03063
6
2ej

Universidad Nacional Autónoma de México

Unidad Académica de los Ciclos Profesional y de Posgrado del Colegio de Ciencias y Humanidades

Estudio y Evaluación de la Programación Orientada a Objetos en una Arquitectura Paralela.

Maestría en Ciencias de la Computación.

Jorge Luis Ortega Arjona.

Director: Dr. Fabián García Nocetti.

junio 1996.

TESIS CON
FALLA DE ORIGEN

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

TESIS

COMPLETA

"La fuerza es la moral de los hombres
que se distinguen de los demás,
y es la mía".

Ludwig van Beethoven.

Indice General.

Contenido.

Introducción. 1

Capítulo I. Del Paralelismo. 5

A. Definiciones y elementos. 5

1. Definiciones. 6

a) Proceso. 6

i) Procesos disjuntos. 7

ii) Procesos cooperativos. 7

b) Programación concurrente, paralela y distribuida. 7

c) Organización de la memoria. 10

i) Memoria compartida. 12

ii) Memoria distribuida. 12

d) Mecanismos basados en variables compartidas: semáforos, regiones críticas y monitores. 13

i) Semáforos o secciones críticas. 13

ii) Regiones críticas. 15

iii) Monitores. 16

e) Mecanismos basados en paso de mensajes: canales y llamadas a procedimientos remotos. 17

i) Canales. 18

ii) Llamadas a procedimientos remotos. 19

f) Abrazo mortal (*Deadly embrace o dead-lock*). 20

2. Elementos. 21

a) Número de procesadores. 21

b) Topologías de conexión. 21

i) No restringida (*unconstrained*). 22

ii) Lineal. 23

iii) Anillo. 23

iv) Estrella. 25

v) Malla. 25

vi) Arbol. 26

vii) Hipercubo. 27

c) Métodos de paralelización. Granularidad y balance de trabajo. 28

i) Paralelismo algorítmico. 31

ii) Paralelismo geométrico. 32

iii) Paralelismo *farm*. 33

d) Métricas de desempeño. 34

i) Tiempos de procesamiento y de comunicación. 34

ii) Speedup. 35

iii) Eficiencia. 36

iv) Fracción serial. 36

B. Características. Ventajas y desventajas. 37

1. Características. 38

a) Paralelismo. 38

b) Secuencialidad. 38

c) Comunicación y sincronización. 40

d) No determinismo. 42

2. Ventajas. 43

a) Aumento en la velocidad. 43

b) Eficiencia en el procesamiento. 43

c) Simplicidad en la expresión de modelos. 44

d) Aplicaciones. 44

e) Tamaño y costo. 44

3. Desventajas. 45

a) Responsabilidades del programador. 45

b) Seguridad contra errores. 45

c) Eficiencia en la programación. 46

d) Problema de *hardware*. 46

e) Problema de *software*. 46

Capítulo II.De la Orientación a Objetos. 49

A. Definiciones y elementos. 49

1. Definiciones. 50

a) Objeto, Clase y Programación Orientada a Objetos. 50

b) Genericidad. 54

c) Polimorfismo y sobrecarga de funciones. 54

d) Ligadura estática y dinámica. 55

e) Métricas de la orientación a objetos. 56

2. Elementos. 59

a) Abstracción. 59

b) Encapsulación. 61

c) Modularidad. 61

d) Jerarquización. Relaciones entre clases. 64

i) Herencia. 64

ii) Agregación. 66

iii) Asociación. 66

iv) Uso. 67

e) Tipificación. 67

f) Persistencia. 68

g) Concurrencia. 69

B. Características. Ventajas y desventajas. 70

1. Características. 70

a) Estructura modular basada en objetos. 70

b) Abstracción de datos. 71

c) Manejo automático de memoria. 71
d) Clases. 71
e) Herencia. 72
f) Polimorfismo y ligadura dinámica. 73
g) Herencia múltiple y repetida. 74
2. Ventajas. **74**
a) Uniformidad. 74
b) Entendimiento. 75
c) Flexibilidad y mantenimiento. 75
d) Estabilidad. 76
e) Reusabilidad. 76
f) Tamaño del código. 77
g) Especificación y verificación. 77
3. Desventajas. **78**
a) Complejidad en el diseño. 79
b) Desempeño. 80
c) Dependencia en etapas de análisis y diseño. 81
d) Coherencia y localización. 81
e) Costos iniciales. 83

Capítulo III. Programación Paralela Orientada a Objetos. 85

A. Objetivo del estudio. 86

B. La programación paralela y la programación orientada a objetos. 87
1. ¿Por qué paralelismo orientado a objetos? 87
2. El paralelismo orientado a objetos. 88
a) Objetos y mensajes. 88
i) Paralelismo interobjeto e intraobjeto. 89
ii) Objetos activos y pasivos. 91
b) Modelos de programación. 92
i) Tareas Paralelas. 92
ii) Paralelismo de datos. El modelo de colección distribuida. 92
iii) Objetos comunicantes. El modelo de actores. 95
c) Incorporación del paralelismo a la programación orientada a objetos. 99
i) Bibliotecas de clases. 99
ii) Extensiones del Lenguaje. 100

C. Análisis de ventajas y desventajas. 101
1. Ventajas. 102
a) Alto nivel de abstracción. 102
b) Descomposición implícita de actividades paralelas. 102
c) Mayor nivel de encapsulación. 102
d) Transparencia en sincronización. 103
e) Localización y autocontenido. 103
f) Diseño basado en paralelización incremental. 103
g) Dinamicidad. 103

h) Multigranularidad. 104
i) Aplicaciones. 104
2. Desventajas. 105
a) Dependencia de datos. 105
b) Identificación de objetos. 105
c) Comunicación y sincronización asociada al paralelismo. 106
d) *Deadlock* recursivo de los sistemas concurrentes. 106
e) Desempeño. 107
f) Herencia estática y dinámica. 108
g) Herencia anómala. 109

D. Un lenguaje paralelo orientado a objetos: INMOS Parallel C++, de Glockenspiel. 109
1. Paralelismo. 111
a) Modelo Interobjeto. 111
b) Modelo Intraobjeto. 113
2. Secuencialidad. 116
a) Modelo Interobjeto. 116
b) Modelo Intraobjeto. 118
3. Comunicación y Sincronización. 118
a) Modelo Interobjeto. 118
b) Modelo Intraobjeto. 118
4. No determinismo. 122
a) Modelo Interobjeto. 122
b) Modelo Intraobjeto. 123
5. Temporizadores. 125
6. Objetos activos paralelos. 127

Capítulo IV. Ejemplo de Aplicación: La Transformada Rápida de Fourier. 129

A. Procesamiento de señales e imágenes. 130
1. Técnicas de transformación en el procesamiento de señales e imágenes. 130
2. Transformada Rápida de Fourier. 131
a) Señales. FFT en una dimensión. 131
b) Imágenes. FFT en dos dimensiones. 133

B. Análisis y diseño orientado a objetos. 134
1. Identificación y definición de objetos y métodos. 135
2. Organización de interfaces entre objetos. 136
3. Definición y organización de una jerarquía clases. 138

C. Análisis y diseño paralelo. 138
1. Identificación de paralelismo potencial. 138
2. Definición de la configuración de la arquitectura. 139

a) Lineal (*pipeline*). 141
b) Hipercubo. 141

D. Diseño paralelo orientado a objetos. 143
1. Configuración utilizando dos objetos. 148
2. Configuración utilizando cuatro objetos. 148

E. Métricas y observaciones. 150
1. Métricas del paralelismo. 150
a) Tiempo de ejecución. 150
b) *Speedup*. 152
c) Eficiencia. 153
d) Fracción serial. 154
2. Métricas de la orientación a objetos. 155
a) Métodos por clase. 155
b) Profundidad del árbol de herencia. 156
c) Acoplamiento entre objetos. 156
d) Cohesión en métodos. 156
3. Observaciones. 157

Capítulo V. Conclusiones generales y orientaciones. 159

A. Conclusiones. 159

B. Orientaciones. 160

Apéndice A. Antecedentes del paralelismo. 163
Apéndice B. Antecedentes de la orientación a objetos. 175
Apéndice C. Códigos fuente de los programas para el ejemplo de aplicación de la FFT 183
Bibliografía. 201

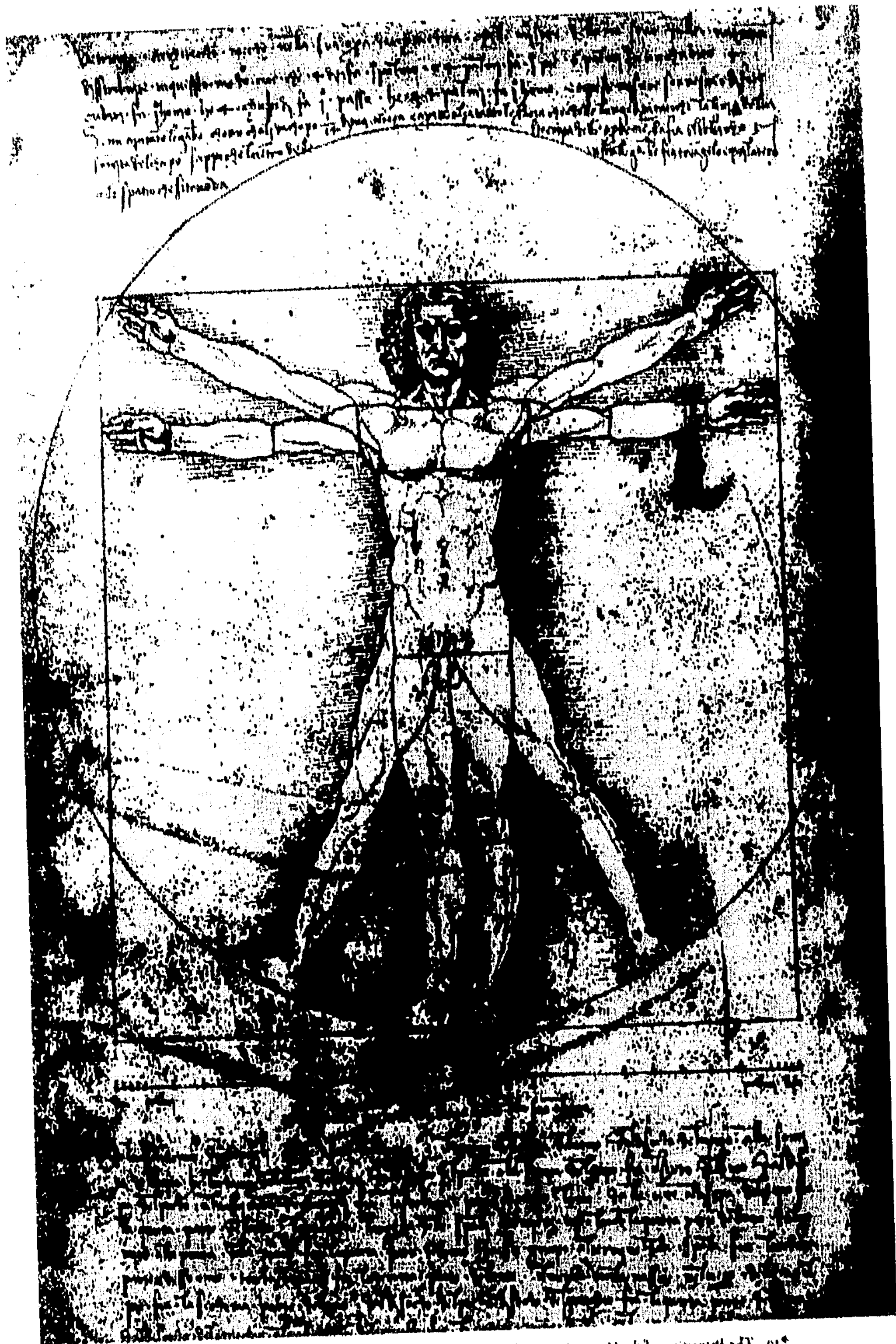
Figuras.

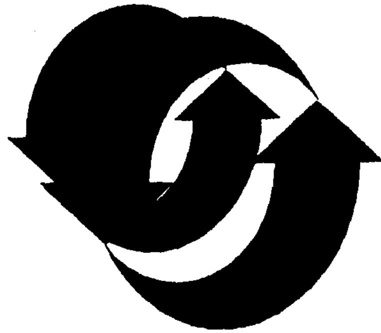
I.1, Procesos disjuntos. 8
I.2, Procesos cooperativos. 8
I.3, Procesos concurrentes. 11
I.4, Procesos paralelos. 11
I.5, Esquema de memoria compartida. 14
I.6, Esquema de memoria distribuida. 14
I.7, Topologías de conexión. 24
I.8, Paralelismo. 39

I.9, Secuencialidad. 39
I.10, Comunicación. 41
I.11, No determinismo. 41
II.1, Anatomía de un objeto. 52
II.2, Clase Párrafo con tres instancias. 52
II.3, Las ventajas del polimorfismo. 55
II.4, Un objeto representa una abstracción de la realidad. 60
II.5, La encapsulación oculta los detalles internos del objeto. 60
II.6, Modularidad. 63
II.7, Herencia. 63
III.1, Objetos y mensajes. 90
III.2, Tareas paralelas. 90
III.3, Paralelismo de datos. 94
III.4, Objetos comunicantes. 94
III.5, Paralelismo interobjeto. 114
III.6, Paralelismo intraobjeto. 114
III.7, Secuencialidad interobjeto. 117
III.8, Secuencialidad intraobjeto. 117
III.9, Comunicación interobjeto. 119
III.10, Comunicación intraobjeto. 119
III.11, No determinismo interobjeto. 124
III.12, No determinismo intraobjeto. 124
III.13, Clase de objetos paralelos. 127
IV.1, Gráfica del flujo de datos en una FFT. 137
IV.2, Jerarquía de clases para la FFT paralela. 137
IV.3, FFT para N=4 con sobreescrituras. 140
IV.4, FFT para N=4 en arreglos de procesadores. 140
IV.5, FFT en hipercubo para N=8. 142
IV.6, Configuración en dos procesadores. 149
IV.7, Configuración en cuatro procesadores. 149
IV.8, Tiempos de ejecución para el cálculo de la FFT. 151
IV.9, *Speedup* para el cálculo de la FFT. 152
IV.10, Eficiencia para el cálculo de la FFT. 153
IV.11, Fracción serial para el cálculo de la FFT. 155

Tablas.

IV.1, Tiempos de ejecución (ANSI C paralelo). 150
IV.2, Tiempos de ejecución (ANSI C++ paralelo). 151
IV.3, *Speedup* (ANSI C paralelo). 152
IV.4, *Speedup* (ANSI C++ paralelo). 152
IV.5, Eficiencia (ANSI C paralelo). 153
IV.6, Eficiencia (ANSI C++ paralelo). 153
IV.7, Fracción serial (ANSI C paralelo). 154
IV.8, Fracción serial (ANSI C++ paralelo). 154
IV.9, Métodos ponderados por clase para el cálculo de la FFT. 155





Introducción.

"Lo último que uno encuentra al realizar una obra es saber qué debe colocarse primero".
Blaise Pascal.

En 1946, John Von Neumann de la Universidad de Princeton, estableció que:

"Más allá de la capacidad de ejecutar las operaciones básicas por separado, una máquina computadora debe ser capaz de realizarlas de acuerdo a la secuencia - o mejor, el patrón lógico - en la cual generen la solución del problema matemático, lo cual es el propósito actual de tal cálculo." [58]

De esta forma, Von Neumann propone que las instrucciones de una máquina computadora pueden ser codificadas, leídas y escritas en una memoria única, permitiendo que sean fácilmente modificadas y su ejecución se realice en un órden cronológico preestablecido. Con esto se inicia la computación moderna tal y como se conoce hoy día, ya que la idea de procesamiento secuencial de instrucciones se encuentra aún vigente en los sistemas de cómputo actuales. Sin embargo, aún cuando se ha comprobado tanto teórica como prácticamente la validez del **Paradigma de Von Neumann** (como se conoce comúnmente a la afirmación precedente), la ejecución secuencial de instrucciones almacenadas en memoria representa una limitante para algunas aplicaciones, en especial para aquéllas en las que los parámetros como el tiempo de procesamiento, la confiabilidad de resultados y la tolerancia a fallas son primordiales.

Una tendencia que ha intentado dar solución a la limitante del Paradigma de Von Neumann es la utilización de arquitecturas paralelas. El paralelismo consiste en la repartición de la labor de procesamiento entre varias unidades semejantes que cooperan

entre sí. Esta tendencia ha sido considerada principalmente como solución al problema de crear computadoras cada vez más veloces. Actualmente, parece no existir un límite para la velocidad de los sistemas de cómputo. Los avances en la tecnología del *hardware* han permitido la aparición de dispositivos electrónicos cada vez más sofisticados, de mayor velocidad y menor costo. Pero es un hecho que ha de llegar un momento en que el límite de velocidad de los dispositivos electrónicos sea alcanzado. En tal momento, el paralelismo se plantea como el único medio para aumentar el desempeño de los sistemas.

Por otra parte, la importancia de los paradigmas del *software* en los últimos años han tomado un realce significativo debido a la dificultad que representa el desarrollo de una programación correcta, confiable y sobre todo, entendible [51]. Esta dificultad se acentúa especialmente cuando se utilizan arquitecturas paralelas como plataforma de procesamiento. El estilo de programación tradicional y la forma de pensar acerca de un problema determinado no son aplicables cuando la solución se genera por una colección de procesadores en paralelo que cooperan entre sí. De esta manera, el paralelismo se presenta como una oportunidad de reconsiderar la programación desde nuevas perspectivas. Tales perspectivas reciben el nombre de paradigmas. Un **paradigma** es un modelo del mundo real, que se utiliza para formular una solución mediante una computadora, para un problema determinado [51]. Existen varios paradigmas utilizados para la programación en arquitecturas paralelas, como por ejemplo, el paradigma sistólico, de reducción, de vector/arreglo, funcional de flujo de datos, de datos paralelos y orientado a objetos. Es este último en el cual se enfoca la atención del presente trabajo.

Muchos expertos están de acuerdo que la programación orientada a objetos ofrece mejorar en gran medida el desarrollo del *software* en los próximos años. Las principales ventajas del paradigma orientado a objetos descansan en su posibilidad de hacer frente a dos temas esenciales de la ingeniería de software: gestión de la complejidad y mejora de la productividad en el proceso de desarrollo del *software* [79]. Esto se realiza fomentando las siguientes estrategias:

- escribir código reutilizable,
- escribir código posible de mantener,
- depurar módulos de código existentes, y
- compartir código con otros.

En el método orientado a objetos, los objetivos del diseño pasan de modelar el comportamiento del mundo real, a modelar los objetos que existen en el mismo y sus comportamientos individuales. Si se practican correctamente las técnicas de programación orientada a objetos, la arquitectura de una aplicación sigue mucho más cerca la estructura del problema. Esto hace que el desarrollo, empleo y mantenimiento de una aplicación sean más regulares, fáciles y rápidos.

