidad de la luz. Existe una creencia de que los problemas de cómputo intensivo jamás serán resueltos incrementando el rendimiento de computadoras individuales y que la única solución es basarnos en el viejo dicho de "divide y vencerás", es decir, dividir nuestro problema en un número determinado de tareas y que estas sean resueltas en forma independiente por varios procesadores.

Se puede definir al procesamiento paralelo como la actividad de varias entidades (idénticas o heterogéneas), trabajando en conjunto para lograr un fin común. Las entidades son los procesadores y un fin puede ser lograr una de las actividades descritas líneas arriba.

Las arquitecturas de computadoras representan un árbol con muchas ramas y hojas. Las computadoras paralelas representan la parte más verde del árbol ya que es esta el área donde se están invirtiendo los esfuerzos de muchos diseñadores de computadoras ya que ofrecen un alto rendimiento y bajo costo. Para una mayor comprensión a continuación se expone la clasificación de las computadoras paralelas.

2.2.1 CLASIFICACIÓN DE LAS COMPUTADORAS PARALELAS

La clasificación que estableció Flynn en 1972 no comprende con exactitud la diversidad de las computadoras modernas pero es todavía muy popular ya que impone un orden y mantiene el nivel de simplicidad que es requerido de acuerdo a la rica diversidad de las arquitecturas recientes. Flynn observó que varios tipos de arquitecturas de computadoras paralelas pueden ser distinguidas por la relación entre las instrucciones ejecutadas en una máquina y por los datos en los que esas instrucciones operan. Un flujo (stream) de instrucciones fue definido como una

secuencia de instrucciones ejecutadas por un procesador, mientras que un stream de datos es la secuencia de datos a ser operados por las instrucciones. Flynn clasificó las arquitecturas de computadoras de acuerdo a la multiplicidad de dichos streams, resultando las cuatro ramas siguientes [8]:

Flujo único de instrucciones, flujo único de datos (SISD: Single Instruction stream, Single Data stream). Este tipo de arquitectura está representada por las computadoras secuenciales convencionales, como se mencionó párrafos atrás, las que contienen el diseño de Von Neumann. En este modelo, en cualquier tiempo dado, solo un dato puede ser operado a la vez y esta operación es definida solamente por una instrucción.

En un principio el paralelismo fue introducido en este modelo para ganar rendimiento. Los primeros sistemas en los que apareció fue en mecanismos de look ahead cuya meta era la de tratar de encimar el ciclo de fetch y el de ejecución. Pronto este método generalizó en el *pipelining* (explotación del paralelismo en tiempo) y operaciones de vectores (explotando paralelismo espacial) donde una instrucción puede operar sobre muchos datos.

Flujo único de instrucciones, flujo múltiple de datos (SIMD: Single Instruction stream, Multiple Data stream). Aquí se tienen computadoras con una unidad de control, recibiendo una instrucción, la cual dirige muchas unidades de procesamiento , cada una realizando las acciones requeridas por la unidad de control sobre los diferentes datos. Ejemplos de estas computadoras son los arreglos de procesadores (arrays) y las CRAY's.

Flujo múltiple de instrucciones, flujo único de datos (MISD: Multiple Instruction stream, Single Data stream). En esta categoría un número de procesadores ejecutan distintos flujos de instrucciones en el mismo flujo de datos. No existe alguna realización práctica de este tipo de arquitectura.

Flujo múltiple de instrucciones, flujo múltiple de datos (MIMD: Multiple Instruction stream, Multiple Data stream). Esta clase contiene la mayoría de los sistemas multiproceso y multicomputadoras. Los sistemas MIMD consisten en distintos procesadores los cuales son autónomos y operan en forma asíncrona, cada uno en

un conjunto distinto de datos sin mecanismos de control. Diferentes instrucciones son ejecutadas simultáneamente en diferentes procesadores y actuando sobre diferentes datos con el requerimiento de que todos ellos contribuyan a la solución de un problema común.

Las arquitecturas MIMD nos llevan al terreno de los multiprocesadores y las multicomputadoras, a pesar de que estos dos términos han sido usados invariablemente, la convención aceptada es que un multiprocesador se refiere a una máquina donde muchos procesadores tienen acceso a una memoria común compartida, mientras que en las multicomputadoras la memoria es distribuida, cada procesador tiene acceso sólo a su propia memoria [15].

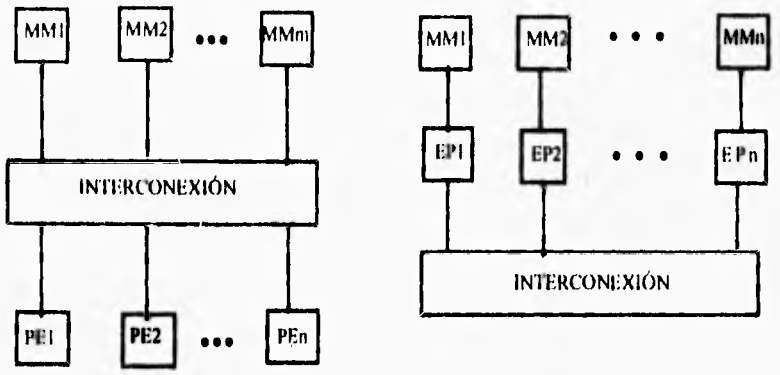
Para un mayor entendimiento de lo anterior, se describen los sistemas de memoria distribuida y los de memoria compartida.

2.2.2 SISTEMAS DE MEMORIA DISTRIBUIDA Y COMPARTIDA

Un sistema de memoria distribuida es aquel que permite que cada procesador perteneciente al sistema tenga acceso a su propia memoria, la comunicación entre estos procesadores es lograda por el paso de un mensaje entre los elementos que desean comunicarse.

Por otro lado, los sistemas de memoria compartida permiten que los procesadores tengan acceso a un espacio de memoria común. Los procesadores se comunican mediante la escritura y lectura a una sección de la memoria compartida [8].

Estos dos tipos de arquitecturas MIMD pueden ser conceptualizados como se ilustra en la figura 2.1. Los MM son módulos de memoria y los EP elementos de procesamiento. Los módulos de memoria pueden ser considerados como una partición del espacio total direccionable en el sistema MIMD.



a) Memoria compartida b) Memoria distribuida

Figura 2.1 Arquitecturas MIMD

En el sistema de memoria compartida (figura 2.1 (a)), el número de elementos de procesamiento y módulos de memoria puede no ser igual. La red de interconexión permite conectar cualquier EP a cualquier MM. Para los sistemas de memoria distribuida (figura 2.1 (b)), el número de elementos de procesamiento y el de módulos de memoria es el mismo, cada MM es direccionado por un solo EP. Si alguno de los elementos de procesamiento requiere de datos almacenados en la memoria de otro, entonces el primero deberá enviarle un mensaje de petición a través de la red de interconexión.

Un ejemplo de multicomputadora basada en la arquitectura de memoria distribuida es el transputer, el cual será definido a continuación.

2.3 EL TRANSPUTER

El transputer es una de las máquinas MIMD antes descritas. Pertenece, como ya se dijo, a la arquitectura de memoria distribuida o de paso de mensajes. El paso de mensajes es la técnica que los procesadores deben usar para intercambiar información con otro, y, debido a que los accesos remotos a memoria no son posibles son llamadas también máquinas sin acceso remoto a memoria o NORMA (NO Remote Memory Access) [15].

El Transputer nace en 1985 cuando la compañía INMOS introduce un nuevo concepto en circuitos de muy alta escala de integración (VLSI: Very Large Scale of Integration) : un solo circuito conteniendo un procesador, memoria local y cuatro puertos seriales de entrada-salida.

El circuito era una computadora en sí, conteniendo como se dijo, un procesador, alguna memoria para almacenar programas y datos y varios puertos para intercambio o transferencia de información con otro transputer o con el mundo exterior. Se diseñaron entonces estos circuitos para que pudieran ser conectados entre sí con la misma facilidad con la que los transistores pueden ser conectados en una computadora, de ahí su nombre: **Trans**istor-Com**puter** [15].

El Transputer soporta explícitamente concurrencia y sincronización (sus mayores características). La sincronización es requerida cuando diferentes partes de un sistema, cada una con diferentes tiempos relativos de ejecución, necesitan interactuar o cooperar con otra de alguna manera; un ejemplo muy sencillo de lo anterior es tratar de comunicarlos, es decir, enviar un mensaje de un proceso a otro. Cada parte a interactuar debe estar preparada para que la interacción tome lugar.

El Transputer es en realidad una familia de microprocesadores relacionados pero con diferentes capacidades.

Por ejemplo,

IMS T212	Procesador de 16 bits
IMS T414	Procesador de 32 bits
IMS T800	Procesador de 32 bits con unidad de punto flotante.

El Transputer utilizado durante el desarrollo de esta tesis es el T805 (ver apéndice A), perteneciente a la familia T800. Este es un procesador de 32 bits con una unidad de punto flotante de 64 bits y cuatro links de comunicación serial, 4Kbytes de memoria y una interfaz de memoria externa, todo dentro del mismo chip lo que lo hace una computadora en un solo chip [6].

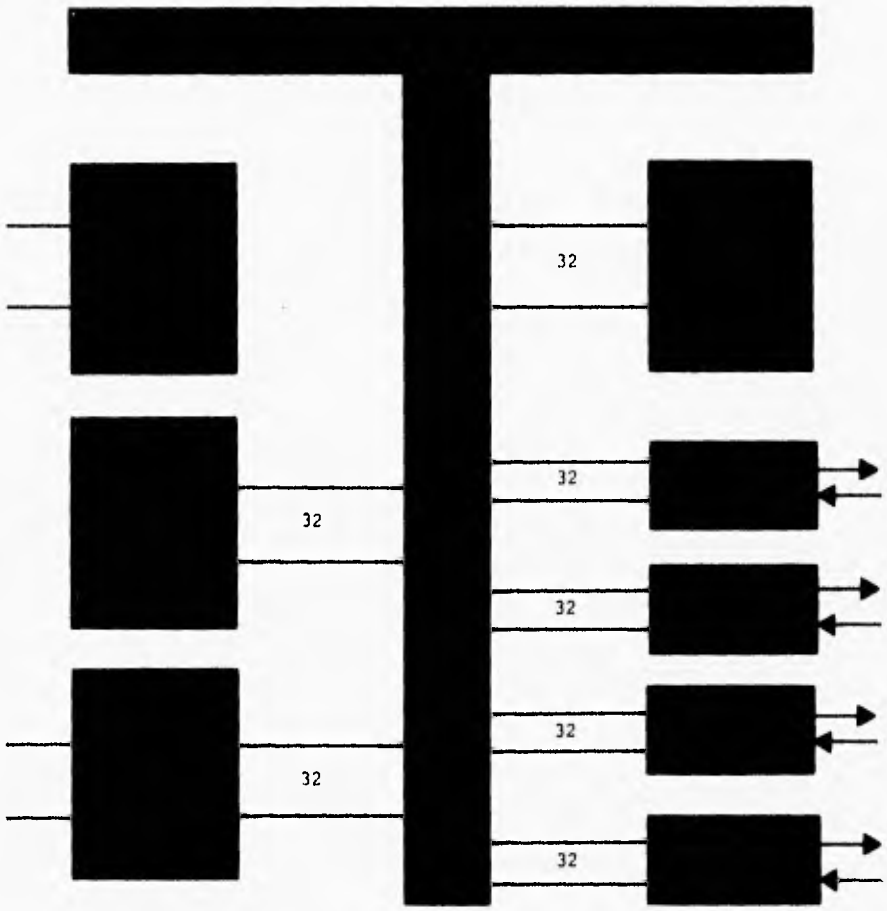


Figura 2.2 Diagrama a bloques del transputer T805.

El procesador corriendo a 30 MHz puede realizar 30 MIPS (millones de instrucciones por segundo), un conjunto reducido de instrucciones y un pequeño número de registros de 32 bits fueron diseñados para ejecutar los programas a mayor velocidad. Contiene un programador de tareas (scheduler) el cual es el encargado de distribuir tiempo de ejecución del procesador entre los diferentes procesos concurrentes. De acuerdo a lo anterior, un transputer admite no solo la ejecución de programas en forma secuencial sino que también admite procesos concurrentes.

En este momento es importante remarcar la diferencia entre paralelismo y concurrencia, el primero de ellos se refiere a la ejecución en el mismo instante de dos o más procesos corriendo cada uno de ellos en un procesador distinto. Por otro lado, la concurrencia se realiza cuando varios procesos se ejecutan en un solo procesador, dando la impresión de realizarse en paralelo pero solo comparten tiempo del procesador, así, mientras uno está activo los demás esperan su turno de ejecución.

Se considera necesario señalar cómo se realiza la concurrencia en el transputer, este procesador soporta concurrencia por medio del scheduler antes mencionado, el cual comparte el tiempo entre los procesos concurrentes y que ahora se describe.

2.3.1 EL SCHEDULER (PROGRAMADOR DE TAREAS)

En cualquier momento un proceso concurrente puede estar:
- Activo: corriendo en ese instante o listo para ser ejecutado.
- Inactivo: esperando un evento.

Eventos típicos son la terminación de una operación de entrada o salida o la toma de tiempo del procesador.

Mientras el scheduler esté activado selecciona la ejecución de un proceso de un conjunto de procesos activos. La elección de un nuevo proceso está basada de acuerdo a la prioridad de cada uno de ellos; en el caso de que dos o más tengan la misma prioridad se utiliza una técnica FIFO (First Input First Output : Primero en Entrar Primero en Salir), es decir, de acuerdo al primer proceso que haya realizado la solicitud, éste se ejecutará primero. Sólo dos niveles de prioridad son establecidos y la estructura de datos del scheduler consiste en dos colas, cada una de estas colas es una lista ligada conteniendo cada una las direcciones de memoria de los procesos listos para ejecutarse, tanto la cabeza como el final de la cola son contenidos en dos registros especiales. Los contenidos de estos registros son apuntadores al espacio de memoria del primer y último proceso de la cola como se ve en la figura 2.3.

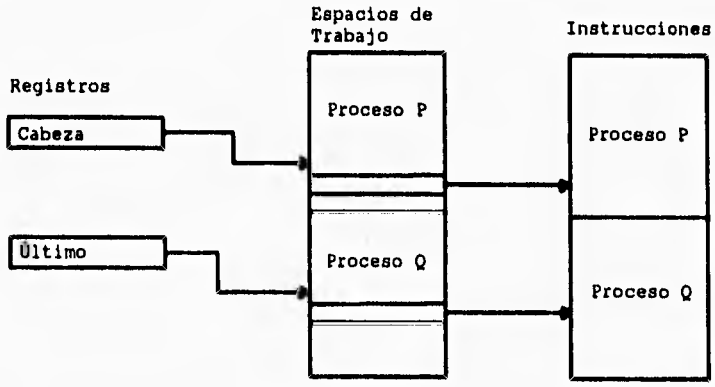


Figura 2.3 Cola del programador de tareas (scheduler)

En la asignación de tiempo del procesador a cada uno de los procesos, el scheduler obviamente favorece a los de prioridad alta (prioridad 0). Un proceso con prioridad baja puede ser llamado por el scheduler sólo si la cola de los procesos de alta prioridad está vacía, lo que nos dice que todo proceso con prioridad baja no será ejecutado hasta que todos los de prioridad alta hayan terminado [5]. Los procesos con prioridad alta correrán hasta su terminación o hasta que ellos tengan que esperar un evento (ejemplo, comunicación). Los procesos con prioridad baja pueden ser interrumpidos por el scheduler si hacen uso del procesador por largo tiempo. El transputer incrementa cada microsegundo un registro del reloj, una división de un período de tiempo corresponde a 1024 ticks del reloj. Cuando dos divisiones continuas ocurren y el mismo proceso ha estado ejecutándose continuamente, el procesador tratará de interrumpirlo. Este proceso de interrupción se lleva a cabo durante la ejecución de algunas instrucciones específicas; cuando esto sucede, los apuntadores al espacio de memoria del proceso son puestos al final de la cola y el primer proceso en ella se convierte en la cabeza.

2.3.2 CANALES Y COMUNICACIÓN.

La comunicación mediante canales entre dos procesos es lograda usando las instrucciones de mensaje de entrada (in) y mensaje de salida (out). La comunicación entre dos procesos es punto a punto, sincronizada y sin almacenaje en memoria, es decir, no es usada la memoria para guardar el mensaje siendo transferido durante la

comunicación; el mensaje es enviado como es, desde el primer byte hasta el último. De acuerdo a lo anterior, un mensaje no necesita una cola de mensajes ni cola de procesos.

Un canal puede ser, o bien una localidad de memoria (canal interno) o uno de los links de los transputers ya mencionados (canal externo), de acuerdo a si el paso de mensajes ocurre en procesos corriendo en un mismo procesador o en procesadores diferentes, respectivamente [5].

Como cada link del Transputer tiene una dirección de memoria reservada, cada canal es siempre identificado por una dirección de memoria y el procesador puede fácilmente discriminar entre los dos tipos de canales. Lo anterior significa que cada comunicación entre procesos puede ser implementada por medio de las mismas instrucciones, independientemente del tipo de canal que estemos utilizando.

Las instrucciones de in y out son funciones indirectas con operandos implícitos alojados en la pila del procesador. Estos operandos son:

- La longitud del mensaje
- La dirección del canal.
- Un apuntador a la dirección de memoria del mensaje.

COMUNICACIÓN INTERNA ENTRE CANALES

Un canal interno es implementado por medio de una palabra de memoria llamada palabra de control de canal. En cualquier instante esta palabra de control almacena el valor especial *vacío* (empty) o el descriptor del proceso. El canal debe ser inicializado como vacío pero cuando ocurra un proceso de entrada o salida el procesador debe realizar tres funciones especiales:

- Almacenar el apuntador del mensaje en el espacio de trabajo del proceso.
- Almacenar el descriptor del proceso en la palabra de control de canal e,
- Interrumpir el proceso.

Cuando el segundo proceso está listo para comunicarse, el procesador encuentra que el canal está ocupado y, como el descriptor de éste, el cual espera por una transferencia de mensaje, se encuentra también en la palabra de control de canal, el procesador vuelve a realizar tres funciones:

- Copiar el mensaje de la dirección de memoria del canal de salida a la del canal de entrada.
- Añadir el espacio de trabajo del proceso de espera a su correspondiente lista dentro del scheduler e,
- Inicializar el canal como vacío otra vez.

La figura 2.4 muestra este proceso.

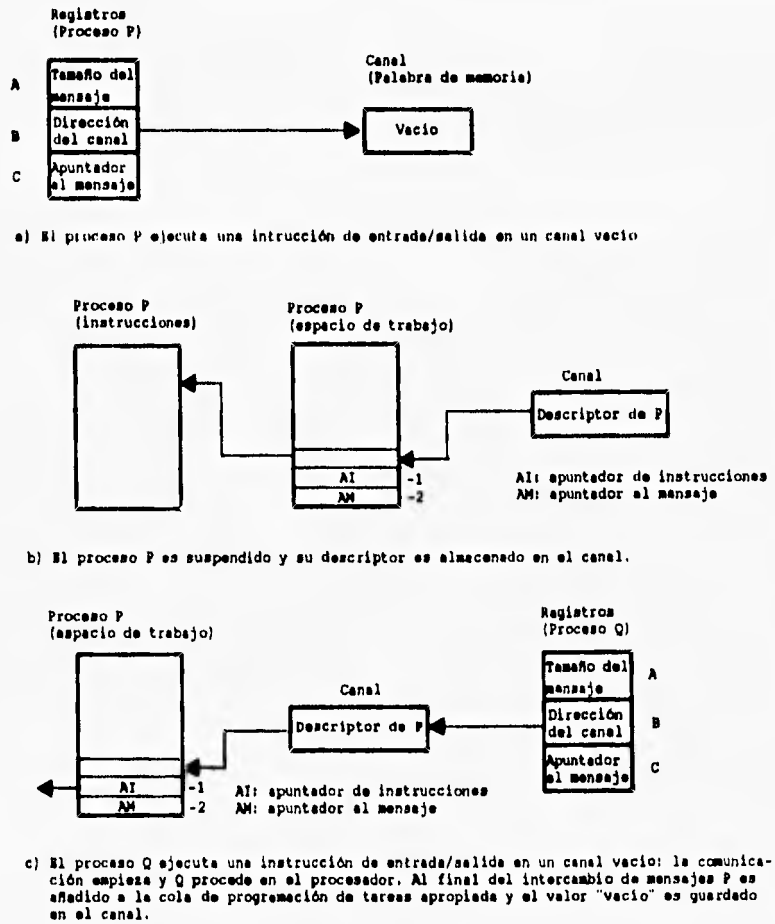
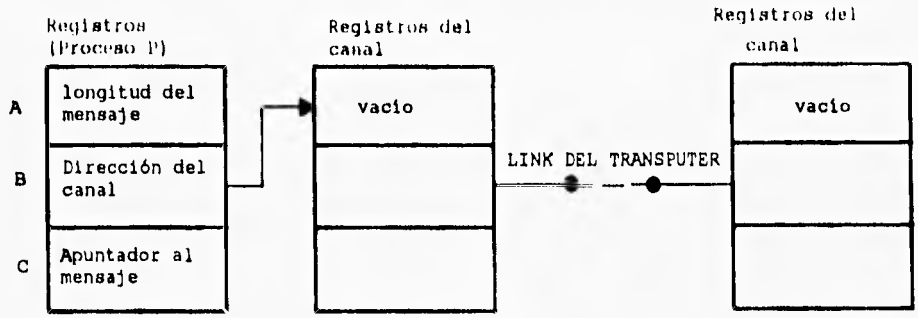


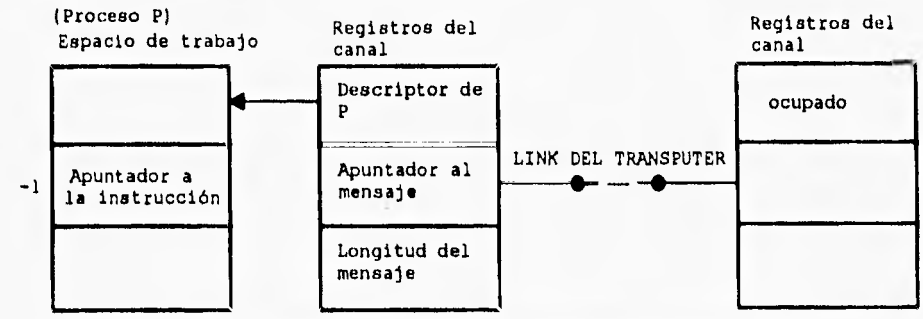
Figura 2.4 Comunicación interna entre canales

COMUNICACIÓN EXTERNA ENTRE CANALES.

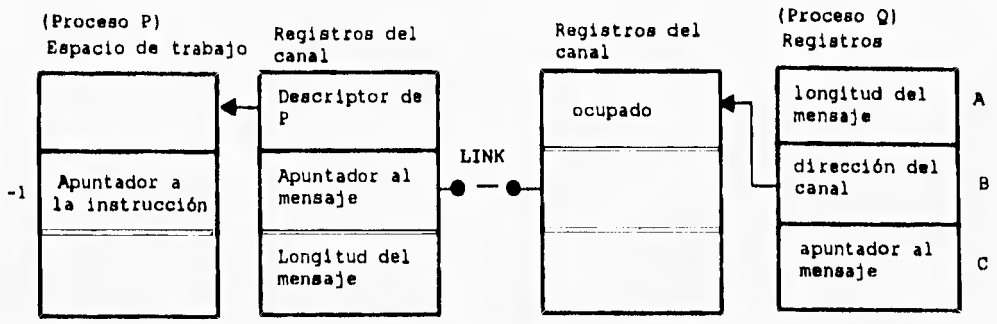
La comunicación externa entre procesos o canales es realizada de una forma similar a la de la comunicación interna solo que el procesador delega la tarea de transmitir el mensaje a una interfaz de ligado (link) autónoma, la cual es capaz de tener acceso directo a memoria en el transputer. Ocho (o bien cuatro bidireccionales) comunicaciones, con actividad normal de procesamiento, pueden ejecutarse concurrentemente en el mismo Transputer. Cada link contiene tres registros dedicados a almacenar los apuntadores al espacio de trabajo, el apuntador al mensaje y la longitud del mismo. Cuando alguno de los procesos ejecuta una instrucción de entrada o salida, el procesador inicializa los registros del link e interrumpe el proceso. Similarmente, cuando el segundo proceso se encuentra listo para comunicarse, los registros en la interfaz de ligado del transputer ejecutando este proceso son inicializados, el proceso es interrumpido y el mensaje es transferido a través del link. Cuando el intercambio de mensajes ha terminado, los procesos que han intervenido en la comunicación están listos para ejecutarse en su propio procesador y son añadidos a sus respectivas listas de procesos. Esto se muestra en la figura 2.5.



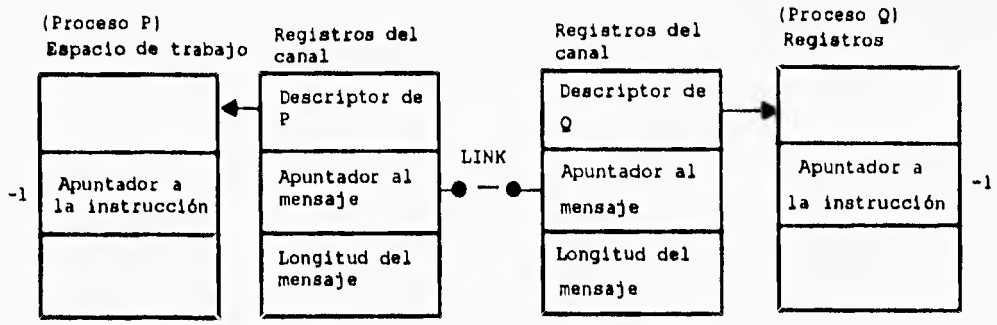
a) El proceso P ejecuta una instrucción de entrada/salida en un canal vacio.



b) El proceso P es suspendido y su descriptor es almacenado en el canal.



c) El proceso Q ejecuta una instrucción de entrada/salida en un canal ocupado.



d) El proceso Q es suspendido, su descriptor es almacenado en el canal y la comunicación comienza. Al final de la misma, tanto P como Q son añadidos a sus respectivas colas.

Figura 2.5 Comunicación externa entre canales

2.3.3 MEMORIA

Además de su RAM interna, el Transputer puede también tener acceso a un gran espacio de memoria externa (4 Gbytes para Transputers con palabras de 32 bits). Tanto la memoria interna como la externa son parte del mismo espacio lineal de direcciones. La interfaz de memoria del Transputer puede dar cabida tanto a RAM's estáticas como dinámicas, así como ROM's y EPROM's y puede ser configurada de acuerdo a la velocidad de acceso a las mismas.

En el caso del transputer T805 utilizado en este trabajo, este posee una memoria externa de 8 Mbytes, cantidad ideal para el procesamiento de imágenes ya que contienen una gran cantidad de información.

2.4 LENGUAJES PARALELOS

Para el diseño del conjunto de instrucciones del Transputer se tomaron en cuenta la mayoría de las operaciones más comúnmente ejecutadas en microprocesadores. Fue visto que sólo un pequeño número de instrucciones simples participaban en la creación de todo programa. Estas incluían la carga en registros, almacenamientos, sumas y operaciones de salto y llamadas a subrutinas. El conjunto de instrucciones del Transputer contiene un número relativamente pequeño de operaciones y cada instrucción consiste en un byte dividido en dos partes de cuatro bits [10]. La parte más significativa es el código de la función mientras que la parte menos significativa corresponde a los valores de los datos (operando). Esta representación permite 16 funciones (2^4), cada una con valor de datos en el rango de 0 a 15. Algunas de estas funciones son usadas para extender la longitud del operando y para extender la función misma para acomodar las instrucciones restantes del conjunto total de instrucciones. Varias mediciones han demostrado que cerca del 70 % de instrucciones ejecutadas pueden ser codificadas en un solo byte usando este esquema, y como en el Transputer muchas de estas requieren solo un ciclo de procesador, hacen de este un procesador muy rápido [8].

El Transputer puede ser programado en un gran número de lenguajes científicos de alto nivel como C, Pascal y Fortran. Estos lenguajes son extendidos para soportar las características que permiten a los transputers ser usados en sistemas multiprocesadores. Cuando el Transputer fue concebido no existía algún lenguaje de alto nivel que estuviera diseñado específicamente para el modelo de concurrencia en el cual el Transputer está basado. Es por ello que fue creado el lenguaje Occam, para programar sistemas concurrentes y distribuidos [2].

2.4.1 OCCAM

Occam puede ser descrito como una colección de procesos concurrentes los cuales se comunican usando una estructura de datos llamada *canal*. Este lenguaje está basado en el modelo teórico de Procesos Secuenciales Comunicados (CSP, Communicating Sequential Processes), el cual es una notación matemática para especificar el comportamiento de procesos concurrentes [6]. Bajo esta teoría, un programa es una colección de procesos secuenciales los cuales pueden ser

ejecutados concurrentemente con otros. Estos procesos sólo pueden interactuar con los demás vía operaciones entre ellos, como los de entrada-salida. Esta comunicación es síncrona y no se almacena en un buffer, por ejemplo, el proceso que ejecuta primero una instrucción de entrada o salida en un canal dado espera hasta que el otro proceso ejecute su correspondiente operación de entrada o salida. El efecto combinado de la ejecución de un par de operaciones de entrada-salida correspondientes es la transferencia de datos entre el proceso emisor y el receptor y la sincronización entre ellos. Cada proceso ejecuta un cierto número de acciones y termina. Estas acciones pueden ser ejecutadas secuencialmente (como en los lenguajes convencionales) o en forma paralela. En otras palabras, cada proceso puede ser estructurado como un conjunto de programas concurrentes que son activados por un comando apropiado. Un programa en Occam y, de forma más general, un programa escrito para un lenguaje basado en CSP, tiene una arquitectura jerárquica paralela que es definida por el anidamiento de activaciones de procesos paralelos.

Los procesos creados en Occam pueden ser ejecutados concurrentemente en el mismo transputer, por la división de períodos de tiempo como se explicó en la parte concerniente al scheduler, o en forma paralela en una red de procesadores.

Se puede citar que una de las ventajas de Occam es que es fácil de aprender y fácil de usar, además de que soporta la mayoría de las aplicaciones que pueden ser desarrolladas en un ambiente de transputers. Los transputers fueron diseñados para una ejecución eficiente del lenguaje Occam, lo opuesto es también cierto [5].

CONCEPTOS BÁSICOS DE OCCAM.

Los tres procesos básicos o primitivos de Occam son definidos por:

- El proceso de asignación. Asignar un valor a una variable.
- El proceso de entrada. Recibir un dato en una variable.
- El proceso de salida. Enviar un dato de una variable.

Existen dos procesos primitivos restantes, ellos son SKIP y STOP, el primero empieza y termina sin realizar ninguna acción, en síntesis es el proceso nulo y es

usado con ciertas formas sintácticas para indicar, explícitamente, que no es requerida ninguna acción; el segundo mientras tanto, empieza, no realiza ninguna acción y nunca termina.

STOP tiene el efecto de inhibir cualquier proceso padre, esta acción es claramente drástica y es sólo utilizada cuando se ha generado una condición de error.

PROCESOS PRIMITIVOS

Asignación.

Este proceso asigna un valor a una variable y tiene la siguiente forma:

variable : = expresión.

donde:

variable: es un identificador de una variable definida por el usuario, y,

expresion: es una expresión de occam. El valor de la expresión es asignado a la variable. En occam una expresión puede tomar diversas formas, puede ser una constante, una combinación aritmética u otra variable.

Una restricción es que los tipos de las variables involucradas en una expresión deben ser iguales a los de la variable a la que se realizará la asignación.