Los objetos son relativamente fáciles de definir, realizar y mantener porque reflejan la modularidad natural de una aplicación. Con el refuerzo de la modularidad y a través de mecanismos de herencia, los objetos de *software* pueden utilizarse de nuevo en aplicaciones futuras y de esta forma reducir substancialmente la cantidad de código nuevo que debe escribirse [79]. El código orientado a objetos puede ser lo bastante general como para volverse a utilizar sin modificación. De tal manera, la programación orientada a objetos mejora no sólo el proceso de desarrollo del *software*, sino también da flexibilidad y utilidad al *software* resultante.

El paradigma orientado a objetos puede modelar con precisión la mayor parte de los objetos y procesos del mundo real. Sin embargo, debido a que los principales desarrollos de la programación orientada a objetos se han realizado en arquitecturas secuenciales (o tipo Von Neumann), el paradigma se complica cuando intenta modelar procesos concurrentes. Establecer un orden lineal de sucesos potencialmente simultáneos durante el procesamiento puede desfigurar significativamente el proceso del mundo real que está modelando.

La mejor solución para modelar sucesos concurrentes es no simular el paralelismo de acciones [79], sino proporcionar en realidad un medio en el que los objetos puedan ejecutarse simultáneamente. Esto es, se requiere de arquitecturas paralelas para lograr un modelo más preciso de los objetos del mundo real.

La programación orientada a objetos y las arquitecturas paralelas, cada una por su parte, están cambiando rápidamente el mundo de la computación. Ambas se consideran entre las tendencias más importantes de la programación actual, ya que ofrecen soluciones novedosas a problemas demasiado complejos para la programación tradicional. Por un lado, el procesamiento en paralelo, con base en un *hardware* especializado, proporciona una mayor velocidad y capacidad en el proceso de cómputo, utilizando la repartición y simultaneidad de acciones y tareas. Por otro lado, la programación orientada a objetos propone una nueva manera de desarrollar *software* de mayor complejidad tecnológica, mediante la utilización de niveles más altos de abstracción.

Varios autores han propuesto una programación paralela y orientada a objetos [51][79], con la posibilidad de obtener las ventajas que representan cada una de estas tendencias. Se han desarrollado lenguajes distribuidos que soportan la concurrencia, permitiendo que objetos de diferentes máquinas se comuniquen a través del envío y recepción de mensajes. Se trata principalmente de extensiones a lenguajes orientados a objetos tradicionales, como C++ y Smalltalk.

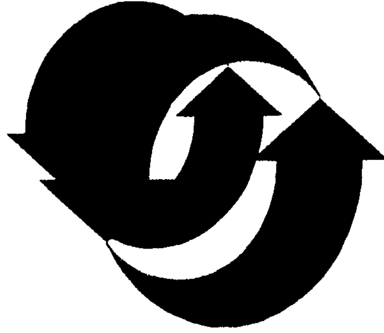
El propósito de este trabajo es realizar un estudio y evaluación de la programación orientada a objetos en una arquitectura paralela, teniendo en consideración las ventajas

que ambas tendencias plantean por separado. La discusión se realiza en dos partes: la primera, el estudio, hace un análisis del paralelismo, la orientación a objetos y del paralelismo orientado a objetos; la segunda parte, de evaluación, se refiere a un análisis práctico del paralelismo orientado a objetos, utilizando un lenguaje experimental para ejemplificarlo, realizando una evaluación mediante la implementación de una solución a un problema determinado.

La primera parte de estudio se presenta en los capítulos I, II y III, tratando por separado ambas tendencias del paralelismo y la orientación a objetos y, a continuación, analizando su combinación en una programación paralela orientada a objetos.

En el capítulo I se hace una breve reseña del paralelismo, mencionando sus principales conceptos, definiciones y elementos básicos, así como sus características más importantes. Por otra parte, el capítulo II realiza un tratamiento semejante con la programación orientada a objetos. Finalmente, el capítulo III trata la propuesta de realizar una programación orientada a objetos en una arquitectura paralela, analizando las ventajas y desventajas que se pueden obtener de cada uno, y comentando el C++ paralelo como una implantación del paralelismo en un lenguaje orientado a objetos.

La segunda parte de evaluación se forma de los capítulos IV y V, en los cuales se considera esencialmente un ejemplo de aplicación y las conclusiones y directivas para futuro desarrollo. De esta forma, en el capítulo IV se presenta un ejemplo de análisis y diseño orientado a objetos en una arquitectura paralela, mediante una aplicación de procesamiento de señales e imágenes basándose en una operación común a ambos: la transformada rápida de Fourier, a fin de medir el desempeño desde el punto de vista del paralelismo y de la orientación a objetos. Para finalizar, el capítulo V se encarga de realizar una síntesis del estudio y la evaluación, a fin de llegar a una conclusión del trabajo y presentar algunas orientaciones al lector.



Capítulo I.
Del Paralelismo.

"Trabajo pesado es, por lo general,
la acumulación de tareas livianas que no se hicieron a tiempo".
Anónimo.

Este capítulo pretende realizar una breve presentación de los principales conceptos del paralelismo, introduciendo las definiciones y elementos más relevantes dentro del modelo paralelo. En seguida se hace mención de sus características más importantes, así como un análisis de las ventajas y desventajas que, como una nueva tecnología, el paralelismo presenta.

A. Definiciones y elementos.

"Si uno es bueno, entonces 10 ó 1,000 debe ser mejor"[8]. En los últimos años, ideas como ésta son las que han dado origen al procesamiento en paralelo. Aún cuando tal afirmación no es necesariamente cierta, propone una atracción primaria hacia las arquitecturas paralelas. Es un hecho que quienes más han contribuido en el desarrollo de la computación actual, han aceptado al paralelismo como una manera novedosa de solucionar algunos problemas de programación. El mismo Von Neumann, reconoció el valor del paralelismo en el *hardware*[58,51], aún cuando sus trabajos y diseños se enfocaron principalmente en modelos secuenciales guiándose por consideraciones prácticas de su tiempo, como el costo y la confiabilidad.

En el desarrollo de esta sección y para fines del presente trabajo, se definen varios conceptos de la programación concurrente, paralela y distribuida. Conjuntamente, se analizan algunos elementos de la programación paralela que especialmente influyen en la creación de sistemas de procesamiento paralelo.

1. Definiciones.

a) Proceso.

Tradicionalmente, la mayoría de los autores acepta que *"un programa en ejecución se llama proceso"* [46,72]. Tal definición establece la diferencia entre un programa "pasivo" almacenado en disco, y un programa "activo" que se encuentra en la memoria del procesador. Sin embargo, aún cuando tal definición considera una característica básica del proceso, es extremadamente parcial, ya que no lo determina completamente.

Una segunda aproximación consiste en considerar que un proceso *"... es esencialmente una secuencia de operaciones que pueden ser realizadas por un solo procesador"* [50]. En esta definición se considera implícitamente la característica activa del proceso, su estructura básica y su relación con el procesador. Sin embargo, tampoco esta definición se considera completa, ya que aísla al proceso de los demás componentes de *software*.

Así, desde un punto de vista más amplio, se considera al proceso o conjunto de procesos como parte de un sistema de *software*: *"Un sistema de software es un conjunto de mecanismos para realizar ciertas acciones sobre ciertos datos"* [55], es decir, datos y procesos forman la estructura de un programa. Tal idea es semejante a *"... usar la palabra proceso para describir el patrón de comportamiento de un objeto, de tal manera que (éste) pueda ser descrito en términos del conjunto limitado de eventos seleccionados como su alfabeto"* [38]. Aún cuando considera también al proceso como parte de un sistema, esta última definición amplía más el concepto, extendiéndolo hasta considerar al proceso o procesos que realiza un objeto como una descripción del propio objeto.

Basándose en las definiciones anteriores, para este trabajo se considera que:

Un proceso es cualquier secuencia de operaciones que están ejecutándose en memoria activa, que realiza una o varias acciones sobre ciertos datos y permite describir el comportamiento de un objeto

mediante las acciones que sea determinado a realizar.

Existen algunas características de los procesos que son especialmente importantes dentro del desarrollo del procesamiento paralelo. Así, se consideran los procesos disjuntos y procesos cooperativos.

i) Procesos disjuntos.

Se dice que dos o más procesos concurrentes son disjuntos si se ejecutan simultáneamente en diferentes bloques de un programa, sin posibilidad de accederse entre sí; ambos inician su ejecución simultáneamente y proceden concurrentemente hasta que terminan [23,11,38]. La ejecución de tales procesos concurrentemente tiene el mismo efecto que su ejecución secuencial en cualquier orden. Dado que no comparten variables, los procesos concurrentes disjuntos no se comunican entre sí (Figura I.1).

ii) Procesos cooperativos.

En contraste, se tienen los procesos concurrentes cooperativos. Estos tienen la posibilidad de comunicarse para cooperar entre sí en la realización de una tarea, compartiendo recursos en común, por lo que requieren de alguna forma de sincronización para evitar la corrupción de los datos del programa (Figura I.2) [23,35,11,36,12,38,2]. Se han desarrollado varias soluciones para el problema de sincronización entre procesos, como son los semáforos [23], regiones críticas [35,11], monitores [36,12] y monitores anidados [38].

Al ejecutarse en un solo procesador, este tipo de procesos se ven forzados a competir por recursos críticos, como son el propio procesador, la memoria, etc.[2].

b) Programación concurrente, paralela y distribuida.

Un programa concurrente, paralelo o distribuido se especifica por dos o más procesos que se ejecutan simultáneamente en un tiempo determinado, y que cooperan entre sí para realizar una tarea [51,23,12,38,2]. Cada proceso representa un programa secuencial que ejecuta una serie de instrucciones. Los procesos cooperan entre sí, utilizando alguna forma de comunicación. Esta puede lograrse utilizando variables compartidas o mediante paso de mensajes

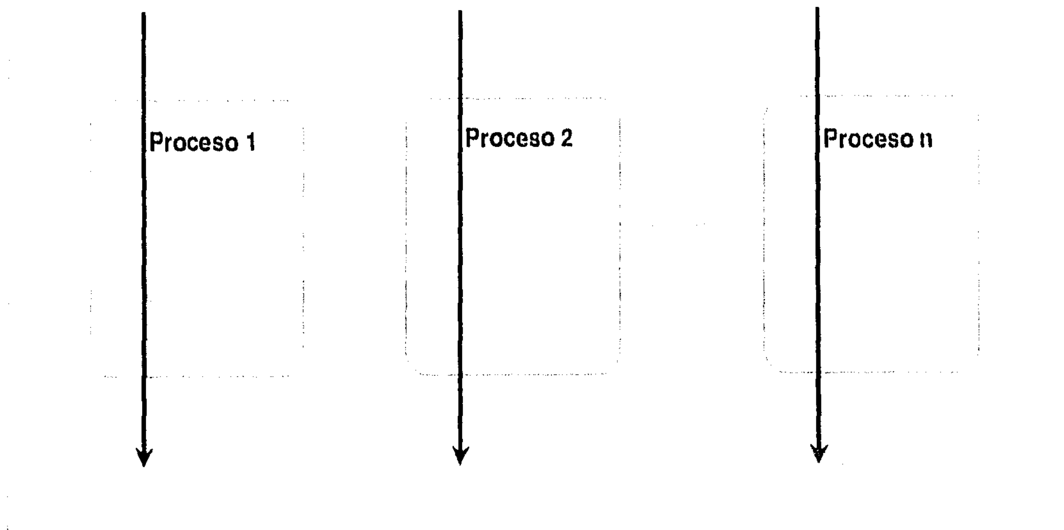


Figura I.1, Procesos disjuntos.

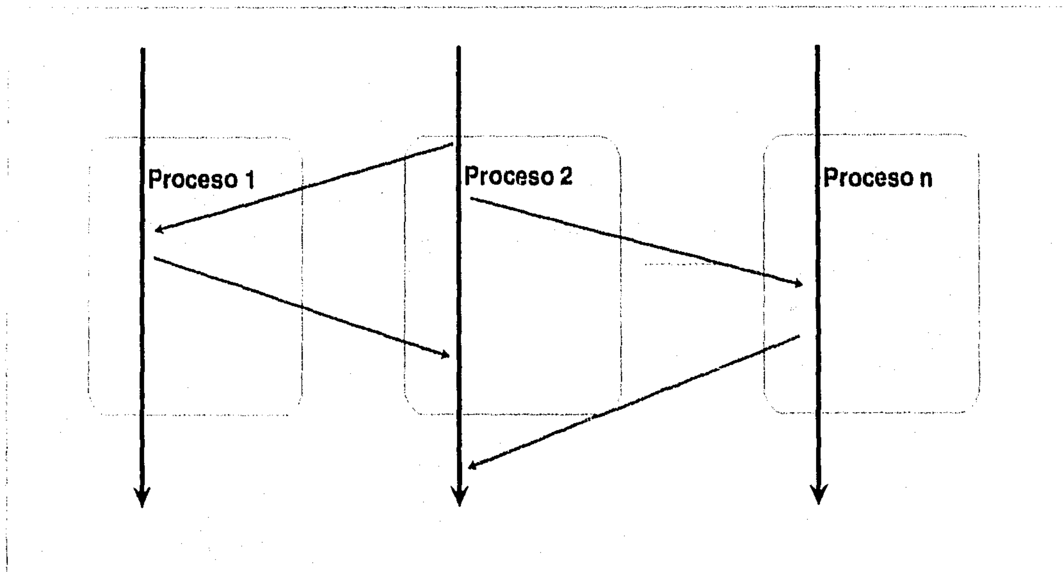


Figura I.2, Procesos cooperativos.

[38,2]. Cuando se utilizan variables compartidas, un proceso escribe en una variable que es leída por otro proceso [23,33,65]. En el caso de paso de mensajes, un proceso envía un mensaje que es recibido por otro proceso [37,13].

Al ejecutarse un programa secuencial, éste sigue una sola "hebra de control", esto es, inicia con una operación atómica (indivisible) del proceso, y se mueve a través del proceso conforme las operaciones atómicas se van ejecutando. La ejecución de un programa concurrente, paralelo o distribuido resulta en el seguimiento de múltiples hebras de control (cada una constituyendo un proceso secuencial), que se comunican entre sí [2].

Un programa concurrente puede ser ejecutado permitiendo a los procesos compartir uno o más procesadores, o ejecutando cada proceso en un solo procesador. La primera aproximación se conoce como **multiprogramación** y se basa en un proceso o *kernel* que multiplexa los procesos en uno o varios procesadores [2,3,34]. El paralelismo se lleva a cabo a nivel interno de cada procesador, repartiendo el tiempo del procesador entre los procesos (Figura I.3), realizando una o varias instrucciones atómicas de cada uno de los procesos en un tiempo diferente, pero dando la impresión que el procesador ejecuta todos los procesos concurrentes en un tiempo determinado. A este tipo de paralelismo se le conoce con el nombre de **paralelismo temporal** [31]. A partir de la multiprogramación, se elabora la siguiente definición:

Un programa concurrente es aquél formado por varios procesos que se ejecutan (en general) **en un solo procesador, siguiendo un esquema de paralelismo temporal.**

La segunda aproximación es conocida como **multiprocesamiento**, y es el caso de un programa que se ejecuta en varios procesadores que comparten una memoria común, o como **procesamiento distribuido o multiprocesador** si los procesadores se comunican entre sí mediante una red de conexiones [2,3,34]. A diferencia de la multiprogramación, el paralelismo se expresa físicamente en varios procesadores que operan simultáneamente (Figura I.4). La ejecución de un programa en varios procesadores es lo que se considera como un **paralelismo real o espacial** [31]. Por esto es que los sistemas basados en este esquema se consideran como paralelos. Sin embargo, se hace una diferencia atendiendo a las

características de la red de interconexión. De esta manera, se definen la **programación paralela** y **distribuida** como sigue:

Un programa paralelo o distribuido es aquél formado por varios procesos que se ejecutan en varios procesadores conectados entre sí mediante una red de comunicación, siguiendo un esquema de paralelismo espacial. Si la red se forma por conexiones dentro de una sola computadora, se considera un sistema de programación paralela; por otro lado, si la red se forma de conexiones entre diferentes computadoras, se considera un sistema de programación distribuida.

Esto es, un sistema multiprocesador puede formarse de una computadora con varios procesadores en el caso de la programación paralela, o por varias computadoras uniprocesador comunicadas por una red en el caso de la programación distribuida. Ambos enfoques aceptan una organización de memoria compartida o distribuida, como se menciona más adelante.

Es posible también considera aproximaciones híbridas entre las programaciones concurrente, paralela y distribuida, es decir, sistemas de computadoras paralelas conectadas en red, cuyos procesadores permiten la ejecución concurrente de procesos.

c) Organización de la memoria.

En los sistemas multiprocesador existen recursos considerados comunes o compartidos por todos los procesadores, como es el caso de la memoria y los periféricos. Durante la ejecución de un programa paralelo o distribuido, los procesadores se sirven de tales recursos compartidos [I.A.1.b] para comunicarse entre sí. El grado de interacción entre los procesos (y en consecuencia, entre los procesadores) se clasifica en dos tipos: **fuertemente acoplados** (*tightly-coupled*), para el caso en que el grado de interacción es alto, y **débilmente acoplados** (*loosely-coupled*), en caso contrario [23,31].

Existen dos tipos de mecanismos disponibles para soportar la interacción y comunicación de los procesadores: memoria compartida y memoria distribuida. Como su nombre lo indica, la manera en que la memoria es "repartida" entre los procesadores determina el mecanismo de comunicación que se utiliza: variables

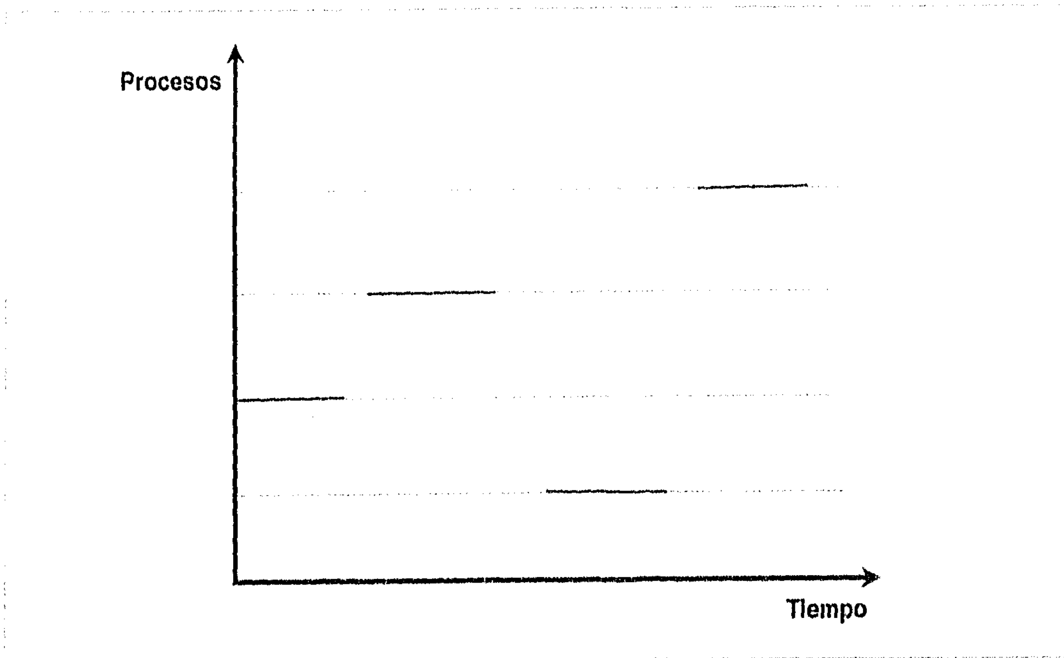


Figura I.3, Procesos concurrentes.

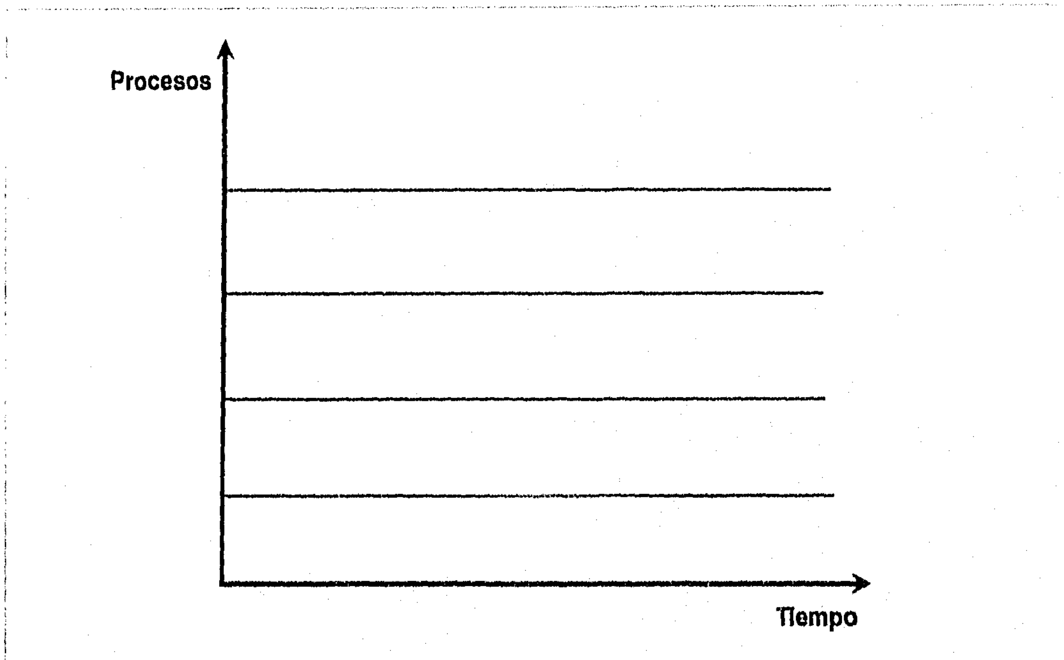


Figura I.4, Procesos paralelos.

compartidas o paso de mensaje. En general, los sistemas fuertemente acoplados utilizan un esquema de variables compartidas para la comunicación entre procesadores, mientras que los sistemas débilmente acoplados se basan principalmente en un esquema de paso de mensajes [71].

i) Memoria compartida.

Un sistema multiprocesador con memoria compartida es aquél que permite el acceso de cualquier procesador del sistema a cualquier localidad de una memoria común, mediante una red de conexión.

La red de conexión es controlada en la mayoría de los casos completamente por un sistema de control de *hardware* que no es visible al programador, quien sólo observa una memoria compartida central y continua [50,31]. Cada dirección de memoria es única e idéntica para cada procesador (Figura I.5).

La comunicación se realiza por lectura y escritura de variables en la misma sección de la memoria compartida. Cuando un procesador lee o escribe en una dirección específica de memoria, la actividad de la red de conexión y la selección del bloque de memoria apropiado son manejados automáticamente.

El esquema de lectura o escritura de variables en la memoria compartida se conoce como **variables compartidas.** Para asegurar la integridad de los datos en la memoria compartida, se deben desarrollar mecanismos para el soporte de comunicaciones, así como un ambiente de programación que provea de planeación, localización, sincronización y coordinación de los procesos.

Los mecanismos que han sido desarrollados para el manejo de variables compartidas son los **semáforos** [23,33,65], **regiones críticas** [35,11] y **monitores** [36,12].

ii) Memoria distribuida.

Un sistema multiprocesador de memoria distribuida es aquél en el que cada procesador utiliza su propia memoria privada, comunicándose con otros procesadores mediante una red de interconexión.

La red de interconexión se forma a partir de un número de procesadores conectados entre sí, basándose en alguna topología. Durante la ejecución de un programa la red puede permanecer estática o modificarse según el programa lo requiera [51].

La comunicación se realiza mediante envío o recepción de datos. Cada procesador reconoce la diferencia entre su memoria local y la memoria remota de otro procesador (Figura I.6). El procesador puede leer o escribir datos en su memoria local libremente. Cuando dos procesadores desean intercambiar datos, esto debe realizarse explícitamente mediante envío o recepción de tales datos.

El esquema de envío o recepción de datos en un sistema de memoria distribuida se conoce como **paso de mensajes**. La comunicación entre procesos se considera como punto a punto, unidireccional y no *bufferizada*.

Los mecanismos básicos utilizados para el uso de paso de mensajes son los **canales** [37,38], para la programación paralela, y las **llamadas a procedimientos remotos** [13,46,72], en el caso de la programación distribuida.

d) Mecanismos basados en variables compartidas: semáforos, regiones críticas y monitores.

En el apéndice A se muestra una semblanza del desarrollo de la programación concurrente. En esos artículos se mencionan varios mecanismos que han sido desarrollados para resolver los problemas de seguridad y sincronización en el uso de variables compartidas. En esta sección se presentan algunas definiciones de los mecanismos desarrollados para resolver tales problemas.

i) Semáforos o secciones críticas.

Como se menciona en el apéndice A, la programación concurrente tiene origen en el trabajo de E.W. Dijkstra en 1968 [23,2]. En este artículo, Dijkstra propone y define un mecanismo que resuelve los problemas de integridad de los datos en un sistema concurrente cooperativo de variables compartidas, mediante la sincronización de los procesos que lo forman. A tales mecanismos se les conoce como **semáforos**.

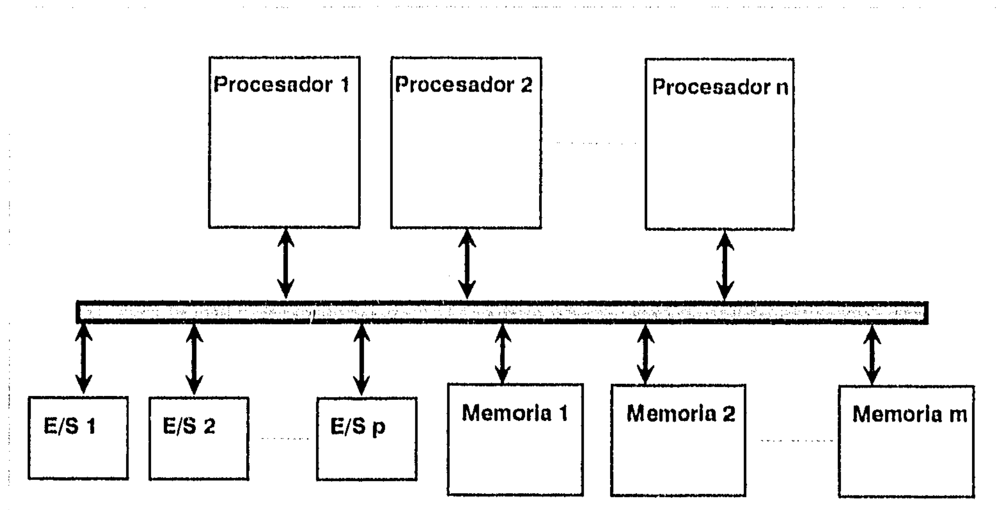


Figura I.5, Esquema de memoria compartida.

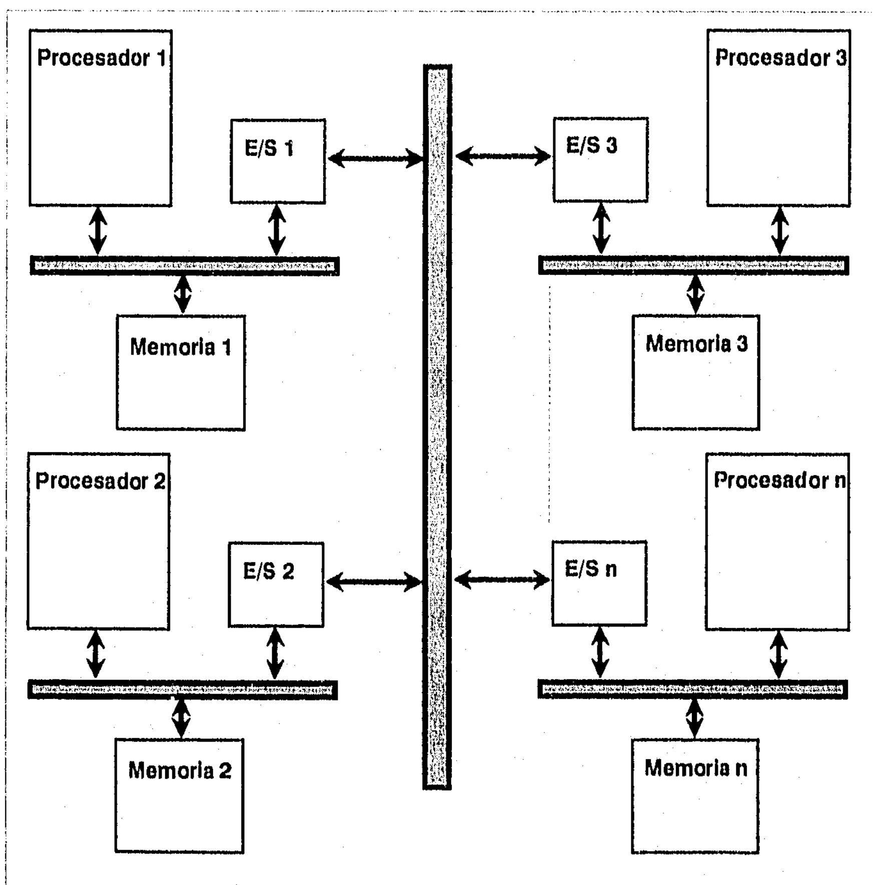


Figura I.6, Esquema de memoria distribuida.

De acuerdo a Dijkstra [23], los semáforos son variables enteras no negativas de propósito especial, las cuales se afectan únicamente mediante dos primitivas de ejecución indivisible llamadas **P** y V:

- V es una operación de un argumento, identificado como un semáforo. Su función es incrementar el valor del semáforo en 1. Este incremento es restringido a ocurrir como una operación indivisible.

- **P** es una operación de un argumento, identificado como un semáforo. Su función consiste en decrementar el valor del semáforo en 1, siempre y cuando el valor resultante del decremento sea no negativo. El decremento es restringido a ocurrir como una operación indivisible.

En el caso en que se aplique una operación **P** a un semáforo cuyo valor es cero, tal operación no podrá completarse hasta que otro proceso realice una operación V sobre el mismo semáforo. Sin embargo, en el caso de la ejecución concurrente de varios procesos, es probable que más de un proceso haya iniciado una operación **P** sobre el mismo semáforo, de tal manera que sólo un proceso al azar puede completar su operación **P**.

Los semáforos regulan el acceso a las variables compartidas, asegurando que sólo un proceso a la vez realice acciones sobre las variables compartidas. A tales acciones se les conoce con el nombre de **sección crítica** [23,2]. En general, se puede identificar como aquella sección de código que se encuentra entre las operaciones **P** y V sobre un mismo semáforo.

Los semáforos y secciones críticas fueron los primeros mecanismos propuestos para solucionar los problemas de exclusión mutua y sincronización entre procesos concurrentes [23,33,65]. Su utilización sigue vigente en la actualidad, principalmente en el desarrollo de sistemas operativos [72,2].

ii) Regiones críticas.

Aún cuando los semáforos representan una solución sencilla para la sincronización de procesos concurrentes, no son una solución

definitiva, debido a que el anidamiento de secciones críticas tiende a dificultar su uso, al generar "protocolos" de acceso complejos. Para simplificar esto, la solución propuesta son las **regiones críticas**.

Desarrolladas a partir de los conceptos de semáforos por Hoare y Brinch-Hansen [35,11,2], las regiones críticas surgen a partir del la idea de que los procesos concurrentes interactúan entre sí debido a que comparten un recurso limitado. De hecho, la ejecución de cada proceso es independiente hasta que requiere entrar a una región de código en el programa donde se encuentra el recurso compartido. Se determina que mediante una variable compartida (que hace las veces de un semáforo), sólo un proceso puede hacer uso de tal recurso dentro de la región crítica [35,11]. Al finalizar su acción sobre el recurso compartido, el proceso lo libera a fin de que otro proceso pueda usarlo.

Hoare y Brinch-Hansen proponen cada uno una notación para las regiones críticas. Ambas se componen por una variable compartida o recurso **v**, y la región crítica **C**. Utilizando esto último, la notación de Hoare [35] es:

```
with v do C
```

Por otro lado, la notación utilizada por Brinch-Hansen [11] es como sigue:

```
region v do C
```

Ambas notaciones expresan el funcionamiento de las regiones críticas: la variable compartida v se encarga de regular la entrada a la región crítica **C**. Las regiones críticas referidas a la misma variable (v, por ejemplo) se excluyen entre sí en el tiempo.

iii) Monitores.

Un siguiente nivel dentro de la solución a los problemas de exclusión mutua y sincronización entre procesos concurrentes, fue el desarrollo por los mismos Hoare y Brinch-Hansen [36,12]. Este mecanismo se conoce como **monitor**.

Inicialmente, es notable que a partir del problema de exclusión mutua, dos procesos no pueden hacer uso de una variable compartida al mismo tiempo. Es necesario sincronizar la acción de

los procesos concurrentes que acceden a una misma variable. Los semáforos solucionan estos problemas, insertando en el código las primitivas que controlan el uso de las variables dentro de la sección crítica. Más adelante, las regiones críticas representan el código donde se accede al recurso compartido, y se colocan en una zona fuera del código del programa. Finalmente, los monitores se desarrollan como un siguiente paso: se considera como un recurso compartido no únicamente las variables compartidas, sino también a los procedimientos y funciones que actúan sobre tales datos, restringiendo su uso por los procesos concurrentes. Su implementación se realiza en general mediante semáforos.

La notación propuesta por Hoare [36] para los monitores se basa en la notación de *clases* de **Simula67**. Un monitor tiene la siguiente forma:

```
monitorname:monitor
  begin...declaraciones de datos locales del monitor
    procedure procname(...parámetros formales...)
      begin ... cuerpo del procedimiento ... end;
    ... otros procedimientos locales del monitor;
          ... inicialización de los datos del monitor;
  end;
```

La invocación de un monitor desde el programa principal se propone de la siguiente manera:

```
monitorname.procname(...parámetros actuales...);
```

Por su parte, Brinch-Hansen utiliza esta notación de monitores en el desarrollo de Pascal Concurrente [12], proponiendo además una forma de implementación mediante colas y estados de espera.

e) Mecanismos basados en paso de mensajes: canales y llamadas a procedimientos remotos.

También en el apéndice A se muestran dos artículos que marcan el inicio de lo que respectivamente se ha definido como programación paralela y distribuida. En esos artículos se mencionan los mecanismos utilizados en el esquema de paso de mensajes. En esta sección se presentan algunas definiciones de tales mecanismos.

i) Canales.

Los **canales** son introducidos inicialmente como formas simples de comunicación entre procesos concurrentes por Hoare [37]; su definición y utilización se profundiza más adelante en un libro publicado posteriormente [38].

Hoare determina el funcionamiento de los canales mediante la comunicación entre un proceso origen y un proceso destino, en el cual sólo podrá llevarse a cabo la comunicación en el momento en que ambos procesos hagan una entrega (del proceso origen) o recepción (del proceso destino) del mensaje. Si algún proceso no se encuentra listo para realizar la comunicación, el otro proceso entrará en un estado de espera hasta que el envío pueda realizarse. En el caso de que un proceso se encuentre en la situación de enviar o recibir un mensaje y no logre comunicarse al finalizar el programa, se determina como un estado erróneo. Las principales características de la comunicación utilizando canales en el esquema de paso de mensajes son entonces [41,66]:

- **Punto a punto.** La comunicación entre procesos se realiza entre dos procesos exclusivamente.
- **No *bufferizada.*** No existe ninguna clase de almacenamiento momentáneo para el mensaje. La comunicación se establece o se entra en un estado de espera.
- **Unidireccional.** La comunicación se establece entre un proceso origen o emisor, hacia otro proceso destino o receptor.

El envío y recepción de mensajes se realiza mediante nuevas primitivas introducidas por Hoare [37]. Su notación es como sigue:

```
<proceso origen> ? <variable de entrada>

<proceso destino> ! <expresión de salida>
```

Como se puede observar, Hoare propone que ambas primitivas de entrada y salida requieran únicamente los valores a enviar o recibir, y los identificadores de los procesos que se

comunican. La programación de los canales se considera entonces transparente para el programador. Sin embargo, en la realidad, la implementación de las primitivas de Hoare utilizando los identificadores de los procesos resulta demasiado compleja. Debido a esto, la mayoría de los lenguajes paralelos (como Occam y C paralelo) incluyen a los canales como tipos de datos especiales [50,41,66].

ii) Llamadas a procedimientos remotos.

Dentro de la programación distribuida, en la que los procesos se comunican utilizando una red de computadoras, es común que tal comunicación se realice mediante las **llamadas a procedimientos remotos** (o RPC).

Las RPC's fueron introducidas por Brinch-Hansen [13] como un mecanismo de comunicación entre procesos de un sistema distribuido. En este esquema, un programa se forma por un grupo de procesos secuenciales que se definen mediante una designación, y se forman por procedimientos y variables propias, y por procedimientos comunes. Siguiendo este esquema, un proceso puede realizar únicamente dos operaciones: ejecutar sus procedimientos propios, o realizar un requerimiento o llamada externa.

Un proceso se encuentra en ejecución de sus procedimientos locales, excepto cuando se encuentra detenido debido a una operación concurrente o a una llamada de otro proceso. En el caso de un proceso *servidor*, éste se encuentra desocupado hasta recibir un requerimiento de un proceso *cliente* [46,72].