Entrada y salida

Los procesos de entrada y salida operan vía canales y proveen la comunicación entre procesos concurrentes. Un canal es una liga de comunicación en un solo sentido entre dos procesos. El canal es usado para pasar datos desde un proceso concurrente a otro. Un canal solo puede ser compartido entre dos procesos que se comunican, un proceso enviará datos a través de él y el otro los recibirá.

Entrada

El proceso de entrada permite que un valor pueda ser transportado desde un canal definido en Occam y que el valor pueda ser asignado en una variable. El proceso de entrada tiene la forma:

canal ? variable

donde:
canal : es un identificador de canales en occam, y,
variable : es una variable de occam donde se recibe el valor transportado a través del canal.

El valor de entrada debe ser del mismo tipo de datos que la variable a recibir el dato.

Salida

El proceso de salida envía un valor de una expresión a través de un canal. Tiene la forma:

canal ! expresión

donde:
canal: es de nueva cuenta un identificador de un canal, y,
expresión : es una expresión en occam.

CONSTRUCCIONES

Las construcciones o constructores proveen al lenguaje con la capacidad de agrupar procesos primitivos. Existen cinco distintos tipos de constructores los cuales son: SEQ, PAR, WHILE, IF y ALT.

SEQ

La construcción secuencial causa que las componentes de uno o más procesos deban ser ejecutados una tras de otra, justo como lo hacen las computadoras tradicionales. Sin embargo, debido a que con occam el programador tiene la opción de ejecutar programas en forma paralela o secuencial, la ejecución secuencial debe ser dada explícitamente.

Cada construcción secuencial es introducida mediante la implementación de la palabra reservada SEQ, la cual es seguida por una lista de procesos que serán ejecutados en secuencia.

Se debe tener el siguiente formato:

```
SEQ
    proceso 1
    proceso 2
    .
    .
    proceso n
```

donde las sentencias proceso 1,2,...n , representan una colección de instrucciones a ser ejecutadas en secuencia y cada una debe estar indentada a dos espacios de la palabra reservada SEQ.

La construcción termina hasta que el último proceso termina, una construcción SEQ sin procesos componentes se comporta como un proceso SKIP.

PAR

La construcción paralela causa que los procesos involucrados sean ejecutados concurrentemente, cada uno a su propia velocidad. Cada proceso dentro de una construcción paralela empieza su ejecución al mismo tiempo que los demás.

Una construcción paralela involucra la palabra reservada PAR seguida por una lista de procesos que serán ejecutados en paralelo.

La construcción paralela tiene el siguiente formato:

```
PAR
    proceso 1
    proceso 2
      .
      .
      .
    proceso n
```

donde los procesos 1,2,3...n , representan la colección de instrucciones a ser ejecutadas en paralelo. De igual forma los procesos deben estar indentados a dos espacios de la palabra reservada.

ALT

La construcción de alternancia permite a un proceso particular de una lista de procesos componentes, ser elegido para su ejecución. En el caso más simple, cada proceso componente es aguardado por una instrucción de entrada. Cada alternativa es elegida para su ejecución de acuerdo a qué proceso efectuará una instrucción de entrada primero.

Una construcción de alternancia está dada por la implementación de la palabra reservada ALT y es seguida por una lista de procesos de entrada.

```
ALT
   entrada 1
      proceso 1
   entrada 2
      proceso 2
   .
   .
   entrada n
      proceso n
```

donde los procesos 1,2...n, representan los procesos asociados. Cada proceso de entrada debe estar indentado dos espacios después de la construcción ALT y cada proceso asociado deberá ser indentado dos espacios más.

Esta construcción ALT se comporta como un multiplexor, aceptando datos de un número de canales de entrada y enviando esos datos a través de un sólo canal de salida.

2.4.2 C PARALELO

El compilador de C paralelo es una extensión del C estándar diseñado por Kernighan y Ritchie con características que permiten la programación paralela para un transputer o red de transputers [1].

Al igual que Occam, el modelo abstracto usado en C paralelo está basado en el modelo de Procesos Secuenciales Comunicados (CSP) de programación paralela. Este modelo mapea fácilmente en transputers para proveer de código paralelo eficiente.

Los procesos concurrentes en C son independientes, pueden ser anidados uno dentro de otro y su único lazo pueden ser los canales. Una función, de las que normalmente son utilizadas en C, puede ser definida como un proceso concurrente usando un conjunto de instrucciones provistas por la librería estándar [1].

La siguiente figura muestra los elementos principales del modelo CSP. Los procesos pueden ser anidados uno dentro de otro y pueden comunicarse, bien unidireccionalmente (un proceso enviando datos a otro), o bien bidireccionalmente (dos procesos intercambiando datos y trabajando de una forma cooperativa). En aplicaciones reales normalmente un proceso se comunica con al menos otro proceso del sistema.

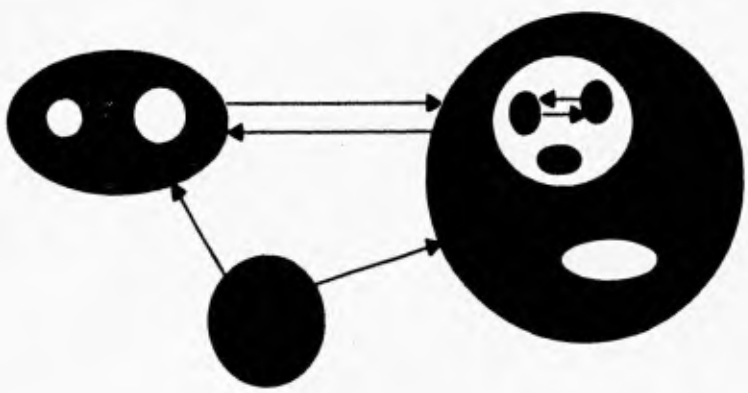


Figura 2.6 Procesos secuenciales comunicantes.

ESTRUCTURAS DE C PARA SOPORTAR CONCURRENCIA.

Podemos decir que C paralelo incluye dos tipos especiales de estructuras que lo hacen diferente del C convencional para procesos secuenciales, estas son *procesos* y *canales*.

Los procesos son los elementos principales del modelo CSP mencionado. Un proceso describe el comportamiento de un componente separable de una aplicación; puede consistir de otros procesos, operaciones secuenciales o cualquier combinación de los mismos. Una aplicación puede ser dividida en procesos y estos pueden ser mapeados en una red de transputers.

Los canales facilitan la comunicación entre procesos y es mediante ellos por donde los procesos intercambian información. Los canales, al igual que en Occam, son conexiones unidireccionales punto a punto, esto es, que solo pueden conectar dos procesos, no más, y la transferencia de datos es en un solo sentido.

Los canales tienen dos funciones, proveer la comunicación entre procesos ejecutándose independientemente y servir para sincronizar la comunicación entre ellos. Procesos que envían datos no podrán continuar hasta que la contraparte reciba los datos. En este sentido, la comunicación es segura ya que ningún dato será transmitido hasta que ambos procesos estén listos.

De acuerdo a lo anterior, dos nuevos tipos de datos son implementados para soportar la concurrencia, estos son:

Process: una estructura para almacenar información acerca de un proceso.
Channel: un tipo de dato para implementar canales.

Process

Cuando un programa empieza hay un proceso principal ejecutándose, una función de C puede ser usada para definir un proceso, se requiere que el primer parámetro de la misma sea un apuntador a un proceso. La función de C a ser usada como un proceso paralelo tiene la siguiente forma:

```
void función (Process* p, .....){
/*cuerpo de la función*/
}
```

Esta interfaz consta de un parámetro fijo seguida por un número determinado de parámetros definidos por el usuario. El parámetro fijo (el primer parámetro), es un apuntador a una estructura de un proceso (Process *). Los parámetros definidos por el usuario pueden ser los ya definidos por el ANSI C más el nuevo tipo de datos que son los canales.

Las funciones provistas para procesos y que aparecen en la libería <process.h> son las siguientes:

*Process ProcAlloc(void (*función) ( ),int tamaño, int parámetros,...)*

Esta función reserva espacio de memoria para un proceso e inicializa la estructura Process. *ProcAlloc* toma un apuntador al código de la función, reserva memoria en el stack para el proceso y aloja también los parámetros en memoria.

tamaño: especifica el tamaño del stack
parámetros: es el número de parámetros a ser pasados a la función, excluyendo el apuntador al proceso.

void ProcStop(void)

Al igual que en Occam, C proporciona un proceso de stop, este causara que un proceso espere un suceso indefinidamente y nunca termine.

```
void ProcRun(Process* p);
void ProcRunHigh(Process* p);
void ProcRunLow(Process* p);
```

Estas funciones son utilizadas para ejecutar procesos asincrónicamente, ProcRun ejecuta un programa sin ninguna prioridad, mientras que ProcRunHigh y ProcRunLow le asignan prioridades a los procesos, ésto es importante cuando se tienen varios procesos corriendo en el mismo procesador, en algunos casos es más importante lograr una comunicación con otro procesador que realizar otra acción, al primer proceso se le asignaría una prioridad alta y al restante una prioridad baja.

void ProcPar (Process p1,.....,NULL);*
*void ProcParList (Process** lista_de_procesos,NULL);*
void ProcPriPar (Process p_alta, Process* p_baja);*

Estas funciones son definidas para ejecutar procesos síncronos, cada una de ellas comienzan un número de procesos y esperan por su terminación para retornar al proceso que los haya llamado. ProcPar toma una lista de apuntadores de procesos terminando con NULL y los ejecuta sin prioridad alguna. ProcParList toma un arreglo de apuntadores a procesos y realiza la misma acción que ProcPar. ProcPriPar ejecuta dos procesos, el primero con prioridad alta y el segundo con prioridad baja.

Channel

Las funciones provistas para canales y que aparecen en la libería <channel.h> son las siguientes:

Channel ChanAlloc(void)*

Los canales pueden ser declarados como variables de cualquier otro tipo pero para usarlos deben ser primero, inicializados correctamente. La función que los inicializa es la función ChanAlloc la cual aloja espacio en memoria para el canal.

Para escribir un valor a un canal se necesitan las siguientes funciones:

```
void ChanOut(Channel* canal, void* cp, int n);
void ChanOutChar(Channel* canal, unsigned char ch);
void ChanOutInt(Channel* canal, int n);
```

Las dos funciones ChanOutChar y ChanOutInt transfieren un carácter o entero respectivamente. ChanOut es una función que transfiere *n* bytes de un arreglo *cp* y puede ser usada para cualquier tipo de dato.

Para leer un valor de un canal se encuentran disponibles las siguientes funciones:

```
void ChanIn(Channel* canal, void* cp, int n);
unsigned char ChanInChar(Channel* canal);
int ChanInInt(Channel* canal);
```

Las dos funciones ChanInChar y ChanInInt leen un carácter o un entero respectivamente. ChanIn es una función general que lee *n* bytes del arreglo *cp* y puede ser usado, al igual que ChanOut, para todo tipo de datos.

Existen casos donde los procesos tratan de obtener datos de varios canales y desean detectar cual de ellos está listo para recibir primero. Esto es logrado usando las siguientes funciones que determinan qué alternativa elegir.

```
int ProcAlt (Channel* canal1, Channel* canal2,....);
int ProcAltList (Channel** lista_de_canales);
int ProcSkipAlt (Channel* canal1, Channel* canal2,....);
int ProcSkipAltList (Channel** lista_de_canales);
```

Todas estas funciones toman parámetros que son listas de apuntadores a canales terminados por un apuntador nulo (NULL). ProcAlt y ProcSkipAlt retornan un índice (empezando en cero) de un canal que está listo para transferir datos.

Hasta el momento se han visto las características de los lenguajes de programación paralelos más populares: Occam y C. Ambos contienen las características necesarias para realizar aplicaciones en forma paralela.

Todas las aplicaciones desarrolladas en este trabajo de tesis fueron hechas en C y es donde asoma inmediatamente una pregunta, ¿por qué usar C cuando el transputer soporta un lenguaje especialmente diseñado para él como Occam?

Podríamos citar varios ejemplos de ventajas y desventajas de uno y otro pero la principal razón de utilizar este lenguaje es que la tendencia en cuanto a metodologías de programación (aún en la programación paralela) es la Programación Orientada a Objetos [16] y C es un lenguaje que puede migrar fácilmente hacia esa metodología. C permite una característica muy importante, la reusabilidad [11], con un pequeño número de modificaciones al código original éste puede adaptarse a la nueva aplicación. No hay que perder de vista que este trabajo de tesis corresponde a un proyecto que se desarrolla actualmente en el IIMAS (Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas) y que obviamente se tratarán de añadir aplicaciones o modificar las ya establecidas, por lo que tener un código fácil de entender y cambiar es una razón muy importante para escoger C paralelo.

2.5 MÉTRICAS DE DESEMPEÑO

El desempeño es uno de los aspectos más importantes de la programación paralela. Una de las razones por la cual se desarrollan programas para sistemas paralelos es la de resolver problemas de una forma más rápida comparado con un sistema secuencial común. Medir el rendimiento de un programa paralelo es una forma de saber qué tan buenos y eficientes han sido nuestros esfuerzos en dividir nuestro problema principal en pequeños módulos cooperativos los cuales son ejecutados en paralelo [9].

La métrica de desempeño más visible y fácil de recordar es el tiempo de ejecución. Midiendo el tiempo que necesita nuestro programa para ser ejecutado es la forma de obtener este parámetro. Para saber qué tan bien resuelve nuestra aplicación el sistema paralelo comparado con un sistema secuencial podemos obtener la razón entre los dos, esta medida es llamada *speedup*, y está asociada al número de procesadores que se utilizan en la implementación y se describe a continuación.

2.5.1 SPEEDUP

El speed up es una medida relacionada con el tiempo de ejecución, mide qué tan rápido se ejecuta un programa en una máquina paralela comparado con lo que lo haría en una máquina secuencial [15]. Se define de la siguiente manera:

$$SpeedUp = \frac{Tiempo\ de\ ejecución\ serial}{Tiempo\ de\ ejecución\ paralela} = \frac{T(1)}{T(p)}$$

donde $T(p)$ representa el tiempo tomado por el programa corriendo en p procesadores, y $T(1)$ es el tiempo de la mejor implementación serial de la aplicación medida en un sólo procesador.

Lo ideal de una gráfica del speedup es que sea una recta con un ángulo de inclinación de 45^0, esto implica que el tiempo de ejecución de la aplicación va decayendo proporcionalmente con el número de procesadores utilizados, así, si el tiempo de ejecución secuencial de un programa es de 600 mseg, para dos procesadores idealmente sería de 300 mseg, para 3 de 200 mseg y así sucesivamente.

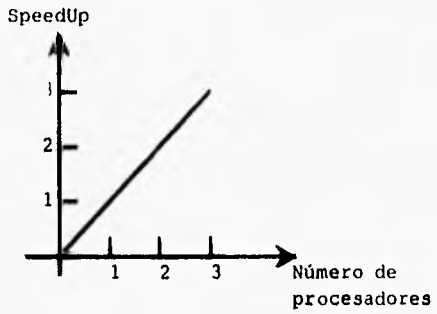


Figura 2.7 Gráfica de un speedup ideal

Obtener una gráfica lineal cuando el número de procesadores se incrementa es bastante difícil, el speedup es una medida del desempeño en función del número de procesadores utilizados en la ejecución de un programa y del tiempo de ejecución, no considera otros factores que influyen en el desempeño como la topología utilizada, el balance de cargas y la granularidad [7], conceptos que serán descritos en el capítulo 4.

2.5.2 EFICIENCIA

La eficiencia es definida como el promedio de la contribución de todos los procesadores implicados en la solución de un problema. La relación matemática de la efiencia es la razón del speedup entre el número de procesadores.

$$Eficiencia(p) = \frac{Speedup(p)}{p}$$

El valor obtenido refleja la eficiencia al aprovechar los recursos de hardware del sistema, es por ello que su valor siempre será inferior a uno, ya que el tiempo de ejecución de un programa paralelo se ve afectado por el tiempo total de comunicaciones entre los procesadores. El valor de la eficiencia para el caso de un solo procesador se considera como la unidad.

2.5.3 EFICACIA

Otra medida de rendimiento de los sistemas paralelos y similar a la eficiencia es la eficacia, muy importante para determinar el mejor costo-efecto del número de procesadores para una aplicación particular.

La eficacia se define como:

$$Eficacia(p) = \frac{(SpeedUp(p))^2}{p}$$

$$= Eficiencia(p) * SpeedUp(p)$$

Una importante propiedad de la eficacia cuando su curva es graficada con respecto al número de procesadores es que su primer máximo corresponde al punto óptimo de operación del sistema [15].

2.5.4 FRACCIÓN SERIAL

La ley de Amdahl es definida de la siguiente manera:

$$T(p) = T_s + \frac{T_p}{p}$$

Donde T_s es el tiempo tomado por la parte del programa que debe ejecutarse secuencialmente y T_p es el tiempo de la tarea que se ejecuta en paralelo. Obviamente, el tiempo para un procesador es $T(1) = T_s + T_p$. La fracción serial se define como $f = T_S / T(1)$, de acuerdo a lo anterior, la ecuación pasada puede reescribirse como:

$$T(p) = T(1)f + \frac{T(1)(1-f)}{p}$$

o, en términos del speedup (s)

$$\frac{1}{s} = f + \frac{1-f}{p}$$

Resolviendo la ecuación anterior en términos de la fracción serial.

$$f(p) = \frac{1/s - 1/p}{1 - 1/p}$$

Un índice pequeño de fracción serial es muy deseable, ya que eso implica que solo una mínima parte de nuestra aplicación se ejecuta secuencialmente con lo que se logra utilizar al máximo nuestros recursos [12][9].

A diferencia de las tres métricas de rendimiento descritas anteriormente, la fracción serial nos puede indicar qué es lo que está sucediendo con nuestro sistema, si está perfectamente balanceado (si las tareas que desarrollan cada uno de los procesadores requieren la misma cantidad de tiempo) o si existe un overhead de comunicaciones (comunicación excesiva con un tiempo de procesamiento mínimo).

Cuando existe un desbalanceo de cargas, el speedup se reduce notablemente y la fracción serial se incrementa de igual manera. Si el sistema se encontrara totalmente desbalanceado la fracción serial sería 1, indicando que existe un problema en la distribución de tareas, esto es imposible notarlo con el speedup o la eficiencia.

El overhead es una función monótona creciente del número de procesadores. Conforme aumenta dicho overhead, el speedup se reduce provocando un ligero incremento en la fracción serial cada vez que se añada un nuevo procesador al sistema. Cuando esta situación se presenta, se dice que la granularidad del sistema es muy fina. Por último, cuando el valor de la fracción serial es constante y la eficiencia baja, se debe a que el algoritmo ha llegado a su límite de paralelización.

2.6 REFERENCIAS

[1] ANSI C TOOLSET, **USER GUIDE**, INMOS 1992.

[2] BENÍTEZ, H. **DISEÑO DE UNA ARQUITECTURA PARA PROCESAMIENTO PARALELO DE SEÑALES DOPPLER DE ULTRASONIDO.** TESIS DE LICENCIATURA, F.I., 1994.

[3] BURNS, A. **PROGRAMMMING IN OCCAM 2**. ADDISON WESLEY, 1992.

[4] CLEMENTS, A. **MULTIPROCESSOR SYSTEMS**. ELECTRONIC AND WIRELESS WORLD, 1991.

[5] DE CARLINI, U. Y DE VILANO, U. **TRANSPUTERS AND PARALLEL ARCHITECTURES**. MIT PRESS, 1991.

[6] GALLETLY, J. **OCCAM2**. PITMAN PUBLISHING, 1990.

[7] GARCÍA, F. Y ORTEGA, J.L. **PROGRAMACIÓN PARALELA ORIENTADA A OBJETOS**. IIMAS 1995.

[8] GREEN, S. **PARALLEL PROCESSING FOR COMPUTER GRAPHICS**. MIT PRESS, 1991.

[9] HARP, A. Y FLATT, H. **MEASURING PARALLEL PROCESSOR PERFORMANCE**. COMMUNICATIONS OF THE ACMD, 1990.

[10] INMOS, **THE TRANSPUTER DATABOOK**, 1992.

[11] KERNIGHAN, B. Y RITCHIE, D. **EL LENGUAJE DE PROGRAMACIÓN C**. PRENTICE-HALL, 1991.

[12] LEWIS, G. **INTRODUCTION TO PARALLEL COMPUTING**. PRENTICE HALL, 1992.

[13] MORRIL, J. **PARALLEL PROCESSING**. REVISTA BYTE 1988. VOL. 13. NO. 11. P.P. 250-261. 1988.

[14] STEIN, RICHARD M. **T800 AND COUNTING**. REVISTA BYTE. VOL. 13. NO. 11. P.P. 287-300. 1988.

[15] THIÉBAUT, D. **PARALLEL PROGRAMMING IN C FOR THE TRANSPUTER**. 1995.

[16] VOSS, G. **OBJECT ORIENTED PROGRAMMING**. MC. GRAW HILL, 1993.

3

ALGORITMOS DE INTERPOLACIÓN DE IMÁGENES DIGITALES

3.1 INTRODUCCIÓN

En este capítulo se muestran los diferentes algoritmos de interpolación que serán implementados en nuestro sistema paralelo para darle solución al problema planteado en el primer capítulo.

Diversos son los algoritmos que aquí se muestran, como lo son repetición de pixeles (orden cero), interpolación lineal (orden uno), splines de primero, segundo y tercer orden y, por último, el algoritmo de convolución cúbica.

Es importante también conocer algunas técnicas de procesamiento digital de imágenes, por lo cual se describen brevemente las más comunes de ellas a continuación.

3.2 PROCESAMIENTO DIGITAL DE IMÁGENES

El término procesamiento digital de imágenes se refiere generalmente al procesamiento de una imagen de dos dimensiones en una computadora digital. En un contexto más amplio implica el procesamiento digital de datos bidimensionales cualesquiera que sean. Una definición de imagen digital es la de ser un arreglo de números reales o bien complejos que pueden ser representados mediante un número finito de bits [6].

Además del procesamiento, los sistemas en los que intervienen imágenes digitales son divididos generalmente en dos unidades principales más, la adquisición de la imagen y el despliegue de la misma [3]. Se definen a continuación estas dos unidades:

Adquisición: para adquirir la imagen es necesario tener un sensor y la capacidad de digitalizar la señal producida por dicho sensor. Para el caso presentado en este trabajo los sensores son cerámicas que realizan un barrido ultrasónico y se utiliza un convertidor A/D para digitalizar la señal.

Despliegue: monitores a color o monocromáticos son los principales dispositivos usados para desplegar la imagen, es decir, son los medios por los cuales podemos visualizar nuestra información.

Aunados a estas dos unidades podemos incluir otros dos elementos importantes dentro de los sistemas de imágenes, el almacenaje y la comunicación de la información [4]; el primero de ellos involucra dispositivos rápidos para guardar los datos, la elección de estos dispositivos es usualmente una tarea difícil en el diseño de sistemas de imágenes. El segundo elemento se refiere a la comunicación local entre sistemas de procesamiento de imágenes y comunicación remota de un punto a otro.

Esta tesis aborda la unidad de procesamiento. El campo del procesamiento digital de imágenes ha crecido vigorosamente y muchas técnicas de procesamiento de imágenes son usadas para resolver una variedad de problemas. En medicina, por ejemplo, se utilizan algoritmos para mejorar el contraste y para la reducción e interpolación de las imágenes médicas. Se procesan imágenes además, en el campo de espectros tomados vía satélite, transmisión y almacenamiento de la información, radares, sonares e inspección automatizada de partes industriales [4][8].

Algunas de las técnicas de procesamiento de imágenes digitales más utilizadas fueron creadas para corregir defectos en las imágenes adquiridas debido a las imperfecciones de los detectores, a una inadecuada o no uniforme iluminación o a un indeseable punto de referencia desde el cual la imagen fue adquirida [11]. Hay que enfatizar que estas correcciones son aplicadas después de que la imagen ha sido digitalizada, así que no se podrá obtener una imagen de mucho mayor calidad que la que se lograría con la optimización del sistema de adquisición [11].

Técnicas como las de ajuste de brillo y contraste, balanceo de colores, detección de bordes, grabación en relieve (embossing), difuminación (blurring), son muy usadas. No es el propósito de esta tesis describir cada una de ellas puesto que este trabajo se centra tan solo en la interpolación de imágenes, la cual será abarcada en la siguiente sección; sin embargo, se presentan ejemplos de las técnicas citadas líneas arriba para una mayor comprensión.

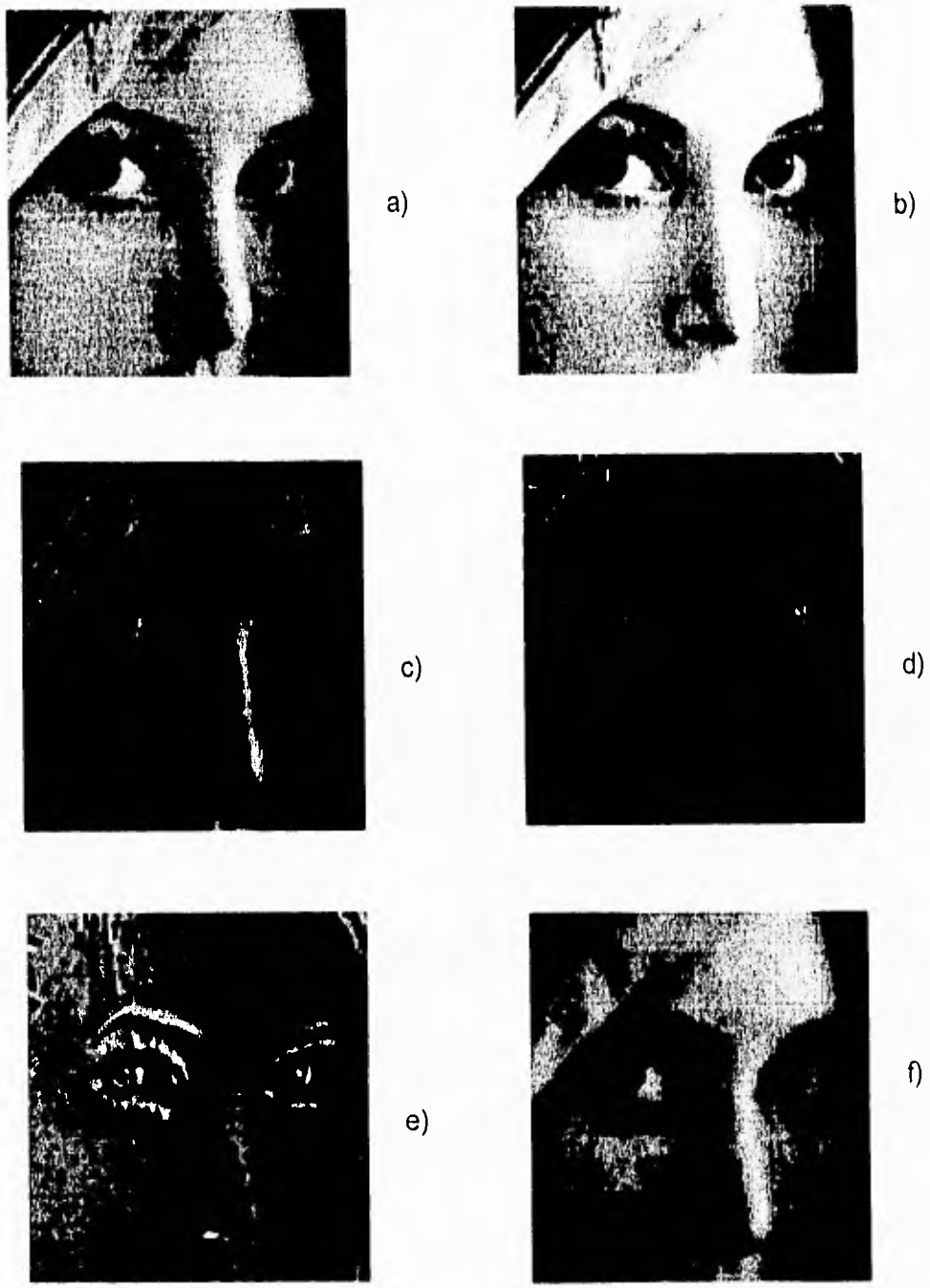


Figura 3.1. a) imagen original, b) con ajuste de brillo, c) ajuste de colores, d) detección de bordes, e) embossing, f) blurring.

El ajuste de brillo consiste en sumarle una cantidad constante a cada uno de los valores de los pixeles mientras que el balanceo de colores se realiza mediante el incremento de uno o alguno de los componentes rgb (red-green-blue: rojo-verde-azul) de la paleta de colores.

La detección de bordes se realiza cuando se descubre un cambio abrupto en el nivel de gris de los pixeles, es decir, si existe una transición de un nivel bajo, en cuanto al pixel, a un nivel alto, entonce éste se trata de un borde.

La difuminación de la imagen (blurring) se logra filtrando la imagen con un filtro paso bajas mientras que el relieve (embossing) de la misma se obtiene con filtro paso altas y la suma de una constante a los valores de los pixeles.

Esta tesis está basada en una de las tantas técnicas de procesamiento existentes, la interpolación de imágenes, la cual se define a continuación.

3.3 INTERPOLACIÓN

Interpolación es el proceso de estimar los valores intermedios de un evento continuo de muestreos discretos [7]. La interpolación es usada con mucha frecuencia en el procesamiento digital de imágenes para incrementar o reducir imágenes y corregir distorsiones espaciales. De acuerdo a la cantidad de datos asociados a una imagen se debe buscar un algoritmo de interpolación que sea bastante eficiente.

Algunos de los algoritmos de interpolación de imágenes digitales son: interpolación de orden cero (repetición de pixeles), interpolación lineal , splines y convolución cúbica [4][6][7], estos dos últimos serán descritos en la siguiente sección pues requieren un análisis más profundo mientras que los demás son algoritmos más simples.

En primera instancia se pondrán a consideración los algoritmos menos complejos como lo son el de orden cero y la interpolación lineal.

3.3.1 INTERPOLACIÓN DE ORDEN CERO

La forma más simple y más fácil de interpolar una imagen es la de repetir cada pixel un número determinado de veces [10], este tipo de interpolación es conocida como de orden cero. Un ejemplo de ello sería el tratar de agrandar una imagen por un factor de 2, entonces lo que se realizaría sería poner una matriz de 2*2 pixeles por cada pixel de la matriz original, dando lo siguiente:

a) imagen original

b) imagen interpolada

Figura 3.2 Interpolación de una imagen por el algoritmo de orden cero

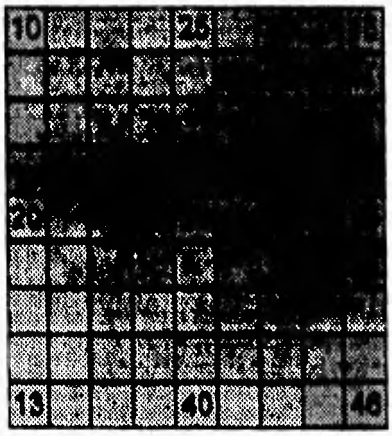
Se puede ver claramente que el cómputo requerido para realizar la interpolación es mínimo (sólo la asignación es necesaria) pero esta técnica ofrece una desventaja que es la de que se puede percibir cuadrados del mismo tono en la nueva imagen. La nueva imagen es más grande que la original, sin embargo luce un poco más burda.

3.3.2 INTERPOLACIÓN LINEAL

Otra técnica muy utilizada es la llamada interpolación por pixeles (*pixel interpolation*) o interpolación lineal, ésta consiste en lo siguiente: supóngase que se

pretende interpolar una imagen por un factor de 4, los datos de la imagen original son esparcidos en la nueva imagen de tal forma que por cada dato de la imagen original serán obtenidos tres nuevos datos tanto en el eje x como y, como se ve en la figura [6].

a) imagen original



b) los datos originales se esparcen sobre la nueva imagen

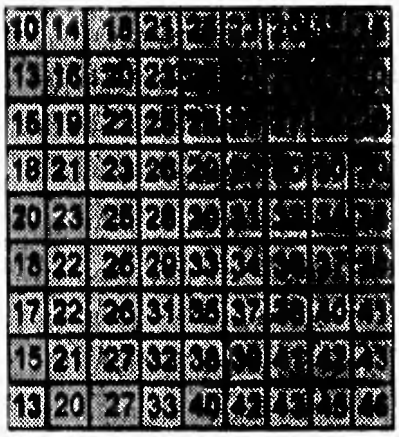
Para calcular los valores de los nuevos pixeles se realizan operaciones entre los pixeles que se encuentren más próximos. Así, si empezamos a contar las columnas y renglones desde cero tenemos que el elemento (0,1) de la matriz se obtiene de la ecuación de una recta que pasa por los puntos originales. En este caso los puntos de la recta son (0,10) y (4,25), sacando la ecuación de la recta que pasa por estos puntos tenemos:

$$y = \frac{15}{4}x + 10$$

Obteniendo los valores para x=1,2,3 y pensando que los datos de la imagen son datos enteros tenemos los siguientes resultados:

$$y(1) = \frac{15}{4}(1) + 10 = 13.75 \approx 14$$
$$y(2) = \frac{15}{4}(2) + 10 = 17.5 \approx 18$$
$$y(3) = \frac{15}{4}(3) + 10 = 21.25 \approx 21$$

El mismo proceso se realiza para obtener los demás datos lo cual arrojaría la siguiente imagen:



c) Imagen interpolada

Figura 3.3 a),b),c) Interpolación lineal

Este algoritmo de interpolación es bastante rápido, sin embargo, ofrece la desventaja de que si existe un cambio muy brusco en los valores de pixeles contiguos, el valor del pixel interpolado quedará muy distante de ambos valores, es por ello necesario utilizar algoritmos de interpolación de orden mayor para que así el valor interpolado no sea el promedio de los dos valores contiguos sino que intervengan los pixeles vecinos de tal forma que el error al interpolar los datos se reduzca.

Algoritmos en los que intervienen más datos vecinos son los que se verán a continuación, tal es el caso de los splines cúbicos y el algoritmo de interpolación cúbica.

3.4 SPLINES

En muchas ocasiones es mejor utilizar un polinomio de interpolación de orden menor que el número de datos, esto es debido a que al utilizar polinomios de alto orden es frecuente caer en resultados erróneos [9]. En el caso de las imágenes sería también muy costoso, en cuanto a tiempo de procesamiento, el utilizar funciones de interpolación de un orden alto ya que, como sabemos, las imágenes se caracterizan por contener grandes volúmenes de datos. Ejemplos de polinomios de interpolación de menor orden son los splines.

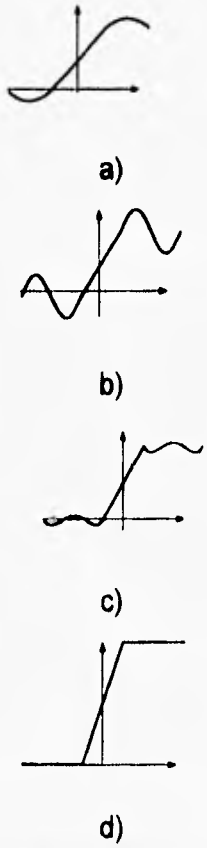


Figura 3.4 Situación visual donde los splines son superiores a polinomios de interpolación de orden superior. La función a ser interpolada presenta un cambio abrupto en el origen. Las partes a), b) y c) indican que el cambio abrupto provoca oscilaciones en los polinomios de orden superior mientras que el spline cúbico d), nos proporciona una mejor aproximación.

3.4.1 SPLINES LINEALES.

Como sabemos, la forma más simple de conectar dos puntos es una línea recta. Los splines de primer orden para un grupo ordenado de puntos puede ser definido como un conjunto de funciones lineales.

$$f(x) = f(x_0) + m_0(x - x_0) \qquad x_0 \le x \le x_1$$
$$f(x) = f(x_1) + m_1(x - x_1) \qquad x_1 \le x \le x_2$$

.

.

.

$$f(x) = f(x_{n-1}) + m_{n-1}(x - x_{n-1}) \quad x_{n-1} \le x \le x_n$$

donde las m_i representan las pendientes de las rectas que unen los puntos:

$$m_i = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{1}$$

Estas ecuaciones pueden ser usadas para evaluar la función de cualquier punto en el intervalo $[x_0, x_n]$. Como se puede ver, el método es idéntico al de la interpolación lineal.

3.4.2 SPLINES CUADRÁTICOS

Una de las desventajas de los splines de primer orden es que la primera derivada en los puntos previamente dados es discontinua, esto quiere decir que la pendiente en los puntos donde dos splines se encuentran, cambia abruptamente. Es por ello que se hace uso de polinomios de orden mayor como los splines cuadráticos.

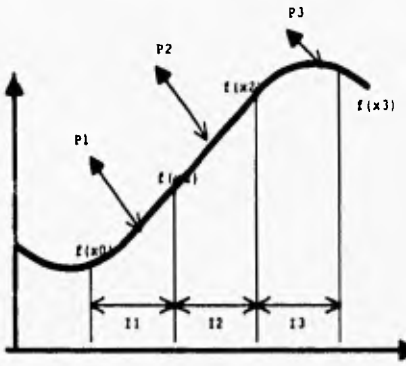
Para asegurar que la m-ésima derivada es continua en todos los puntos debe usarse por lo menos un spline de orden *m+1*. En la práctica, polinomios de tercer orden o splines cúbicos son los más usados.

Los splines cuadráticos aseguran la continuidad de las primeras derivadas en los puntos dados previamente mas no así de las segundas derivadas.

El objetivo de un spline cuadrático es derivar un polinomio de segundo orden para cada intervalo entre puntos. Los polinomios para cada intervalo pueden ser representados generalmente como:

$$f_i(x) = a_i x^2 + b_i x + c_i \qquad (2)$$

La siguiente figura se incluye para hacer más clara la notación.



$P1 = a_1x^2 + b_1x + c_1$ $P2 = a_2x^2 + b_2x + c_2$ $P3 = a_3x^2 + b_3x + c_3$

I1 = intervalo 1 I2= intervalo 2 I3= intervalo 3

Figura 3.5 Notación usada para derivar los splines cuadráticos

Para n+1 puntos (i=0,1,2,.....,n) hay *n* intervalos y, consecuentemente, *3n* valores desconocidos (las *a's*, *b's* y *c's*) a evaluar. Se requieren por lo tanto, *3n* ecuaciones o condiciones para obtener los valores de las incógnitas. Estas son [9]:

1.- Los valores de la función deben ser iguales en los puntos dados, esta condición puede ser representada como

$$a_{i-1}x_{i-1}^2 + b_{i-1}x_{i-1} + c_{i-1} = f(x_{i-1}) \qquad (3)$$
$$a_i x_{i-1}^2 + b_i x_{i-1} + c_i = f(x_{i-1}) \qquad (4)$$

Esta condición es válida para $i=2,...,n$ ya que solo los puntos interiores deben ser usados, así que las ecuaciones (3) y (4) nos proporcionan cada una *n-1* condiciones, o bien un global de *2n-2* condiciones de un total de *3n*.

2.- La primera y la última función deben pasar a través de los puntos finales, esto añade dos condiciones más

$$a_1 x_0^2 + b_1 x_0 + c_1 = f(x_0) \quad (5)$$
$$a_n x_n^2 + b_n x_n + c_n = f(x_n) \quad (6)$$

Para un total de $2n$ condiciones.

3.- La primera derivada en los puntos interiores debe ser igual. La primera derivada de la ecuación (2) es

$$f'(x) = 2ax + b$$

Entonces la condición puede ser representada generalmente como

$$2a_{i-1} x_{i-1} + b_{i-1} = 2a_i x_{i-1} + b_i \quad (7)$$

De nueva cuenta, estas condiciones sólo se aplican para $i=2,\ldots,n$, esto proporciona otras $n-1$ condiciones para un total de $3n-1$. Como tenemos $3n$ coeficientes desconocidos debemos determinar otra condición, ésta puede ser arbitraria ya que nos basta con sólo determinar el valor de las constantes. Podemos seleccionar la condición de que

4.- La segunda derivada es cero en el primer punto. Como la segunda derivada es $2a_i$, esta condición puede ser desarrollada matemáticamente como

$$a_1 = 0 \quad (8)$$

La interpretación visual de esta condición es de que los primeros dos puntos son conectados a través de una línea recta.

3.4.3 SPLINES CÚBICOS

El objetivo de los splines cúbicos es el derivar un polinomio de tercer orden para cada intervalo entre puntos internos de la forma

$$f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i \quad (9)$$

Entonces, para *n+1* puntos (*i=0,1,2...,n*), hay n intervalos y, consecuentemente, *4n* coeficientes indefinidos a evaluar. Como para los splines cuadráticos, *4n* ecuaciones son requeridas para evaluar las incógnitas. Estas son, de forma simplificada [9]:

1.- Los valores de la función deben ser iguales en los puntos interiores (*2n-2 condiciones*)

2.- La primera y la última funciones deben pasar a través de los puntos finales (*2 condiciones*)

3.- La primera derivada en los puntos interiores debe ser igual (*n-1 condiciones*).

4.- La segunda derivada en los puntos interiores debe ser igual (*n-1 condiciones*).

5.- La segunda derivada en los puntos finales es cero (*2 condiciones*).

La interpretación visual de la quinta condición es de que la función se convierte en una línea recta en los puntos finales.

Las cinco condiciones arriba citadas nos proveen de las *4n* condiciones requeridas para obtener los *4n* coeficientes, podemos resolver este sistema de *4n* incógnitas y obtener la solución.

A continuación se desarrollará una técnica que requiere solo la solución de *n-1* ecuaciones lo que aminora el esfuerzo computacional y, por ende, el tiempo de procesamiento.

El primer paso en la derivación es basado en la observación de que debido a que cada par de puntos interiores está conectado por un polinomio cúbico, la segunda derivada dentro de cada intervalo es una línea recta. La ecuación (9) puede ser derivada dos veces para verificar la observación anterior. De acuerdo a

ello, la segunda derivada puede ser representada por un polinomio de interpolación de Lagrange.

$$f_i''(x) = f''(x_{i-1})\frac{x - x_i}{x_{i-1} - x_i} + f''(x_i)\frac{x - x_{i-1}}{x_i - x_{i-1}} \tag{10}$$

Donde $f_i''(x)$ es el valor de la segunda derivada en cualquier punto x dentro del i-ésimo intervalo. Esta ecuación es una línea recta conectando la segunda derivada en el primer punto interior $f''(x_{i-1})$ con la segunda derivada en el segundo punto interior $f''(x_i)$.

Otro paso es que la ecuación (10) deba ser integrada dos veces para encontrar una expresión para $f_i(x)$, sin embargo, esta expresión contendrá dos constantes desconocidas de integración. Estas dos constantes pueden ser evaluadas invocando las condiciones de igualdad por intervalos, $f(x)$ debe ser igual a $f(x_{i-1})$ en x_{i-1} y $f(x)$ debe ser igual a $f(x_i)$ en x_i. De acuerdo a lo anterior tenemos

$$\begin{aligned} f_i(x) = {} & \frac{f''(x_{i-1})}{6(x_i - x_{i-1})}(x_i - x)^3 + \frac{f''(x_i)}{6(x_i - x_{i-1})}(x - x_{i-1})^3 \\ & + \left[\frac{f(x_{i-1})}{x_i - x_{i-1}} - \frac{f''(x_{i-1})(x_i - x_{i-1})}{6}\right](x_i - x) \\ & + \left[\frac{f(x_i)}{x_i - x_{i-1}} - \frac{f''(x_i)(x_i - x_{i-1})}{6}\right](x - x_{i-1}) \end{aligned} \tag{11}$$

Podríamos decir que esta última ecuación es mucho más compleja que la ecuación (9), sin embargo, en esta última ecuación sólo se tienen dos coeficientes indefinidos, las segundas derivadas al comienzo y final del intervalo. Si pudiéramos determinar las segundas derivadas en cada extremo del intervalo, la ecuación (11) se convertirla en un polinomio de tercer grado que podríamos utilizarlo como función de interpolación.

Las segundas derivadas pueden ser evaluadas invocando la condición de que las primeras derivadas en los puntos extremos de los intervalos deben ser continuas, es decir,

$$f_{i-1}'(x_i) = f_i'(x_i) \tag{12}$$

La ecuación (12) puede ser diferenciada para dar una expresión para la primera derivada. Si lo anterior se realiza los intervalos (i-1)ésimo e iésimo y los dos resultados son igualados de acuerdo a la ecuación anterior, se obtiene la siguiente ecuación:

$$\begin{aligned}(x_i - x_{i-1})f''(x_{i-1}) + 2(x_{i+1} - x_{i-1})f''(x_i) \\ +(x_{i+1} - x_i - x_i)f''(x_{i+1}) \\ = \frac{6}{(x_{i+1} - x_i)}[f(x_{i+1}) - f(x_i)] \\ +\frac{6}{(x_i - x_{i-1})}[f(x_{i-1}) - f(x_i)]\end{aligned} \tag{13}$$

Si la ecuación (13) es escrita para todos los puntos extremos de cada uno de los intervalos se obtendrán *n-1* ecuaciones con *n+1* incógnitas que serán las segundas derivadas. Sin embargo, como se pretende utilizar splines cúbicos naturales, las segundas derivadas en los puntos extremos son iguales a cero, por lo que nuestro problema se reduce ahora a la solución de un sistema de *n-1* ecuaciones con *n-1* incógnitas.

Hay que notar que el sistema de ecuaciones es tridiagonal. Así que no sólo se han reducido el número de ecuaciones, sino que hemos convertido el sistema de ecuaciones en uno más fácil de resolver.

La ecuación para la función de interpolación por splines cúbicos por intervalos es la siguiente:

$$\begin{aligned}f(x_i) = \frac{f''(x_{i-1})}{6(x_i - x_{i-1})}(x_i - x)^3 + \frac{f''(x_i)}{6(x_i - x_{i-1})}(x_i - x_{i-1})^3 \\ +\left[\frac{f(x_{i-1})}{x_i - x_{i-1}} - \frac{f''(x_{i-1})(x_i - x_{i-1})}{6}\right](x_i - x) \\ +\left[\frac{f(x_i)}{x_i - x_{i-1}} - \frac{f''(x_i)(x_i - x_{i-1})}{6}\right](x - x_{i-1})\end{aligned} \tag{14}$$

Como se puede ver, esta ecuación contiene sólo dos incógnitas -las segundas derivadas al final de cada intervalo. Estas incógnitas pueden ser evaluadas utilizando la siguiente ecuación:

$$(x_i - x_{i-1})f''(x_{i-1}) + 2(x_{i+1} - x_{i-1})f''(x_i) + (x_{i+1} - x_i)f''(x_{i+1})$$
$$= \frac{6}{(x_{i+1} - x_i)}[f(x_{i+1}) - f(x_i)] + \frac{6}{(x_i - x_{i-1})}[f(x_{i-1}) - f(x_i)]$$

Si esta ecuación es escrita para todos los puntos interiores tendríamos como resultado *n-1* ecuaciones con *n-1* incógnitas que era a lo que se pretendía llegar.

3.5 ALGORITMO DE INTERPOLACIÓN POR CONVOLUCIÓN CÚBICA

La interpolación por convolución cúbica es una nueva técnica empleada para el muestreo de datos discretos. Tiene un gran número de características que la hacen muy eficiente para el procesamiento digital de imágenes y que pueden ser desarrolladas eficientemente en una computadora digital. La interpolación por convolución cúbica converge uniformemente a la función interpolada tanto como el incremento de muestreo se acerque a cero.

El algoritmo de convolución cúbica está basado en el algoritmo de convolución desarrollado por Rifman y Bernstein [7].

Una función de interpolación es una función especial de aproximación. Una propiedad fundamental de las funciones de interpolación es que ellas deben coincidir con los datos muestreados en los nodos de interpolación, o puntos de muestreo. Eso quiere decir que si *f* es la función de muestreo, y *g* es la correspondiente función de interpolación, entonces $g(x_k) = f(x_k)$ para cualquier x_k que es un nodo de interpolación.

Para datos igualmente espaciados, muchas funciones de interpolación pueden ser escritas de la siguiente forma:

$$g(x) = \sum_k c_k \, u\left(\frac{x - x_k}{h}\right) \tag{15}$$

Entre las funciones de interpolación que pueden ser caracterizadas se encuentran los splines cúbicos y funciones lineales de interpolación.

En la ecuación 15, h representa el incremento de muestreo, las x_k *'s* son los nodos de interpolación, *u* el kernel de interpolación y *g* es la función de interpolación. Los parámetros c_k*'s* son parámetros que dependen de los datos muestreados, estos son seleccionados de tal forma que cumplan la prerrogativa antes mencionada, es decir, $g(x_k) = f(x_k)$, para cada *k*.

El kernel de interpolación en la ecuación dada convierte datos discretos en funciones continuas en una operación similar a la convolución. Los kernels de interpolación tienen un impacto importante en el comportamiento numérico de las funciones de interpolación. Debido a sus influencias en precisión y eficiencia, los kernels pueden ser usados para crear nuevos algoritmos de interpolación. El algoritmo de convolución cúbica es derivado de un conjunto de condiciones impuestas en el kernel de interpolación las cuales son diseñadas para maximizar la precisión para un nivel dado de esfuerzo computacional.

3.5.1 EL KERNEL DE CONVOLUCIÓN CÚBICA

El kernel de convolución cúbica está compuesto de piezas de un polinomio de tercer grado definido en los subintervalos (-2,-1), (-1,0), (0,1) y (1,2). Fuera del intervalo (-2,2), el kernel es cero. Como una consecuencia de esta condición, el número de datos muestreados para cumplir esta condición es reducido a 4.

El kernel de interpolación debe ser simétrico. De acuerdo a lo anterior se tiene que *u* debe tener la forma:

$$u(s) = \begin{cases} A_1 |s|^3 + B_1 |s|^2 + C_1 |s| + D_1 & 0 < |s| < 1 \\ A_2 |s|^3 + B_2 |s|^2 + C_2 |s| + D_2 & 1 < |s| < 2 \\ 0 & 2 < |s| \end{cases} \tag{16}$$

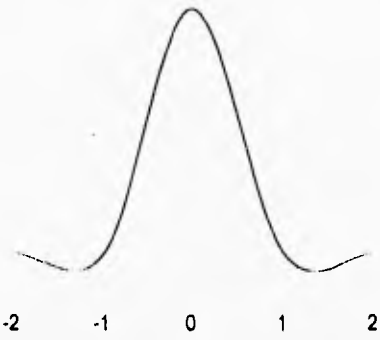


Figura 3.6 El kernel de interpolación por convolución cúbica

El kernel de interpolación debe asumir los valores $u(0) = 1$ y $u(n) = 0$ donde n es cualquier valor entero diferente de cero. Esta condición como veremos más adelante tiene un gran significado computacional. Como h es el incremento de muestreo, la diferencia entre los nodos de interpolación x_j y x_k es $(j-k)h$. Si se sustituye x_j por x en (15) se convierte en

$$g(k_j) = \sum_k c_k u(j-k) \qquad (17)$$

Como $u(j-k)$ es cero a menos de que $j = k$, el lado derecho de la ecuación (17) se reduce a c_j. La condición de interpolación requiere de que $g(x_j) = f(x_j)$. De ahí que $c_j = f(x_j)$, en otras palabras, las c_k 's en la ecuación (15) son reemplazadas por los datos muestreados. Esta es una diferencia substancial en cuanto a procesamiento con respecto a lo ya visto para el método de splines cúbicos. El kernel de interpolación del spline cúbico no es cero para los enteros diferentes de cero, como vimos, se tiene que resolver una matriz para obtener estos datos.

Además de ser 0 o 1 en los nodos de interpolación, el kernel debe ser continuo y tener una primera derivada continua. De acuerdo a estas dos condiciones, un conjunto de ecuaciones pueden ser derivadas de los coeficientes en (16).

Las condiciones $u(0) = 1$ y $u(1) = u(2) = 0$ nos proporcionan cuatro ecuaciones para estos coeficientes:

$$1 = u(0) = D_1$$
$$0 = u(1) = A_1 + B_1 + C_1 + D_1$$
$$0 = u(1^+) = A_2 + B_2 + C_2 + D_2$$
$$0 = u(2) = 8A_2 + 4B_2 + 2C_2 + D_2$$

Otras tres ecuaciones son obtenidas del hecho de que *u'* es continua en los nodos 0,1 y 2:

$$-C1 = u'(0^-) = u'(0^+) = C$$
$$3A_1 + 2B_1 + C_1 = u'(1^-) = u'(1^+) = 3A_2 + 2B_2 + C_2$$
$$12A_2 + 4B_2 + C_2 = u'(2^-) = u'(2^+) = 0$$

Vemos que se tienen 7 ecuaciones y 8 constantes por lo que tenemos que añadir una condición más para obtener una solución única. Rifman y Bernstein usan la condición de que A_2= -1. Sin embargo, se obtendrá el valor de A_2 de tal forma que la función *g* se aproxime a la función original *f* en el grado más alto posible.

Asumamos que $A_2 = a$, por lo que los siguientes siete coeficientes pueden ser determinados en términos de *a* de las siete ecuaciones anteriores.

$$\begin{aligned} D_1 &= 1 \\ A_1 + B_1 + C_1 + D_1 &= 0 \\ a + B_2 + C_2 + D_2 &= 0 \\ 8a + 4B_2 + 2C_2 + D_2 &= 0 \\ C_1 &= 0 \\ 3A_1 - 3a + 2B_1 - 2B_2 + C_1 - C_2 &= 0 \\ 12a + 4B_2 + C_2 &= 0 \end{aligned}$$

Resolviendo este sistema de ecuaciones nos da el conjunto solución para el kernel de interpolación en términos de *a*.

$$u(s) = \begin{cases} (a+2)|s|^3 - (a+3)|s|^2 + 1 & 0 < |s| < 1 \\ a|s|^3 - 5a|s|^2 + 8a|s| - 4a & 1 < |s| < 2 \\ 0 & 2 < |s| \end{cases} \qquad (18)$$

Ahora, si se supone que x es cualquier punto en el cual los datos muestreados van a ser interpolados, entonces x debe estar entre dos nodos de interpolación consecutivos que pueden ser denotados por x_j y x_{j+1}.

Si $s = (x-x_j)/h$ podemos hacer el siguiente arreglo: $(x-x_k)/h = (x-x_j+x_j-x_k)/h = s+j-k$, por lo que la ecuación (15) puede ser reescrita de la siguiente forma:

$$g(x) = \sum_k c_k u(s+j-k) \tag{19}$$

Como existe la condición de que la función u es cero excepto en el intervalo (-2 , 2) y como 0 < s < 1, entonces la ecuación (19) se reduce a:

$$g(x) = c_{j-1}u(s+1) + c_j u(s) + c_{j+1}u(s-1) + c_{j+2}u(s-2) \tag{20}$$

De la ecuación (18) se obtiene entonces:

$$\begin{aligned}
u(s+1) &= a(s+1)^3 - 5a(s+1)^2 + 8a(s+1) - 4a \\
&= as^3 - 2as^2 + as \\
u(s) &= (a+2)s^3 - (a+3)s^2 + 1 \\
u(s-1) &= -(a+2)(s-1)^3 - (a+3)(s-1)^2 + 1 \\
&= -(a+2)s^3 + (2a+3)s^2 - a \\
u(s-2) &= -a(s-2)^3 - 5a)(s-2)^2 - 8a(s-2) - 4a \\
&= -as^3 + as^2
\end{aligned}$$

Si sustituimos las ecuaciones de arriba en la ecuación (20) y agrupamos términos en s tenemos:

$$\begin{aligned}
g(x) = &-[a(c_{j+2} - c_{j-1}) + (a+2)(c_{j+1} - c_j)]s^3 \\
&+[2a(c_{j+1} - c_j - 1) + 3(c_{j+1} - c_j) + a(c_{j+2} - c_j]s^2 \\
&-a(c_{j+1} - c_{j-1})s + c_j
\end{aligned} \tag{21}$$

Si la función de muestreo f tiene al menos las tres primeras derivadas continuas en el intervalo $[x_j, x_{j+1}]$, entonces de acuerdo al teorema de Taylor:

$$c_{j+1} = f(x_{j+1}) = f(x_j) + f'(x_j)h + f''(x_j)h^2/2 + 0h^3 \tag{22}$$

donde $h=x_{j+1}-x_j$. $0h^3$ representa los términos de orden h^3, los cuales tienden a cero a una razón proporcional a h^3. De manera similar podemos obtener las expresiones para c_{j+2} y c_{j-1}.

$$c_{j+2} = f(x_j) + 2hf'(x_j) + 2h^2 f''(x_j) + 0h^3 \quad (23)$$
$$c_{j-1} = f(x_j) - hf'(x_j) + h^2 f''(x_j)/2 + 0h^3 \quad (24)$$

Si sustituimos las ecuaciones (22), (23) y (24) en la ecuación (21) tenemos la función de interpolación por convolución cúbica.

$$\begin{aligned} g(x) = &-(2a+1)[2hf'(x_j) + h^2 f''(x_j)]s^3 \\ &+[(6a+3)hf'(x_j) + (4a+3)h^2 f''(x_j)/2]s^2 \\ &-2ahf'(x_j)s + f(x_j) + 0h^3 \end{aligned} \quad (25)$$

Como $sh=(x-x_j)$, la expansión en series de Taylor para $f(x)$ alrededor de x es:

$$f(x) = f(x_j) + shf'(x_j) + s^2 h^2 f''(x_j)/2 + 0 \quad (26)$$

Restando (25) a (26)

$$\begin{aligned} f(x) - g(x) = &(2a+1)[2hf'(x_j) + h^2 f''(x_j)]s^3 \\ &-(2a+1)[3hf'(x_j) + h^2 f''(x_j)]s^2 \\ &+(2a+1)shf'(x_j) + 0h^3 \end{aligned} \quad (27)$$

Podemos ver que si la función de interpolación $g(x)$ pretende tener por lo menos los tres primeros coeficientes de la expansión en series de Taylor de la función $f(x)$, entonces la ecuación (27) debe ser igual a cero, esto se logra si el valor de a=-1/2.

Esta última aseveración nos da la condición final para el kernel de interpolación: A_2= -1/2

Entonces $f(x)-g(x)= 0h^3$ (28)

Esta última ecuación implica que los errores de interpolación tienden a cero unifórmemente a una razón proporcional a h^3, el cubo del incremento de muestreo.

En otras palabras, *g* es una aproximación de tercer orden de *f*. La condición de que A_2=-1/2 está diseñada para asegurar una aproximación de tercer orden, mientras que otro valor asegura tan sólo una aproximación de primer orden.

Sustituyendo la condición de que A_2= -1/2, el kernel de interpolación de la convolución cúbica quedará de la siguiente manera:

$$u(s) = \begin{cases} \frac{3}{2}|s|^3 - \frac{5}{2}|s|^2 + 1 & 0 < |s| < 1 \\ -\frac{1}{2}|s|^3 + \frac{5}{2}|s|^2 - 4|s| + 2 & 1 < |s| < 2 \\ 0 & 2 < |s| \end{cases} \tag{29}$$

3.5.2 CONDICIONES DE FRONTERA

La función *f* muestreada está definida para todos los números reales pero en la realidad *f* puede ser sólo observada en un intervalo finito. Como el dominio de *f* está restringido a un intervalo finito, digamos [a, b], necesitamos definir las condiciones de frontera.

Antes que nada, los puntos de muestreo x_k deben estar definidos dentro de un intervalo de la imagen, [a,b]. Digamos que $x_k=x_{k-1}+h$ para k= 0, 1, 2, 3..., N.

En el intervalo [a,b], la función de interpolación por convolución cúbica puede ser escrita de la siguiente forma:

$$g(x) = \sum_{k=-1}^{N+1} c_k u\left(\frac{x - x_k}{h}\right) \tag{30}$$

Para determinar *g* para toda *x* en el intervalo [a,b] los valores de c_k para *k*= -1,0,1,2,3,...,N+1 son necesarios. Para *k*=0,1, 2 ..., N, *c(k)* = *f(k)*, pero para *k*= -1 y

k = N+1 los valores de f son desconocidos, como x_{-1} y x_{N+1} caen fuera del intervalo de observación los valores de c_{-1} y c_{N+1} son condiciones de frontera.

Las condiciones de frontera deben ser escogidas de tal forma que $g(x)$ sea una aproximación de tercer orden de $f(x)$ para toda x contenida en el intervalo [a, b]. Para encontrar la condición apropiada para el término a la izquierda del intervalo se debe suponer que x es un punto en el subintervalo $[x_0, x_1]$. De acuerdo a este valor de x, la función de interpolación se reduce a:

$$g(x) = c_{-1}u(s+1) - c_0 u(s) + c_1 u(s-1) + c_2 u(s- \qquad (31)$$

donde $s = (x-x_0)/h$.

Sustituyendo la ecuación (29) en la anterior y agrupando términos

$$g(x) = s^3[c_2 - 3c_1 + 3c_0 - c_{-1}]/2 - s^2[c_2 - 4c_1 + 5c_0 - 2c_{-1}]/2 \qquad (32)$$
$$+ s[c_1 - c_{-1}]/2 + c_0$$

Si la función g es una aproximación de tercer orden de la función f, entonces el término en s^3 debe ser cero. Esto significa que el coeficiente c_{-1} puede ser escogido de la siguiente forma:

$$c_{-1} = c_2 - 3c_1 + 3c_0 \qquad \text{o bien,}$$

$$c_{-1} = f(x_2) - 3f(x_1) + 3f(x_0) \qquad (33)$$

Después de sustituir (33) en (32), la función de interpolación se convierte en:

$$g(x) = s^2[f(x_2) - 2f(x_1) + f(x_0)]/2 + s[-f(x_2) + 4f(x_1) - 3f(x_0)]/2 + f(x_0) \qquad (34)$$

Todo lo que queda es demostrar que esta última ecuación es una aproximación de tercer orden para $f(x)$. Primero expandemos las funciones $f(x_2)$ y $f(x_1)$ en una serie de Taylor alrededor de x_0.

$$f(x_2) = f(x_0) + 2f'(x_0)h + 2f''(x_0)h^2 + 0(h^3) \qquad (35)$$
$$f(x_1) = f(x_0) + f'(x_0)h + f''(x_0)h^2/2 + 0(h^3) \qquad (36)$$

Reemplazando $f(x_2)$ y $f(x_1)$ en la ecuación (34) tenemos el siguiente resultado:

$$g(x) = f(x_0) + f'(x_0)sh + f''(x_0)s^2h^2/2 + 0h^3 \quad (37)$$

Como $sh=x-x_0$, la expansión en series de Taylor de la función $f(x)$ alrededor de x_0 es:

$$f(x) = f(x_0) + f'(x_0)sh + f''(x_0)s^2h^2/2 + 0h^3 \quad (38)$$

Restando la ecuación (37) a la (38)

$$f(x) - g(x) = 0h^3$$

Así, la condición de frontera obtenida en la ecuación (33) también cumple que la aproximación sea de tercer orden para $f(x)$ en el intervalo $x_0 < x < x_1$. Haciendo un análisis similar al anterior podemos encontrar la condición para c_{N+1}. Si x se encuentra en el intervalo $[x_{N-1}, x_N]$, la condición de frontera es la siguiente:

$$c_{N+1} = 3f(x_N) - 3f(x_{N-1}) + f(x_{N-2}) \quad (39)$$

En conclusión, usando el kernel de interpolación cúbica definido en la ecuación (29) y las condiciones de frontera dadas en (33) y (39) se puede dar una completa descripción de la función de interpolación por convolución cúbica.