Respecto a la notación, Brinch-Hansen propone la siguiente: un proceso **P** puede llamar a un procedimiento **R** definido en otro proceso **Q.** Esto se describe:

```
call Q.R (...expresiones, variables...)
```

Las RPC's son utilizadas principalmente dentro de los sistemas operativos que incluyen facilidades para programación distribuida y comunicaciones, como es el caso de las versiones comerciales de Unix [46,72].

f) Abrazo mortal (*Deadly embrace o dead-lock*).

Las dificultades de la programación secuencial tradicional se pueden resumir en dos principalmente: asegurar que el programa tiene un inicio y un final, es decir, que en algún momento termine y, por otro lado, que los resultados obtenidos sean congruentes con los resultados esperados, esto es, que sean correctos.

Sin embargo, dentro de la programación concurrente aparecen nuevos problemas, además de los existentes en la programación secuencial. Entre ellos, tal vez el más complejo y difícil de contender es el abrazo mortal o *dead-lock.*

Dijkstra menciona su existencia a partir de la observación de los procesos concurrentes con semáforos, como una situación insegura entre dos o más procesos [23]. Es un estado en el cual todos los procesos esperan un evento que nunca sucederá, como por ejemplo, la situación en la que dos o más procesos podrán continuar solamente si alguno de ellos desaparece.

Dijkstra considera que el problema del abrazo mortal aparece por el hecho de compartir recursos entre procesos y que la solución a este problema no es trivial. Finalmente expone un ejemplo de cómo es posible evitar el abrazo mortal [23].

El abrazo mortal puede verse como un bloqueo entre los procesos de un programa concurrente, en el cual ningún proceso es capaz de continuar. Este problema parece en ambos esquemas de memoria compartida y distribuida, con diferentes características [31].

Se han realizado varios estudios alrededor de este problema y la manera de evitarlo [33,65]. La complicación de determinar su existencia aumenta considerablemente respecto al número de procesos involucrados, así como la cantidad de comunicaciones entre ellos.

Aún cuando no es posible determinarlo precisamente, es posible disminuir la probabilidad de que ocurra, minimizando las comunicaciones entre los procesos concurrentes, facilitando también la tarea de detectar la parte del programa donde ocurre o puede ocurrir.

2. Elementos.

La programación concurrente, paralela y distribuida se basa en un conjunto de elementos que influyen dentro del diseño e implementación de sistemas paralelos, y que además determinan en cierta medida su funcionamiento. Los elementos de la programación paralela en particular se presentan a continuación.

a) Número de procesadores.

Uno de los parámetros a considerar al realizar un programa paralelo es el número de procesadores en los cuales se va a ejecutar. El número de procesadores involucrados es uno de las principales características que reflejan la existencia y capacidad de un sistema paralelo: ya el hecho de ser más de un procesador presupone la creación de procesos paralelos y, dependiendo del número de procesadores, es posible estimar la capacidad de procesamiento del sistema. Sin embargo, esto último no es determinante, ya que la capacidad de procesamiento depende también de otros elementos de la programación paralela.

Como se menciona en el apéndice A, las leyes sobre el desempeño de un sistema paralelo utilizan como parámetros para medir su capacidad a los tiempos de procesamiento y al número de procesadores [26,42].

Actualmente se han realizado varios desarrollos en el campo del *hardware* tendientes en integrar un gran número de procesadores en un solo circuito, minimizando su tamaño [54]. De esta manera, el concepto de número de procesadores se convierte más bien en una cantidad fija contenida en un circuito. Sin embargo, en el desarrollo de este trabajo, se considera a los procesadores como unidades discretas (*transputers*), cuyo número puede variar de acuerdo a la capacidad del sistema.

El número de procesadores se considera como el parámetro determinante en el diseño de un programa paralelo, ya que se relaciona estrechamente con otros elementos, como la topología, granularidad, balance de carga, etc.

b) Topologías de conexión.

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 algún 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** [53]. En un sistema paralelo, los nodos y segmentos corresponden a procesadores y ligas de comunicación respectivamente.

Si una red de comunicación tiene la capacidad de modificarse durante el tiempo de ejecución de un programa, se dice que es una topología **dinámica**. En caso contrario, se trata de una topología **estática**. Dentro de la programación paralela, se utilizan con una mayor frecuencia las topologías de tipo estático [51].

Existen varias topologías estáticas usualmente utilizadas para definir las redes de conexión entre procesadores [51,8,20,61]. De cierta manera, las topologías utilizadas se asemejan mucho a aquéllas usadas para la configuración de redes de computadoras [72]. Entre las más usuales se encuentran la no restringida (*unconstrained*), lineal, anillo, estrella, malla simple y toroidal, árbol e hipercubo (Figura I.7) [51,8,50,20,61]. Obviamente, existen otras topologías, pero pueden considerarse como variantes de las mencionadas [20]. A continuación, se mencionan algunas características de cada arreglo.

i) No restringida (*unconstrained*).

Como su nombre lo indica, la topología no restringida 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 (Figura I.7.a). Aún cuando el esquema es muy simple, generalmente es poco práctica en la realidad, ya que al aumentar el número de nodos y segmentos, el número de conexiones se incrementa rápidamente [20].

Sin embargo, la topología no restringida presenta la ventaja de tener un alto grado de acoplamiento, ya que los segmentos son dedicados exclusivamente a la comunicación entre dos nodos eliminando el problema de que otro nodo entre en conflicto por uso del mismo segmento [20].

Si el número de segmentos entre nodos de la topología no restringida se aumenta hasta lograr que cada nodo pueda comunicarse con todos los demás nodos, resulta una topología totalmente conectada (Figura I.7.b). De hecho, para conectar

totalmente **N** nodos, son necesarios **N(N+1)/2** segmentos [51], lo que provoca que el uso de una topología totalmente conectada sea costosa de diseñar en la realidad.

ii) Lineal.

La topología lineal es la organización más sencilla de una red de comunicación. Consiste en conectar en línea todos los nodos [20]. Cada nodo se conecta como máximo a otros dos nodos, excepto los extremos, que sólo tienen una conexión a otro nodo (Figura I.7.c).

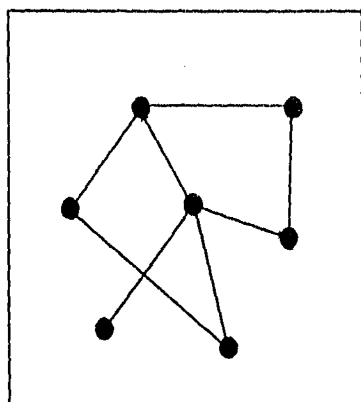
La simplicidad es la mayor ventaja de la topología lineal. Se requiere de **N-1** segmentos para crear una red de **N** nodos. Sin embargo, la comunicación entre nodos no adyacentes representa su mayor desventaja: se estima que en promedio, se requieren de **N/3** "saltos" (*hops*) para enviar un mensaje de un nodo emisor a un receptor. El número de saltos en el ruteo de un mensaje se refiere al número de segmentos que el mensaje debe atravesar desde el origen para alcanzar su destino [51].

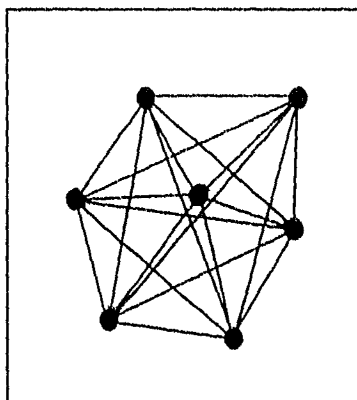
Esta topología es utilizada muy frecuentemente en el diseño de sistemas paralelos con pocos procesadores [51,8,61].

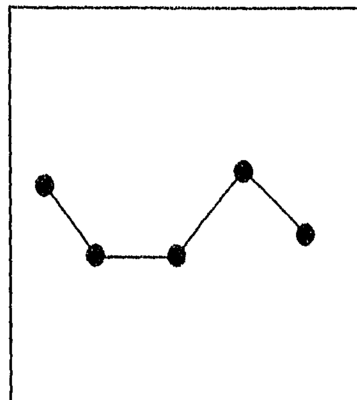
iii) Anillo.

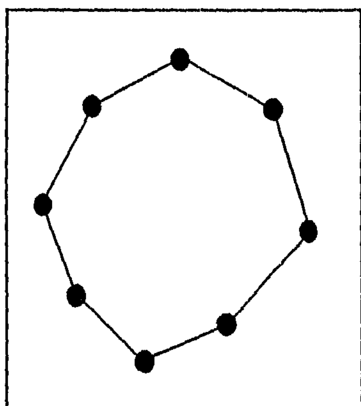
Es posible disminuir el número promedio de saltos de la topología lineal, incrementando su dimensionalidad a un anillo [51]. En un anillo, cada uno de los nodos tiene conexión a otros dos. Cada uno de ellos recibe el nombre de **vecino anterior** y **vecino posterior** [20]. Un nodo recibe información de su vecino anterior o posterior y la pasa a su vecino posterior o anterior respectivamente, de tal manera que la información fluye a través de la red en dos sentidos, pasando toda la información por cada nodo en el anillo (Figura I.7.d).

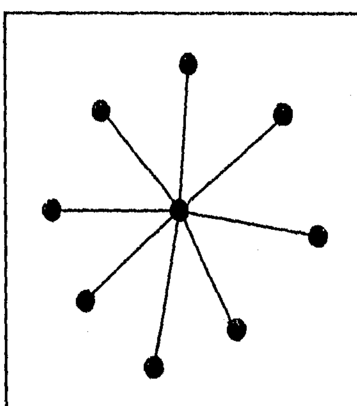
Se ha dicho que un anillo reduce el número de saltos promedio en una red de **N** nodos a un valor aproximado **N/16**, pues cualquier mensaje puede ser dirigido a través del camino más corto, dividiendo en ciertos casos la distancia a la mitad. Además, al aumentar el tamaño de un anillo en un nodo, éste dobla el número de caminos posibles entre cualquier par de nodos que interconecta [51].

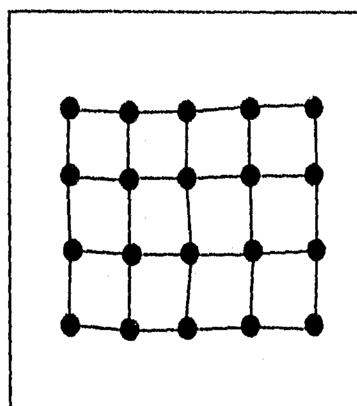

a) No restringida.

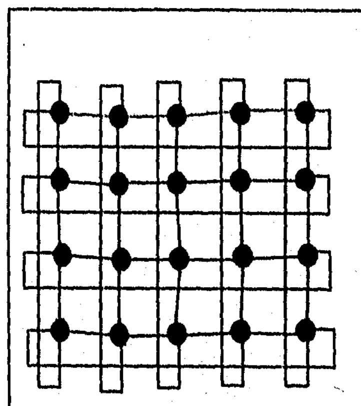

b) Totalmente conectada.

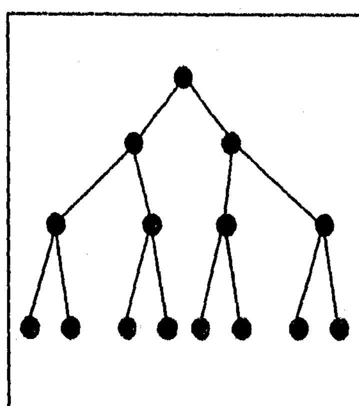

c) Lineal.


d) Anillo.


e) Estrella.


f) Malla.


g) Malla toroidal.


h) Arbol binario.

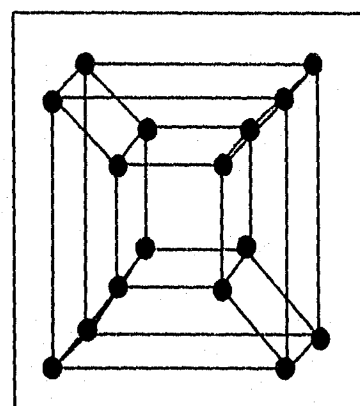

i) Hipercubo.

Figura I.7, Topologías de conexión.

La topología lineal es un ejemplo de una red de conexión unidimensional, mientras que el anillo representa una red de conexión bidimensional. La topología de anillo presenta mayores ventajas en algunas clases de procesos débilmente acoplados, y representa una de las formas más populares de comunicación [8,20,61]. Para aumentar su potencialidad, es posible extenderlo a una tercera dimensión, conectando anillos entre sí [20].

iv) Estrella.

Otro ejemplo de una red bidimensional es la estrella (Figura I.7.e). La topología de estrella emplea un nodo como el elemento de comunicación centralizada entre otros dos nodos conectados a él [20]. De hecho, cada nodo de una red de **N** nodos cuenta únicamente con un canal de comunicación, excepto el nodo central, que cuenta con **N-1** canales, y hacia donde se dirigen todos los nodos para intercambiar información.

La principal ventaja de la topología de estrella es que reduce el tiempo de comunicación en la red, ya que se requiere un total de dos saltos para enviar un mensaje entre dos nodos, sin importar el tamaño de la red. De hecho, requiere un número menor de canales que cualquier topología no restringida [20].

Por otro lado, sin embargo, la topología de estrella tiene la desventaja de depender completamente en el desempeño de su nodo central. Si el nodo central falla, el sistema entero falla. Además, debe ser más veloz que los demás nodos, si se desea que el sistema sea eficiente [20].

v) Malla.

La topología de malla se forma por un arreglo bidimensional de nodos, que se conectan entre sí (Figura I.7.f). En general, se utiliza un arreglo de tipo rectangular, donde cada nodo se conecta a otros cuatro mediante un número igual de canales.

La ventaja de la topología de malla es que aumenta la capacidad de comunicación entre los nodos de la red y a su vez, permite el uso de rutas o caminos alternos de comunicación, esto es, entre dos nodos que se comunican entre sí, existe más de un camino. Sin embargo, el número de saltos requeridos para llegar de un nodo a

otro en una malla de NXM nodos, en el peor caso, es precisamente NXM.

Las desventajas de la malla son, primero, el número de canales de comunicación requeridos: no es posible añadir únicamente un solo nodo, ya que para mantener la estructura de malla, es necesario aumentar toda una fila, ya sea vertical u horizontal y, además, por cada nodo añadido por un lado de la malla, es necesario aumentar el número de canales en tres.

Por otro lado, el número de saltos para la comunicación entre dos nodos cualesquiera de la malla no es constante, es decir, depende directamente de la ruta o camino seguido. Entonces, existe una ruta eficiente con mínimo número de saltos y una cantidad de rutas "alternas" con un número de saltos mayor. Por lo tanto, la ventaja de contar con un número mayor de canales decae considerablemente debido a la cantidad de saltos. Este problema se ha resuelto utilizando otro tipo de conexiones dentro de la malla, generando lo que se conoce como malla toroidal (Figura I.7.g).

Una malla toroidal se forma de manera semejante a un anillo a partir de una topología lineal: partiendo de una malla común, se conectan los canales de comunicación del los lados de la malla entre sí, así como los canales de la parte superior e inferior de la malla. La topología resultante es, por su descripción, de tipo tridimensional.

En la malla toroidal se aumenta el número de caminos entre dos nodos, lo que minimiza el número de saltos para la comunicación. Es posible, entonces, encontrar más de una ruta eficiente para comunicar dos nodos.

La topología de malla es utilizada principalmente en el procesamiento de imágenes, ya que cada componente puede ser responsable de una parte de la imagen que se procesa, interactuando con sus nodos vecinos para resolver las condiciones en las fronteras [64].

vi) Arbol.

Otra topología bidimensional utilizada comúnmente en el diseño de redes de conexión, es la de árbol, y en especial la de árbol binario (Figura I.7.h). En esta última, cualquier par de nodos en el

árbol se comunican entre sí atravesando el árbol binario en busca de un nodo común.

El árbol binario presenta la ventaja de poder realizar varias comunicaciones entre nodos simultáneamente, ya que potencialmente durante la comunicación entre dos nodos, es muy probable que no se requieran todos los canales de la red. La desventaja del árbol binario recae en la situación de falla de cualquier nodo o segmento, lo que puede causar desde la pérdida de un solo nodo, o hasta la mitad de los nodos de la red [20].

La cantidad de saltos requeridos para comunicar dos nodos entre sí en una red de **N** nodos depende de la profundidad del árbol, y en el peor de los casos es de $2\log_2(N+1) - 2$.

vii) Hipercubo.

Es posible observar en los incisos anteriores que una topología mejora considerablemente sus características de comunicación al aumentar su dimensionalidad, esto es, su expresión en una línea, un plano, un espacio o un hiperespacio [51]. Por lo tanto, es fácil concluir que una topología que pueda aumentar su dimensionalidad fácilmente tendrá mayores ventajas de comunicación que otras. Este es el caso del hipercubo (Figura I.7.i).

Un sistema n-dimensional multiproceso conectado como un hipercubo binario se forma por $N=2^n$ nodos. En cada vértice del hipercubo se encuentra un elemento de procesamiento y se conecta exactamente a otros **n** nodos vecinos [20].

Es posible identificar cada nodo de un hipercubo mediante una dirección binaria de **n** *bits*, utilizando el código *Hamming* [51] para numerar los nodos, es decir, la dirección de los nodos vecinos de un nodo cualquiera difieren respecto a la dirección de este último en sólo un *bit*.

La construcción de un hipercubo n-dimensional se realiza recursivamente tomando un hipercubo de n-1 dimensiones, aumentando a cada dirección un bit 0 como más significativo, y conectándolo a otro hipercubo igual cuyas direcciones han sido aumentadas en su bit más significativo con 1. Esto es, que un hipercubo de n dimensiones puede ser subdividido en dos hipercubos

de n-1 dimensiones, y cada uno de éstos a su vez en dos hipercubos de n-2 dimensiones y así consecutivamente [20].

El interés en el hipercubo se fundamenta en que esta topología se ajusta particularmente a un cierto tipo de algoritmos. En especial, aquellos que involucran la evaluación de transformadas rápidas de Fourier o problemas de física que exhiben cierto grado de regularidad [31,20]. Su popularidad ha crecido en los últimos años, a partir del primer hipercubo, construido en Caltech en 1983, conocido como el *Cosmic Cube* y basado en 64 procesadores 8086 [20,61]. Actualmente, varios sistemas multiprocesador se basan en la arquitectura de Caltech [51,8,31,61].

Escoger el esquema de interconexión o topología a utilizar resulta una labor complicada [64]. La meta del diseño de toda red de conexión utilizando un topología determinada es principalmente reducir los costos de *hardware* en cuanto a ligas de conexión y, al mismo tiempo, minimizar el tiempo requerido para enviar un mensaje, reduciendo el número de saltos [51,8,61]. Sin embargo, existe un fuerte compromiso entre el crecimiento de ligas de conexión contra el descenso en el número de saltos, por lo que no se ha determinado una topología que se ajuste a todos problemas. De hecho, la práctica común es considerar, dependiendo de la naturaleza del problema, la topología a utilizar.