Cuando $x_k < x < x_{k+1}$, la función de interpolación es:

$$g(x) = c_{k-1}(-s^3 + 2s^2 - s)/2 + c_k(3s^3 - 5s^2 + 2)/2 + c_{k+1}(-3s^3 + 4s^2 + s)/2 + c_{k+2}(s^3 - s^2)/2$$

Donde:

$s = (x - x_k)/h$

$c_k = f(x_k)$ para $k = 0,1,2,\ldots,N$;

$c_{-1} = 3f(x_0) - 3f(x_1) + f(x_2)$ y

$c_{N+1} = 3f(x_N) - 3f(x_{N-1}) + f(x_{N-2})$.

3.6 REFERENCIAS

[1] BURT, PETER Y ADELSON, E. **THE LAPLACIAN PYRAMID AS A COMPACT IMAGE CODE.** IEEE TRANSACTIONS ON COMMUNICATIONS, 1983.

[2] CHELLAPPA, R. **DIGITAL IMAGE PROCESSING**. IEEE COMPUTER SOCIETY PRESS.

[3] ESCALANTE, B. **PERCEPTUALLY ASSESSED DIGITAL PROCESSING OF MEDICAL IMAGES**. TESIS DOCTORAL, 1992.

[4] GONZÁLEZ, R. Y WOODS, R. **DIGITAL IMAGE PROCESSING**. ADDISON WESLEY, 1992.

[5] HELD, G. **DATA COMPRESSION**. ADDISON WESLEY, 1987.

[6] JAIN, A. **FUNDAMENTALS OF DIGITAL IMAGE PROCESSING**. PRENTICE HALL, 1989.

[7] KEYS, R. **CUBIC CONVOLUTION INTERPOLATION FOR DIGITAL IMAGE PROCESSING**. IEEE TRANSACTIONS ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, 1981.

[8] LOW, A. **INTRODUCTORY COMPUTER VISION AND IMAGE PROCESSING**. Mc. GRAW-HILL, 1993.

[9] NAKAMURA, S. **MÉTODOS NUMÉRICOS APLICADOS CON SOFTWARE**. PRENTICE HALL, 1992.

[10] OLIVER, D. Y ANDERSON, S. **TRICKS OF THE GRAPHICS GURUS**. SAMS PUBLISHING, 1993.

[11] RUSS, J. **THE IMAGE PROCESSING HANDBOOK**. CRC PRESS,1992.

[12] SCLAROFF, S. Y PENTLAND, P. **ON MODAL MODELING FOR MEDICAL IMAGES: UNDERCONSTRAINED SHAPE DESCRIPTION AND DATA COMPRESSION**., MIT MEDIA LABORATORIES, 1994.

4

SISTEMA DE VISUALIZACIÓN ULTRASÓNICA

4.1 INTRODUCCIÓN

Gran parte de la investigación y desarrollo en el campo de la visión computarizada ha sido dirigido a la producción de dispositivos que puedan presentar un modelo visual de lo que no es visible para el ojo humano debido a obstrucciones no removibles [5]. En particular, mucho trabajo médico no podría haberse logrado sin el conocimiento preciso de la presentación en tres dimensiones de órganos individuales de nuestro cuerpo. Un ejemplo de ello es el scanner para el cerebro, el cual presenta al especialista una serie de cortes en dos dimensiones del cerebro que más tarde permitirán la creación de un modelo tridimensional completo interno y externo del mismo.

De igual forma, dispositivos de barrido ultrasónico, como el que se describirá a continuación, son usados en medicina para inspecciones fetales. En estos dispositivos se requiere una reconstrucción previa al despliegue para poder obtener una imagen con mejor calidad.

En este capítulo, como se mencionó en el párrafo anterior, se describirá el sistema completo de visualización ultrasónica, además citar sus características más importantes.

4.2 SISTEMA DE VISUALIZACIÓN

El diagrama a bloques del sistema global puede ser mostrado de la siguiente manera:

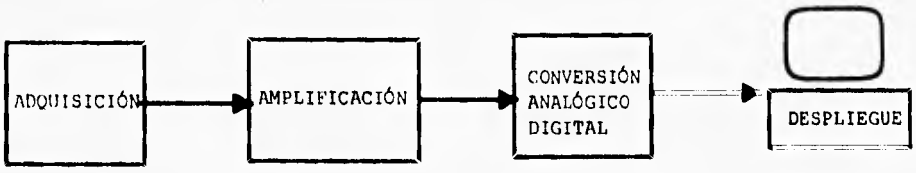


Figura 4.1 Diagrama a bloques del sistema

El dibujo anterior nos da una visión muy general del sistema, sin embargo, se describirá cada una de las partes más a fondo.

Como se vio en la figura 4.1, el sistema en principio consta de una etapa de adquisición, otra de acondicionamiento de la señal y el despliegue. Cada una de ellas presenta ciertas características que a continuación se presentan.

4.2.1 ADQUISICIÓN

Las técnicas para la adquisición de datos de cuerpos sólidos involucran la medición de la reflexión de una señal transmitida sobre el cuerpo mencionado o por la medición de la absorción de otra señal [7]. Radares y ultrasonido son ejemplos de las primeras, mientras que la tomografía potencial aplicada (ejemplo, la medición de resistencia a través del cuerpo para crear un tomograma) puede verse como una combinación de las dos técnicas [7].

En este proyecto se utiliza la técnica de reflexión de una señal. La parte de adquisición consiste en un arreglo de 64 sensores colocados en línea recta, los cuales serán excitados en grupos de 8 por una señal eléctrica, en primera instancia serán disparados los sensores del 1 al 8, despues del 2 al 9, seguidos del 3 al 10 y así sucesivamente hasta disparar los 56 grupos de sensores posibles. Al encontrar un cuerpo sólido, parte de la señal será reflejada y parte transmitida, las diferentes señales reflejadas a través del conjunto de transductores son sumadas para obtener una señal total; esta señal es de una magnitud muy pequeña, por lo que debe amplificarse para contener los niveles de voltaje necesarios para su conversión digital.

A la salida del convertidor analógico-digital se obtendrán 256 bytes representando la primer columna de la imagen. Siendo 56 los disparos de grupos de sensores, se obtendrá al final una imagen de 256 renglones por 56 columnas que ya se encuentran listos para su despliegue. La figura 4.2 muestra claramente esta fase de adquisición y acondicionamiento de la señal.

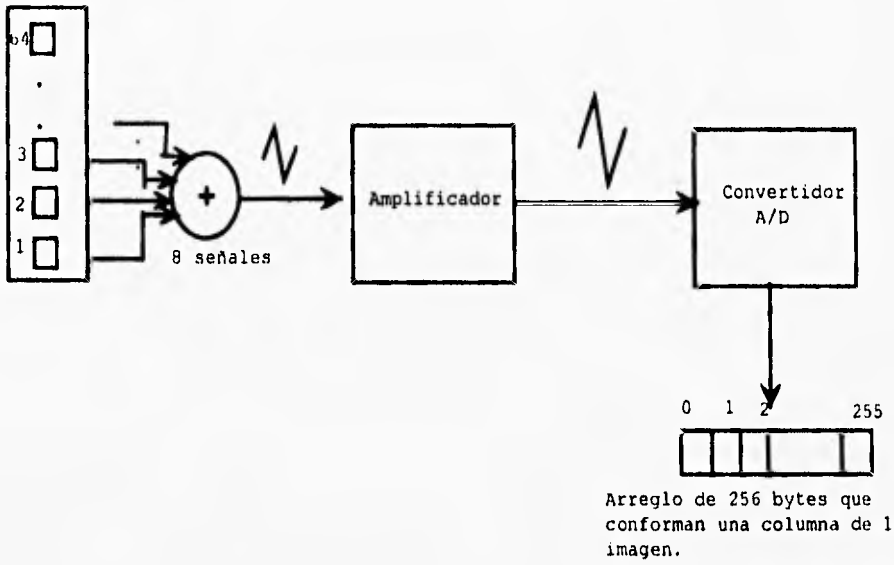


Figura 4.2 Sistema de aquisición de la imagen, se excitan los primeros 8 sensores (1-8), se hace un corrimiento y se excitan 8 más (2-9), y así sucesivamente hasta completar los 56 grupos.

La tasa de adquisición de los datos en el convertidor es de 1 Mbyte/seg por lo que conformar una imagen total tardaría 14.336 mseg.

4.2.2 DESPLIEGUE

El despliegue se realiza en una computadora personal con monitor VGA y resolución de 640*480 pixeles, que es un modo de video estándar. El despliegue se realiza a partir de la recepción, por el bus de la computadora, de los datos enviados por el convertidor, los cuales son mapeados diréctamente a la memoria de video.

El tamaño de la imagen es de 256 columnas por 56 renglones, se puede ver que las dimensiones de la misma son pequeñas con respecto a la resolución del monitor, es por ello necesario interpolar la imagen al triple sobre las columnas para que el observador pueda apreciar de una mejor manera la imagen.

Es típico que los sistemas de tomografía ultrasónica proporcionen una imagen de pequeño tamaño debido al uso de pocos sensores. Para reconstruir la imagen es necesario utilizar algoritmos de interpolación rápidos y fáciles de implementar [2][4].

La interpolación más básica, como se vio en el capítulo 3, es la repetición de pixeles y esta fue la primera que se implementó. Siendo tan simple, el mismo procesador central podría realizar este tipo de procesamiento (figura 4.3). Como uno de los objetivos es el desplegar al menos 25 imágenes o cuadros por segundo está justificado utilizar este algoritmo ya que se obtienen alrededor de 33 cuadros por segundo.

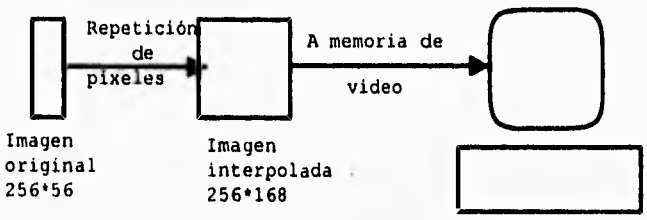


Figura 4.3 Despliegue

Sin embargo, este tipo de algoritmos a pesar de ser fáciles de implementar no proporcionan buenos resultados, se requiere utilizar algoritmos de orden mayor como el de convolución cúbica [3][4] que proveen una mejor calidad de la imagen.

Es evidente que el número de operaciones y la cantidad de procesamiento es mucho mayor, es por ello necesario añadir una etapa más a nuestro diagrama de la figura 4.1, la de procesamiento.

Siendo un cómputo intensivo y de acuerdo al volumen de información manejado, se utiliza una arquitectura paralela basada en transputers que nos ofrece la capacidad de tener varios procesadores trabajando en conjunto para resolver la aplicación. Existe una ventaja más de utilizar una arquitectura paralela, la de poder incorporar nuevos módulos sin necesidad de modificar nuestro sistema global. Así, si se pretende realizar otro tipo de procesamiento a la imagen, como el filtrado, almacenaje o realce, entre otras técnicas de procesamiento digital de imágenes, no

es necesario vover a diseñar nuestro sistema, sino basta incorporar un nuevo conjunto de procesadores (en comunicación con la red existente) que se encarguen de realizar la nueva aplicación deseada.

Así, el esquema global de nuestro sistema queda formado como se muestra en la figura 4.4.

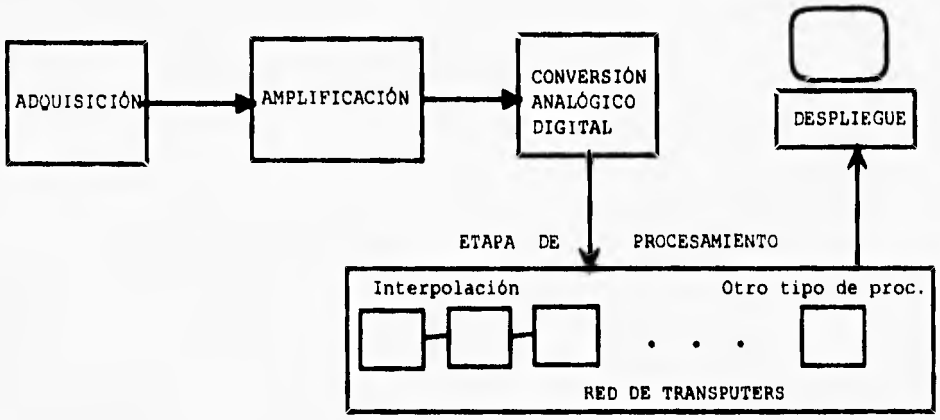


Figura 4.4 Esquema global del sistema de visualización

Cabe señalar que este sistema de visualización ultrasónica representa una alternativa muy económica en comparación con los sistemas de ultrasonido comerciales los cuales ascienden a varios miles de dólares. Su simplicidad, calidad y bajo costo hacen de este, un proyecto innovador en su ramo.

El siguiente capítulo muestra la etapa de procesamiento sobre la red de transputers, es decir, la implementación paralela de los algoritmos de interpolación vistos en el capítulo 3.

4. 3 REFERENCIAS

[1] CHELLAPPA, R. **DIGITAL IMAGE PROCESSING**. IEEE COMPUTER SOCIETY PRESS.

[2] ESCALANTE, B. **PERCEPTUALLY ASSESSED DIGITAL PROCESSING OF MEDICAL IMAGES**. TESIS DOCTORAL, 1992.

[3] GONZÁLEZ, R. Y WOODS, R. **DIGITAL IMAGE PROCESSING**. ADDISON WESLEY, 1992.

[4] JAIN, A. **FUNDAMENTALS OF DIGITAL IMAGE PROCESSING**. PRENTICE HALL, 1989.

[5] LOW, A. **INTRODUCTORY COMPUTER VISION AND IMAGE PROCESSING**. Mc. GRAW-HILL, 1993.

[6] OLIVER, D. Y ANDERSON, S. **TRICKS OF THE GRAPHICS GURUS**. SAMS PUBLISHING, 1993.

[7] RUSS, J. **THE IMAGE PROCESSING HANDBOOK**. CRC PRESS,1992.

[8] SCLAROFF, S. Y PENTLAND, P. **ON MODAL MODELING FOR MEDICAL IMAGES: UNDERCONSTRAINED SHAPE DESCRIPTION AND DATA COMPRESSION**., MIT MEDIA LABORATORIES, 1994.

5

IMPLEMENTACIÓN PARALELA DE ALGORITMOS DE INTERPOLACIÓN

5.1 INTRODUCCIÓN

Para realizar un sistema paralelo se deben tomar en cuenta varios factores, algunos de ellos son la granularidad, balanceo de cargas, la topología de la red de procesadores y el modelo mediante el cual se paralelizará nuestro algoritmo. De la correcta elección de cada uno de estos factores dependerá la eficiencia de nuestro sistema, es por eso importante describir cada uno de ellos.

El presente capítulo, además de describir cada uno de estos conceptos, comprende la implementación de los algoritmos de interpolación mediante el modelo farming, el cual trata de explotar al máximo el grado de paralelismo de nuestro algoritmo. Se incluye además, los programas de la aplicación en pseudocódigo para una mayor comprensión de las partes que conforman nuestro sistema.

5.2 GRANULARIDAD Y BALANCEO DE CARGAS

Un programa que utiliza procesamiento en paralelo para su solución es una colección de procesos que se ejecutan al mismo tiempo en diferentes procesadores como se vio en el capítulo 2. La manera de repartir los procesos en los procesadores debe ser tal que la ejecución del programa se realice en el tiempo más corto.

Desarrollar un programa paralelo no es una tarea sencilla, se debe realizar un modelo que obtenga los máximos beneficios de utilizar más procesadores. Idealmente es posible tomar el código de una aplicación secuencial y de alguna manera "paralelizarla" para explotar su paralelismo inherente, sin embargo, sus dependencias secuenciales hacen esto inapropiado [5]. Lo más común es desarrollar algoritmos que exploten su paralelismo en una forma más explícita; estos algoritmos se basan en la partición, repartición y ejecución óptima del problema global, de tal forma que el tiempo de ejecución sea mínimo.

Al mapear un problema a una arquitectura paralela es necesario primero dividir el problema en segmentos que se ejecuten en paralelo, determinando la manera como los procesadores se comunican y sincronizan entre sí

independientemente de la naturaleza del problema. Dentro de la etapa de asignación de procesos a los procesadores existen dos elementos de la programación paralela que tienen una intensa relación y determinan el grado de paralelismo de nuestro sistema, estas son la granularidad y el balance de trabajo [8], los cuales se definen a continuación.

5.2.1 GRANULARIDAD

La granularidad, como se mencionó anteriormente, es una medida del paralelismo del sistema, es un indicador de la cantidad de procesamiento que cada procesador puede realizar independientemente, en relación con el tiempo que ocupa para intercambiar información con otros procesadores.

Una granularidad fina (fine granularity) es usada cuando cada tarea consiste en unas cuantas instrucciones u operaciones que consumen típicamente fracciones de segundo de tiempo del procesador. Esto da un incremento en el grado de paralelismo que permite establecer mecanismos para lograr una distribución uniforme de las tareas. Sin embargo, esto se logra muchas veces a costa de un gran overhead de comunicaciones (entiéndase por overhead de comunicaciones al tiempo, en la mayoría de los casos excesivo, de recepción y transmisión de datos por parte de un procesador) para mantener y distribuir las tareas y agrupar los resultados. En muchos casos, el tiempo de procesador consumido por estas comunicaciones puede exceder al tiempo mismo consumido por la ejecución de la sola tarea.

En constraste a lo anterior, la granularidad gruesa (coarse granularity) da un incremento en complejidad de las tareas, las cuales pueden consistir en un gran número de instrucciones que consumen mucho más tiempo del procesador. Existe menos potencial para explotar el paralelismo ya que existen pocas tareas a ser realizadas, pero el overhead de comunicaciones se reduce, obviamente debido al poco tránsito de información [5].

No existe una solución establecida en cuanto al problema de seleccionar el tamaño del grano en una aplicación paralela. La granularidad de una aplicación se

obtiene en algunos casos a partir del grado de interacción dentro de las tareas de un problema y algunas veces de la arquitectura paralela a ser usada. Un sistema compuesto de un grupo reducido de procesadores unidos entre sí por ligas de comunicación relativamente lentas se pueden ajustar a un esquema de granularidad gruesa puesto que en estas el número de comunicaciones es reducido. Por otro lado, un sistema con elementos de procesamiento de poca capacidad pero que se comunican con frecuencia, se ajustan al esquema de granularidad fina [8].

5.2.2 BALANCEO DE CARGAS

El balanceo de cargas es el proceso por el cual los elementos del dominio de datos son asignados a los procesadores con la finalidad de maximizar la utilización de los mismos y reducir al mínimo el tiempo de ejecución del programa [9]. Podemos decir que un programa está balanceado si su cómputo es igualmente distribuido en todos los procesadores del sistema. Valiosos ciclos de procesador son perdidos si algunos nodos tienen que esperar a que los otros terminen, es por ello que se debe asegurar un balanceo de trabajo para que la eficiencia de nuestro sistema sea alta.

Si la carga de trabajo es conocida se puede estáticamente realizar un distribución balanceada de trabajo, por otro lado, si la carga no es conocida entonces se puede realizar un balanceo dinámico.

BALANCEO ESTÁTICO

El balanceo estático de trabajo requiere una estimación u obtención del perfil del programa, de tal forma que se conozca de antemano la carga de trabajo por cada procesador que conforma la red. Esta técnica tiene la finalidad de que todos los procesos terminen al mismo tiempo y así reducir el tiempo en el que los procesadores se encuentran desocupados [9].

El balanceo estático proporciona un alto rendimiento pero, desafortunadamente, la estimación no es siempre segura y el modelado requiere un íntimo conocimiento de la estructura de control de la aplicación, de las propiedades del dominio de datos y del hardware.

BALANCEO DINÁMICO

Si no es posible conocer la carga de tabajo por procesador, el balanceo se puede ajustar dinámicamente durante la ejecución de un programa. Estas técnicas se aplican mediante un sistema operativo o el uso de algún tipo de software durante la ejecución.

El overhead está siempre asociado con este tipo de técnicas. Antes de incorporar un esquema de balanceo dinámico en un algoritmo se debe tener en cuenta la reducción en tiempo requerido para completar el trabajo contra el tiempo requerido para balancear la carga. Si se requiere más tiempo para balancear la carga que para terminar el trabajo entonces no es algo práctico utilizar este método [5].

El balanceo dinámico siempre depende de la forma en la que el problema es distribuido en el sistema. Distribuir un problema entre los diferentes nodos de una computadora paralela puede ser realizado mediante la descomposición del dominio o descomposición del control. En la descomposición del dominio, el dominio de los datos de entrada son divididos y cada división de trabajo es asignada a cada procesador. En la descomposición del control no son los datos los divididos sino más bien es el problema en sí el que se particiona y se distribuye entre los procesadores [6].

Existen diferentes técnicas para particionar los algoritmos y alcanzar mejores niveles de paralelismo, estos son descritos en la siguiente sección.

5.3 ESTRATEGIAS PARA EXPLOTAR EL PARALELISMO

Como su contraparte secuencial, las aplicaciones paralelas tienen también una fase de análisis, diseño, implementación y prueba. Sin embargo, a pesar de que estas fases tienden a lo mismo que la parte secuencial: la buena implementación de la aplicación, ellas tienen ciertas características que las hacen diferentes.

Se dice que la mayor diferencia se encuentra en la fase de diseño (considerada la más importante de las citadas), ya que es imperativo explotar el paralelismo al máximo para no desperdiciar recursos. De acuerdo a lo anterior, la elección de la estrategia que descomponga el problema global en subproblemas de menor nivel para ser mapeados en diferentes procesadores, es importante para obtener el mayor grado de paralelismo [3].

Diferentes estrategias pueden ser adoptadas para extraer paralelismo de un algoritmo. Como cada algoritmo permite varios tipos alternativos de paralelismo para ser explotados al mismo tiempo, es importante conocer los pros y los contras de cada posible solución así como escoger el más apropiado para obtener los resultados que queremos de la aplicación.

Se presentan a continuación tres de las estrategias más utilizadas: el paralelismo geométrico, el paralelismo algorítmico y el proceso farming, como se verá más adelante, estos algoritmos atienden aspectos de mucha importancia como lo son el máximo grado de paralelismo a ser explotado, la posibilidad de variar el tamaño del grano, el uso de la memoria en cada procesador y la cantidad de comunicación requerida.

5.3.1 PARALELISMO GEOMÉTRICO

La primera de las estrategias a ser descrita es el paralelismo geométrico. Muchos problemas físicos tienen una estructura geométrica regular, cada objeto o grupo de datos teniendo interacción con los objetos o grupos de datos vecinos. En estos casos el llamado paralelismo geométrico puede ser usado y el dominio del

problema es descompuesto en varios *granos,* cada uno de los cuales es asignado a un procesador diferente [3]. Supongamos que se tiene un cubo grande, este puede ser dividido en un número de cubos más pequeños, así, la geometría espacial del sistema es dividida en una forma simétrica, asumiendo una distribución uniforme de los datos sobre la geometría para permitir una solución más fácil de agrupar. Cada una de esas pequeñas unidades actúa como una entidad cuasi independiente, es decir, responsable de sus datos y de su región espacial. El cómputo realizado por cada unidad es sumado para obtener la solución al problema. Este tipo de estrategia es también conocida como descomposición de estructuras de datos [4].

El paralelismo geométrico es indudablemente una de las más simples y más eficientes técnicas para desarrollar aplicaciones paralelas. Una de las principales ventajas de la descomposición del problema global radica en el hecho de que no solamente puede ser variado el tamaño del grano asignado a cada procesador sino que también es muy simple evaluar su efecto (junto con el efecto de los parámetros de hardware del sistema) en el rendimiento global y la eficiencia del procesamiento [3].

5.3.2 PARALELISMO ALGORÍTMICO

Otro tipo fundamental de paralelismo puede ser explotado partiendo el algoritmo usado en un cierto número de procesos de acuerdo al flujo de datos (descomposición de flujo de datos). Cada uno de esos procesos se ejecuta completamente en paralelo con sus propios datos o con los datos recibidos por otros procesos. Con el fin de explotar el paralelismo algorítmico es necesario identificar las secciones de procesamiento que pueden ser ejecutadas en paralelo, cada sección implementa sólo una parte del algoritmo y es asignada a un procesador. La interconexión lógica entre los procesadores que intervienen es definida por el flujo estático de datos entre los módulos secuenciales previamente identificados y es en forma definitiva altamente dependiente del algoritmo usado.

La presencia de dependencias secuenciales en los datos conduce normalmente a la descomposición del problema en una o más estructuras en *pipeline,* donde un flujo de datos es conducido a través de un arreglo lineal de

procesadores ejecutando diferentes operaciones sobre sus datos de entrada. El *pipelining* de procesadores es una forma muy flexible de descomponer las tareas, consiste en traslapar los procesamientos y comunicaciones [1]. El pipelining es una forma de explotar el paralelismo en el tiempo.

El paralelismo algorítmico se relaciona con el modelo de flujo de datos, ya que el procesamiento se conduce a partir de la disponibilidad de los datos, ignorando el orden de las instrucciones. Este modelo es equivalente al modelo de programación funcional, en el que el proceso se sigue mediante una secuencia determinada de instrucciones. La mayoría de los programas secuenciales basados en el modelo funcional pueden interpretarse paralelamente mediante el modelo de flujo de datos [8].

El paralelismo algorítmico se considera como una de las formas más populares para paralelizar una aplicación.

5.3.3 EL PROCESO FARMING

El concepto de farmer es aplicable a problemas cuya solución puede ser llevada a cabo mediante la descomposición del problema en partes pequeñas, donde estas partes son independientes unas de otras. Como las partes son independientes, cada una puede ser ejecutada concurrentemente y al final ser sumadas para darle solución al problema global. La solución es análoga a la de un granjero (farmer) supervisando las tareas de los trabajadores de la granja, cada trabajador realiza una tarea independiente de la que realizan los demás trabajadores, de ahí su nombre [4].

El modelo farming fue propuesto por May y Shepard [7] para la solución de problemas con un alto grado de cómputo comparado con el de comunicaciones. Este modelo se ha convertido en uno de los más populares por la comunidad involucrada en el procesamiento paralelo. Sus aplicaciones son muy diversas pero se utiliza principalmente en la realización de gráficos y el procesamiento digital de imágenes.

Un modelo de este tipo puede ser diseñado por un conjunto de procesos de acuerdo al modelo de CSP mencionado en el capítulo 2. Un proceso puede ser nominado como el granjero, el cual controlará la organización y desarrollo del trabajo, y se encargará de asignar trabajo a sus trabajadores subordinados. Los procesos llamados trabajadores (workers) son modelados mediante programas idénticos, es decir, cada uno de ellos estará realizando la misma acción pero en forma paralela. Cuando cada trabajador termina su tarea, el proceso farmer se encarga de asignarle una nueva inmediatamente, de tal forma que el procesador esté ocioso la menor cantidad de tiempo. Resumiendo entonces, una granja de trabajadores divide el trabajo, se distribuye entre los mismos trabajadores y en cuanto terminan una tarea se les asigna otra hasta que todo el problema haya llegado a su fin. En este tipo de aplicaciones es necesario utilizar comunicaciones entre procesos ejecutándose en el mismo procesador por lo que es necesario utilizar una configuración adecuada para evitar un overhead.

5.4 TOPOLOGÍAS

Parte importante también en el desarrollo de sistemas paralelos y la cual es muy necesaria tomar en consideración para obtener el mejor rendimiento del sistema es la topología o configuración de la red de procesadores. La buena elección de la misma influenciará determinantemente en el resultado obtenido de la aplicación.

Para lograr el intercambio de información entre un cierto número de procesadores es necesario que estos se comuniquen, conectándose entre sí mediante una red, siguiendo un esquema específico o topología.

El término topología se relaciona con un conjunto de elementos o nodos, y un conjunto de parejas no ordenadas de nodos diferentes llamados segmentos. En un sistema paralelo los nodos corresponden a los procesadores, mientras que los segmentos representan las ligas entre ellos [8]

Dentro de los tipos de topologías que se conocen encontramos dos vertientes principales, las estáticas y las dinámicas. Las primeras se refieren a aquellas

topologías que no pueden modificar su estructura y que permanecen a lo largo de la ejecución del programa. Las topologías dinámicas pues, son las opuestas a las anteriores, estas sí pueden modificar su estructura.

Los sistemas multiprocesos pueden ser descritos por su topología y por su nivel de interconexión. El nivel de interconexión es una medida del número de unidades por las cuales un mensaje debe pasar para llegar de un procesador *X* a un procesador *Y*. En cuanto a las topologías, podemos citar las cuatro básicas que son: no restringida, lineal, el anillo y la estrella. Otro tipo de topologías como la de árbol, malla e hipercubos son variantes de esas cuatro topologías puras [2].

A continuación se citan las características de cada una de ellas.

5.4.1 NO RESTRINGIDA

Esta topología se basa en un esquema de interconexión libre, es decir, no sigue ningún modelo geométrico en su configuración. Cada procesador es conectado directamente a otro con el cual se desea que entable comunicación. Aún cuando el esquema es muy simple, generalmente es poco práctico ya que al aumentar el número de nodos y segmentos, el número de conexiones se incrementa rápidamente. La ventaja de estos sistemas es que se puede lograr un alto nivel de acoplamiento, como los buses son dedicados a la comunicación entre dos procesadores, no hay conflicto con los otros que esperan el acceso al mismo bus.

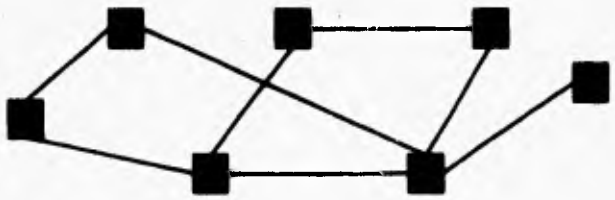


Figura 5.1 La topología no restringida no es muy usada mas que por los sistemas más simples. Si el número de procesadores crece, el el número de interconexiones se vuelve muy grande.

5.4.2 Lineal

La topología lineal es considerada como la organización más sencilla de una red de comunicación, consiste en conectar en línea todos los nodos de la red. Cada nodo se conecta como máximo a otros dos nodos, excepto los extremos, que sólo se conectan a uno solo.

Su mayor ventaja radica en la simplicidad de su implementación, se requieren de N-1 canales para comunicar N nodos, sin embargo, la comunicación entre nodos no adyacentes representa quizá la mayor desventaja, y es que se necesita que los mensajes pasen por nodos intermedios antes de llegar al nodo destino, lo que implica una pérdida de tiempo por la espera y sincronización. Este tipo de topologías son utilizadas en sistemas donde el número de procesadores es pequeño.

Figura 5.2 Configuración lineal, para N procesadores se requieren N-1 canales para comunicarlos.

5.4.3 ANILLO

En una topología de anillo cada procesador se conecta únicamente a sus dos procesadores vecinos. Uno de ellos es denominado el vecino anterior y el restante el vecino posterior. Un nodo recibe información del vecino anterior y se la transfiere al nodo posterior, de esta forma, la información fluye a través del anillo solamente en una dirección y los paquetes de datos son pasados a través de todos los nodos del anillo, cabe decir que cada paquete de información contiene una dirección de destino, así, cuando un nodo recibe un paquete verifica la dirección destino y si esta corresponde con la dirección del nodo lee el paquete, de otra forma lo transfiere al nodo posterior.

La topología de anillo ofrece ciertas ventajas para algunas clases de sistemas multiprocesadores débilmente acoplados y, en caso de fallar algún nodo no afecta el desempeño del sistema pues se puede designar una ruta alterna para alcanzar el nodo final [2].

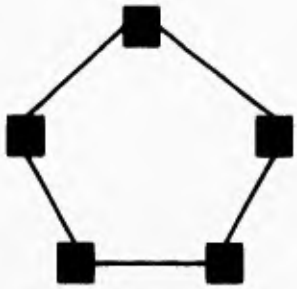


Figura 5.3 Topología de anillo. Cada procesador es conectado solamente a sus dos vecinos más cercanos. La información fluye en una sola dirección.