Adicionalmente, dentro del problema de escoger una topología en particular, existen otras consideraciones prácticas, como la tecnología del *hardware* existente, en cuanto a la manera de realizar las ligas de comunicación (circuito impreso, cable, con o sin interruptores). Esta consideración incrementa o disminuye la capacidad de configuración, manejabilidad y complejidad de la red. Otra consideración es el tamaño de la red: para una red muy grande, la topología a usar puede ser difícil de determinar. Esto se simplifica si la red tiene una estructura regular. Estas consideraciones deben ser tomadas en cuenta durante el diseño en general de una aplicación paralela [64].

c) Métodos de paralelización. Granularidad y balance de trabajo.

Como se ha mencionado anteriormente [I.A.1.b], un programa paralelo es una colección de procesos que se ejecutan simultáneamente en diferentes procesadores. La manera de repartir los procesos entre los procesadores disponibles debe ser tal que la ejecución del programa se realice en el menor tiempo [51]. Aún cuando los sistemas paralelos son

poderosas máquinas, un sistema multiprocesador cuyos procesadores no se encuentran apropiadamente coordinados puede requerir más tiempo para completar una tarea que un sistema uniprocesador [39].

La principal dificultad para el diseño de un programa paralelo, es transformar la estructura del problema a solucionar, de tal manera que corresponda a la estructura de una computadora paralela [39]. Una programación paralela óptima determina tanto la repartición y ejecución de cada proceso, de tal manera que el tiempo de ejecución sea mínimo [51].

Se han propuesto varios métodos de programación, cada uno de los cuales constituye una manera diferente de expresar una solución paralela a un problema dado. Es importante mencionar que el desempeño de un método de paralelización depende directamente de la topología utilizada, pero ésta no lo determina totalmente. Algunos métodos se expresan más claramente sobre ciertas topologías que otros, como puede ser el caso de el procesamiento de imágenes en una malla, o el cálculo de la tranformada rápida de Fourier en un hipercubo [51].

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 o el *hardware* en el que el programa se ejecute [39]. Dentro del problema de asignación de los procesos a los procesadores, existen dos elementos de la programación paralela que se relacionan entre sí y determinan el nivel de paralelismo utilizado. Estos son la granularidad y el balance de trabajo.

- Granularidad.

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 [51,39].

Una aplicación de granularidad gruesa (*coarse-grained*) puede ser dividida en partes lógicas formadas por procesos de secuencias independientes, con poca sincronización o comunicación. En un esquema de granularidad gruesa, la relación de tiempo de procesamiento entre tiempo de comunicación es alta [39].

Por otro lado, en una aplicación de granularidad fina (*fine-grained*) se realizan una cuantas instrucciones entre las comunicaciones entre

procesadores. En el esquema de granularidad fina, la relación de tiempo de procesamiento entre tiempo de comunicación es baja [39].

La finalidad de la granularidad es determinar la "mejor" forma de dividir el problema en conjuntos de tareas, de tal manera que el tiempo de ejecución de cada conjunto sea mínimo. El tamaño del conjunto puede ser alterado añadiendo o eliminando tareas. A tales conjuntos de tareas, se les conoce como **granos** [51,31].

Si la granularidad es gruesa, el tamaño de los granos es demasiado grande y, por consecuencia, el nivel de paralelismo disminuye ya que algunas tareas que son potencialmente paralelas se han agrupado en granos que se ejecutan secuencialmente en un procesador. En caso contrario, cuando la granularidad es muy fina, el tamaño de los granos es tan pequeño que existe una pérdida de tiempo de ejecución por un aumento en los tiempos de comunicación entre procesadores [51].

No hay una solución general al problema de selección del grano a utilizar. 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 a partir de la arquitectura paralela a ser usada. Un sistema formado por pocos procesadores potentes y unidos entre sí por ligas de comunicación relativamente lentas, se ajusta más a un esquema de granularidad gruesa; en el otro extremo, un sistema con elementos de procesamiento muy pequeños, pero que se comunican relativamente rápido, se presta para un esquema de granularidad fina [39].

- Balance de trabajo.

Se considera que la granularidad de un programa paralelo hace referencia a la división de tareas de un problema en conjuntos de tareas que se ejecutan en un procesador. Por su parte, el balance de trabajo se refiere a la manera de repartir el total de tareas a realizar entre el total de procesadores disponibles [31].

Un balance de trabajo óptimo es aquel que mantiene a cada procesador ocupado y procura que todos los procesadores terminen casi al mismo tiempo. Se dice que un programa está balanceado si su procesamiento se encuentra distribuido igualmente entre todos los procesadores [51].

El balance de trabajo puede realizarse de dos maneras principalmente: estática y dinámicamente [31].

Una aplicación puede ser analizada con el fin de conocer si es posible balancearla, esto es, conocer de antemano la carga de trabajo por procesador, lo que determina estáticamente una distribución de trabajo entre los procesadores durante la compilación. A esto se le conoce como balance de trabajo estático [51].

Si por el contrario, no es posible conocer previamente la carga de trabajo por procesador, el balance de trabajo se puede ajustar dinámicamente durante la ejecución del programa. A esto se conoce como un balance de carga dinámico [51].

Existen varias técnicas para realizar de la mejor forma el balance de trabajo de un programa paralelo. Las técnicas basadas en balance estático pueden ser utilizadas directamente por el programador, mientras que las técnicas de balance dinámico se aplican ya sea mediante un sistema operativo o el uso de algún tipo de *software* durante la ejecución del programa [51,31,64].

La organización de programas alrededor del paralelismo ha sido realizado de varias maneras. Es posible mencionar una gran cantidad de métodos de paralelización que se utilizan para la programación de aplicaciones. La mayoría se puede clasificar en general dentro de las siguientes categorías [29]:

i) Paralelismo algorítmico.

Se basa en la paralelización de tareas cuasi-independientes que ejecutan secciones del algoritmo solución del problema (y por lo tanto, no idénticos), de tal manera que los datos y resultados obtenidos son pasados a través de las tareas como lo indica el algoritmo [29].

El paralelismo algorítmico, también conocido como descomposición en flujo de datos [51], se refiere a la idea de obtener un cierto grado de paralelismo a partir del algoritmo usado para resolver un problema. El paralelismo es introducido considerando la manera como el algoritmo puede ser dividido en tareas separadas cuasi-independientes. Cada tarea puede ser ejecutada en paralelo,

con la comunicación de datos o resultados entre las tareas como sea necesario, es decir, cada tarea realiza algún proceso sobre los datos, pasando su resultado a una tarea subsecuente [29].

El paralelismo algorítmico se relaciona con el modelo de flujo 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. De hecho, la mayoría de los programas secuenciales basados en el modelo funcional pueden interpretarse paralelamente mediante el modelo de flujo de datos [51].

El paralelismo algorítmico se considera como una de las formas más populares de paralelizar una aplicación, ya que parece dirigido naturalmente al paralelismo. Sin embargo, son pocas las aplicaciones que se han desarrollado utilizando este método [51].

ii) Paralelismo geométrico.

Se desarrolla mediante tareas cuasi-independientes idénticas, las cuales procesan una porción de los datos e interactúan con o se ven afectados por las tareas vecinas, de acuerdo a la geometría del problema [29].

El paralelismo geométrico, estructural o de datos [51] se introduce utilizando cualquier geometría espacial regular o estructura presente en un problema. Consiste en dividir los datos del problema de una manera simétrica en unidades cuasi-independientes, considerando una distribución uniforme (geométrica). El procesamiento se realiza sobre cada una de las unidades, siendo cada proceso responsable únicamente de una región espacial de datos. El efecto total se espera como una acción de cada proceso sobre las unidades de datos. La interacción entre unidades vecinas puede incorporarse con el fin de presentar una solución más realista [29].

Se acostumbra utilizar el paralelismo geométrico con un grado de granularidad fino, por lo que es aplicable ampliamente para aquellas máquinas con un gran número de procesadores sencillos, comunicándose mediante redes de alta velocidad. A este tipo de sistemas paralelos se les conoce como **masivamente paralelos** [51].

El método de paralelismo geométrico es una forma simple y natural de resolver problemas con una gran cantidad de datos. Existe una gran variedad de aplicaciones relacionadas con el paralelismo de datos, como por ejemplo, el diseño de circuitos VLSI, la simulación de partículas, la modelación de fluidos, la visión por computadora, el ajuste de secuencias de proteínas, la compilación de información, el aprendizaje de máquinas y las gráficas por computadora, entre otras [51].

iii) Paralelismo *farm*.

Se forma por tareas totalmente independientes e idénticas, que procesan datos en cualquier orden [29].

El paralelismo en *farm*, o supervisor/trabajador, se aplica a problemas cuya solución puede descomponerse en pequeñas partes independientes entre sí. Debido a esto último, cada parte puede ejecutarse aisladamente, dando la solución como el efecto de la acción sumada de todas las partes. Sin embargo, debido a la independencia de los "trabajadores", para obtener la solución global se requiere de un "supervisor" (o *farmer*) que distribuya y controle el trabajo entre los trabajadores [29].

El modelo *farm* de procesos es utilizado para la solución de problemas con una alta relación entre tiempos de procesamiento y comunicación. Es el modelo más popular, debido a la variedad de aplicaciones en las que se utiliza. Estas van desde la implementación de algoritmos para procesamiento de gráficos e imágenes, a problemas numéricos y de simulación [31].

Son varias las características que han hecho de este modelo el más utilizado. Principalmente, el modelo permite el uso de un número arbitrario de procesadores, utilizando uno como controlador y los demás como trabajadores, usualmente conectados en una topología lineal. El código de la aplicación se ejecuta en cada trabajador al mismo tiempo. Se utiliza además un sistema simple de ruteo de mensajes a través de la red, consistente en unas cuantas líneas de código, lo que consume poco tiempo de procesador, y actúa como interface entre la aplicación que está ejecutándose en el procesador y el resto de la red. Además de su característica de sencillez, el sistema de comunicación es altamente independiente de la aplicación, por lo que puede ser usado en una variedad de

aplicaciones [31].

Una característica importante del modelo *farm* es que provee un mecanismo automático de balanceo dinámico de la carga de trabajo, basado en la **demanda de trabajo** por procesador, esto es, que las tareas se pasan a los procesadores para su ejecución según se requieran, con el fin de utilizar todo el potencial de procesamiento del sistema, lo que permite una gran eficiencia de la red de procesadores en comparación con el uso de un solo procesador [31].

d) Métricas de desempeño.

El desempeño de los sistemas paralelos en términos de tiempo ha sido su mayor ventaja. Esto último ha sido motivo de discusión desde su aparición.

En la práctica, se ha demostrado una mejora considerable en la ejecución de un programa paralelamente respecto a su contraparte secuencial. Considerando esto último, se han generado una serie de métricas de desempeño para sistemas paralelos, basados en los tiempos de ejecución de un programa en un número determinado de procesadores. De esta manera, es posible mencionar el propio tiempo de ejecución, el *speedup*, la eficiencia [26] y la fracción serial [42].

i) Tiempos de procesamiento y de comunicación.

La característica más importante que se ha considerado respecto al desempeño de sistemas paralelos es el tiempo de ejecución de un programa. Con un sistema paralelo se pretende que tal tiempo sea disminuido en proporción al tiempo utilizado en un sistema uniprocesador. En base a esta relación y al tiempo de ejecución en sí refleja el desempeño global de un sistema.

El tiempo de ejecución de un programa en un sistema paralelo puede subdividirse en una componente secuencial y una paralela, o respecto a la granularidad, en un tiempo de procesamiento y un tiempo de comunicación [I.A.2.c]. Estos últimos forman la base sobre la que se sustenta la mayoría de las métricas de desempeño de un sistema paralelo.

Una aplicación paralela trivial es aquella que no requiere comunicación entre los procesadores. El tiempo de ejecución de una

aplicación de este tipo tiene un comportamiento completamente lineal [51]. Sin embargo, en la realidad las aplicaciones paralelas no son triviales, pues la comunicación entre procesadores es inevitable en la mayoría de los casos.

Al tiempo de retraso introducido por la comunicación entre procesos se le conoce como **retraso de comunicación** (*communication delay*), y se genera a partir de la espera en alguno de los procesadores al intentar una comunicación [51].

En contraste con el retraso de comunicación, existe el concepto de **tiempo de procesamiento**, es decir, el tiempo efectivo que toma a un procesador realizar las tareas que le son asignadas, independientemente de las comunicaciones.

El desempeño de un sistema paralelo depende entonces de ambos tiempos, encontrándose un punto óptimo en función de la cantidad de procesamiento realizado por procesador y la comunicación entre ellos. Para obtener un reflejo del desempeño de un sistema paralelo, se utilizan una serie de métricas basadas principalmente en el tiempo de procesamiento global y el número de procesadores, como se menciona a continuación.

ii) Speedup.

Es posible categorizar diferentes configuraciones paralelas respecto una aplicación determinada utilizando exclusivamente el tiempo de ejecución como parámetro de comparación. Sin embargo, es importante realizar una evaluación más objetiva del desempeño. Tal medición puede ser el *speedup*.

El *speedup* es una relación del tiempo de ejecución de un programa ejecutándose en un solo procesador sobre el tiempo de ejecución del mismo programa ejecutándose en **n** procesadores. Esta relación se analiza en el apéndice A, puede obtenerse de la siguiente forma:

$$S(n) = \frac{T(1)}{T(n)}$$

donde **n** es el número de procesadores utilizados, **T(1)** es el tiempo

de ejecución en un solo procesador y **T(n)** es el tiempo de ejecución en **n** procesadores [26].

El *speedup* es una medida del desempeño en función únicamente del número de procesadores utilizados en la ejecución de un programa y del tiempo de ejecución. No considera otros factores que influyen sobre el desempeño, como la topología utilizada, el balance de trabajo, la granularidad, etc.

iii) Eficiencia.

Otra medida importante del desempeño de la programación paralela, ligada directamente con el tiempo de ejecución y el *speedup*, es la eficiencia. La medida de la eficiencia de un sistema paralelo se asocia a la idea que *"n trabajadores deben hacer el trabajo en una fracción 1/n del tiempo que le lleva a un solo trabajador"* [51]. La relación matemática que describe esto es:

$$E(n)=\frac{T(1)}{nT(n)}=\frac{S(n)}{n}$$

en donde **n** es el número de procesadores utilizados, **T(1)** es el tiempo de ejecución del programa en un solo procesador, **T(n)** es el tiempo de ejecución para **n** procesadores, y **S(n)** es el *speedup* [26,42].

Es importante notar que el valor de la eficiencia refleja la eficiencia al aprovechar los recursos de *hardware* del sistema [42]. Por lo tanto, su valor siempre es inferior a cero, ya que el tiempo de ejecución de un programa paralelo se ve afectado por el tiempo total de comunicaciones entre los procesadores, excepto en el caso de un sistema paralelo trivial. El valor de la eficiencia para el caso de un solo procesador se considera como la unidad.

iv) Fracción serial.

El tiempo de ejecución, el *speedup* y la eficiencia son métricas útiles que permiten evaluar de una manera sencilla el desempeño de un sistema paralelo. Sin embargo, carecen de cierta información importante, que no puede ser obtenida a partir de ellas. Aumentar el número de procesadores obviamente disminuye el

tiempo de ejecución de un programa, lo que se refleja en cambios del *speedup* y la eficiencia del sistema. Sin embargo, es necesario considerar los otros factores que afectan el desempeño de un sistema paralelo. Para esto se ha creado otra métrica del desempeño de un sistema paralelo, conocido como fracción serial [42].

La fracción serial puede ser deducida a partir de la Ley de Amdahl (que se menciona en el apéndice A), mediante la siguiente expresión:

$$f = \frac{\frac{1}{S} - \frac{1}{n}}{1 - \frac{1}{n}}$$

donde **S** se refiere al *speedup*, y **n** es el número de procesadores involucrados.

La fracción serial es útil al tomar en cuenta la manera en que varía respecto a la influencia de otros factores del paralelismo [42]:

- El balance de carga se refleja en un cambio irregular de la fracción serial al incrementar el número de procesadores.

- La granularidad demasiado fina representa incrementos suaves de la fracción serial al incrementar el número de procesadores.

- En el caso de procesamientos vectoriales, una posible reducción de la longitud de los vectores conduce a un suave incremento en la fracción serial al aumentar el número de procesadores.

B. Características. Ventajas y desventajas.

Como se ha mencionado al principio de este capítulo, existe una atracción primaria a los sistemas paralelos para la solución de problemas que requieren una mejora en los tiempos de ejecución. Tal atracción debe justificase basándose en las características de la programación paralela que permiten mejorar el desempeño. A partir de tales características es posible determinar las ventajas y desventajas que presenta la programación paralela.

1. Características.

La programación de un lenguaje paralelo se caracteriza principalmente por su capacidad de expresar el paralelismo, la secuencialidad, el no determinismo y la sincronización entre procesos.

a) Paralelismo.

Un lenguaje paralelo se describe por la ejecución paralela de procesos, mediante diferentes declaraciones. Su existencia responde a la necesidad de que en los lenguajes de programación secuencial no se cuenta con una construcción que exprese el paralelismo. La necesidad de tales declaraciones ha sido notoria desde los inicios de la programación concurrente [23,35,37].

Como se menciona en el apéndice A, Dijkstra propone una extensión al lenguaje ALGOL60, utilizando una estructura basada en las instrucciones **parbegin ... parend**, formando lo que llamó una composición paralela, la cual implica la iniciación simultánea de las instrucciones que los comprenden. Su ejecución termina cuando todas y cada una de tales instrucciones internas finalizan.

Las consideraciones de Dijkstra han dado como resultado lo que constituye en varios lenguajes paralelos la **instrucción paralela**, la cual representa la activación simultánea de un conjunto de instrucciones que generan procesos disjuntos, los cuales poseen una velocidad de ejecución independiente entre sí (Figura I.8). La instrucción paralela finaliza exitosamente únicamente cuando todos los procesos generados terminan con éxito. De esta manera, el tiempo de procesamiento de la instrucción paralela corresponde al tiempo que tome para terminar al proceso más lento.

Existen varias derivaciones de la instrucción paralela de los lenguajes, como por ejemplo las instrucciones **cobegin...coend**, de Pascal Concurrente [12,13], los procesos **P1||P2||...||Pn**, de CSP [37,38], la instrucción **PAR**, de Occam [41,66,29], etc.

b) Secuencialidad.