5.4.4 ESTRELLA

Este tipo de topologías emplean un procesador central encargado de organizar y distribuir la información a todos los elementos que conforman la red. La ventaja de esta topología es que reduce el tiempo de comunicación en la red, sólo se requieren dos saltos para comunicar dos nodos, no importando el tamaño de la misma.

Por otro lado, la topología de estrella es tan buena como su nodo central; si éste falla, el sistema entero falla. Para asegurar la eficiencia de la red es necesario que el nodo central sea más rápido que los otros nodos, de tal forma que pueda distribuir la información rápidamente y ningún procesador tenga tiempo ocioso.

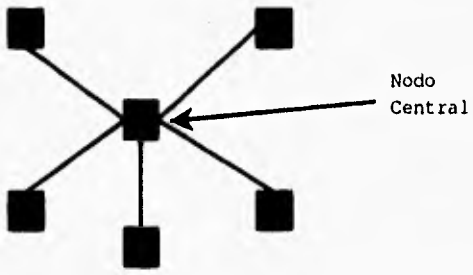


Figura 5.4 Topología de estrella. La labor del procesador central es la de coordinar el trabajo de los demás procesadores de la red.

5.4.5 TOPOLOGÍAS DERIVADAS

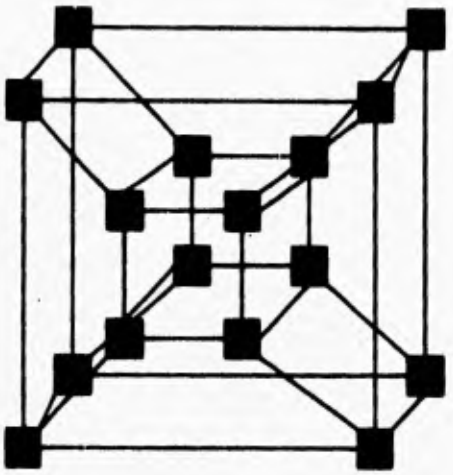
Como se mencionó en la sección anterior, existen topologías derivadas de las cuatro principales ya citadas, ellas son las de hipercubo, malla y árbol (véase la figura 5.5).

Un hipercubo de n dimensiones puede conectar $N=2^n$ procesadores en la forma de un cubo binario de dimensión *n*. Cada esquina (vértice o nodo) del hipercubo consiste de un elemento de procesamiento y su memoria asociada. En esta topología, cada nodo está conectado directamente a n vecinos.

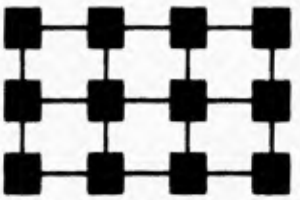
La topología de malla se forma por un arreglo bidimensional de nodos que se conectan entre sí. En general, se emplea un arreglo de tipo rectangular donde cada nodo se conecta a otros cuatro mediante igual número de canales. La ventaja de esta topología es que aumenta la capacidad de comunicación entre los nodos de la red y, además, permite el uso de rutas o caminos alternos de comunicación, esto es, que un nodo pueda transferir información a otro por una o más rutas.

Esta topología es utilizada comúnmente en el diseño de redes de conexión, una variante, el árbol binario, se aplica constantemente en este tipo de sistemas. En el árbol binario cualquier par de nodos se comunican entre sí atravesando la estructura en busca de un nodo común.

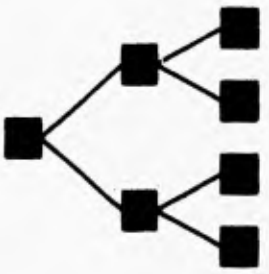
Presentan la ventaja de poder realizar varias comunicaciones entre nodos simultáneamente, ya que durante la comunicación entre dos nodos es probable que no se utilicen todos los canales de la red. Su principal desventaja reside en la situación de falla de cualquier nodo, si alguno falla es posible que pueda fallar incluso la mitad de la red.



a) Hipercubo de orden 4



b) Malla



c)Árbol

Figura 5.5 Topologías derivadas

5.5 IMPLEMENTACIÓN

Se ha llegado a la parte de la implementación de nuestro algoritmo, el cual como se mencionó anteriormente debe ser diseñado correctamente para obtener el máximo grado de paralelismo de la aplicación y obtener el menor tiempo de ejecución. De acuerdo a lo anterior, se escogió el modelo de proceso farming para la implantación paralela de la aplicación porque contiene características que proporcionan un mejor rendimiento de los sistemas.

El proceso farming es muy utilizado por aplicaciones cuyo nivel de cómputo es bastante grande, en este caso sabemos que al hablar de imágenes, implícitamente intuimos que la cantidad de datos e información es enorme, por lo cual este modelo se ajusta bastante bien. Existen, por supuesto, otras características que lo hacen más viable, y es que el proceso farming puede ser implementado en un número indeterminado de procesadores sin tener que hacerle modificaciones al código. Por otro lado presenta la característica de contener un

mecanismo intrínseco de balanceo dinámico de cargas, esto se explicará más adelante.

De acuerdo a los aspectos que se deben considerar para realizar un sistema paralelo vistas al principio de este capítulo, además del balanceo de cargas, como lo son la granularidad y el nivel de interconexión de los procesadores (topologías), el proceso farming es una opción inmejorable. El tamaño del grano puede variarse sin tener que modificar todo el sistema y, puede ser implementado en cualquier tipo de topología, desde la más simple como lo es la lineal, hasta la más compleja como lo pueden ser los hipercubos.

El proceso farming consiste en dos tipos de procesadores: un controlador simple y uno o más trabajadores. El controlador genera nuevas tareas y se las envía a los trabajadores tan pronto como estas sean requeridas, otra de sus funciones es la de agrupar los resultados originados por los trabajadores de la red. Los procesadores trabajadores (workers) consisten en dos partes específicas, la aplicación en sí y el ruteo de información, el cual se divide en dos procesos: el de ruteo de tareas y el de ruteo de resultados. El modelo farming puede ser ilustrado de la siguiente forma:

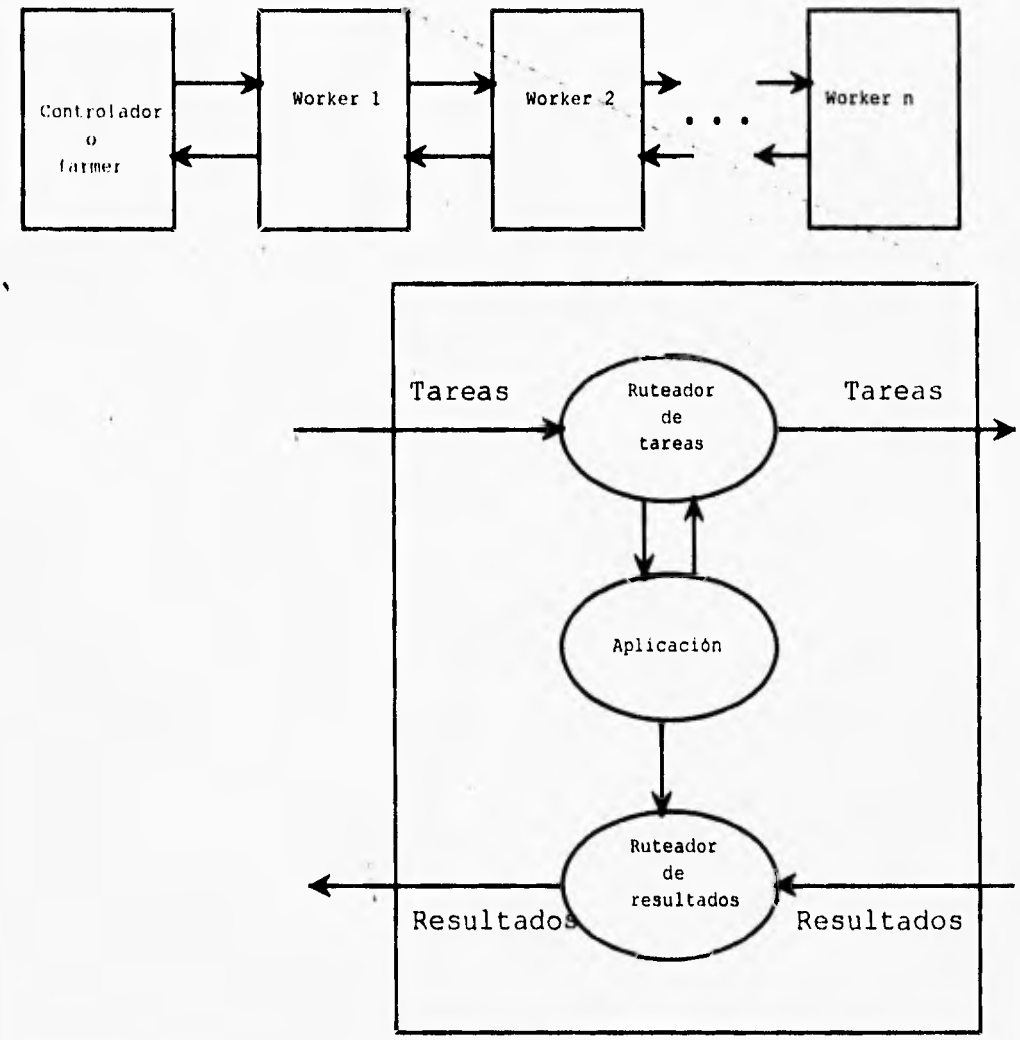


Figura 5.8 Esquema del proceso farming

Consideremos de la figura 5.8 que cada elipse representa un subproceso que se encuentra ejecutándose concurrentemente con los otros. El primero de ellos será considerado como proceso *ruteador de tareas*, el segundo como proceso de *aplicación* (en este caso el algoritmo de interpolación) y el último como proceso *ruteador de resultados*.

El proceso *ruteador de tareas* en cada trabajador contiene un buffer para almacenar las tareas, así, si el procesador recibe una nueva tarea se almacena en

el buffer si este se encuentra vacío, de no ser así, se la envía al siguiente trabajador. Cuando una tarea ha sido procesada es transferida al ruteador de resultados el cual la retorna al proceso controlador o farmer. Cada vez que se haya realizado una tarea, el controlador envía otra para reemplazar la que ha sido terminada. El sistema es inicializado de tal forma que se envía trabajo suficiente para llenar la capacidad de los trabajadores y de sus buffers.

Para ilustrar la simplicidad de este modelo, se explicarán los dos procesos ruteadores que forman parte de cada trabajador. El mecanismo por el cual el ruteador de tareas opera es el siguiente: las tareas son comunicadas desde el controlador en una dirección de izquierda a derecha, como se ve en la figura 5.8. Para cualquier trabajador si el proceso encargado del procesamiento de la aplicación se encuentra desocupado, entonces el proceso de ruteo de tareas tan pronto reciba un paquete se lo asignará para que se realice el procesamiento del mismo. Por otro lado, si el proceso de la aplicación se encuentra ocupado, entonces pueden suceder dos cosas, la primera de ellas es que la tarea recién llegada sea almacenada en el buffer si este se encuentra vacío, o bien asignarle la tarea al siguiente trabajador. Cuando el proceso de la aplicación se encuentra desocupado, es entonces que la tarea es removida del buffer y asignada a este proceso.

El proceso de la aplicación debe enviar un mensaje al proceso de ruteo de tareas de que se encuentra desocupado para que este proceso le envíe una tarea inmediatamente, como se ve, se busca que el procesador se encuentre siempre trabajando.

El proceso de ruteo de resultados es un proceso simple que realiza la recabación de los resultados provenientes del proceso de la aplicación del mismo procesador así como de aquellos provenientes de los trabajadores a lo largo de la cadena, como se ve también en la figura anterior. Los resultados son dirigidos hacia el proceso controlador.

El procesador controlador o farmer es responsable de la distribución de tareas hacia los trabajadores y del recibimiento y manejo de los resultados de los mismos. El controlador inicia el procesamiento entre los trabajadores enviando a la "granja" un número equivalente al doble del número de trabajadores existentes, una

tarea para el proceso de aplicación y la otra para el buffer. Después de esta primera etapa solo se enviará una tarea tan pronto se reciba un resultado.

La forma general del proceso farming en pseudocódigo es la siguiente:

```
ProcPar
    proceso farmer
    proceso trabajador 1
    proceso trabajador 2
    .
    proceso trabajador n
```

Todos los trabajadores y el farmer que es el controlador, deben trabajar en paralelo corriendo en diferentes procesadores.

Los trabajadores constan de tres subprocesos corriendo concurrentemente, estos son el del *ruteador de tareas*, el de la *aplicación* y el del *ruteador de resultados*. El primero y último procesos deben ejecutarse con una prioridad alta ya que es muy importante el establecer comunicación con otros trabajadores, bien sea para recibir la nueva tarea o para enviar el trabajo realizado. El proceso encargado de realizar la aplicación se ejcuta en prioridad baja, de tal forma que sólo es ejecutado cuando no existe comunicación. Es prioritario que exista comunicación, de lo contrario corremos el riesgo de que el procesador se encuentre desocupado durante varios ciclos de reloj.

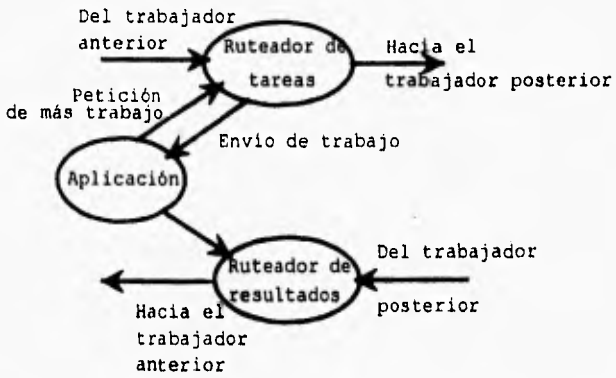


Figura 5.9 Esquema de los subprocesos de cada trabajador y sus canales de comunicación.

Expresado en pseudocódigo se tendría la siguiente estructura:

```
        /* Procesador  trabajador (worker) */
ProcRunHigh     proceso ruteador de tareas
ProcRunLow      proceso encargado de ejecutar la aplicación
ProcRunHigh     proceso ruteador de resultados
```

Asimismo podemos expresar cada uno de estos subprocesos de la siguiente manera:

```
/*Subproceso ruteador de tareas */

MIENTRAS haya tareas disponibles
    PROC ALT
        - recibir petición por parte del proceso de la aplicación
        - recibir una nueva tarea del controlador
     SI se cumple la primer condición (petición del proceso de aplicación) {
            SI buffer=lleno
               enviar datos del buffer a este proceso
            SI NO
               proceso de la aplicación se encuentra desocupado
    }

    SI se recibe una nueva tarea{
            SI proceso de aplicación se encuentra desocupado
               enviar la tarea a este proceso
            SI  buffer = vacío
               almacenar la tarea en el buffer
            SI NO
               enviar tarea al siguiente trabajador
     }
```

El proceso de la aplicación es el más simple de todos, puede ser expresado de la siguiente forma:

/*Proceso de la aplicación */

Recibir tarea del proceso ruteador de tareas
Realizar procesamiento de la aplicación (en este caso la interpolación de la imagen)
Enviar resultados al proceso ruteador de resultados
Enviar petición de más trabajo al proceso ruteador de tareas

En cuanto al proceso ruteador de resultados este puede ser implementado así:

/*Proceso ruteador de resultados*/

MIENTRAS haya tareas disponibles
PROC ALT
- Recibir tarea terminada por el proceso de la aplicación corriendo en el mismo procesador
- Recibir tarea terminada por parte de los otros trabajadores.

SI se cumple la primer condición (término de la tarea en el mismo procesador)
recibir el trabajo
enviar la información al farmer
SI se reciben tareas terminadas por los otros trabajadores
recibir el trabajo
enviar la información al farmer

Por último tenemos al proceso que actúa como farmer o controlador, este proceso es el encargado de agrupar el total de las tareas y es el que sincroniza a los demás trabajadores, su implementación en pseudocódigo es la siguiente:

/* proceso farmer */

Enviar trabajo suficiente para cada procesador de tal forma que se llene el buffer y la aplicación empiece a procesar.
MIENTRAS número de tareas no se haya terminado
recibir tarea terminada de los trabajadores
enviar una nueva tarea

La aplicación consiste, de acuerdo al planteamiento de nuestro problema en el capítulo 1, en la interpolación de la imagen que es enviada del sistema de adquisición.

Se mencionaba en párrafos anteriores que el proceso farming contiene intrínsecamente un balanceo de cargas, esto se puede explicar comparando el proceso farming con un proceso común, el cual consiste en recibir una tarea, procesarla, enviarla al procesador central y enviar una petición de una nueva tarea.

El esquema de este proceso sería el que a continuación se muestra.

Procesador central	Procesador Trabajador
	Enviar petición de nueva tarea
Recibir la petición Enviar unidad de trabajo	
	Recibir unidad de trabajo Procesar unidad de trabajo Enviar nueva petición

Tabla 5.1 Diagrama de tiempo ocioso del procesador con un esquema secuencial

Los cuadros en blanco de la tabla enterior representan tiempos en los cuales el procesador se encuentra ocioso. Por el contrario, el proceso farming, al utilizar un buffer que almacena una tarea y la cual es enviada al proceso encargado de realizar la aplicación tan pronto este se halle desocupado, minimiza el tiempo ocioso del procesador. Aquí se muestra su esquema.

Procesador Central	Buffer	Procesador trabajador
	buffer = vacío	Enviar petición de tarea
Recibir petición Enviar unidad de trabajo		
	Se almacena en el buffer *buffer*=lleno	Enviar nueva petición
Recibir petición Enviar unidad de trabajo	Enviar lo contenido en el buffer al proceso encargado de desarrollar la aplicación Se recibe nueva tarea *buffer* = lleno	Se recibe la unidad de trabajo y se empieza a procesar
	buffer = lleno	Procesando unidad de trabajo
	enviar tarea almacenada al proceso de la aplicación	Término del procesamiento Enviar nueva petición

Tabla 5.2 Diagrama de tiempo ocioso (proceso farming)

Con el almacenaje de una tarea más se logra que el trabajador realice un traslape entre lo que sería la petición de una nueva tarea y la recepción de la misma con el procesamiento de la tarea presente. De acuerdo al grado en el cual este traslape sea logrado se puede lograr un alto rendimiento del sistema. Por ejemplo, si el tiempo requerido para procesar el grano es mucho más grande que la comunicación, entonces podremos observar casi un traslape total entre las comunicaciones y el procesamiento. Este esquema logra que el tiempo ocioso del procesador sea mínimo pues siempre contará con una tarea extra que empezará a ser procesada tan pronto acabe con la primera tarea. Otra de las ventajas del proceso farming radica en que todas las aplicaciones de los trabajadores son iguales, de tal forma que todas acabarán al mismo tiempo y se minimiza el riesgo de que un procesador tenga que esperar a otros a que acaben.

En el siguiente capítulo se muestran los resultados obtenidos bajo este esquema.

5.6 REFERENCIAS

[1] BENÍTEZ, H. **DISEÑO DE UNA ARQUITECTURA PARA PROCESAMIENTO PARALELO DE SEÑALES DOPPLER DE ULTRASONIDO**. TESIS DE LICENCIATURA, F.I., 1994.

[2] CLEMENTS, A. **MULTIPROCESSOR SYSTEMS**. ELECTRONIC AND WIRELESS WORLD, 1991.

[3] DE CARLINI, U. Y DE VILANO, U. **TRANSPUTERS AND PARALLEL ARCHITECTURES**. MIT PRESS, 1991.

[4] GALLETLY, J. **OCCAM2**. PITMAN PUBLISHING, 1990.

[5] GREEN, S. **PARALLEL PROCESSING FOR COMPUTER GRAPHICS**. MIT PRESS, 1991.

[6] LEWIS, G. **INTRODUCTION TO PARALLEL COMPUTING**. PRENTICE HALL, 1992.

[7] MAY, D. **OCCAM**. PRENTICE HALL, 1993.

[8] ORTEGA, J.L. **ESTUDIO Y EVALUACIÓN DE LA PROGRAMACIÓN ORIENTADA A OBJETOS EN UNA ARQUITECTURA PARALELA**. TESIS DE MAESTRÍA. 1995.

[9] THIÉBAUT, D. **PARALLEL PROGRAMMING IN C FOR THE TRANSPUTER**. 1995.

6

ANÁLISIS DE TIEMPOS DE EJECUCIÓN Y MÉTRICAS DE DESEMPEÑO

6.1 INTRODUCCIÓN

Existen métricas fundamentales para evaluar el desempeño de un sistema paralelo, como el speedup, la eficiencia, la fracción serial y la eficacia, todas ellas fueron descritos en el capítulo 2.

En este capítulo, además de analizar el rendimiento del sistema para cada una de las implementaciones realizadas, se hará una comparación de la calidad de las imágenes interpoladas por los algoritmos de interpolación ya descritos en el capítulo 3, así como un análisis costo-beneficio de los mismos.

Hay que enfatizar claramente que el sistema en general pretende ser una alternativa muy económica de tomografía ultrasónica, por lo que se debe utilizar el mínimo número de recursos para su implementación, de ahí la importancia del análisis costo-beneficio.

6.2 REDUCCIÓN DE LOS ALGORITMOS DE INTERPOLACIÓN

Es conocido que, dentro de cualquier procesador, las operaciones aritméticas requieren más ciclos de procesador que algunas otras instrucciones como los saltos y las cargas a registros. De a cuerdo a lo que se viene planteando en este trabajo de tesis acerca del uso mínimo de recursos, es necesario realizar el menor número de estas operaciones.

En el transputer utilizado en este trabajo (ver apéndice A), las operaciones aritméticas toman tiempos que a continuación se citan: 350 nseg para la suma y resta, 550 nseg para la multiplicación y 850 nseg para la división. Se puede ver entonces que se debe buscar la forma de reducir el número de multiplicaciones y, sobre todo, de divisiones para mejorar los tiempos de procesamiento.

Para ilustrar lo anterior, se describen a continuación dos ejemplos de implementaciones, la del algoritmo lineal y la de convolución cúbica, donde se minimiza el número de multiplicaciones y divisiones.

Recordemos del capítulo 3, que la interpolación lineal está basada en una recta que pasa por los puntos extremos de cada intervalo, como se ve en la figura 6.1.

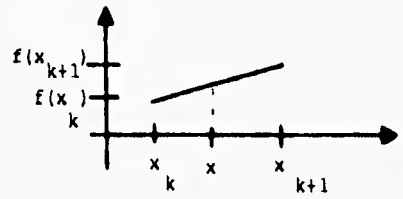


Figura 6.1 Interpolación lineal

El algoritmo de interpolación lineal puede ser descrito por la siguiente fórmula:

$$g(x) = f(x_k) + m_k(x - x_k) \qquad x_k < x < x_{k+1}$$

$$\text{donde } m_k = \frac{f(x_k) - f(x_{k+1})}{x_k - x_{k+1}},$$

f(x) es la función original y

g(x) la función interpolada

$$g(x) = f(x) \qquad x = x_k.$$

Siendo los datos de la imagen equiespaciados, entonces la diferencia $x_k - x_{k+1} = -1$, con lo que simplifica nuestra ecuación a:

$$g(x) = f(x_k) + (f(x_{k+1}) - f(x_k))(x - x_k), \quad x_k < x < x_{k+1} \qquad (1)$$
$$g(x) = f(x_k), \qquad x = x_k.$$

La ecuación 1 representa una forma general de expresar la función de interpolación lineal para imágenes digitales, sin embargo, de acuerdo al problema considerado, la interpolación será constante, al triple sobre las columnas, por lo que $(x - x_k)$ dará 1/3 o 2/3 dependiendo de si es el primer dato interpolado dentro del intervalo, o el segundo, respectivamente. Sintetizando la ecuación 1 para nuestro problema particular:

$g(x_1) = f(x_k)$, el primer dato interpolado es igual al de la imagen original.

$$g(x_2) = f(x_k) + \frac{1}{3}(f(x_{k+1}) - f(x_k)) \quad \text{, para el segundo dato interpolado} \qquad (2)$$

$$g(x_3) = f(x_k) + \frac{2}{3}(f(x_{k+1}) - f(x_k)) \text{, para el tercer dato.}$$

Se puede observar que para el primer dato interpolado se utiliza solo la asignación, para el segundo una resta, una suma, una división y la asignación, mientras que para el tercero se usa una suma, una resta, una multiplicación, una división y una asignación, lo que nos da un total de 3 asignaciones, 2 sumas, 2 restas, una multiplicación y dos divisiones.

Sin embargo, mediante una pequeña modificación a estas últimas ecuaciones se pueden obtener mejores resultados. Igualemos $\frac{1}{3}(f(x_{k+1}) - f(x_k))$ a una variable auxiliar:

$$aux = \frac{1}{3}(f(x_{k+1}) - f(x_k))$$

Con lo que la ecuación 2 puede ser expresada como:

$$g(x_1) = f(x_k) \qquad (3)$$
$$aux = \frac{1}{3}(f(x_{k+1}) - f(x_k))$$
$$g(x_2) = f(x_k) + aux$$
$$g(x_3) = g(x_2) + aux$$

La cual nos da un total de 4 asignaciones, 2 sumas, una resta y una división, que, comparándola con la ecuación 2, nos permite utilizar menos ciclos del procesador y, por ende, se reduce el tiempo de procesamiento.

De igual forma, se puede hacer una simplificación del algoritmo de interpolación cúbica visto en la sección 5 del capítulo 3.

La función de interpolación del algoritmo de convolución cúbica es la siguiente:

$$g(x) = c_k \qquad x = x_k \qquad (4)$$
$$g(x) = c_{k-1}(-s^3 + 2s^2 - s)/2 + c_k(3s^3 - 5s^2 + 2)/2 + c_{k+1}(-3s^3 + 4s^2 + s)/2 + c_{k+2}(s^3 - s^2)/2 \qquad x_k < x < x_{k+1}$$

Donde:

$s = (x - x_k)/h$

$c_k = f(x_k)$ para $k = 0,1,2,...,N$;

$c_{-1} = 3f(x_0) - 3f(x_1) + f(x_2)$ y

$c_{N+1} = 3f(x_N) - 3f(x_{N-1}) + f(x_{N-2})$.

De acuerdo al problema considerado, s puede ser 1/3 o 2/3 por las mismas razones descritas para el algoritmo lineal. Sustituyendo este valor en la ecuación (4) tenemos:

$$g(x) = c_k \qquad x = x_k \qquad (5)$$
$$g(x_2) = -\frac{2}{27}c_{k-1} + \frac{7}{9}c_k + \frac{1}{3}c_{k+1} - \frac{1}{27}c_{k+2}$$
$$g(x_2) = -\frac{1}{27}c_{k-1} + \frac{1}{3}c_k + \frac{7}{9}c_{k+1} - \frac{2}{27}c_{k+2}$$

Dando un total de 3 asignaciones, 4 sumas, 4 restas, 4 multiplicaciones y 8 divisiones, pero, utilizando el mínimo común denominador podemos modificar este conjunto de ecuaciones de la siguiente forma:

$$g(x_1) = c_k \qquad (6)$$
$$g(x_2) = \frac{-2c_{k-1} + 21c_k + 9c_{k+1} - c_{k+2}}{27}$$
$$g(x_3) = \frac{-c_{k-1} + 9c_k + 21c_{k+1} - 2c_{k+2}}{27}$$

Ocupando 3 asignaciones, 4 sumas, 4 restas, 6 multiplicaciones y 2 divisiones, con lo cual se reduce en mucho, el esfuerzo computacional.

6.3 TIEMPOS DE COMUNICACIÓN

Antes de implementar el modelo farming descrito en la sección 5 del capítulo anterior, es importante determinar el tiempo que toma a dos procesadores el

comunicarse, para obtener una idea de cuánto tiempo disponemos para el procesamiento. De acuerdo a las características del sistema, se necesitan procesar al menos 25 cuadros por segundo, o bien, un cuadro cada 40 mseg.

Así, es necesario transferir una matriz de 256*56 elementos, interpolarla al triple sobre las columnas y regresar la imagen interpolada (en este caso de 256*168). Es importante notar que esto puede hacerse enviando pequeñas submatrices que en conjunto conformen la imagen original. Para esto se variará el tamaño de las submatrices para determinar la dimensión óptima, es decir, el tamaño del grano ideal que nos permita obtener el menor tiempo en cuanto a comunicación y sincronización.

Cabe señalar que aunque la comunicación de un byte es proporcional a la de transmitir un millón de ellos, se obtienen tiempos diferentes al enviar submatrices de tamaño menor ya que es necesario utilizar ciclos más grandes e incrementar contadores mayores (véase figura 6.2), en síntesis: realizar más operaciones.

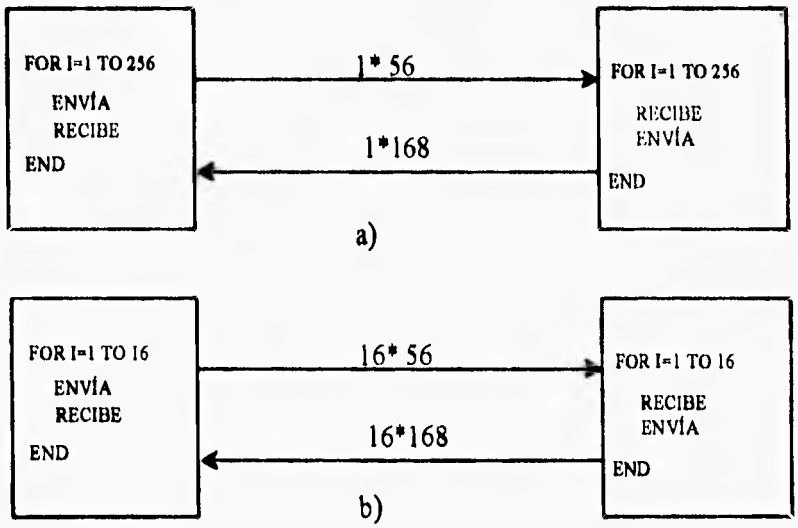


Figura 6.2 Esquema de la comunicación de submatrices, a) se envían 256 vectores de 56 elementos, b) se envían 16 matrices de 16*56 elementos

La tabla 6.1 muestra los tiempos de transmisión de la imagen original del procesador central hacia otro procesador de la red. El valor *n* representa el número de vectores de 56 elementos que serán enviados.

n	Tiempo de comunicación [mseg]
1	36.898
2	34.096
3	33.369
4	33.000
5	32.757
8	32.463
16	32.181
32	32.041
64	31.973

Tabla 6.1 Tiempos de comunicación

Su gráfica asociada es la que se muestra:

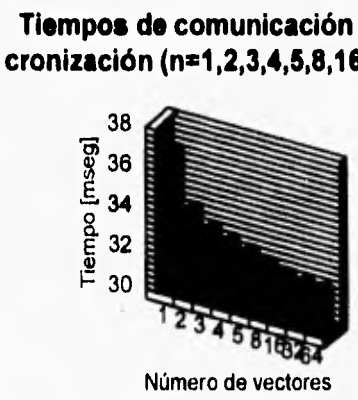


Figura 6.3 Gráfica de los tiempos de comunicación entre transputers

De la gráfica se puede observar que el menor tiempo de comunicación se obtiene cuando el tamaño de la submatriz es mayor, esto debido, como se mencionó, al menor uso de ciclos y contadores.

Los resultados anteriores muestran además que el tiempo de comunicación de una imagen completa es muy alto, con lo que para lograr el objetivo planteado en esta tesis de desplegar al menos 25 cuadros por segundo sería muy complejo, ya que se debe procesar cada cuadro en 1/25 seg = 40 mseg, de los cuales se ocuparían 31.973 mseg en comunicaciones y restarían 8.027 mseg para realizar la interpolación. Incrementando el número de procesadores se pueden lograr abatir los tiempos de comunicación, idealmente se reduciría por un factor 1/p, donde p es el número de procesadores, sin embargo, para efectos de costo, es necesario que el número de procesadores involucrados en la aplicación sea el mínimo.

Para lograr lo anterior es necesario comunicar la menor cantidad de datos, es decir, que no exista redundancia, y esto se puede lograr manteniendo en un buffer los datos de la imagen original en el procesador central, la cual será enviada por este a cada uno de los procesadores que conforman la red, esto no cambia con respecto al esquema anterior, sin embargo, estos últimos procesadores devolverán al procesador central sólo los nuevos datos interpolados ya no los de la imagen original, con lo que se reduce en 1/5 el tamaño de los datos a comunicarse, transfiriéndose por tanto una imagen de 256*56 y regresándose una de 256*112. El esquema de lo dicho anteriormente es mostrado a continuación.

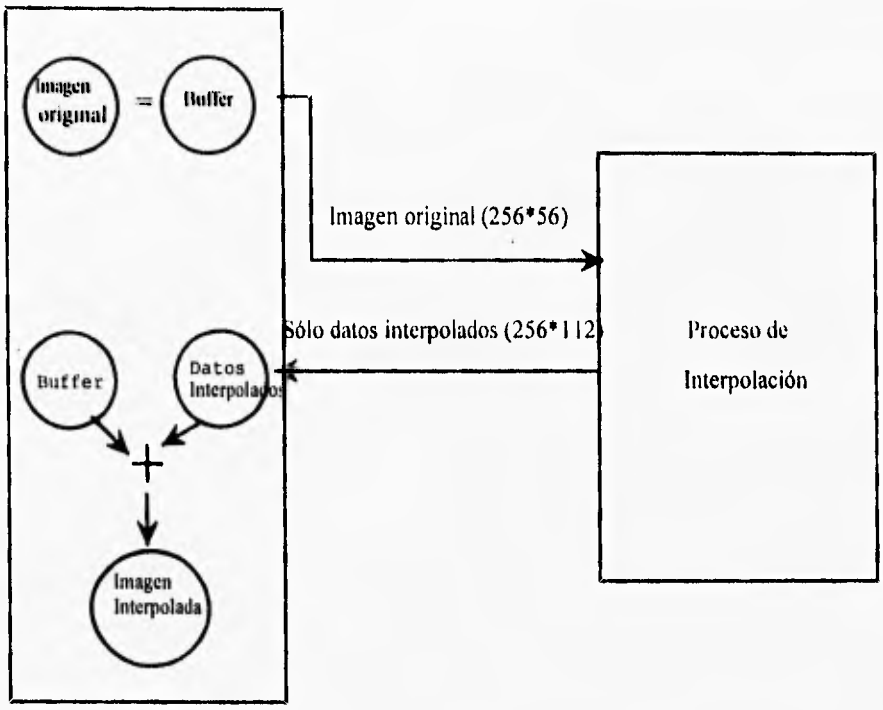


Figura 6.4 Esquema de comunicación, se mantiene en un buffer la imagen en el procesador central para no transferir información redundante.

De acuerdo a este nuevo esquema los tiempos de comunicación se muestran a continuación:

n	Tiempo de comunicación [mseg]
1	28.979
2	26.229
3	25.484
4	25.115
5	24.868
8	24.578
16	24.297
32	24.156
64	24.087

Tabla 6.2 Tiempos de comunicación con el nuevo esquema

Dando la siguiente gráfica:

Figura 6.5 Tiempos de comunicación bajo el nuevo esquema.

De igual forma que en el primer esquema, la disminución en el tiempo de comunicación es debida al poco número de operaciones para sincronizar submatrices más grandes. Convendría, por tanto, elegir submatrices de 8,16, 32 o 64 renglones pero eso solo sería con respecto a la comunicación, faltaría medir los tiempos de procesamiento para poder determinar el tamaño del grano que permita obtener un tiempo menor.

6.4 TIEMPOS DE PROCESAMIENTO

Para la implementación del sistema considerado en este trabajo se utilizaron tres algoritmos de interpolación: de primero, segundo y tercer orden que son splines lineales, splines cuadráticos y la convolución cúbica respectivamente.

Los tiempos de procesamiento de cada uno de estos algoritmos en forma secuencial (sólo interpolación, sin contar comunicación) se muestran a continuación, donde *n* representa el número de vectores de 56 elementos que fueron procesados para dar el mismo número de vectores pero con 112 elementos (se recuerda que sólo se obtendrán y transferirán los nuevos datos interpolados, los datos originales

de la imagen serán almacenados en un buffer en el procesador central y no se necesita procesarlos).

n	Splines lineales [mseg]	Splines cuadráticos [mseg]	Convolución cúbica [mseg]
1	89.920	134.592	166.016
2	81.472	126.272	158.208
3	78.713	125.803	156.075
4	77.954	124.608	154.432
5	77.800	123.768	153.861
8	77.690	123.840	153.668
16	77.500	123.620	153.368

Tabla 6.3 Tiempos de procesamiento de los tres algoritmos

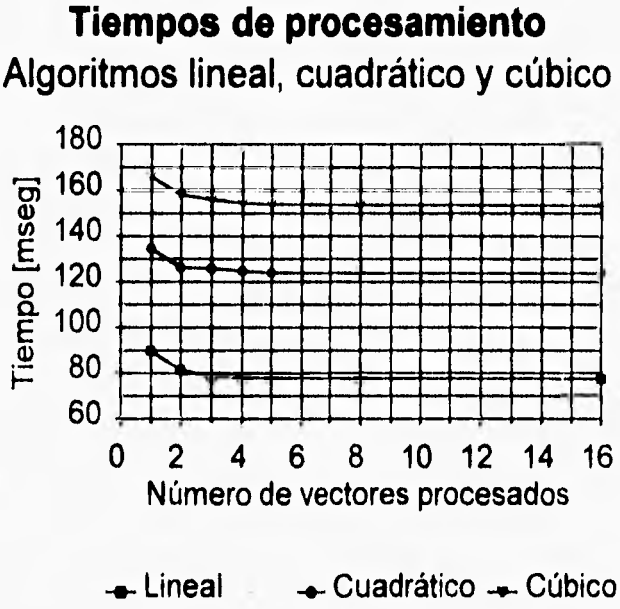


Figura 6.6 Tiempos de procesamiento de los algoritmos de interpolación

De igual forma que en la comunicación, se obtiene un tiempo menor cuando el tamaño de la submatriz es mayor, en este caso es de 16 vectores. Otra razón por la que se debe elegir este número de vectores, es debido a que la imagen original

está conformada por 256 renglones, cada uno con 56 elementos, es deseable que la dimensión de la submatriz, en cuanto a los renglones, sea un divisor de 256, pues así evitamos el tener que enviar matrices de tamaño variable con lo cual se tendría que modificar la aplicación para cada tamaño de la submatriz. Siendo el tamaño de 16 renglones entonces basta con realizar 16 veces el proceso para conformar una imagen.

Hasta el momento se tienen tiempos de comunicación y de procesamiento por separado, para poder realizar una comparación con nuestro proceso farming es necesario determinar el tiempo que se necesitaría para ejecutar nuestra aplicación en forma secuencial.

La tabla 6.4 muestra los resultados obtenidos:

n	Splines lineales [mseg]	Splines cuadráticos [mseg]	Convolución cúbica [mseg]
1	130.916	173.307	211.277
2	120.194	165.297	206.055
3	117.982	164.932	199.161
4	116.840	164.9	199.158
5	114.704	163.41	197.725
8	114.506	163.234	197.700
16	124.20	163.073	197.622

Tabla 6.4 Tiempos de proceso secuencial

La tabla 6.4 comprende los tiempos requeridos para realizar la aplicación secuencialmente en dos procesadores; el primero de ellos se encarga de distribuir los datos, mientras que el segundo lleva a cabo la interpolación

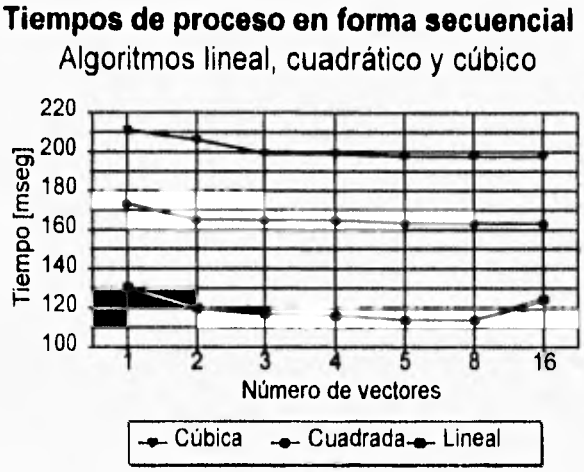


Figura 6.7 Tiempo de proceso total (interpolación y comunicación) en forma secuencial

De la gráfica anterior se puede observar que para los algoritmos de segundo y tercer orden, el tamaño óptimo del grano es de 16 vectores, mientras que para el algoritmo lineal es de 8. Esto es debido a que el tiempo de procesamiento lineal es mucho menor comparado con los dos restantes, lo que implica que exista un mayor tiempo de espera cuando el tamaño de la submatriz se incrementa.

6.5 TIEMPOS DE PROCESAMIENTO BAJO EL ESQUEMA FARMING

De lo visto anteriormente, se puede concluir que el tamaño del grano influye determinantemente en la eficiencia de la aplicación secuencial, es importante hacerlo notar porque igualmente lo será bajo un esquema farming paralelo. Para la implementación de este esquema se utiliza el mismo proceso descrito en el capítulo anterior.

Recordemos que el proceso farming consta de 2 tipos de procesadores, el farmer, el cual es el encargado de distribuir las tareas y recolectar el trabajo de los

trabajadores que conforman la red de procesadores, estos últimos tienen la función de recibir el trabajo, procesarlo y enviarlo al farmer.

De igual forma, se implementaron los tres algoritmos de interpolación ya mencionados en dos procesadores (uno actuando como farmer y el otro como trabajador), dando los siguientes resultados.

n	Splines lineales [mseg]	Splines cuadráticos [mseg]	Convolución cúbica [mseg]
1	146.231	195.383	229.614
2	118.695	162.719	207.404
4	104.609	150.936	188.947
8	100.805	150.610	186.265
16	105.999	147.768	183.777

Tabla 6.5 Tiempos de procesamiento con el modelo farming (un trabajador)

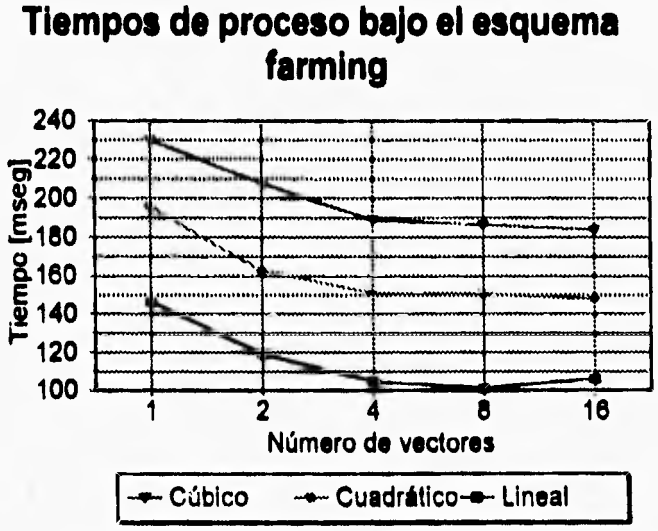


Figura 6.8 Tiempos de proceso con el esquema farming para los algoritmos de interpolación

Es importante hacer notar que para el algoritmo lineal, el tamaño del grano óptimo es de 8 vectores mientras que para los otros dos algoritmos es de 16. De

haber escogido un grano compuesto por uno o dos vectores se obtendría un tiempo mayor que el tiempo secuencial (arriba de los 15 mseg por implementación enviando un solo vector, y cerca de los 2 mseg si se enviaran 2 vectores), situación que no es deseable ya que no se estaría explotando el paralelismo.

Los mejores tiempos en la implementación secuencial fueron de 114.506 mseg, 163.073 mseg y 197.622 mseg para los algoritmos de primero, segundo y tercer orden, respectivamente, mientras que bajo el proceso farming son de 100.805 mseg, 147.766 mseg y 183.777 mseg. Se observa claramente que se justifica el uso de esta estrategia pues nos ayuda a extraer el máximo grado de paralelismo de nuestro algoritmo.

De acuerdo a los resultados descritos, los algoritmos de interpolación son implementados en una red de procesadores configurados en diferentes topologías.

6.5.1 TIEMPOS DE PROCESAMIENTO EN UNA TOPOLOGÍA LINEAL

La topología lineal consiste en conectar a cada procesador con otros dos nodos, excepto los de los extremos que solo se conectan a uno. Uno de esos extremos es el procesador que actúa como farmer.

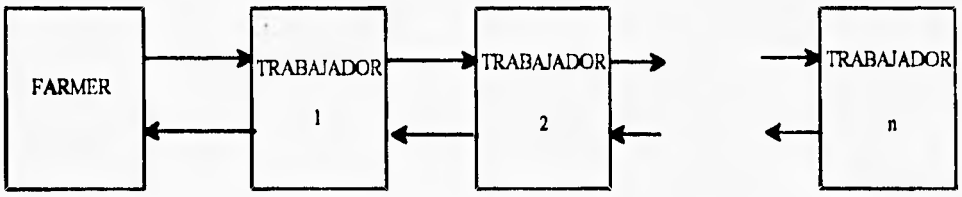


Figura 6.9 Proceso farming sobre una topología lineal

La implementación de los algoritmos de interpolación utilizando el proceso farming en una red de transputers bajo una topología lineal arrojó los siguientes resultados.

# de procesadores	Splines lineales	Splines cuadrados	Interpolación cúbica
1	100.805	147.766	183.786
2	52.696	75.564	93.207
3	39.635	51.6	63.185
4	38.65	40.875	49.971
5		39.231	41.344
6			40.89

Tabla 6.6 Tiempos de ejecución en una configuración lineal

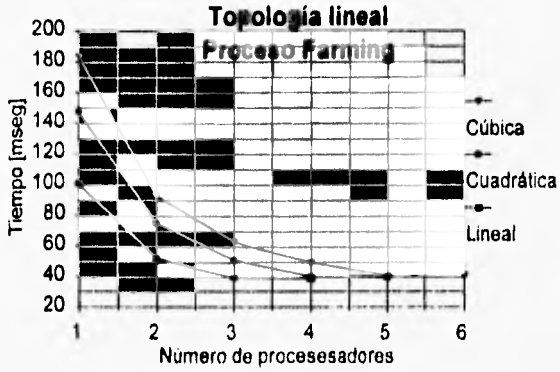


Figura 6.10 Tiempos de proceso bajo el modelo farming (topología lineal)

Podemos ver que para los tres algoritmos se obtienen tiempos menores a los 40 mseg, lo que implica que se pueden procesar los 25 cuadros por segundo requeridos por el sistema. Sin embargo, se puede ver que las tres curvas son asintóticas sobre los 40 mseg y al aumentar de 3 a 4 el número de procesadores en el algoritmo lineal, de 4 a 5 en el cuadrático y de 5 a 6 en el cúbico no existe gran disminución en los tiempos de procesamiento.

Para entender por qué sucede esto se realiza un análisis de desempeño por medio de las métricas descritas en el capítulo 2.

Las tablas 6.7, 6.8 y 6.9 muestran las métricas de desempeño para cada uno de los algoritmos utilizados, así como sus tiempos de ejecución y el número de cuadros por segundo que se pueden procesar.

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	100.805	9.920	1.000	1.000	1.000	0.000
2	52.696	18.977	1.913	0.956	1.830	0.046
3	39.635	25.230	2.543	0.848	2.156	0.090
4	38.650	25.873	2.608	0.652	1.701	0.178

Tabla 6.7 Métricas de desempeño para el algoritmo lineal.

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	147.766	6.767	1.000	1.000	1.000	0.000
2	75.564	13.234	1.956	0.978	1.912	0.023
3	51.600	19.380	2.864	0.955	2.734	0.024
4	40.875	24.465	3.615	0.904	3.267	0.035
5	39.231	25.490	3.767	0.753	2.837	0.082

Tabla 6.8 Métricas de desempeño para el algoritmo de splines cuadráticos

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	183.786	5.441	1.000	1.000	1.000	0.000
2	93.207	10.729	1.972	0.986	1.944	0.014
3	63.185	15.827	2.909	0.970	2.820	0.016
4	49.971	20.012	3.678	0.919	3.382	0.029
5	41.344	24.187	4.445	0.889	3.952	0.031
6	40.890	24.456	4.495	0.749	3.367	0.067

Tabla 6.9 Métricas dedesempeño para el algoritmo de convolución cúbica.

De los datos proporcionados por la tabla 6.7 se puede observar que para el algoritmo de interpolación lineal, el valor de eficacia más alto corresponde a la implementación con tres trabajadores, no obstante presenta una eficiencia del 84% que, como se verá más adelante, puede ser mejorada.

Para el algoritmo de segundo orden el valor máximo de eficacia se presenta para 4 trabajadores; para este valor se observa una eficiencia del 90% que es bastante buena y un procesamiento de 24.46 cuadros por segundo.

Respecto a la implementación del algoritmo de convolución cúbica se puede observar que para 5 trabajadores se logra el mejor valor de eficacia con una utilización de los recursos del 88% y un procesamiento de 24 cuadros por segundo. Hay que hacer notar que en esta implementación a pesar de que el número de procesadores se hace más grande, la eficiencia sigue siendo alta, la pregunta surge ¿por qué a partir de cierto número de procesadores la eficiencia cae rápidamente?

Esta pregunta la podemos responder con la métrica que hasta el momento no se ha mencionado: la fracción serial, para ello se muestra la gráfica de las fracciones seriales resultantes de la implementación de los tres algoritmos.

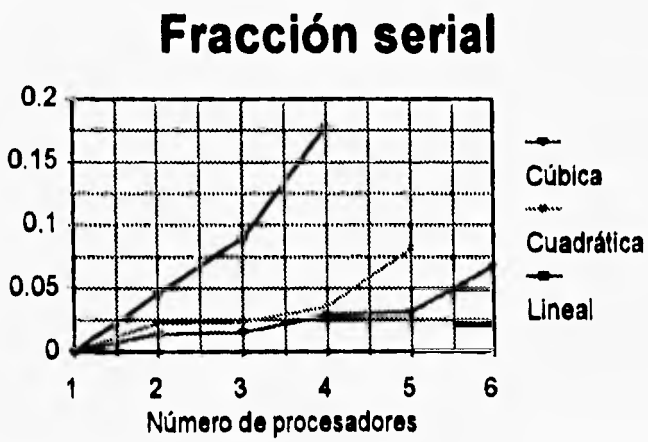


Figura 6.11 Fracciones seriales para cada implementación (topología lineal)

De la figura 6.11 se puede ver que la fracción serial se incrementa drásticamente en la implementación lineal comparada con las dos restantes, esto es debido a que en el algorimo lineal el tiempo de procesamiento es muy pequeño, lo que implica que exista una gran cantidad de comunicaciones que hace que la

eficiencia baje, se puede decir que la granularidad aquí es fina: poco tiempo de procesamiento comparado con el de comunicaciones.

Por otro lado, en la implementación del algoritmo de segundo y tercer orden existe un cambio abrupto cuando el número de procesadores aumenta de 4 a 5 y de 5 a 6 respectivamente, ¿qué provoca este cambio?. La respuesta es la topología o tipo de conexión de procesadores la que lo provoca, la topología lineal tiene la desventaja de que para que llegue la información a los últimos procesadores de la línea, estos tienen que esperar una mayor cantidad de tiempo, por lo que se encuentran inactivos una parte de tiempo y es lo que provoca un desbalanceo de cargas. Por otro lado, el primer procesador presenta un gran número de comunicaciones pues a través de él pasará toda la información dirigida a los demás procesadores.

6.5.2 TOPOLOGÍA DE ESTRELLA.

Con el objeto de solucionar la desventaja que posee la topología lineal es preciso implementar los algoritmos en otro tipo de configuración. Una alternativa es la topología de estrella, la cual puede utilizar los canales restantes del transputer de acuerdo al esquema mostrado en la figura 6.12.

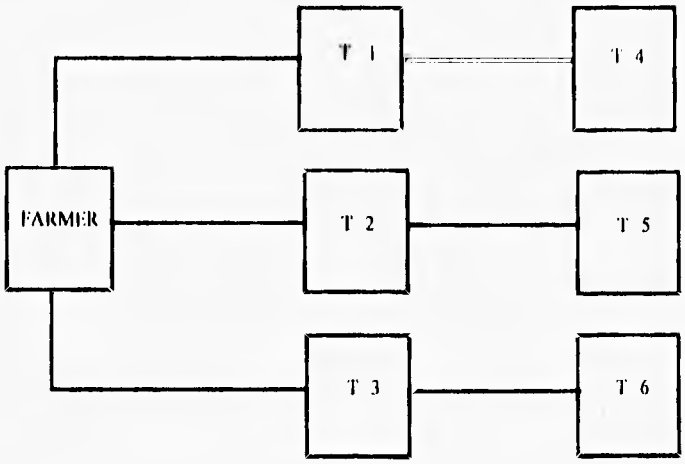


Figura 6.12 Proceso farming sobre una topología de estrella

# de procesadores	Splines lineales	Splines cuadrados	Interpolación cúbica
1	100.805	147.766	183.786
2	50.499	74.045	92.037
3	40.333	49.555	61.562
4	36.719	37.917	46.775
5			38.544

Tabla 6.10. Tiempos de proceso (configuración estrella)

La tabla 6.10 muestra los tiempos de ejecución para los tres algoritmos considerados. Se puede observar que los tiempos se reducen entre 1 y 2 mseg por implementación, lo que es muy positivo si consideramos que se necesitan 40 mseg para procesar un cuadro. La topología de estrella presenta mayores ventajas que la topología lineal, el tiempo que tarda en llegar la información a cada procesador es menor y con ello el tiempo global se reduce.

Es importante decir que solo se pueden utilizar 3 ramas para la topología de estrella, debido a que el transputer contiene solo 4 links o ligas de comunicaciones con las cuales puede transferir información a otros procesadores. Una de ellas es utilizada para establecer comunicación con el procesador host.

La gráfica de tiempos de procesamiento bajo la topología estrella es la siguiente:

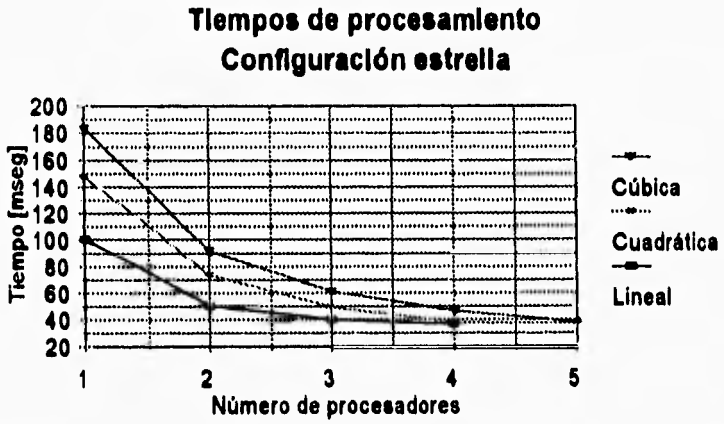


Figura 6.13 Tiempos de proceso (configuración estrella)

Para hacer un análisis de la implementación se presentan a continuación las métricas de desempeño:

# De Trabajadores	T. de procesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	100.805	9.920	1.000	1.000	1.000	0.000
2	50.499	19.802	1.996	0.998	1.992	0.002
3	40.333	24.794	2.499	0.833	2.082	0.100
4	36.719	27.234	2.745	0.686	1.884	0.152

Tabla 11 Métricas de rendimiento para el algoritmo lineal

# De Trabajadores	T. de procesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	147.766	6.767	1.000	1.000	1.000	0.000
2	74.045	13.505	1.996	0.998	1.991	0.002
3	49.555	20.180	2.982	0.994	2.964	0.003
4	37.917	26.373	3.897	0.974	3.797	0.009
5	37.646	26.563	3.925	0.785	3.081	0.068

Tabla 6.12 Métricas de rendimiento para el algoritmo de splines cuadráticos

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	183.786	5.441	1.000	1.000	1.000	0 000
2	92.037	10.865	1.997	0.998	1.994	0.002
3	61.562	16.244	2.985	0.995	2.971	0.002
4	46.775	21.379	3.929	0.982	3.860	0.006
5	38.544	25.944	4.768	0.954	4.547	0 012

Tabla 6.13 Métricas de rendimiento para el algoritmo de convolución cúbica

De igual forma que como se hizo para la implementación lineal veremos en qué punto se obtienen los mejores resultados para cada implementación.

Se observa que para la configuración lineal, el valor máximo de eficacia se encuentra cuando se usan 3 trabajadores pero con una eficiencia del 83% que implica que no se están utilizando al máximo todos los recursos; notar que la fracción serial crece tremendamente con este valor, esto se comentará un poco más adelante.

En el caso de la implementación de segundo orden se pueden encontrar mejores resultados, con 4 trabajadores se obtiene una eficiencia del 97%, la cual es bastante alta si pensamos que el número de procesadores es alto, con este número de procesadores se obtienen 26.373 cuadros por segundo.

En la implementación del algoritmo de convolución cúbica se puede ver que los valores de eficiencia son también altos, del 95% para 5 trabajadores y un procesamiento de casi 26 cuadros por segundo.

De nueva cuenta se analizará la gráfica de la fracción serial para cada una de las implementaciones.

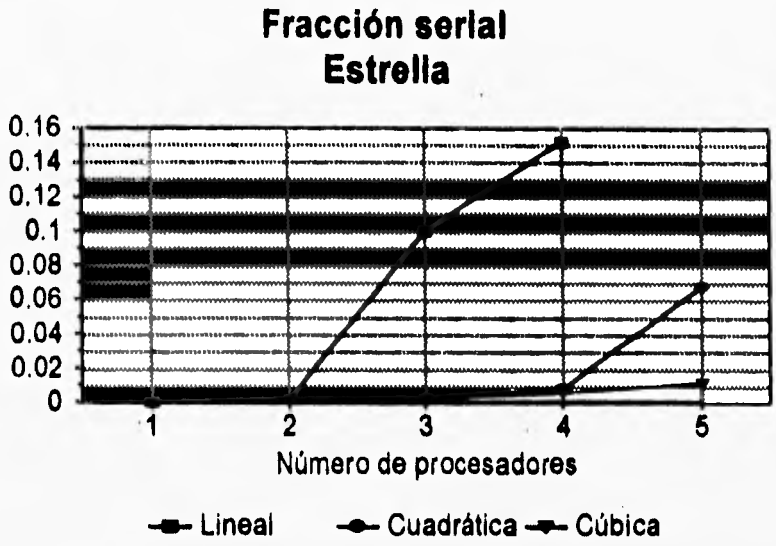


Figura 6.14 Fracciones seriales de las diferentes implementaciones (T. de estrella)

El incremento tan drástico que vemos en la gráfica en la implementación lineal es debido, sin lugar a dudas, a la granularidad del sistema, el procesamiento es muy pequeño comparado con el de comunicaciones, lo que provoca que haya un incremento en las mismas y que los procesadores ocupen más tiempo en comunicación que en procesamiento. En la implementación cuadrática solo ocurre este salto cuando el número de procesadores cambia de 4 a 5, se debe en parte a que existe un desbalanceo de cargas, al último procesador le toca hacer menos trabajo que a los demás, no hay una repartición equitativa de las tareas, lo que implica que no se utilicen correctamente los recursos.

En la implementación del algoritmo de convolución cúbica la fracción serial se comporta como una línea recta con pendiente muy pequeña, casi cero, esto es lo ideal en cualquier sistema e implica que el balanceo de cargas es muy bueno y el tamaño del grano es el adecuado. Hay que subrayar que obtener ese nivel de eficiencias con un número tan alto de procesadores es difícil de lograr.

6.6 FARMER ACTIVO

Como se describió anteriormente, las implementaciones realizadas utilizan *n* número de procesadores trabajadores y uno actuando como farmer, el cual tiene como única función el enviar y recibir datos hacia y de los trabajadores. Por tanto, hasta ahora se utilizan *n+1* procesadores.

Uno de los objetivos de este trabajo es el de utilizar el menor número de recursos para hacer de esta una implementación costeable, lo que implica reducir el número de procesadores involucrados en la aplicación. De lo anterior surge la idea de modificar el proceso actuando como farmer, ¿por qué no hacer que también procese alguna información mientras se encuentra esperando para enviar o recibir datos?

El farmer entonces, presentará una nueva estructura, es decir, tendrá dos procesos corriendo concurrentemente, uno de ellos tendrá la misma función que el farmer descrito anteriormente y el nuevo proceso se encargará de procesar información. Con esto podemos considerarlo como otro trabajador más. Su estructura se muestra a continuación:

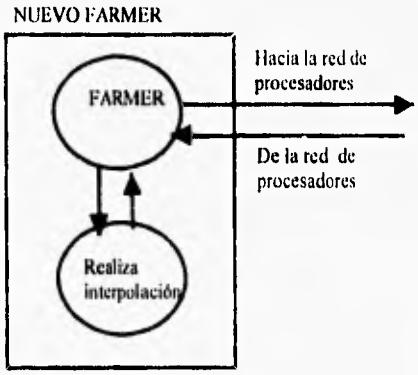


Figura 6.15 Estructura del nuevo farmer

Se podría pensar que el proceso capaz de procesar información dentro del nuevo farmer realiza un menor número de tareas que los demás trabajadores pero no es así, la comunicación entre los dos procesos concurrentes es más rápida que con otro procesador ya que el canal es interno, es decir, una zona de memoria y no un dispositivo de comunicación serial como lo son los links. Por tanto, las tareas del nuevo proceso son semejantes a la de los demás trabajadores.

En síntesis, la función del nuevo farmer será la de enviar y recibir datos de los trabajadores y del proceso que se ejecuta concurrentemente con él.

Las tablas 6.14, 6.15 y 6.16 resumen los resultados obtenidos de las implementaciones de los algoritmos bajo este nuevo esquema en una topología estrella, que es la que ofrece mejores resultados por lo ya descrito en la sección anterior.

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	100.805	9.920	1.000	1.000	1.000	0.000
2	50.563	19.777	1.994	0.997	1.987	0.003
3	34.082	29.341	2.958	0.986	2.916	0.007
4	28.632	34.926	3.521	0.880	3.099	0.045

Tabla 6.14 T. de proceso y métricas de rendimiento para el algoritmo lineal

# De Trabaja-dores	T. de pro-cesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	147.766	6.767	1.000	1.000	1.000	0.000
2	74.308	13.458	1.989	0.994	1.977	0.006
3	49.727	20.110	2.972	0.991	2.943	0.005
4	37.928	26.366	3.896	0.974	3.795	0.009
5	31.927	31.321	4.628	0.926	4.284	0.020

Tabla 6.15 T. de proceso y métricas de rendimiento para el algoritmo de splines cuadráticos

# De Trabajadores	T. de procesamiento [mseg]	# de cuadros por segundo	Speedup	Eficiencia	Eficacia	Fracción Serial
1	183.786	5.441	1.000	1.000	1.000	0.000
2	92.41	10.821	1.989	0.994	1.978	0.006
3	61.833	16.173	2.972	0.991	2.945	0.005
4	46.728	21.400	3.933	0.983	3.867	0.006
5	38.045	26.285	4.831	0.966	4.667	0.009
6	33.4	29.940	5.503	0.917	5.046	0.018

Tabla 6.16 T. de proceso y métricas de rendimiento para el algoritmo de convolución cúbica

Como se puede ver, este esquema ofrece mejores resultados que el anterior ya que aminora el tiempo de comunicación. Se alcanzan en todas las implementaciones casi 30 cuadros por segundo con eficiencias bastante altas (en todas las implementaciones se supera el 88 %). A su vez, la fracción serial no presenta grandes saltos pues existe un mejor balanceo de cargas que en el esquema anterior.