El concepto de secuencialidad de instrucciones se presenta como característica básica en la mayoría de los lenguajes de programación, ya que parte directamente del Paradigma de Von Neumann. Sin embargo, dentro de

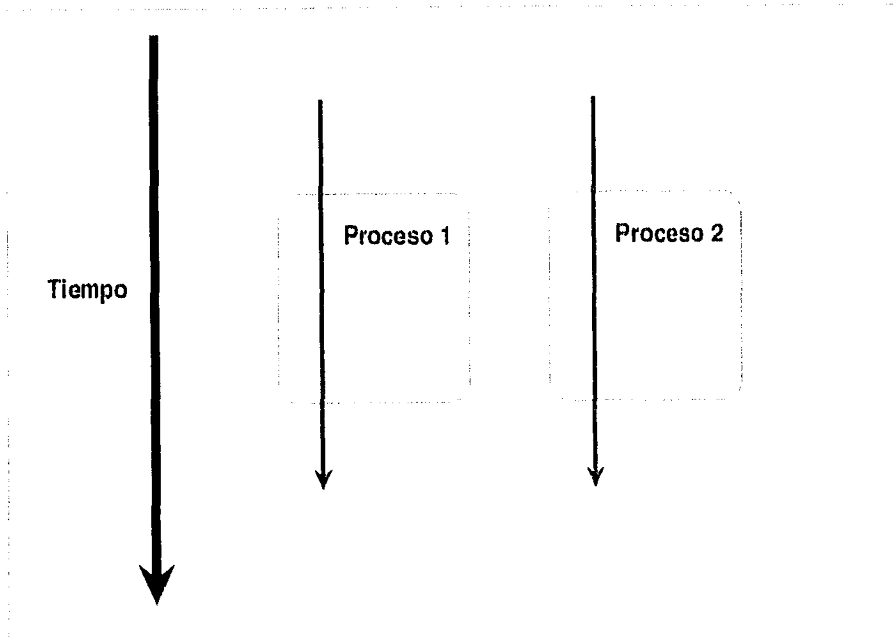


Figura I.8, Paralelismo.

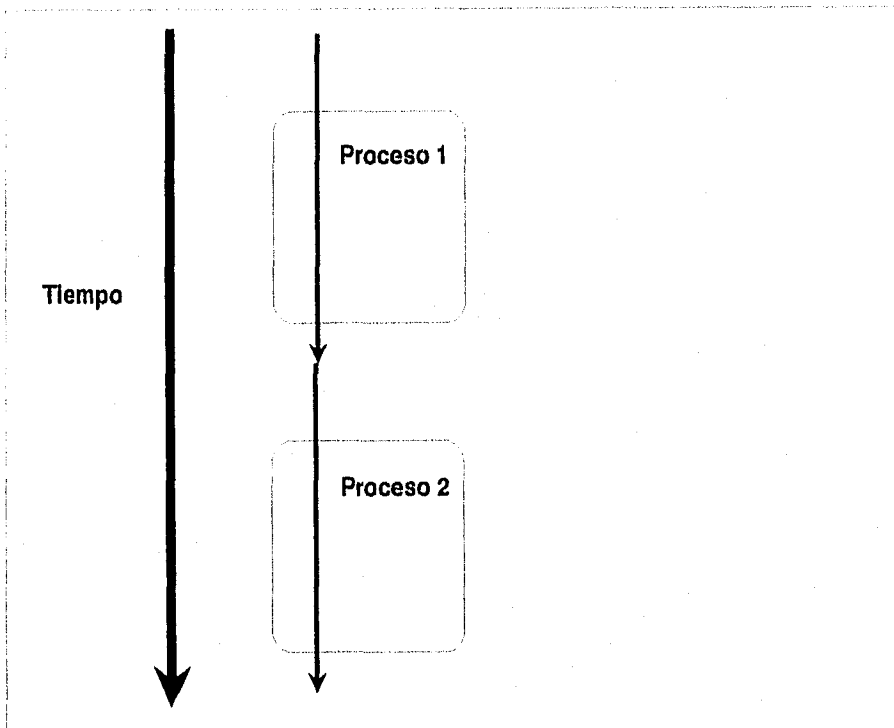


Figura I.9, Secuencialidad.

la programación paralela es necesario representar la secuencialidad de forma expresa, con el fin de contrastar su acción con la ejecución paralela de instrucciones.

La **instrucción secuencial** expresa un conjunto de procesos disjuntos que se activan en secuencia conforme se encuentran dentro de la instrucción (Figura I.9). Finaliza exitosamente si todos los procesos de la secuencia finalizan, en caso contrario queda interrumpida y falla. Debido a que se encuentran en un orden, el tiempo de ejecución de una instrucción secuencial es igual a la suma de los tiempos de ejecución de los procesos que lo forman.

A partir de los primeros lenguajes, la secuencialidad de un conjunto de instrucciones se ha expresado mediante la inclusión del símbolo ; entre las instrucciones de la secuencia. Lenguajes como ALGOL60 [23], Pascal [80], C y otros cuentan con tal característica, la cual ha sido utilizada también en algunos lenguajes paralelos, como por ejemplo, la instrucción **P1;P2;...;Pn**, de Pascal concurrente[12,13] y CSP [37,38]. Otros lenguajes, como Occam [41,66,29], introducen específicamente un constructor secuencial **SEQ**.

c) Comunicación y sincronización.

Como se menciona anteriormente [I.A.1.d,I.A.1.e], existen varios mecanismos de comunicación y sincronización entre procesos, basados en los esquemas de memoria compartida y memoria distribuida. La comunicación y sincronización de procesos paralelos en un esquema de memoria distribuida por paso de mensajes, se basa a su vez en un par de instrucciones de envío y recepción de mensajes [37,38], con las siguientes características en la comunicación entre dos procesos emisor (**P1**) y receptor (**P2**) (Figura I.10):

- La instrucción de envío en **P1** especifica de alguna manera como destino al proceso **P2**.
- La instrucción de recepción en **P2** especifica de alguna manera como fuente al proceso **P1**.
- El tipo de dato de la instrucción de recepción debe ser el mismo que el tipo de dato de la instrucción de envío.

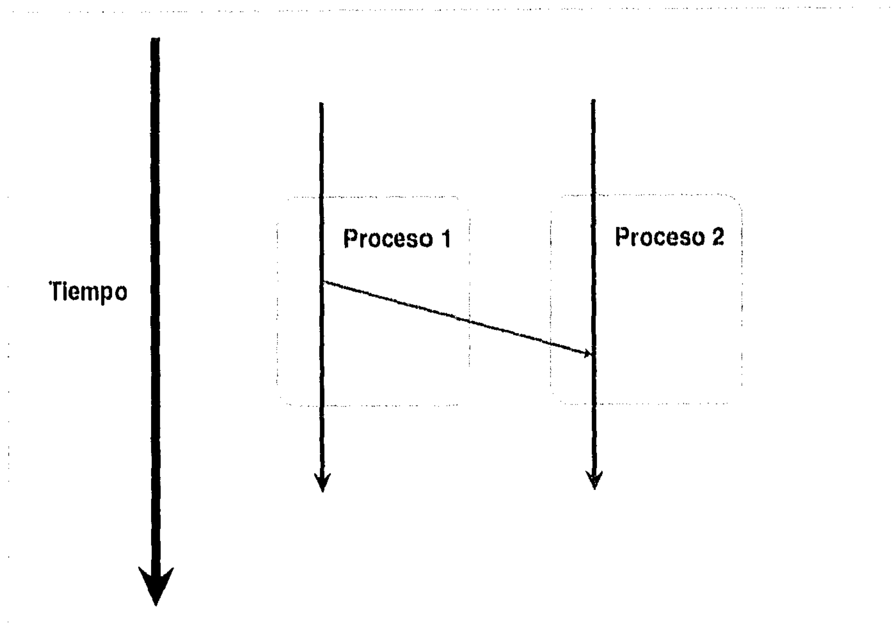


Figura I.10, Comunicación.

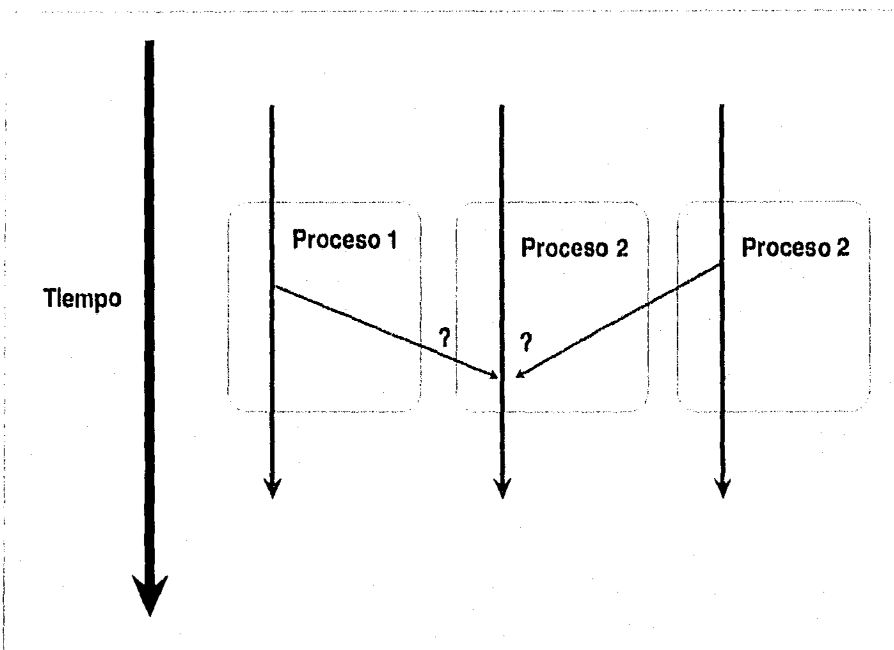


Figura I.11, No determinismo.

- La instrucción de envío o recepción en un proceso permanecerá bloqueada hasta que su contraparte en otro proceso se ejecute con éxito, lo que establece la comunicación, esto es, las instrucciones de envío y recepción se corresponden.

- Una vez completada la comunicación, los procesos continúan su ejecución independiente.

- En el caso de que finalice el programa paralelo, y un proceso no haya realizado con éxito una instrucción de envío o recepción, la instrucción falla.

Ejemplos de instrucciones de envío y recepción son las siguientes:

En CSP [37,38]:

P1 ! P2 (instrucción de envío)
P2 ? P1 (instrucción de recepción).

En Occam (utilizando un canal **C**) [41,66,29]:

C ! P1 (instrucción de envío)
C ? P2 (instrucción de recepción).

d) No determinismo.

Dentro de la programación paralela, un proceso tiene la capacidad de intercambiar mensajes con otro u otros procesos a lo largo de la ejecución de un programa. El orden en que es posible enviar o recibir tales mensajes es totalmente arbitrario, lo que no permite conocer previamente la manera en que los procesos se comunican entre sí. Esto se conoce comúnmente como la característica de no determinismo de un programa paralelo (Figura I.11).

Sin embargo, en ocasiones no es permisible recibir los mensajes en un orden totalmente aleatorio. Es necesario que antes se verifiquen ciertas condiciones para aceptar el mensaje [35]. Esto es posible lograrlo utilizando una expresión lógica o *Booleana* que condicione la ejecución de algún proceso en particular. De esta manera se introduce una **instrucción alternativa**, la cual se especifica mediante un conjunto de **procesos custodiados**. A las expresiones lógicas que controlan la ejecución de los

procesos custodiados, y a la instrucción de recepción del mensaje, se les conoce como **custodias** [38].

En una instrucción alternativa, todas las custodias se evalúan al mismo tiempo, ejecutándose sólo aquel proceso asociado a la custodia que tenga éxito, es decir, la expresión lógica que la define es verdadera y la recepción del mensaje tiene éxito. En el caso en que más de una custodia sea verdadera, la instrucción selecciona arbitrariamente al proceso asociado de una de esas custodias. Por otro lado, la instrucción alternativa permanece bloqueada esperando la recepción de un mensaje, hasta que se verifique alguna de sus custodias. Si todas las custodias fallan, la instrucción alternativa falla [38].

Ejemplos de la instrucción alternativa son la instrucción ***[C1→P1[]C2→P2[]...[]Cn→Pn]** de CSP[37,38], y la instrucción **ALT** de Occam [41,66,29].

2. Ventajas.

En base a sus objetivos y características, la programación paralela presenta una serie de ventajas respecto a la programación secuencial. Algunas de estas ventajas son:

a) Aumento en la velocidad.

Como se ha dicho anteriormente, la mayor ventaja que los sistemas paralelos presentan es el incremento en la velocidad de procesamiento de un programa, disminuyendo de los tiempos de ejecución mediante el uso de varias unidades de procesamiento que cooperan entre sí.

Es el aumento en la velocidad de ejecución la ventaja más importante que actualmente representan los sistemas paralelos, debido a que gran parte de las aplicaciones en cómputo presentan el problema de altos tiempos de respuesta. En especial, aquellas aplicaciones asociadas con el cómputo de gran cantidad de datos (también conocido como denso o pesado) y de tiempo real tienen en el procesamiento paralelo una solución.

b) Eficiencia en el procesamiento.

Igualmente como se menciona anteriormente, otra ventaja de los sistemas paralelos se refiere a la eficiencia en el procesamiento. Esto es que debido a su velocidad de funcionamiento, un procesador es capaz de

realizar varias acciones concurrentemente, es decir, ocupar aquellos tiempos de inactividad del procesador para realizar otra actividad necesaria. De esta manera, el tiempo de procesamiento es utilizado de una manera más eficiente, al aumentar la eficiencia del procesamiento mediante concurrencia.

c) Simplicidad en la expresión de modelos.

En general, un programa es un modelo matemático de una parte de la realidad. Sin embargo, la mayoría de los fenómenos reales se componen de elementos que actúan en el tiempo simultáneamente, comunicándose entre sí. La programación paralela permite una expresión más clara de los modelos, dando una característica espacial y temporal a sus elementos.

La programación paralela provee de un marco conceptual simple, en el cual el programador puede representar un problema y obtener una solución expresada de una manera más natural [35].

d) Aplicaciones.

La programación paralela tiene una gran variedad de aplicaciones en las que se aprovecha el *hardware*, conjugando varios procesos simultáneos entre sí.

La programación paralela tiene aplicaciones en áreas como la implementación de sistemas operativos, sistemas en tiempo real, simulación, programación heurística, procesamiento masivo, etc. [8,35].

Las computadoras convencionales se desarrollan cada vez más y más considerando un cierto paralelismo. Aún más, los fabricantes de supercomputadoras también están incluyendo características paralelas a sus productos. Actualmente, la apertura de los sistemas a arquitecturas paralelas cada vez es más común [8,52].

e) Tamaño y costo.

En un futuro, las arquitecturas paralelas se presentan como una solución viable al problema del incremento de la densidad computacional, esto es, de un *software* más extenso y complejo. Esto permite una mayor capacidad en un espacio más pequeño y a un costo menor por unidad de procesamiento [52].

La esperanza de un *hardware* de procesamiento más veloz, más barato y más pequeño es una de las bases del desarrollo actual de la tecnología computacional. Es posible que el uso de sistemas multiprocesador llegue a ser ampliamente aceptado en el mundo práctico comercial de la computación. Al llegar al extremo del desempeño de computadoras uniprocesador, los sistemas multiprocesador consistentes de 4, 8, 16 o más procesadores serán adicionados a sistemas personales, representando en un espacio menor y a un costo bajo, la capacidad de soportar una mayor densidad computacional [52].

3. Desventajas.

Así como es posible mencionar las ventajas que se pueden obtener de la programación paralela, también es importante considerar las desventajas derivadas del paralelismo, algunas de las cuales han contribuido para inhibir el uso de sistemas paralelos, y de las que se pueden mencionar las siguientes:

a) Responsabilidades del programador.

Al realizar un programa secuencial, las responsabilidades del programador se pueden resumir en dos aspectos principalmente: que el programa sea correcto, esto es, que entregue resultados correctos, y además, que su ejecución termine en algún momento.

En el caso de un programa paralelo, a los aspectos de corrección y terminación se añaden otros, como la eficiencia en el proceso, el método de paralelización a utilizar, el balance de trabajo, la granularidad, etc., para los cuales, el programador no cuenta con alguna referencia precisa, ya que tales aspectos del paralelismo varían entre aplicaciones. Esto significa que los problemas de desarrollo de *software* tienden a complicarse aún más en el caso de los sistemas paralelos.

b) Seguridad contra errores.

Realizar un programa paralelo formado por procesos cooperativos involucra introducir uno o varios de los mecanismos de comunicación y sincronización [I.A.1.d,I.A.1.e], desarrollados como elementos del lenguaje básicos y de bajo nivel, lo que simplifica su uso en programas relativamente cortos. Sin embargo, en el caso de una programación extensa, el uso de los mecanismos de comunicación y sincronización entre procesos puede complicarse hasta resultar poco comprensible.

Por su naturaleza, los programas paralelos son propensos a presentar errores en la comunicación que dependen del tiempo, los cuales no pueden ser detectados fácilmente, aún durante varias ejecuciones de prueba [35].

c) Eficiencia en la programación.

La programación paralela puede mejorar notablemente el desempeño de un sistema. Sin embargo, el añadir características paralelas a un lenguaje o programa produce en general una carga extra en el tiempo de ejecución o en el tamaño del programa por procesador [35], por lo que las mejoras obtenidas al incluir el paralelismo se compensan hasta cierto punto. Lograr una alta eficiencia en la programación es también una responsabilidad del programador [42].

d) Problema de *hardware*.

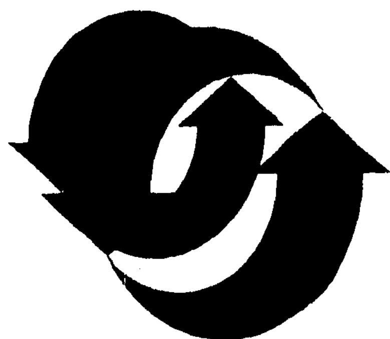
Aún cuando los costos de producción del *hardware* han disminuido notablemente en los últimos años, es evidente que cualquier máquina o sistema paralelo consistente en varias unidades de proceso y memoria, posee un mayor número de elementos y tiene un costo mayor que un sistema simple uniprocesador. Además, más procesadores no necesariamente igualan un desempeño óptimo [8]. Dependen para ello del *software* y de otros elementos de comunicación que incrementan aún más su costo.

e) Problema de *software*.

La mayor dificultad para el uso de los sistemas paralelos recae en el *software*. Los algoritmos existentes no han sido diseñados para el paralelismo, y aún nuevos algoritmos no son fáciles de paralelizar [8]. En general, cualquier algoritmo programado es susceptible a tener un cierto grado de paralelización por alguno de los métodos, es decir, algunas tareas o datos pueden ser tomados como granos del proceso, a fin de paralelizar una tarea [29]. Es común que al observar el desarrollo de una labor secuencial, un programador pueda detectar cuáles tareas o datos pueden ser paralelizados en un programa. Sin embargo, existen algunas tareas o datos que aunque parezca que pueden ejecutarse o procesarse simultáneamente, no lo son en la realidad [8].