Podemos concluir de este análisis que conviene utilizar el segundo esquema del proceso farming (farmer activo), ya que reduce el número de procesadores para la solución del problema. De igual forma se pudo observar que la topología de estrella ofrece un uso óptimo de los recursos, las eficiencias son muy altas y, por ende, el tiempo de procesamiento menor.

Con respecto a la elección del mejor de los algoritmos implementados se debe tomar en cuenta dos factores principales: bajo costo y calidad de la imagen. Desafortunadamente ambos factores no van de la mano, mientras mayor sea el orden de los algoritmos, mayor es el número de operaciones y el número de procesadores involucrados en la solución del problema; por otro lado, utilizando un algoritmo de orden uno o lineal se reduce el número de procesadores pero la calidad de la imagen decrece de la misma forma.

No hay que perder de vista también que uno de los objetivos planteados dentro de este trabajo de tesis es el de procesar al menos 25 cuadros por segundo, sitiuación que se puede lograr con el último esquema y con 3 procesadores para la implementación lineal (29.341 cuadros), 4 para la implementación cuadrática (26.366 cuadros) y 5 procesadores para la implementación por convolución cúbica (26.285 cuadros).

De acuerdo a lo anterior, la solución óptima atendiendo al costo sería utilizar un algoritmo lineal, de acuerdo a la calidad de la imagen, splines cuadráticos o convolución cúbica.

6.7 RESULTADOS DE LAS IMÁGENES INTERPOLADAS

Para la obtención de estas imágenes se construyó un dispositivo que permitía colocar una varilla cilíndrica y un tubo, de 3 y 35 mm de diámetro a distintas profundidades desde el sensor, dentro de un recipiente lleno de agua para simular las características del cuerpo humano. Se formaron imágenes con la varilla y el tubo en profundidades desde 1 cm hasta 13 cm, con focalizaciones cilíndrica y diédrica. La focalización consiste en desviar un haz de luz a través de una lente ultrasónica para dirigirlo a una zona específica, esto se hace buscando una mejor resolución y una intensidad alta del haz en un punto de interés.

Estas imágenes fueron guardadas en archivos para de ahí ser interpoladas y ser guardadas nuevamente en archivos.

Se muestran a continuación resultados de las imágenes interpoladas por los tres tipos de algoritmos de interpolación.

a) Imagen original

b) repetición de pixeles

c) interplación lineal

d) spline cuadrático

e) convolución cúbica

Figura 6.16 resultados de las imágenes interpoladas, enfoque cilíndrico

En la figura 6.16 a) se tiene el tubo mencionado (en el corte se ve como una elipse), debido a la focalización los bordes aparecen más delgados. La figura 6.16 b) representa la imagen interpolada por el algoritmo de orden cero o la repetición de pixeles. Se puede observar que la imagen es un poco más burda y que se pueden apreciar los cuadros formados por pixeles del mismo color, si la imagen es de mucho contraste, el resultado de la imagen es de baja calidad. Las siguientes figuras c), d) y e) son más suaves, a medida que el orden aumenta la calidad de la imagen mejora. Sobre todo en la figura 6.16 c) el resultado es excelente, a pesar de solo interpolar en una dimensión, la imagen no se distorsiona.

Otro ejemplo de resultados de imágenes interpoladas es el siguiente:

a) imagen original

b) repetición de pixeles

c) interpolación lineal

d) spline cuadrático

e) interpolación cúbica

Figura 6.17 resultados de las imágenes interpoladas, enfoque diédrico

La figura 6.17 a) muestra la imagen de la varilla con un enfoque diédrico, se observa que los bordes están perfectamente definidos, no hay ecos secundarios y la parte central aparece más clara que las laterales, esta es una de las características de este tipo de enfoque.

De igual forma que en la imagen anterior, la repetición de pixeles no proporciona buenos resultados, si el contraste es muy alto, la imagen resultante es muy burda. Es preciso decir que las imágenes utilizadas para análisis médico necesitan ser fieles a la original, para no perder los detalles de la misma. De acuerdo a lo anterior, el algoritmo de interpolación cúbica es la alternativa más viable para obtener una imagen de mejor calidad.

7

CONCLUSIONES

7.1 INTRODUCCIÓN

En el presente capítulo se dará una mirada atrás a lo hasta ahora realizado, con el fin de observar los alcances y logros de este trabajo de tesis.

Se pondrán de manifiesto las aportaciones que se dieron, así como una revisión de los objetivos planteados y la solución a los mismos. Por otro lado se expondrán las ideas que se tienen para mejorar y expandir el sistema de visualización ultrasónica.

7.2 CONCLUSIONES

La mayoría de las computadoras existentes están basadas en la arquitectura de Von Neumann, donde las operaciones se realizan secuencialmente durante la ejecución de un programa, sin embargo, muchas de las aplicaciones hoy desarrolladas, como el procesamiento digital de imágenes, visión y simulación, poseen un paralelismo intrínseco. Resolver estas aplicaciones de una forma secuencial requieren de una gran cantidad de procesamiento y tiempos de ejecución muy altos, es por ello necesaria la utilización de una arquitectura paralela con la cual se puede dividir nuestro problema global en un número determinado de tareas para ser resueltas en forma independiente por varios procesadores.

El objetivo general de esta tesis fue el de lograr la interpolación de imágenes ultrasónicas para diagnóstico médico en tiempo real. Este trabajo es tan solo una parte de un proyecto de investigación más amplio, el de crear un sistema de visualización ultrasónica flexible, de alta resolución y bajo costo. Se pueden determinar tres módulos muy importantes dentro del sistema global, la adquisición de los datos, el procesamiento de los mismos y el despliegue.

La aportación de este trabajo se encuentra en la fase de procesamiento, donde, a la imagen adquirida, se le realiza un proceso de interpolación con el fin de adecuarla a las características de la fase de despliegue. Se utilizaron para ello varios algoritmos de interpolación como el lineal, splines cuadráticos y de

convolución cúbica con el fin de determinar cual de ellos nos proporcionaría una mejor calidad de la imagen con un tiempo de procesamiento menor.

Cabe mencionar que realizar un sistema paralelo no es un tarea trivial, se debe encontrar la mejor estrategia para paralelizar el algoritmo y así explotar el máximo grado de paralelismo del mismo. Se eligió el proceso farming pues es muy utilizado por aplicaciones cuyo nivel de cómputo es bastante grande, en este caso, se intuye que al hablar de imágenes se trata de volúmenes muy altos de información. Otra de sus características importantes es que puede ser implementado en un número indeterminado de procesadores sin tener que modificar el código de la aplicación.

Además de la elección de una estrategia adecuada existen dos factores más que determinan en gran medida la eficiencia de nuestro sistema, estas son la granularidad y el balanceo de cargas. Se realizó un estudio de ambos factores encontrando que de la correcta elección de la granularidad y un buen balanceo de las tareas dio como resultado que los tiempos de ejecución del sistema disminuyeran.

Utilizando el proceso farming mencionado y la granularidad adecuada se pudieron disminuir considerablemente, los tiempos de ejecución para cada implementación. Con dos procesadores, por ejemplo, los tiempos de ejecución de las implementaciones de los algoritmos de interpolación mencionados disminuyeron de 114 mseg a 100 mseg, de 163 mseg a 147 mseg y de 197 mseg a 183 mseg, en los algoritmos lineal, splines cuadráticos y convolución cúbica, respectivamente. Un decremento de 14, 16 y 14 mseg, lo cual es significativo si se recuerda que uno de los objetivos es procesar cuando menos 25 cuadros por segundo, es decir, procesar cada cuadro en 40 mseg.

La topología, es decir, la interconexión de la red de procesadores, es un factor que también debe considerarse. La topología de estrella proporciona mejores tiempos de ejecución y mejora el balanceo de cargas ya que presenta la ventaja de poder transmitir información al mismo tiempo a cada una de sus ramas y los procesadores esperan menos tiempo para recibir sus tareas, en oposición a la

configuración lineal donde el último procesador de la línea permanece ocioso un mayor tiempo.

Con una topología de estrella se tuvieron eficiencias muy altas, del 95% con 6 procesadores y procesando 25 cuadros por segundo en la implementación del algoritmo de convolución cúbica, 97% con 5 procesadores y 26 cuadros por segundo en la de splines cuadráticos y 99% con 3 procesadores y 20 cuadros por segundo en la configuración lineal.

Una de las innovaciones desarrolladas en este trabajo de tesis fue la de modificar el esquema farming. Anteriormente la función del procesador actuando como farmer era solo la de sincronizar y distribuir las tareas a la red de procesadores; con el nuevo esquema planteado el procesador actuando como farmer deja de ser subutilizado, además de sincronizar y distribuir las tareas al resto de los trabajadores, también procesa, con lo cual se explota al máximo la capacidad de los procesadores incluidos en el sistema.

Bajo este nuevo esquema, con el farmer activo, y una topología lineal se redujo el número de procesadores utilizados y las eficiencias se mantuvieron altas. Con 5 procesadores se logró una eficiencia del 96% y procesando 26 cuadros para la implementación del algoritmo de convolución cúbica; 97 % de eficiencia, 26 cuadros por segundo utilizando 4 procesadores para la implementación cuadrática y, para la implementación lineal con 3 procesadores, una eficiencia del 98% procesando 29 cuadros por segundo.

En cuanto a la elección del algoritmo que proporciona una mejor calidad de las imágenes interpoladas, sin lugar a dudas es el de interpolación cúbica, como se dijo en el capítulo 6, este hace uso de una vecindad más grande de pixeles lo que conlleva a que el valor del pixel interpolado sea más congruente con los valores existentes. Con este algoritmo se obtiene una imagen más clara y suave, su desventaja es que requiere de mayor procesamiento y un mayor número de procesadores para su implementación en tiempo real. Los otros dos algoritmos (lineal, splines cuadráticos), proporcionan también buenos resultados y requieren menos procesamiento, así si se requiere calidad de la imagen la opción es la

implementación cúbica, atendiendo al bajo costo la implementación lineal o splines cuadráticos.

7.3 TRABAJO FUTURO

Una de las razones por las que se eligió el proceso farming como estrategia para paralelizar los algoritmos de interpolación es que es altamente modular, lo que quiere decir que se pueden ir incorporando nuevos módulos sin modificar el sistema original. Las arquitecturas paralelas permiten hacer exactamente lo mismo, así que el adaptar al sistema una fase de filtrado y posprocesamiento de la imagen no cambiará lo ya existente, se desarrollarán estas aplicaciones por separado y luego se unirán sin modificación alguna.

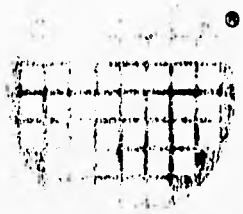
Con respecto a la aplicación de interpolación de imágenes digitales en particular, se está pensando en disminuir el tiempo de comunicación entre los procesadores. Una forma de lograrlo es utilizar un algoritmo de compresión que nos ayude a transferir el mínimo número de información. Se ha pensado en utilizar una técnica simple como es la de Run Length que se basa en sustituir una corrida de pixeles (pixeles del mismo valor) por dos datos (el primero conteniendo el valor del pixel y el segundo el tamaño de la corrida) u otra un poco más compleja como lo es la del código huffman, la cual se basa en un análisis estadístico de la imagen y se le asigna un código con un número de bits menor, al pixel que aparezca con mayor frecuencia y un código de bits mayor, al pixel que aparezca menos.

Por otro lado, el siguiente paso es utilizar la metodología de programación orientada a objetos en forma paralela, para hacer aún más modular nuestro sistema y permitir la reusabilidad en él. Esta tecnología aún se encuentra en fase de análisis pero presenta grandes ventajas como la de permitir crear bibliotecas paralelas que puedan ser utilizadas por usuarios que no tengan claro conocimiento de la implementación.

APÉNDICE A

EL TRANSPUTER

Chapter 6



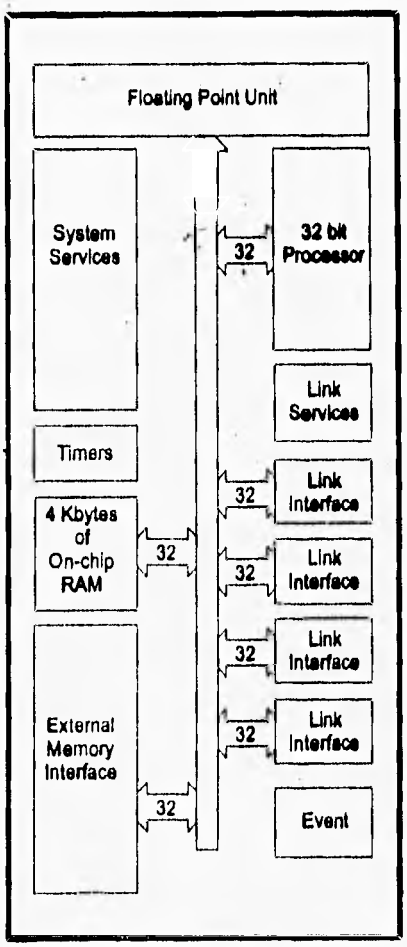
IMS T805 transputer

inmos®

Engineering Data

FEATURES

32 bit architecture
33 ns internal cycle time
30 MIPS (peak) instruction rate
4.3 Mflops (peak) instruction rate
Pin compatible with IMS T800, IMS T425, IMS T400 and IMS T414
Debugging support
64 bit on-chip floating point unit which conforms to IEEE 754
4 Kbytes on-chip static RAM
120 Mbytes/sec sustained data rate to internal memory
4 Gbytes directly addressable external memory
40 Mbytes/sec sustained data rate to external memory
630 ns response to interrupts
Four INMOS serial links 5/10/20 Mbits/sec
Bi-directional data rate of 2.4 Mbytes/sec per link
High performance graphics support with block move instructions
Boot from ROM or communication links
Single 5 MHz clock input
Single +5V ±5% power supply
Packaging 84 pin PGA / 84 pin PLCC / 100 pin CQFP
MIL-STD-883 processing will be available



APPLICATIONS

Scientific and mathematical applications
High speed multi processor systems
High performance graphics processing
Supercomputers
Workstations and workstation clusters
Digital signal processing
Accelerator processors
Distributed databases
System simulation
Telecommunications
Robotics
Fault tolerant systems
Image processing
Pattern recognition
Artificial intelligence

SGS-THOMSON MICROELECTRONICS

April 1992
42 1440 01

3 Floating point unit

The 64 bit FPU provides single and double length arithmetic to floating point standard ANSI-IEEE 754-1985. It is able to perform floating point arithmetic concurrently with the central processor unit (CPU), sustaining 3.3 Mflops on a 30 MHz device. All data communication between memory and the FPU occurs under control of the CPU.

The FPU consists of a microcoded computing engine with a three deep floating point evaluation stack for manipulation of floating point numbers. These stack registers are FA, FB and FC, each of which can hold either 32 bit or 64 bit data; an associated flag, set when a floating point value is loaded, indicates which. The stack behaves in a similar manner to the CPU stack (page 27).

As with the CPU stack, the FPU stack is not saved when rescheduling (page 30) occurs. The FPU can be used in both low and high priority processes. When a high priority process interrupts a low priority one, the FPU state is saved inside the FPU. The CPU will service the interrupt immediately on completing its current operation. The high priority process will not start, however, before the FPU has completed its current operation.

Points in an instruction stream where data need to be transferred to or from the FPU are called *synchronisation points*. At a synchronisation point the first processing unit to become ready will wait until the other is ready. The data transfer will then occur and both processors will proceed concurrently again. In order to make full use of concurrency, floating point data source and destination addresses can be calculated by the CPU whilst the FPU is performing operations on a previous set of data. Device performance is thus optimised by minimising the CPU and FPU idle times.

The FPU has been designed to operate on both single length (32 bit) and double length (64 bit) floating point numbers, and returns results which fully conform to the ANSI-IEEE 754-1985 floating point arithmetic standard. Denormalised numbers are fully supported in the hardware. All rounding modes defined by the standard are implemented, with the default being rounded to the nearest.

The basic addition, subtraction, multiplication and division operations are performed by single instructions. However, certain less frequently used floating point instructions are selected by a value in register A (when allocating registers, this should be taken into account). A *load constant* instruction *ldc* is used to load register A; the *floating point entry* instruction *fpentry* then uses this value to select the floating point operation. This pair of instructions is termed a *selector sequence*.

Names of operations which use *fpentry* begin with *fpu*. A typical usage, returning the absolute value of a floating point number, would be

```
fpuabs;     ldc     fpentry;
```

Since the indirection code for *fpuabs* is **0B**, it would be encoded as

Mnemonic		Function code	Memory code
ldc	*fpuabs*	#4	#4B
fpentry	(op. code #AB)		#2AFB
is coded as			
pfix	#A	#2	#2A
opr	#B	#F	#FB

Table 3.1 *fpentry* coding

The *remainder* and *square root* instructions take considerably longer than other instructions to complete. In order to minimise the interrupt latency period of the transputer they are split up to form instruction sequences. As an example, the instruction sequence for a single length square root is

```
fpusqrtfirst;   fpusqrtstep;   fpusqrtstep;   fpusqrtlast;
```

The FPU has its own error flag *FP_Error*. This reflects the state of evaluation within the FPU and is set in circumstances where invalid operations, division by zero or overflow exceptions to the ANSI-IEEE 754-1985 standard would be flagged (page 49). *FP_Error* is also set if an input to a floating point operation is infinite or is not a number (NaN). The *FP_Error* flag can be set, tested and cleared without affecting the main *Error* flag, but can also set *Error* when required (page 48). Depending on how a program is compiled, it is possible for both unchecked and fully checked floating point arithmetic to be performed.

Further details on the operation of the FPU can be found in the *'Transputer Instruction Set - A Compiler Writer's Guide'*.

Operation	T805-20		T805-25		T805-30	
	Single length	Double length	Single length	Double length	Single length	Double length
add	350 ns	350 ns	280 ns	280 ns	233 ns	233 ns
subtract	350 ns	350 ns	280 ns	280 ns	233 ns	233 ns
multiply	550 ns	1000 ns	440 ns	800 ns	367 ns	667 ns
divide	850 ns	1600 ns	680 ns	1280 ns	567 ns	1067 ns

Timing is for operations where both operands are normalised fp numbers.