El diseño del *software* paralelo es un problema grave, pues una mejora en el desempeño total de un sistema paralelo depende mayormente de los algoritmos utilizados que del *hardware*. Es verdad que los

diseñadores de sistemas de procesamiento paralelo han creado herramientas de *software* para ayudar a los programadores a tratar con las características únicas de sus productos y para producir código paralelo. Como el *hardware*, este *software* varía en su desarrollo y habilidad para realizar su función [8].



Capítulo II. De la Orientación a Objetos.

"Omitamos ahora todas las cosas Abstractas, de manera que cualquier Término se entienda sólo de las Concretas, ya sean substancias, como el Yo, ya sean fenómenos, como el arco iris".

Gottfried Wilhelm Leibniz.

Este capítulo introduce algunas definiciones y elementos relevantes del modelo orientado a objetos, presenta algunas de sus características más relevantes y plantea las ventajas y desventajas que presenta este modelo.

A. Definiciones y elementos.

En los últimos años, la programación orientada a objetos ha generado un cambio revolucionario en la manera de solucionar la mayoría de los problemas computacionales. Sin duda alguna, la orientación a objetos es en la actualidad el cambio más significativo que ha ocurrido en el campo del *software* [21]. Sin embargo, tal cambio no ha surgido abruptamente, es el resultado del desarrollo de varios trabajos y estudios en el área de la programación, que a través de los años se ha presentado en una diversidad de lenguajes [21,9]. Las contribuciones de cada uno de estos han dado como resultado lo que se conoce actualmente como *tecnología orientada a objetos*.

En esta sección se presenta un resumen de algunas definiciones importantes partiendo de las ideas de varios autores sobre el tema de la programación orientada a

objetos. En seguida, se introducen los elementos de la programación orientada a objetos.

1. Definiciones.

a) Objeto, Clase y Programación Orientada a Objetos.

Los conceptos de objeto y clase son fundamentales dentro de la programación orientada a objetos. Existen en la actualidad una diversidad de definiciones de ambos conceptos. Para la mayoría de los autores, definir el concepto de objeto consiste en una percepción primaria. Así por ejemplo, se considera *"...desde una perspectiva de cognición humana, un objeto es cualquiera de los siguientes:*

- *Una cosa tangible y/o visible.*
- *Algo que puede ser aprendido intelectualmente.*
- *Algo hacia lo que un pensamiento o acción es dirigido"* [9].

Inicialmente, el término objeto en Simula 67 era aplicado a un modelo del mundo real, justificando su existencia espacial y temporal, y con el fin de simular algún aspecto de la realidad [9,60].

Una mejor aproximación es considerar que *"Un objeto es una persona, un lugar o cosa. Dentro de un desarrollo orientado a objetos, un objeto representa lo que un sistema necesita saber y hacer con respecto a una persona, lugar o cosa real"* [21]. Esta definición aún considera al objeto como un modelo de la realidad, confiriéndole las capacidades de discernir y actuar como el objeto real al que representa.

Por otro lado, es importante considerar la característica dinámica del objeto, esto es *"objeto es una noción dinámica en tiempo de ejecución; cualquier objeto es una instancia de una cierta clase, creada en tiempo de ejecución..."* [55].

Otra aproximación consiste en considerar al objeto en el contexto de una computadora, es decir, *"un objeto es una región de memoria. Un objeto con nombre tiene una clase de almacenamiento que determina su tiempo de vida. El significado de los valores en un objeto se determina por el tipo de expresión utilizada para accederlo"* [73,74].

Para efectos de este trabajo, se considera que:

Un objeto es un modelo en programación de una parte de la realidad, generado dinámicamente en tiempo de ejecución dentro de un espacio o región de memoria, como una instancia de una clase en particular. Un objeto posee como características inherentes identidad, estado y comportamiento como un modelo del mundo real.

Así, un objeto es como una "pequeña" computadora completa con una memoria (estado) y un conjunto de instrucciones propios (métodos). La memoria se modifica mediante las instrucciones, que definen el comportamiento de la pequeña computadora (Figura II.1) [79]. De hecho, una computadora es un objeto que consiste en un estado en forma de memoria y un funcionamiento en forma de instrucciones.

En la definición anterior se menciona el término de clase a partir de la cual se obtiene un objeto como instancia. Debido a la cercana relación entre el concepto de clase y objetos, es necesario a continuación considerar la siguiente definición de clase (A diferencia del objeto, el concepto de clase comparte semejanzas entre varios autores). Así por ejemplo, tenemos que *"una clase es una agrupación de objetos basada en características comunes. En un desarrollo orientado a objetos, una clase define esas cosas que cada uno de sus objetos sabe y hace"* [21].

Por otro lado, se considera que *"Los sistemas orientados a objetos se construyen como colecciones de clases. Cada clase representa una implementación particular en tipos de datos abstractos o un grupo de implementaciones. Las clases deben ser diseñadas para ser tan generales y reusables como sea posible"* [55]. De acuerdo con esta definición, una clase es una implementación en *tipos de datos abstractos*. Esto últimos se refiere a una descripción matemática formal, definida en base a funciones, precondiciones y axiomas, que generan una descripción no ambigua.

Una tercera aproximación considera que *"Las clases en un programa representan los conceptos fundamentales de una aplicación y, en particular, los conceptos fundamentales de la realidad que está siendo modelada. Una clase es un tipo de dato definido por el usuario"* [73,74].

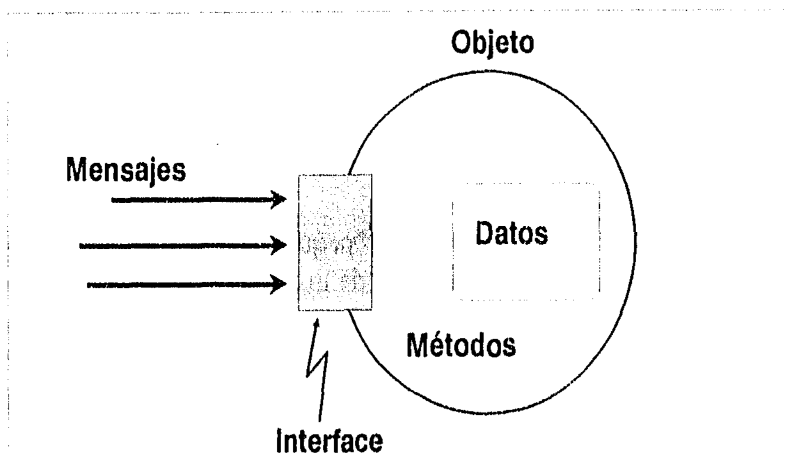


Figura II.1, Anatomía de un objeto.

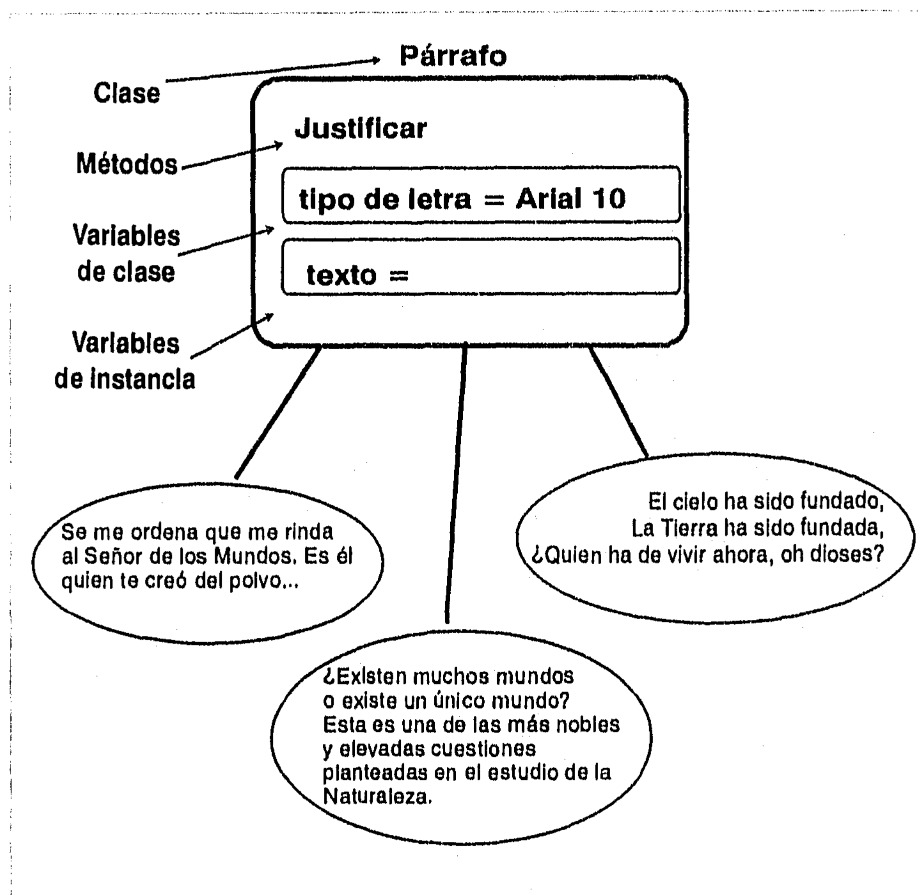


Figura II.2, Clase Párrafo con tres instancias.

De esta manera, y para efectos de este trabajo, se elabora la siguiente definición (Figura II.2):

Una clase es un programa que agrupa, describe y determina características y comportamiento comunes de un grupo de objetos, como una implementación en tipos de datos abstractos, y de acuerdo con una aplicación.

La noción más fundamental de la programación orientada a objetos es que el programa es un modelo de algunos aspectos de la realidad. La programación orientada a objetos utiliza a los objetos como sus elementos lógicos fundamentales; cada objeto es una instancia de alguna clase; la relación de clases entre sí se determina mediante una estructura jerárquica basada en herencia. Estas características de la programación orientada a objetos se derivan de la propuesta de Cardelli y Wegner: *"... un lenguaje es orientado a objetos si y sólo si satisface los siguientes requerimientos:*

- *Soporta los objetos como una abstracción de datos con una interface de operaciones y un estado local oculto.*

- *Los objetos tienen asociado un tipo (o clase).*

- *Los tipos (o clases) pueden heredar atributos de otros supertipos (o superclases)"* [73].

Si un programa no contempla cualquiera de los tres elementos considerados, entonces no es un programa orientado a objetos. Se consideran algunas otras clasificaciones para tales programas, como son **programación con tipos de datos abstractos**, en el caso de un programa que no soporta la herencia, o **programación basada en objetos**, si no provee un soporte directo a la herencia.

Finalmente, dentro del presente trabajo se considera la siguiente definición:

La programación orientada a objetos es un método de implementación en el cual los programas se organizan como colecciones cooperativas de objetos, cada uno representando una instancia de alguna clase, las que pertenecen a una jerarquía de clases, unidas entre sí mediante relaciones de herencia.

b) Genericidad.

La genericidad se refiere a un conjunto de características semejantes que comparten varias clases en general. Así una clase genérica o parametrizada es aquella que sirve como patrón para otras clases. Puede utilizar como parámetros otras clases, objetos u operaciones [9]. Como una característica particular, los parámetros de las clases genéricas deben estar definidos antes de que cualquier objeto de la clase sea creado.

En la mayoría de los casos, las clases genéricas se presentan como un *template* (o clase patrón), que es una especificación de cómo es la forma de un grupo de clases.

El concepto de genericidad aparece en varios lenguajes de programación, tanto orientado a objetos como basados en objetos. Como ejemplos, se tienen los lenguajes Eiffel, C++, Clu, Ada y otros.

c) Polimorfismo y sobrecarga de funciones.

Dos características importantes dentro de la mayoría de los lenguajes orientados y basados en objetos es la definición de funciones utilizando polimorfismo o sobrecarga.

El polimorfismo es un concepto de la teoría de tipos, en el que la declaración de un nombre determinado puede denotar instancias de varias clases diferentes siempre y cuando se encuentren relacionadas mediante una superclase común. Cualquier objeto que se denote por tal nombre puede responder a un cierto conjunto de requerimientos u operaciones de diferentes maneras [9,55].

En contraste, un lenguaje se determina como monomórfico si cada valor o variable puede ser interpretado mediante un y sólo un tipo. Por su parte, en los lenguajes que dan soporte al polimorfismo, los valores o variables pueden tener al mismo tiempo más de un tipo [9].

Una forma especial de polimorfismo se expresa sobre las funciones definidas como sobrecargadas. La sobrecarga (*overload*) de funciones se expresa como la designación de un solo nombre que puede hacer referencia a diferentes funciones en un mismo entorno [73,74]. Esto significa que un nombre puede especificar varias operaciones que comparten características semejantes en su comportamiento, pero que en realidad su implementación es diferente (Figura II.3). En el momento en que el nombre es usado, la

función correcta es seleccionada mediante comparación de tipos de los argumentos actuales con los tipos de los argumentos formales.

El polimorfismo y sobrecarga de funciones aportan características de mayor expresividad y flexibilidad a la programación, permitiendo una expresión más familiar de funciones en el programa.

Solución sin polimorfismo

```
void mostrar(obj){
struct objeto *obj;
switch(obj->TipoObjeto)
      {case PUNTO:
            mostrarPunto(obj);
            break;
      case CIRCULO:
            mostrarCirculo(obj);
            break;
      default:
            printerror("Tipo de objeto incorrecto");
      }
};
```

Solución con polimorfismo

```
void mostrar(obj) {
struct objeto *obj;
obj.mostrar
};
```

Figura II.3, Las ventajas del polimorfismo.

d) Ligadura estática y dinámica.

Una característica importante de los lenguajes de programación, es el concepto de ligadura de tipos en tiempos de compilación y ejecución.

La ligadura estática o temprana de tipos se refiere al momento en el que las variables se relacionan con un tipo de datos, es decir, los tipos de todas las variables y expresiones se fijan en tiempo de compilación. Dentro de la programación orientada a objetos, la ligadura estática denota la asociación de un nombre con una clase, en el momento en que el nombre es declarado durante tiempo de compilación, y antes de la creación del objeto que el nombre designa [9,55].

Por otra parte, se tiene que en un lenguaje con ligadura dinámica o tardía, el tipo de las variables y expresiones de un programa no se conocen hasta que se encuentra en tiempo de ejecución. En programación orientada a objetos, la ligadura dinámica denota la asociación de un nombre con una clase, pero durante tiempo de ejecución, es decir, en el momento en que el objeto es creado [9,55].

e) Métricas de la orientación a objetos.

Como se menciona anteriormente, la orientación a objetos propone una nueva forma de realizar la programación de sistemas, basándose principalmente en métodos de análisis y diseño orientados a objetos. Sin embargo, realizar inicialmente un análisis en objetos, y posteriormente el diseño en clases de abstracciones, no es una tarea sencilla. Requiere en gran medida de obtener las abstracciones del sistema a modelar de manera que las clases y objetos representen un modelo válido.

Es necesario observar que el análisis y diseño orientados a objetos son en realidad procesos iterativos e incrementales. Siempre es posible obtener una mejor modelado de un sistema a partir de un modelo anterior. Sin embargo, aún cuando es posible continuar indefinidamente redefiniendo y mejorando el modelo, es necesario saber en qué momento se considera como válido. Para esto, es necesario realizar alguna clase de medición sobre las abstracciones. Así, se proponen las siguientes características de las abstracciones:

- Acoplamiento.

El acoplamiento es una medida de la asociación que se establece al conectar dos módulos entre sí. Un acoplamiento fuerte tiende a complicar el diseño, pues es difícil de entender, cambiar o corregir si un módulo está fuertemente relacionado con otros. En cambio, en el caso de un acoplamiento débil, es posible tener un mayor control sobre la complejidad del sistema [9].

Como una característica especial de la programación orientada a objetos, se espera que los módulos sean débilmente acoplados, como se menciona más adelante en este mismo capítulo [II.A.2.c].

- Cohesión.

La cohesión mide el grado de conectividad entre elementos de un módulo en particular. La cohesión baja o coincidental refleja que dentro de un módulo se incluyen una serie de abstracciones que en realidad no se relacionan entre sí. Por otro lado, en la cohesión alta o funcional los elementos de una clase o módulo se relacionan entre sí de tal manera que trabajan juntos para representar un comportamiento bien determinado [9].

De manera semejante que el acoplamiento, la cohesión entre módulos o clases es deseable que sea lo más funcional posible. Esta característica se menciona también más adelante en el presente capítulo [II.A.2.c].

- Suficiencia.

La suficiencia significa que un módulo o clase captura suficientes características de la abstracción para permitir una interacción significativa y eficiente entre los módulos. De otra manera, resultan componentes inútiles o redundantes [9]. Es deseable que las abstracciones que forman un módulo sean suficientes en sí mismas, es decir, que dependan más de las características propias del módulo, y menos de las de otros módulos. De esta manera, es posible relacionar la suficiencia con acoplamiento y cohesión.

- Completitud.

La completitud significa que la interface de un módulo o clase representa las características significativas de las abstracciones [9]. Por lo tanto, una clase completa es aquella cuya interface es lo suficientemente general para permitirle ser usada comúnmente por otras clases clientes. Sin embargo, la completitud de una clase o módulo es sumamente subjetiva, ya que es posible proveer una inteface mucho mayor que la que realmente se requiere.

- Primitividad.

La primitividad de un módulo o clase representa el grado de complejidad de las operaciones definidas en el módulo. Se consideran *operaciones primitivas* a aquellas que pueden ser eficientemente implementadas utilizando características inherentes a la abstracción [9].

En la práctica, se considera que los conceptos de acoplamiento y cohesión son el criterio básico para la obtención de abstracciones suficientes, completas y primitivas.

Sin embargo, además de considerar los factores que representan una medida de la calidad del análisis y diseño orientados a objetos en cuanto a las abstracciones, también es importante considerar algunas otras métricas que reflejan la calidad de un programa como resultado del análisis y diseño.

Algunas métricas aplicables a los sistemas orientados a objetos son [9]:

- Métodos ponderados por clase.

  Este método permite medir de una manera relativa la complejidad de una clase en particular. Si además se considera que todos los métodos de la clase son igualmente complejos, también puede tomarse como una medición del número de métodos por clase. En general, una clase con una cantidad de métodos mayor al número de sus posibles llamadas o puertos tiende a ser más específico respecto a alguna aplicación y comúnmente tiende a contener un mayor número de errores.

- Profundidad del árbol de herencia. Número de hijos.

  La profundidad del árbol de herencia y el número de hijos son medidas de la forma y el tamaño de una estructura de clases. Un sistema orientado a objetos correctamente estructurado tiende a ser más bien como un bosque de árboles de clases, en lugar de una sola gran estructura arborecente. En general, las estructuras balanceadas y con una profundidad y anchura aproximada de 2 a 7 clases se consideran de manera empírica correctas.

- Acoplamiento entre objetos.