Table 3.2 Typical floating point operation times for IMS T805

APÉNDICE B

PROGRAMAS DE LA APLICACIÓN

```
/* archivo de configuración de la red de transputers */

r 0
r 1
r 2
s 0 0
s 1 5
s 2 5
c 0 0 3 1 0 0
c 0 0 0 2 0 3


/* archivo de compilación de todos los módulos */
icc f1.c  -t805
ilink f1.tco  -t805 -f startup.lnk

icc w1.c -t805
ilink w1.tco   -t805 -f startrd.lnk

icc w2.c -t805
ilink w2.tco   -t805 -f startrd.lnk

icc w3.c -t805
ilink w3.tco   -t805 -f startrd.lnk

icc w4.c -t805
ilink w4.tco   -t805 -f startrd.lnk

icc w5.c -t805
ilink w5.tco   -t805 -f startrd.lnk

icconf f1.cfs
icollect f1.cfb
iserver -se -sb f1 bll


/* archivo que mapea los procesos lógicos en los procesadores */


/*{{{  Hardware description */
T805 (memory = 1M) farmer_processor;
T805 (memory = 1M) worker1_processor;
T805 (memory = 1M) worker2_processor;
T805 (memory = 1M) worker3_processor;
T805 (memory = 1M) worker4_processor;
T805 (memory = 1M) worker5_processor;


connect farmer_processor.link[1], host;
connect farmer_processor.link[2], worker1_processor.link[1];
connect worker1_processor.link[2], worker2_processor.link[1];
connect farmer_processor.link[3], worker3_processor.link[0];
connect worker3_processor.link[2], worker4_processor.link[1];
connect farmer_processor.link[0], worker5_processor.link[3];

/*}}}*/

/*{{{  Software description */
process (stacksize = 200k, heapsize = 200k,
         interface(input in, output out, input from_worker1, output to_worker1, input from_worker3, output to_worker3,input
from_worker5, output to_worker5)) farmer;

process (stacksize = 20k, heapsize = 20k,
         interface(input from_farmer, output to_farmer,input from_worker2, output to_worker2)) worker1;

process (stacksize = 20k, heapsize = 20k,
```

```
        interface(input from_worker1, output to_worker1)) worker2;

process (stacksize = 20k, heapsize = 20k,
        interface(input from_farmer, output to_farmer,input from_worker4, output to_worker4)) worker3,

process (stacksize = 20k, heapsize = 20k,
        interface(input from_worker3, output to_worker3)) worker4;

process (stacksize = 20k, heapsize = 20k,
        interface(input from_farmer, output to_farmer)) worker5;


input from_host,
output to_host,

connect farmer.in, from_host;
connect farmer.out, to_host;
connect farmer.from_worker1, worker1.to_farmer;
connect farmer.to_worker1, worker1.from_farmer;
connect worker1.from_worker2, worker2.to_worker1;
connect worker1.to_worker2, worker2.from_worker1;

connect farmer.from_worker3, worker3.to_farmer;
connect farmer.to_worker3, worker3.from_farmer;
connect worker3.from_worker4, worker4.to_worker3;
connect worker3.to_worker4, worker4.from_worker3;

connect farmer.from_worker5, worker5.to_farmer;
connect farmer.to_worker5, worker5.from_farmer;


/*}}}*/

/*{{{  Network mapping */
use "f1.lku" for farmer;
use "w1.lku" for worker1;
use "w2.lku" for worker2;
use "w3.lku" for worker3;
use "w4.lku" for worker4;
use "w5.lku" for worker5;

place farmer on farmer_processor;
place worker1 on worker1_processor;
place worker2 on worker2_processor;
place worker3 on worker3_processor;
place worker4 on worker4_processor;
place worker5 on worker5_processor;

place to_host   on host;
place from_host on host;


/*}}}*/

/* archivo que contiene declaración de funciones de canales */
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <channel.h>

Channel *
Checked_ChanAlloc ()

{
        Channel   *chan;
```

```
        if ((chan = ChanAlloc ()) == NULL)  {
                exit (0);
        }
        return chan;
}



Process *
Checked_ProcAlloc (void (*func)(Process*, Channel*, Channel*), int sp, int nparam,
                Channel *c1,Channel *c2)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2);
        if (proc == NULL)  {
                exit (0);
        }
        return proc;
}


Process *
Checkmore_ProcAlloc (void (*func)(Process*, Channel*, Channel*, Channel*), int sp, int nparam,
                Channel *c1,Channel *c2, Channel *c3)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2,c3);
        if (proc == NULL)  {
                exit (0);
        }
        return proc;
}


Process *
Check_ProcAlloc (void (*func)(Process*, Channel*, Channel*, Channel*, Channel*), int sp, int nparam,
                Channel *c1, Channel *c2, Channel *c3, Channel *c4)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2,c3,c4);
        if (proc == NULL)  {
                exit (0);
        }
        return proc;
}

Process *
Chk_ProcAlloc (void (*func)(Process*, Channel*, Channel*, Channel*, Channel*, Channel*), int sp, int nparam,
                Channel *c1, Channel *c2, Channel *c3, Channel *c4, Channel *c5)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2,c3,c4,c5);
        if (proc == NULL)  {
                exit (0);
        }
```

```
        return proc;
}


Process *
C6_ProcAlloc (void (*func)(Process*, Channel*, Channel*, Channel*, Channel*, Channel*,Channel*), int sp, int nparam,
                Channel *c1, Channel *c2, Channel *c3, Channel *c4, Channel *c5, Channel *c6)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2,c3,c4,c5,c6);
        if (proc == NULL)  {
                exit (0);
        }
        return proc;
}


Process *
C8_ProcAlloc (void (*func)(Process*, Channel*, Channel*,Channel*, Channel*, Channel*, Channel*, Channel*,Channel*), int sp,
int nparam,
                Channel *c1, Channel *c2, Channel *c3, Channel *c4, Channel *c5, Channel *c6, Channel *c7, Channel
*c8)

{
        Process  *proc;

        proc = ProcAlloc (func, sp, nparam, c1,c2,c3,c4,c5,c6,c7,c8);
        if (proc == NULL)  {
                exit (0);
        }
        return proc;
}

/* proceso actuando como farmer */  /* esquema farmer activo */

#include <misc.h>
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>
#include <channel.h>
#include <ioctrl.h>
#include <process.h>
#include "parahelp.h"


void principal(Process* p1,Channel* to_worker1, Channel* from_worker1,Channel* to_worker2, Channel*
from_worker2,Channel* to_worker3, Channel* from_worker3, Channel* to_alterno, Channel* from_alterno)
{
        int             bandera=10;
        int             trabajos_enviados=0;
        int             trabajos_hechos=0;
        int             running=1;
        int             j;
        char            indic;
        unsigned char  total[16][113];
        unsigned char  auxi[16][57];
        int             tareas1,tareas2,tareas3,tareas4=1,tareas6=1,tareas5;
        int             reloj1,reloj2,reloj_w1,reloj_w2,reloj_w3,reloj_w4,reloj_w5;
p1=p1;
```

```
reloj1=ProcTime();

            for(j=0;j<2;j++){
            ChanOut(to_worker1,auxi,sizeof(auxi));
            ChanOut(to_worker1,auxi,sizeof(auxi));
            ChanOut(to_worker2,auxi,sizeof(auxi));
            ChanOut(to_worker2,auxi,sizeof(auxi));
            ChanOut(to_worker3,auxi,sizeof(auxi));
            }


/*ChanOut(to_alterno,auxi,sizeof(auxi));*/

trabajos_enviados=10;

while(running)
{


            bandera=ProcAlt(from_worker1,from_worker2,from_worker3/*,from_alterno*/,NULL);
            switch(bandera){

              case 0:
              ChanIn(from_worker1,total,sizeof(total));
              trabajos_hechos++;
        break;
               case 1:
              ChanIn(from_worker2,total,sizeof(total));
              trabajos_hechos++;
        break;
               case 2:
              ChanIn(from_worker3,total,sizeof(total));
              trabajos_hechos++;
        break;
              /*case 3:
              indic=ChanInChar(from_alterno);
              ChanIn(from_alterno,total,sizeof(total));
              trabajos_hechos++;
              break; */
     )


if(trabajos_enviados<384){
            if(bandera==0)
            ChanOut(to_worker1,auxi,sizeof(auxi));
            else if(bandera==1)
            ChanOut(to_worker2,auxi,sizeof(auxi));
      else if(bandera==2)
            ChanOut(to_worker3,auxi,sizeof(auxi));
            /*else{
            ChanOut(to_alterno,auxi,sizeof(auxi));
            tareas6++;
            }*/
            trabajos_enviados++;

            }


              if(trabajos_hechos==384){
                running=0;
                auxi[0][56]=255;
           ChanOut(to_worker1,auxi,sizeof(auxi));
                ChanOut(to_worker2,auxi,sizeof(auxi));
                ChanOut(to_worker3,auxi,sizeof(auxi));
                /*ChanOut(to_alterno,auxi,sizeof(auxi));*/
```

```
            }

}
reloj2=ProcTime();

reloj_w1=ChanInInt(from_worker1);
tareas1=ChanInInt(from_worker1);
reloj_w2=ChanInInt(from_worker1);
tareas2=ChanInInt(from_worker1);
reloj_w3=ChanInInt(from_worker2);
tareas3=ChanInInt(from_worker2);
reloj_w4=ChanInInt(from_worker2);
tareas4=ChanInInt(from_worker2);
reloj_w5=ChanInInt(from_worker3);
tareas5=ChanInInt(from_worker3);

printf("\nEl tiempo es: %d   \n",  (reloj2-reloj1));
printf("%d\n",reloj_w1);
printf("%d\n",reloj_w2);
printf("%d\n",reloj_w3);
printf("%d\n",reloj_w4);
printf("%d\n",reloj_w5);
printf("El numero de tareas en p1 es %d\n",tareas1);
printf("El numero de tareas en p2 es %d\n",tareas2);
printf("El numero de tareas en p3 es %d\n",tareas3);
printf("El numero de tareas en p4 es %d\n",tareas4);
printf("El numero de tareas en p5 es %d\n",tareas5);
/*printf("El numero de tareas en el host es %d\n",tareas6);*/



}



void alterno(Process* p2, Channel* to_principal, Channel* from_principal)
{
unsigned char cad[16][113];
unsigned char aux[16][57];
char  indic;
int cont=0;
int i;

int aux11,aux21,aux31,aux41,aux51,aux61,aux71,aux81,aux91,aux101,aux111,aux121,aux131,aux141,aux151,aux161;
int aux12,aux22,aux32,aux42,aux52,aux62,aux72,aux82,aux92,aux102,aux112,aux122,aux132,aux142,aux152,aux162;
int a,b,c;

int running=1,bandera=10;
p2=p2;

running=0;

while(running){
bandera=ProcAlt(from_principal,NULL);
  switch(bandera){
          case 0:
          ChanIn(from_principal,aux,sizeof(aux));
          if(aux[0][56]==255){
              running=0;
          }
          else{

                  aux11=3*(aux[0][0]-aux[0][1])+aux[0][2];
                      aux21=3*(aux[1][0]-aux[1][1])+aux[1][2];
                      aux31=3*(aux[2][0]-aux[2][1])+aux[2][2];
```

```
aux41=3*(aux[3][0]-aux[3][1])+aux[3][2];
aux51=3*(aux[4][0]-aux[4][1])+aux[4][2];
aux61=3*(aux[5][0]-aux[5][1])+aux[5][2];
aux71=3*(aux[6][0]-aux[6][1])+aux[6][2];
aux81=3*(aux[7][0]-aux[7][1])+aux[7][2];
aux91=3*(aux[8][0]-aux[8][1])+aux[8][2];
aux101=3*(aux[9][0]-aux[9][1])+aux[9][2];
aux111=3*(aux[10][0]-aux[10][1])+aux[10][2];
aux121=3*(aux[11][0]-aux[11][1])+aux[11][2];
aux131=3*(aux[12][0]-aux[12][1])+aux[12][2];
aux141=3*(aux[13][0]-aux[13][1])+aux[13][2];
aux151=3*(aux[14][0]-aux[14][1])+aux[14][2];
aux161=3*(aux[15][0]-aux[15][1])+aux[15][2];
```

```
aux12=3*(aux[0][55]-aux[0][54])+aux[0][53];
aux22=3*(aux[1][55]-aux[1][54])+aux[1][53];
aux32=3*(aux[2][55]-aux[2][54])+aux[2][53];
aux42=3*(aux[3][55]-aux[3][54])+aux[3][53];
aux52=3*(aux[4][55]-aux[4][54])+aux[4][53];
aux62=3*(aux[5][55]-aux[5][54])+aux[5][53];
aux72=3*(aux[6][55]-aux[6][54])+aux[6][53];
aux82=3*(aux[7][55]-aux[7][54])+aux[7][53];
aux92=3*(aux[8][55]-aux[8][54])+aux[8][53];
aux102=3*(aux[9][55]-aux[9][54])+aux[9][53];
aux112=3*(aux[10][55]-aux[10][54])+aux[10][53];
aux122=3*(aux[11][55]-aux[11][54])+aux[11][53];
aux132=3*(aux[12][55]-aux[12][54])+aux[12][53];
aux142=3*(aux[13][55]-aux[13][54])+aux[13][53];
aux152=3*(aux[14][55]-aux[14][54])+aux[14][53];
aux162=3*(aux[15][55]-aux[15][54])+aux[15][53];
```

```
cad[0][0]=(-2*aux11+21*aux[0][0]+9*aux[0][1]-aux[0][2])/27;
cad[1][0]=(-2*aux21+21*aux[1][0]+9*aux[1][1]-aux[1][2])/27;
cad[2][0]=(-2*aux31+21*aux[2][0]+9*aux[2][1]-aux[2][2])/27;
cad[3][0]=(-2*aux41+21*aux[3][0]+9*aux[3][1]-aux[3][2])/27;
cad[4][0]=(-2*aux51+21*aux[4][0]+9*aux[4][1]-aux[4][2])/27;
cad[5][0]=(-2*aux61+21*aux[5][0]+9*aux[5][1]-aux[5][2])/27;
cad[6][0]=(-2*aux71+21*aux[6][0]+9*aux[6][1]-aux[6][2])/27;
cad[7][0]=(-2*aux81+21*aux[7][0]+9*aux[7][1]-aux[7][2])/27;
cad[8][0]=(-2*aux91+21*aux[8][0]+9*aux[8][1]-aux[8][2])/27;
cad[9][0]=(-2*aux101+21*aux[9][0]+9*aux[9][1]-aux[9][2])/27;
cad[10][0]=(-2*aux111+21*aux[10][0]+9*aux[10][1]-aux[10][2])/27;
cad[11][0]=(-2*aux121+21*aux[11][0]+9*aux[11][1]-aux[11][2])/27;
cad[12][0]=(-2*aux131+21*aux[12][0]+9*aux[12][1]-aux[12][2])/27;
cad[13][0]=(-2*aux141+21*aux[13][0]+9*aux[13][1]-aux[13][2])/27;
cad[14][0]=(-2*aux151+21*aux[14][0]+9*aux[14][1]-aux[14][2])/27;
cad[15][0]=(-2*aux161+21*aux[15][0]+9*aux[15][1]-aux[15][2])/27;
```

```
cad[0][1]= (-aux11+9*aux[0][0]+21*aux[0][1]-2*aux[0][2])/27;
cad[1][1]= (-aux21+9*aux[1][0]+21*aux[1][1]-2*aux[1][2])/27;
cad[2][1]= (-aux31+9*aux[2][0]+21*aux[2][1]-2*aux[2][2])/27;
cad[3][1]= (-aux41+9*aux[3][0]+21*aux[3][1]-2*aux[3][2])/27;
cad[4][1]= (-aux51+9*aux[4][0]+21*aux[4][1]-2*aux[4][2])/27;
cad[5][1]= (-aux61+9*aux[5][0]+21*aux[5][1]-2*aux[5][2])/27;
cad[6][1]= (-aux71+9*aux[6][0]+21*aux[6][1]-2*aux[6][2])/27;
cad[7][1]= (-aux81+9*aux[7][0]+21*aux[7][1]-2*aux[7][2])/27;
cad[8][1]= (-aux91+9*aux[8][0]+21*aux[8][1]-2*aux[8][2])/27;
```

```
cad[9][1]= (-aux101+9*aux[9][0]+21*aux[9][1]-2*aux[9][2])/27;
cad[10][1]= (-aux111+9*aux[10][0]+21*aux[10][1]-2*aux[10][2])/27;
cad[11][1]= (-aux121+9*aux[11][0]+21*aux[11][1]-2*aux[11][2])/27;
cad[12][1]= (-aux131+9*aux[12][0]+21*aux[12][1]-2*aux[12][2])/27;
cad[13][1]= (-aux141+9*aux[13][0]+21*aux[13][1]-2*aux[13][2])/27;
cad[14][1]= (-aux151+9*aux[14][0]+21*aux[14][1]-2*aux[14][2])/27;
cad[15][1]= (-aux161+9*aux[15][0]+21*aux[15][1]-2*aux[15][2])/27;
```

```
cad[0][110]=(-2*aux[0][53]+21*aux[0][54]+9*aux[0][55]-aux12)/27;
cad[1][110]=(-2*aux[1][53]+21*aux[1][54]+9*aux[1][55]-aux22)/27;
cad[2][110]=(-2*aux[2][53]+21*aux[2][54]+9*aux[2][55]-aux32)/27;
cad[3][110]=(-2*aux[3][53]+21*aux[3][54]+9*aux[3][55]-aux42)/27;
cad[4][110]=(-2*aux[4][53]+21*aux[4][54]+9*aux[4][55]-aux52)/27;
cad[5][110]=(-2*aux[5][53]+21*aux[5][54]+9*aux[5][55]-aux62)/27;
cad[6][110]=(-2*aux[6][53]+21*aux[6][54]+9*aux[6][55]-aux72)/27;
cad[7][110]=(-2*aux[7][53]+21*aux[7][54]+9*aux[7][55]-aux82)/27;
cad[8][110]=(-2*aux[8][53]+21*aux[8][54]+9*aux[8][55]-aux92)/27;
cad[9][110]=(-2*aux[9][53]+21*aux[9][54]+9*aux[9][55]-aux102)/27;
cad[10][110]=(-2*aux[10][53]+21*aux[10][54]+9*aux[10][55]-aux112)/27;
cad[11][110]=(-2*aux[11][53]+21*aux[11][54]+9*aux[11][55]-aux122)/27;
cad[12][110]=(-2*aux[12][53]+21*aux[12][54]+9*aux[12][55]-aux132)/27;
cad[13][110]=(-2*aux[13][53]+21*aux[13][54]+9*aux[13][55]-aux142)/27;
cad[14][110]=(-2*aux[14][53]+21*aux[14][54]+9*aux[14][55]-aux152)/27;
cad[15][110]=(-2*aux[15][53]+21*aux[15][54]+9*aux[15][55]-aux162)/27;
```

```
cad[0][111]=(-aux[0][53]+9*aux[0][54]+21*aux[0][55]-2*aux12)/27;
cad[1][111]=(-aux[1][53]+9*aux[1][54]+21*aux[1][55]-2*aux22)/27;
cad[2][111]=(-aux[2][53]+9*aux[2][54]+21*aux[2][55]-2*aux32)/27;
cad[3][111]=(-aux[3][53]+9*aux[3][54]+21*aux[3][55]-2*aux42)/27;
cad[4][111]=(-aux[4][53]+9*aux[4][54]+21*aux[4][55]-2*aux52)/27;
cad[5][111]=(-aux[5][53]+9*aux[5][54]+21*aux[5][55]-2*aux62)/27;
cad[6][111]=(-aux[6][53]+9*aux[8][54]+21*aux[6][55]-2*aux72)/27;
cad[7][111]=(-aux[7][53]+9*aux[7][54]+21*aux[7][55]-2*aux82)/27;
cad[8][111]=(-aux[8][53]+9*aux[8][54]+21*aux[8][55]-2*aux92)/27;
cad[9][111]=(-aux[9][53]+9*aux[9][54]+21*aux[9][55]-2*aux102)/27;
cad[10][111]=(-aux[10][53]+9*aux[10][54]+21*aux[10][55]-2*aux112)/27;
cad[11][111]=(-aux[11][53]+9*aux[11][54]+21*aux[11][55]-2*aux122)/27;
cad[12][111]=(-aux[12][53]+9*aux[12][54]+21*aux[12][55]-2*aux132)/27;
cad[13][111]=(-aux[13][53]+9*aux[13][54]+21*aux[13][55]-2*aux142)/27;
cad[14][111]=(-aux[14][53]+9*aux[14][54]+21*aux[14][55]-2*aux152)/27;
cad[15][111]=(-aux[15][53]+9*aux[15][54]+21*aux[15][55]-2*aux162)/27;
```

```
for(i=2;i<110;i+=2){

a=cont+1;
b=cont+2;
c=cont+3;

cad[0][i]=(-2*aux[0][cont]+21*aux[0][a]+9*aux[0][b]-aux[0][c])/27;
                                        cad[0][i+1]=(-aux[0][cont]+9*aux[0][a]+21*aux[0][b]-2*aux[0][c])/27;

cad[1][i]=(-2*aux[1][cont]+21*aux[1][a]+9*aux[1][b]-aux[1][c])/27;

cad[1][i+1]=(-aux[1][cont]+9*aux[1][a]+21*aux[1][b]-2*aux[1][c])/27;
```

```
cad[2][i]=(-2*aux[2][cont]+21*aux[2][a]+9*aux[2][b]-aux[2][c])/27;

cad[2][i+1]=(-aux[2][cont]+9*aux[2][a]+21*aux[2][b]-2*aux[2][c])/27;

cad[3][i]=(-2*aux[3][cont]+21*aux[3][a]+9*aux[3][b]-aux[3][c])/27;

cad[3][i+1]=(-aux[3][cont]+9*aux[3][a]+21*aux[3][b]-2*aux[3][c])/27;

cad[4][i]=(-2*aux[4][cont]+21*aux[4][a]+9*aux[4][b]-aux[4][c])/27;
                                        cad[4][i+1]=(-aux[4][cont]+9*aux[4][a]+21*aux[4][b]-2*aux[4][c])/27;

cad[5][i]=(-2*aux[5][cont]+21*aux[5][a]+9*aux[5][b]-aux[5][c])/27;

cad[5][i+1]=(-aux[5][cont]+9*aux[5][a]+21*aux[5][b]-2*aux[5][c])/27;

cad[6][i]=(-2*aux[6][cont]+21*aux[6][a]+9*aux[6][b]-aux[8][c])/27;

cad[6][i+1]=(-aux[6][cont]+9*aux[6][a]+21*aux[6][b]-2*aux[6][c])/27;

cad[7][i]=(-2*aux[7][cont]+21*aux[7][a]+9*aux[7][b]-aux[7][c])/27;

cad[7][i+1]=(-aux[7][cont]+9*aux[7][a]+21*aux[7][b]-2*aux[7][c])/27;

cad[8][i]=(-2*aux[8][cont]+21*aux[8][a]+9*aux[8][b]-aux[8][c])/27;

cad[8][i+1]=(-aux[8][cont]+9*aux[8][a]+21*aux[8][b]-2*aux[8][c])/27;

cad[9][i]=(-2*aux[9][cont]+21*aux[9][a]+9*aux[9][b]-aux[9][c])/27;

cad[9][i+1]=(-aux[9][cont]+9*aux[9][a]+21*aux[9][b]-2*aux[9][c])/27;

cad[10][i]=(-2*aux[10][cont]+21*aux[10][a]+9*aux[10][b]-aux[10][c])/27;

cad[10][i+1]=(-aux[10][cont]+9*aux[10][a]+21*aux[10][b]-2*aux[10][c])/27;

cad[11][i]=(-2*aux[11][cont]+21*aux[11][a]+9*aux[11][b]-aux[11][c])/27;

cad[11][i+1]=(-aux[11][cont]+9*aux[11][a]+21*aux[11][b]-2*aux[11][c])/27;

cad[12][i]=(-2*aux[12][cont]+21*aux[12][a]+9*aux[12][b]-aux[12][c])/27;

cad[12][i+1]=(-aux[12][cont]+9*aux[12][a]+21*aux[12][b]-2*aux[12][c])/27;

cad[13][i]=(-2*aux[13][cont]+21*aux[13][a]+9*aux[13][b]-aux[13][c])/27;

cad[13][i+1]=(-aux[13][cont]+9*aux[13][a]+21*aux[13][b]-2*aux[13][c])/27;

cad[14][i]=(-2*aux[14][cont]+21*aux[14][a]+9*aux[14][b]-aux[14][c])/27;

cad[14][i+1]=(-aux[14][cont]+9*aux[14][a]+21*aux[14][b]-2*aux[14][c])/27;

cad[15][i]=(-2*aux[15][cont]+21*aux[15][a]+9*aux[15][b]-aux[15][c])/27;

cad[15][i+1]=(-aux[15][cont]+9*aux[15][a]+21*aux[15][b]-2*aux[15][c])/27;

                                cont++;
}

    ChanOutChar(to_principal,indic);
    ChanOut(to_principal,cad,sizeof(cad));
        cont=0;
    }
     break;
```

```
int main (void) {

        Channel*    from_worker1;
        Channel*    to_worker1;
        Channel*    from_worker2;
        Channel*    to_worker2;
        Channel*    from_worker3;
        Channel*    to_worker3;
        Channel*    principal_alterno;
        Channel*    alterno_principal;
        Process         *p_principal,*p_alterno;

        from_worker1 = (Channel *) get_param (3);
        to_worker1   = (Channel *) get_param (4);
        from_worker2 = (Channel *) get_param (5);
        to_worker2   = (Channel *) get_param (6);
        from_worker3 = (Channel *) get_param (7);
        to_worker3   = (Channel *) get_param (8);

        principal_alterno = ChanAlloc();
        alterno_principal = ChanAlloc();

        p_principal =
C8_ProcAlloc(principal,0,8,to_worker1,from_worker1,to_worker2,from_worker2,to_worker3,from_worker3,principal_alterno,
alterno_principal);
        p_alterno   = Checked_ProcAlloc(alterno,0,2,alterno_principal, principal_alterno);

    ProcPriPar(p_principal,p_alterno);


        exit_terminate (0);



/* archivo worker */  /* todos los procesos workers son parecidos*/
#include <misc.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "parahelp.h"




void switcher(Process *p1, Channel* from_farmer, Channel* switch_inter,Channel* mas_trabajo, Channel* senal,Channel*
to_worker2)
{
unsigned char cad[16][57];
unsigned char auxiliar[16][57];
int running=1;

int ocupado=0,bandera,buffered1=0;
int var=0;
int tiempo,uno=1;
```

```
p1=p1;
tiempo=ProcTime();

while(running || buffered1==1){

bandera=ProcAlt(from_farmer,mas_trabajo,NULL);
   switch(bandera){
          case 0:
                ChanIn(from_farmer,cad,sizeof(cad));
                 if(uno==1){
               tiempo=ProcTime();
                      uno=0;uno=0;
            }
                 if(cad[0][56]==255){
                      running=0;
            ChanOutChar(senal,'a');
                      ChanOut(to_worker2,cad,sizeof(cad));
                 }
                 else{
                   if(ocupado==0){
                     ChanOut(switch_inter,cad,sizeof(cad));
                     ocupado=1;
                   }
                   else if(buffered1==0){
                     buffered1=1;
                        memcpy(auxiliar, cad,912*sizeof(unsigned char));
                   }
                      else{
                        ChanOut(to_worker2,cad,sizeof(cad));
                      }
                 }
                break;
          case 1:
                var=ChanInInt(mas_trabajo);
                if(buffered1==1){
                  ChanOut(switch_inter,auxiliar,sizeof(auxiliar));
                  buffered1=0;
                }
                else{
                   ocupado=0;
                }
                break;


}
ChanOutInt(switch_inter,tiempo);
}


void interpolar(Process *p2, Channel *inter_switch, Channel *inter_retro, Channel *mas_trabajo, Channel* senal1, Channel*
senal2)
{
unsigned char aux[16][57];
unsigned char cad[16][113];
int i,running=1;
int tiempo,cont=0, bandera;
int aux11,aux21,aux31,aux41,aux51,aux61,aux71,aux81,aux91,aux101,aux111,aux121,aux131,aux141,aux151,aux161;
int aux12,aux22,aux32,aux42,aux52,aux62,aux72,aux82,aux92,aux102,aux112,aux122,aux132,aux142,aux152,aux162;
int a,b,c;

char k;
p2=p2;

while(running){
          bandera=ProcAlt(inter_switch,senal1, NULL);
          switch(bandera){
```

```
case 0:
    ChanIn(inter_switch,aux,sizeof(aux));
    aux11=3*(aux[0][0]-aux[0][1])+aux[0][2];
        aux21=3*(aux[1][0]-aux[1][1])+aux[1][2];
        aux31=3*(aux[2][0]-aux[2][1])+aux[2][2];
        aux41=3*(aux[3][0]-aux[3][1])+aux[3][2];
        aux51=3*(aux[4][0]-aux[4][1])+aux[4][2];
        aux61=3*(aux[5][0]-aux[5][1])+aux[5][2];
        aux71=3*(aux[6][0]-aux[6][1])+aux[6][2];
        aux81=3*(aux[7][0]-aux[7][1])+aux[7][2];
        aux91=3*(aux[8][0]-aux[8][1])+aux[8][2];
        aux101=3*(aux[9][0]-aux[9][1])+aux[9][2];
        aux111=3*(aux[10][0]-aux[10][1])+aux[10][2];
        aux121=3*(aux[11][0]-aux[11][1])+aux[11][2];
        aux131=3*(aux[12][0]-aux[12][1])+aux[12][2];
        aux141=3*(aux[13][0]-aux[13][1])+aux[13][2];
        aux151=3*(aux[14][0]-aux[14][1])+aux[14][2];
        aux161=3*(aux[15][0]-aux[15][1])+aux[15][2];

    aux12=3*(aux[0][55]-aux[0][54])+aux[0][53];
    aux22=3*(aux[1][55]-aux[1][54])+aux[1][53];
    aux32=3*(aux[2][55]-aux[2][54])+aux[2][53];
    aux42=3*(aux[3][55]-aux[3][54])+aux[3][53];
        aux52=3*(aux[4][55]-aux[4][54])+aux[4][53];
        aux62=3*(aux[5][55]-aux[5][54])+aux[5][53];
        aux72=3*(aux[6][55]-aux[6][54])+aux[6][53];
        aux82=3*(aux[7][55]-aux[7][54])+aux[7][53];
        aux92=3*(aux[8][55]-aux[8][54])+aux[8][53];
        aux102=3*(aux[9][55]-aux[9][54])+aux[9][53];
        aux112=3*(aux[10][55]-aux[10][54])+aux[10][53];
        aux122=3*(aux[11][55]-aux[11][54])+aux[11][53];
        aux132=3*(aux[12][55]-aux[12][54])+aux[12][53];
        aux142=3*(aux[13][55]-aux[13][54])+aux[13][53];
        aux152=3*(aux[14][55]-aux[14][54])+aux[14][53];
        aux162=3*(aux[15][55]-aux[15][54])+aux[15][53];

        cad[0][0]=(-2*aux11+21*aux[0][0]+9*aux[0][1]-aux[0][2])/27;
        cad[1][0]=(-2*aux21+21*aux[1][0]+9*aux[1][1]-aux[1][2])/27;
        cad[2][0]=(-2*aux31+21*aux[2][0]+9*aux[2][1]-aux[2][2])/27;
        cad[3][0]=(-2*aux41+21*aux[3][0]+9*aux[3][1]-aux[3][2])/27;
        cad[4][0]=(-2*aux51+21*aux[4][0]+9*aux[4][1]-aux[4][2])/27;
        cad[5][0]=(-2*aux61+21*aux[5][0]+9*aux[5][1]-aux[5][2])/27;
        cad[6][0]=(-2*aux71+21*aux[6][0]+9*aux[6][1]-aux[6][2])/27;
        cad[7][0]=(-2*aux81+21*aux[7][0]+9*aux[7][1]-aux[7][2])/27;
        cad[8][0]=(-2*aux91+21*aux[8][0]+9*aux[8][1]-aux[8][2])/27;
        cad[9][0]=(-2*aux101+21*aux[9][0]+9*aux[9][1]-aux[9][2])/27;
        cad[10][0]=(-2*aux111+21*aux[10][0]+9*aux[10][1]-aux[10][2])/27;
        cad[11][0]=(-2*aux121+21*aux[11][0]+9*aux[11][1]-aux[11][2])/27;
        cad[12][0]=(-2*aux131+21*aux[12][0]+9*aux[12][1]-aux[12][2])/27;
        cad[13][0]=(-2*aux141+21*aux[13][0]+9*aux[13][1]-aux[13][2])/27;
        cad[14][0]=(-2*aux151+21*aux[14][0]+9*aux[14][1]-aux[14][2])/27;
        cad[15][0]=(-2*aux161+21*aux[15][0]+9*aux[15][1]-aux[15][2])/27;

cad[0][1]= (-aux11+9*aux[0][0]+21*aux[0][1]-2*aux[0][2])/27;
        cad[1][1]= (-aux21+9*aux[1][0]+21*aux[1][1]-2*aux[1][2])/27;
        cad[2][1]= (-aux31+9*aux[2][0]+21*aux[2][1]-2*aux[2][2])/27;
        cad[3][1]= (-aux41+9*aux[3][0]+21*aux[3][1]-2*aux[3][2])/27;
```

```
cad[4][1]= (-aux51+9*aux[4][0]+21*aux[4][1]-2*aux[4][2])/27;
cad[5][1]= (-aux61+9*aux[5][0]+21*aux[5][1]-2*aux[5][2])/27;
cad[6][1]= (-aux71+9*aux[6][0]+21*aux[6][1]-2*aux[6][2])/27;
cad[7][1]= (-aux81+9*aux[7][0]+21*aux[7][1]-2*aux[7][2])/27;
cad[8][1]= (-aux91+9*aux[8][0]+21*aux[8][1]-2*aux[8][2])/27;
cad[9][1]= (-aux101+9*aux[9][0]+21*aux[9][1]-2*aux[9][2])/27;
cad[10][1]= (-aux111+9*aux[10][0]+21*aux[10][1]-2*aux[10][2])/27;
cad[11][1]= (-aux121+9*aux[11][0]+21*aux[11][1]-2*aux[11][2])/27;
cad[12][1]= (-aux131+9*aux[12][0]+21*aux[12][1]-2*aux[12][2])/27;
cad[13][1]= (-aux141+9*aux[13][0]+21*aux[13][1]-2*aux[13][2])/27;
cad[14][1]= (-aux151+9*aux[14][0]+21*aux[14][1]-2*aux[14][2])/27;
cad[15][1]= (-aux161+9*aux[15][0]+21*aux[15][1]-2*aux[15][2])/27;
```

```
cad[0][110]=(-2*aux[0][53]+21*aux[0][54]+9*aux[0][55]-aux12)/27;
cad[1][110]=(-2*aux[1][53]+21*aux[1][54]+9*aux[1][55]-aux22)/27;
cad[2][110]=(-2*aux[2][53]+21*aux[2][54]+9*aux[2][55]-aux32)/27;
cad[3][110]=(-2*aux[3][53]+21*aux[3][54]+9*aux[3][55]-aux42)/27;
cad[4][110]=(-2*aux[4][53]+21*aux[4][54]+9*aux[4][55]-aux52)/27;
cad[5][110]=(-2*aux[5][53]+21*aux[5][54]+9*aux[5][55]-aux62)/27;
cad[6][110]=(-2*aux[6][53]+21*aux[8][54]+9*aux[6][55]-aux72)/27;
cad[7][110]=(-2*aux[7][53]+21*aux[7][54]+9*aux[7][55]-aux82)/27;
cad[8][110]=(-2*aux[8][53]+21*aux[8][54]+9*aux[8][55]-aux92)/27;
cad[9][110]=(-2*aux[9][53]+21*aux[9][54]+9*aux[9][55]-aux102)/27;
cad[10][110]=(-2*aux[10][53]+21*aux[10][54]+9*aux[10][55]-aux112)/27;
cad[11][110]=(-2*aux[11][53]+21*aux[11][54]+9*aux[11][55]-aux122)/27;
cad[12][110]=(-2*aux[12][53]+21*aux[12][54]+9*aux[12][55]-aux132)/27;
cad[13][110]=(-2*aux[13][53]+21*aux[13][54]+9*aux[13][55]-aux142)/27;
cad[14][110]=(-2*aux[14][53]+21*aux[14][54]+9*aux[14][55]-aux152)/27;
cad[15][110]=(-2*aux[15][53]+21*aux[15][54]+9*aux[15][55]-aux162)/27;
```

```
cad[0][111]=(-aux[0][53]+9*aux[0][54]+21*aux[0][55]-2*aux12)/27;
cad[1][111]=(-aux[1][53]+9*aux[1][54]+21*aux[1][55]-2*aux22)/27;
cad[2][111]=(-aux[2][53]+9*aux[2][54]+21*aux[2][55]-2*aux32)/27;
cad[3][111]=(-aux[3][53]+9*aux[3][54]+21*aux[3][55]-2*aux42)/27;
cad[4][111]=(-aux[4][53]+9*aux[4][54]+21*aux[4][55]-2*aux52)/27;
cad[5][111]=(-aux[5][53]+9*aux[5][54]+21*aux[5][55]-2*aux62)/27;
cad[6][111]=(-aux[6][53]+9*aux[6][54]+21*aux[6][55]-2*aux72)/27;
cad[7][111]=(-aux[7][53]+9*aux[7][54]+21*aux[7][55]-2*aux82)/27;
cad[8][111]=(-aux[8][53]+9*aux[8][54]+21*aux[8][55]-2*aux92)/27;
cad[9][111]=(-aux[9][53]+9*aux[9][54]+21*aux[9][55]-2*aux102)/27;
cad[10][111]=(-aux[10][53]+9*aux[10][54]+21*aux[10][55]-2*aux112)/27;
cad[11][111]=(-aux[11][53]+9*aux[11][54]+21*aux[11][55]-2*aux122)/27;
cad[12][111]=(-aux[12][53]+9*aux[12][54]+21*aux[12][55]-2*aux132)/27;
cad[13][111]=(-aux[13][53]+9*aux[13][54]+21*aux[13][55]-2*aux142)/27;
cad[14][111]=(-aux[14][53]+9*aux[14][54]+21*aux[14][55]-2*aux152)/27;
cad[15][111]=(-aux[15][53]+9*aux[15][54]+21*aux[15][55]-2*aux162)/27;
for(i=2;i<110;i+=2){

a=cont+1;
b=cont+2;
c=cont+3;

cad[0][i]=(-2*aux[0][cont]+21*aux[0][a]+9*aux[0][b]-aux[0][c])/27;
                                        cad[0][i+1]=(-aux[0][cont]+9*aux[0][a]+21*aux[0][b]-2*aux[0][c])/27;
```

```
cad[1][i]=(-2*aux[1][cont]+21*aux[1][a]+9*aux[1][b]-aux[1][c])/27;
cad[1][i+1]=(-aux[1][cont]+9*aux[1][a]+21*aux[1][b]-2*aux[1][c])/27;
cad[2][i]=(-2*aux[2][cont]+21*aux[2][a]+9*aux[2][b]-aux[2][c])/27;
cad[2][i+1]=(-aux[2][cont]+9*aux[2][a]+21*aux[2][b]-2*aux[2][c])/27;
cad[3][i]=(-2*aux[3][cont]+21*aux[3][a]+9*aux[3][b]-aux[3][c])/27;
cad[3][i+1]=(-aux[3][cont]+9*aux[3][a]+21*aux[3][b]-2*aux[3][c])/27;

cad[4][i]=(-2*aux[4][cont]+21*aux[4][a]+9*aux[4][b]-aux[4][c])/27;
                                        cad[4][i+1]=(-aux[4][cont]+9*aux[4][a]+21*aux[4][b]-2*aux[4][c])/27;
cad[5][i]=(-2*aux[5][cont]+21*aux[5][a]+9*aux[5][b]-aux[5][c])/27;
cad[5][i+1]=(-aux[5][cont]+9*aux[5][a]+21*aux[5][b]-2*aux[5][c])/27;
cad[6][i]=(-2*aux[6][cont]+21*aux[6][a]+9*aux[6][b]-aux[6][c])/27;
cad[6][i+1]=(-aux[6][cont]+9*aux[6][a]+21*aux[6][b]-2*aux[6][c])/27;
cad[7][i]=(-2*aux[7][cont]+21*aux[7][a]+9*aux[7][b]-aux[7][c])/27;
cad[7][i+1]=(-aux[7][cont]+9*aux[7][a]+21*aux[7][b]-2*aux[7][c])/27;
cad[8][i]=(-2*aux[8][cont]+21*aux[8][a]+9*aux[8][b]-aux[8][c])/27;
cad[8][i+1]=(-aux[8][cont]+9*aux[8][a]+21*aux[8][b]-2*aux[8][c])/27;
cad[9][i]=(-2*aux[9][cont]+21*aux[9][a]+9*aux[9][b]-aux[9][c])/27;
cad[9][i+1]=(-aux[9][cont]+9*aux[9][a]+21*aux[9][b]-2*aux[9][c])/27;
cad[10][i]=(-2*aux[10][cont]+21*aux[10][a]+9*aux[10][b]-aux[10][c])/27;
cad[10][i+1]=(-aux[10][cont]+9*aux[10][a]+21*aux[10][b]-2*aux[10][c])/27;
cad[11][i]=(-2*aux[11][cont]+21*aux[11][a]+9*aux[11][b]-aux[11][c])/27;
cad[11][i+1]=(-aux[11][cont]+9*aux[11][a]+21*aux[11][b]-2*aux[11][c])/27;

cad[12][i]=(-2*aux[12][cont]+21*aux[12][a]+9*aux[12][b]-aux[12][c])/27;
cad[12][i+1]=(-aux[12][cont]+9*aux[12][a]+21*aux[12][b]-2*aux[12][c])/27;
cad[13][i]=(-2*aux[13][cont]+21*aux[13][a]+9*aux[13][b]-aux[13][c])/27;
cad[13][i+1]=(-aux[13][cont]+9*aux[13][a]+21*aux[13][b]-2*aux[13][c])/27;
cad[14][i]=(-2*aux[14][cont]+21*aux[14][a]+9*aux[14][b]-aux[14][c])/27;
cad[14][i+1]=(-aux[14][cont]+9*aux[14][a]+21*aux[14][b]-2*aux[14][c])/27;
cad[15][i]=(-2*aux[15][cont]+21*aux[15][a]+9*aux[15][b]-aux[15][c])/27;
cad[15][i+1]=(-aux[15][cont]+9*aux[15][a]+21*aux[15][b]-2*aux[15][c])/27;
                                cont++;
                        }
```

```
cont=0,
                ChanOut(inter_retro,cad,sizeof(cad));
                ChanOutInt(mas_trabajo,1);
                break,
            case 1:
                k=ChanInChar(senal1);
                running=0;
                ChanOutChar(senal2,k);
            }

}
tiempo=ChanInInt(inter_switch);
ChanOutInt(inter_retro,tiempo);
}


void retro(Process *p3, Channel *retro_inter, Channel *to_farmer, Channel* senal2,Channel* from_worker2)
{
unsigned char cad[16][113];
int bandera,
int running1=1,

int tiempo,tiempo2;
int tareas=0,tareas2=0,
char k,
p3=p3;


while(running1){
            bandera=ProcAlt(retro_inter, senal2,from_worker2,NULL);
            switch(bandera){
            case 0:
                ChanIn(retro_inter,cad,sizeof(cad));
                    tareas++;
                ChanOut(to_farmer,cad,sizeof(cad));
               break;

            case 1:
                    k=ChanInChar(senal2);
                    running1=0;

                    break;
            case 2:
                ChanIn(from_worker2,cad,sizeof(cad));
                    tareas2++;
                ChanOut(to_farmer,cad,sizeof(cad));
               break,
    }

}


tiempo=ChanInInt(retro_inter);
tiempo=ProcTime()-tiempo;
ChanOutInt(to_farmer,tiempo);
ChanOutInt(to_farmer,tareas);
tiempo2=ChanInInt(from_worker2);
ChanOutInt(to_farmer,tiempo2);
ChanOutInt(to_farmer,tareas2);

}


int main (void) {
```

```
Channel* from_farmer;
Channel* to_farmer;
Channel* from_worker2;
Channel* to_worker2;

Channel*      switch_inter;
Channel* inter_retro;
Channel* mas_trabajo;
Channel*      senal1;
Channel* senal2;

Process       *p_switch,*p_interpolar,*p_retro;

from_farmer = (Channel *) get_param (1);
to_farmer   = (Channel *) get_param (2);
from_worker2 = (Channel *) get_param (3);
to_worker2   = (Channel *) get_param (4);



switch_inter = ChanAlloc();
inter_retro  = ChanAlloc();
mas_trabajo  = ChanAlloc();
senal1       = ChanAlloc();
senal2       = ChanAlloc();

p_switch     = Chk_ProcAlloc (switcher  , 0,5, from_farmer, switch_inter, mas_trabajo,  senal1,to_worker2);
p_interpolar  = Chk_ProcAlloc (interpolar, 0, 5, switch_inter , inter_retro, mas_trabajo, senal1, senal2);
p_retro      = Check_ProcAlloc (retro     , 0, 4, inter_retro, to_farmer, senal2,from_worker2);


ProcRunHigh(p_switch);
ProcRunHigh(p_retro);
ProcRunLow(p_interpolar);


exit_terminate(0);
```