  Se refiere a una medición en la conectividad con otros objetos, es decir, la capacidad de comunicación de un objeto respecto a otros con los que intercambia información. Como se menciona anteriormente en esta sección, y siguiendo un esquema tradicional, es deseable el diseño de objetos débilmente acoplados, los cuales tienen un alto potencial de reuso.

- Cohesión en métodos.

  Se trata de una medida de la unidad de la abstracción de la clase. Una clase con poca o baja cohesión entre sus métodos sugiere una abstracción inapropiada, ya sea que deba ser fragmentada en más de una clase o que sus responsabilidades sean delegadas a otras clases existentes.

2. Elementos.

La programación orientada a objetos se basa principalmente en una serie de elementos que en su conjunto se conoce como el modelo de objetos. Contiene en su mayoría varios principios de la programación tradicional, conjuntándolos y relacionándolos entre sí. De esta manera, se cuentan con la abstracción, encapsulación, modularidad y jerarquía como elementos "mayores" de la orientación a objetos, considerando que la falta de alguno de ellos genera un modelo no orientado a objetos. Por otro lado, se tienen la tipificación, concurrencia y persistencia como elementos "menores" del modelo de objetos, ya que su presencia no se considera esencial dentro del modelo [9].

a) Abstracción.

"La abstracción surge de un reconocimiento de similaridades entre ciertos objetos, situaciones o procesos en el mundo real, y de la decisión de concentrarse en estas similaridades e ignorar por ese tiempo las diferencias" [9]. Con estas palabras, Hoare considera a la abstracción como un proceso mediante el cual, cualquier ser humano puede observar una característica o comportamiento semejante entre entidades que en la realidad son diferentes.

Dentro de la Programación Orientada a Objetos, cada objeto posee una identidad, un estado y un comportamiento [21,9,55]. Específicamente, el estado es un modelado de las propiedades de un objeto real y considera los valores de tales propiedades, mientras que el comportamiento modela la manera como el objeto real actúa y reacciona a diferentes situaciones. Sin embargo, es extremadamente complejo considerar un objeto como un modelo exacto de las propiedades y acciones. Decidir el conjunto correcto de propiedades y acciones del objeto real que nos interesa modelar es obtener una abstracción del objeto real, es decir, la función de la abstracción es obtener las propiedades y acciones de la realidad que se consideran importantes para el modelo, y eliminar aquellas que no son preponderantes. La abstracción delimita las características de un objeto que lo distinguen de otros objetos, siempre desde el punto de vista de quien realiza la abstracción.

Sin embargo, el proceso de abstracción de objetos a partir de la información obtenida de un problema particular no es sencillo. Se han propuesto algunos métodos para tratar con esta dificultad [21,9].

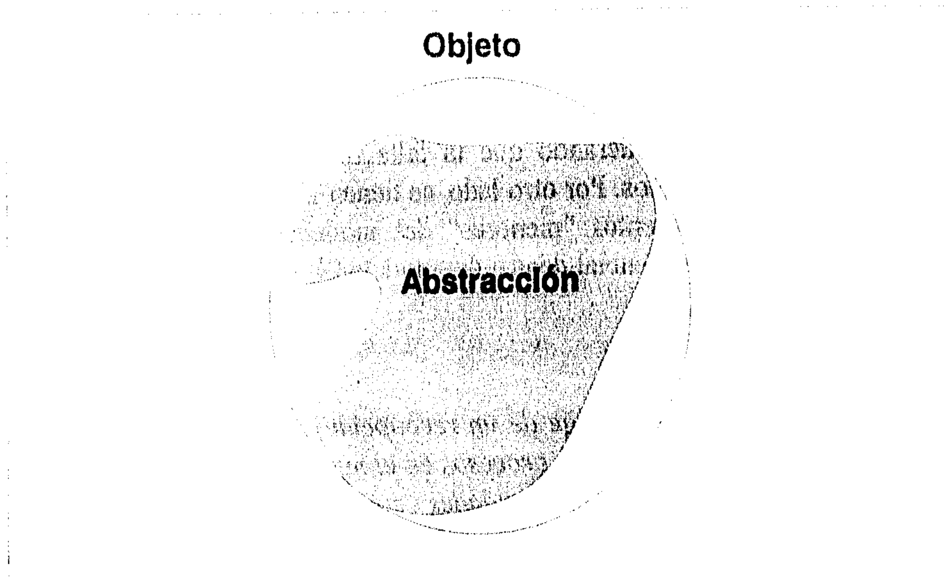


Figura II.4, Un objeto representa una abstracción de la realidad.

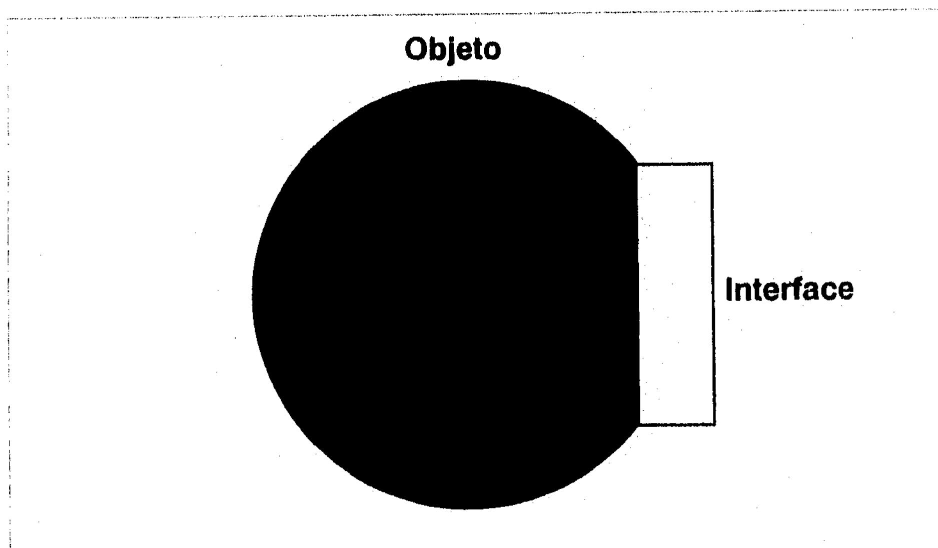


Figura II.5, La encapsulación oculta los detalles internos del objeto.

b) Encapsulación.

En complemento a la abstracción, la encapsulación se enfoca principalmente en la implementación de las características determinadas por la abstracción. La encapsulación es el proceso de ocultar al exterior del objeto aquello que no contribuye directamente a las características esenciales del objeto [9,55]. Los objetos en la abstracción representan un cierto nivel de abstracción. La encapsulación se encarga de los detalles en un nivel de abstracción más bajo, ocultándolos del usuario o cliente.

De manera práctica, para expresar la encapsulación dentro de la Programación Orientada a Objetos, cada clase consta de dos partes: una **interface** y una **implementación** [21,9,55,73,74]. La interface representa la vista externa de los objetos de la clase, es decir, su vía de comunicación a otros objetos de otras clases [9]. Permite enfatizar la abstracción, ya que sirve como una cubierta para ocultar la estructura interna que regula el comportamiento de cada objeto. Por su parte, la implementación de una clase comprende la representación de la abstracción, así como los mecanismos que resultan en el comportamiento deseado [9]. Se puede considerar a la implementación como las operaciones definidas en la interface de una clase.

La interface de la clase sirve como un control de acceso a los detalles de implementación. Se acostumbra considerar tres niveles de acceso dentro de la interface: **público, protegido** y **privado** [9,73,74].

Cualquier elemento de la interface declarado en la clase como público, puede ser accedido por cualquier otra clase dentro de la jerarquía; por otro lado, si un elemento de la interface es declarado como protegido, esto quiere decir que sólo puede ser utilizado por funciones dentro de la clase, por clases declaradas como "amigas", y por las funciones y clases "amigas" de sus clases derivadas; en el caso de un elemento de la clase declarado como privado, éste sólo podrá ser accedido por las funciones dentro de la clase y las clases "amigas" que se hayan declarado [21,9,73,74].

c) Modularidad.

La mayor parte de los sistemas de *software* son susceptibles a ser divididos en unidades operacionales más pequeñas. De hecho, particionar un problema en componentes individuales siempre reduce la complejidad en cierto grado. Casi todos los métodos propuestos para el análisis y diseño

de *software* se basan en dividir el problema total en varios "subproblemas" o módulos.

Se considera a la modularidad de un sistema como la propiedad que tiene de descomponerse en un conjunto de módulos cohesivos y débilmente acoplados. Un programa consiste de varios módulos que pueden ser compilados juntos o por separado. Cada módulo se forma por una secuencia de declaraciones de tipos, funciones, variables y constantes [73,74].

Dentro de la programación orientada a objetos, se requiere que los módulos tengan las características de ser altamente cohesivos y débilmente acoplados, es decir, que agrupen las abstracciones relacionadas lógicamente (cohesivos) y minimicen la dependencia entre ellos (acoplados) [9]. Los módulos del sistema podrán entonces cambiarse independientemente al funcionamiento global del propio sistema. Tal característica determina en gran medida la capacidad de reutilización del *software* [9,55].

La modularidad tiene una relación estrecha con la abstracción y la encapsulación. Tal relación se manifiesta principalmente en la estructura del objeto, en la que la abstracción se oculta mediante una barrera provista por la encapsulación de los módulos [9]

Sin embargo, aún cuando parece sencillo y trivial la división y designación de los diferentes módulos que componen un sistema de *software*, no lo es tanto. Escoger de la gran cantidad de maneras en que un problema puede ser subdividido para resolverse de la forma más eficiente, resulta una labor ardua y dificultosa. La modularización se ve afectada por limitaciones prácticas, como el tamaño real del mayor módulo generable, su ubicación en memoria, etc., o limitaciones organizacionales, como la asignación a diferentes grupos de trabajo del desarrollo de uno o varios módulos. Esto último plantea un serio problema en la definición de los límites e interfaces de los módulos [9,55].

Una solución al problema de modularización propuesta en programación orientada a objetos es considerar el criterio de que las abstracciones deben ser "empaquetadas", de tal manera que sea sencillo establecer las interfaces de los módulos y que resulten ser altamente cohesivos, pero lo más independientes entre sí [9,78,55].

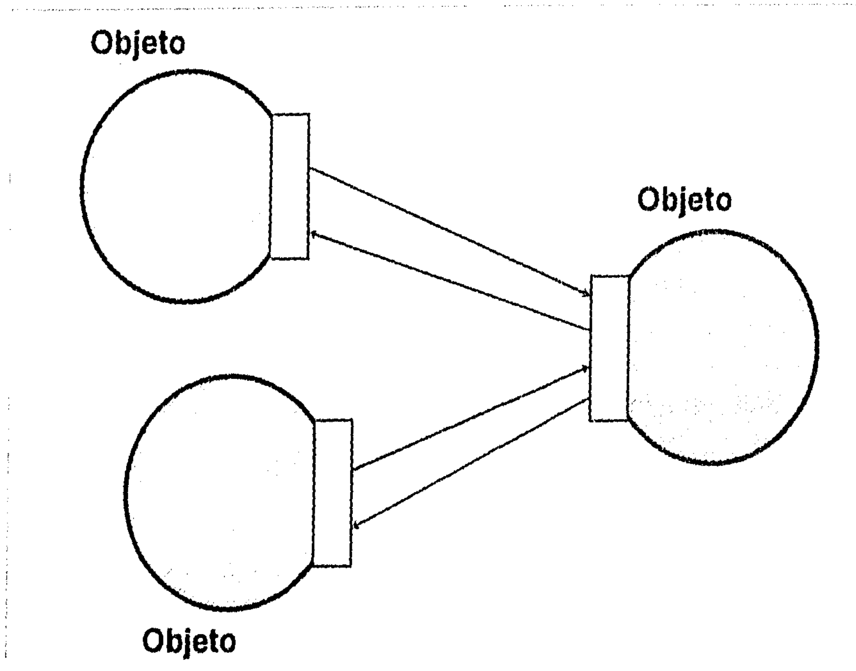


Figura II.6, Modularidad.

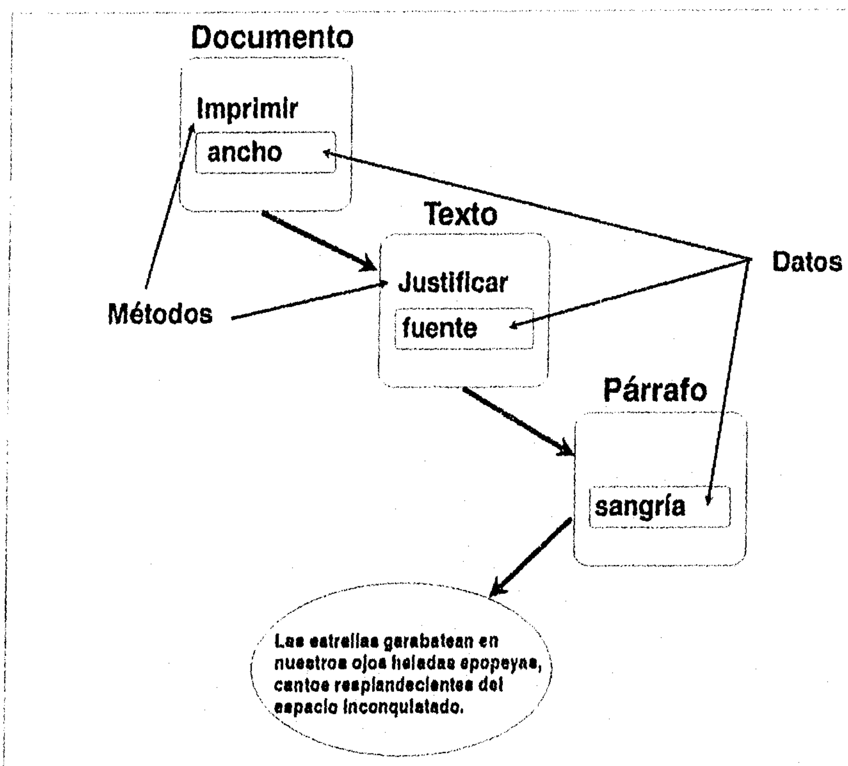


Figura II.7, Herencia.

d) Jerarquización. Relaciones entre clases.

En la realidad, es difícil concebir un objeto aislado: siempre es posible encontrar o definir algún tipo de relación entre objetos. En general, un conjunto de objetos organizados bajo ciertas reglas forman un jerarquía y, mediante su identificación dentro de un diseño, el proceso de comprensión de un problema se facilita grandemente.

Es posible distinguir en la programación orientada a objetos algunas relaciones útiles entre clases de objetos. Considerar cada una o su combinación permiten realizar de cierta forma una jerarquización. Tales relaciones son principalmente: herencia, agregación, asociación y uso [9].

i) Herencia.

La herencia es tal vez la relación más importante que se establece entre clases dentro de la programación orientada a objetos. Expresa relaciones de tipo generalización/especialización, es decir, aquella relación en la que una clase comparte características de estructura y/o comportamiento definida en una (herencia simple) o varias otras clases (herencia múltiple) [9,79].

La relación de herencia se establece entre una subclase que recibe las características heredadas, y una o varias superclases que proveen de tales características. En general, una subclase tiende a presentar un aumento o disminución de las características heredadas. En el primer caso, se dice que la subclase usa la herencia para extensión. En caso contrario, se considera un uso de la herencia para restricción.

Conforme se utiliza dentro de una aplicación, la relación de herencia genera una jerarquización que se representa como un conjunto de categorías de abstracciones. Como se menciona anteriormente [II.A.1.e], se considera que una arquitectura orientada a objetos bien estructurada se forma en realidad de bosques de árboles de herencia, basados en un conjunto de clases base, las cuales representan la mayor generalización en cuanto a grado de abstracción, en contraste con las clases hoja o concretas, las cuales representan la mayor especialización respecto al nivel de abstracción dentro del bosque de árboles de herencia.

La herencia representa una herramienta poderosa en cuanto a la reusabilidad de abstracciones, evitando la repetición directa de código. Sin embargo, su uso presenta ciertas dificultades. En especial en el caso de herencia múltiple, se presentan problemas delicados, como son la colisión de nombres y la herencia repetida [9,55,79].

La colisión de nombres se presenta cuando dos o más superclases utilizan el mismo nombre para un elemento de sus interfaces, ya sean variables o métodos. Esto provoca una ambigüedad en la definición y uso de tal elemento en la subclase [9,79]. Existen tres aproximaciones para dar solución a este problema: primero, considerar la definición de la colisión dentro de la semántica del lenguaje, reconociéndola como una situación ilegal durante la compilación; segundo, el que la semántica del lenguaje permita el uso del mismo nombre en diferentes clases como referencia al mismo atributo, considerándolo en una superclase común dentro de la jerarquía; y tercero, en el que la semántica del lenguaje permita la colisión, pero requiere una referencia completa del nombre y la clase donde se encuentra declarado [9].

El problema de herencia repetida [9,55] se presenta cuando una subclase hereda alguna característica de una superclase más de una vez, por ejemplo, el caso en que dos o más clases son subclases de una superclase común y, a la vez, son superclases de una sola subclase. De esta forma, la característica heredada presenta más de una definición, que además resulta ser exactamente la misma. Este problema representa una de las mayores dificultades, ya que puede presentarse fácilmente dentro de cualquier jerarquía con más de dos niveles de herencia múltiple. Se han planteado principalmente tres aproximaciones de solución a este problema: primero, es posible considerar las ocurrencias de herencia repetida como ilegales, realizando una verificación en tiempo de compilación del programa; segundo, consiste en permitir la repetición de características heredadas, lo que requiere el uso de referencias completas para identificar los miembros de una copia en específico; y tercero, considera que las múltiples referencias a una superclase denoten una referencia directa a esa clase. Así, aún cuando se presente herencia repetida, el código heredado no se repite [9,55].

ii) Agregación.

La agregación es otra importante relación que expresa conexiones del tipo todo/parte entre clases, esto es, por continencia o referencia física entre dos clases de objetos [9,79]. En agregación, se considera que una clase es un elemento o parte de otra.

En cuanto a la jerarquización, la agregación propone que en una jerarquía de clases, aquella clase que contenga o referencíe a una o varias clases se encuentra en un grado o nivel más alto de abstracción [9,55].

A diferencia de la herencia, la agregación no es un concepto exclusivo de los lenguajes orientados a objetos. Sin embargo, la utilización de ambas, agregación y herencia, resultan en un poderoso mecanismo de jerarquización, ya que mientras que la agregación realiza la agrupación física de estructuras lógicamente relacionadas, la herencia facilita la reutilización de tales grupos de abstracciones [79].

La agregación también puede considerarse como una relación de pertenencia, en la cual una modificación de las clases que son parte de otra, no repercute en la identidad de esta última y viceversa. Esto da como resultado que la existencia de las clases relacionadas por agregación sea independiente una de las otras [9,79].

La agregación puede ser fácilmente confundida con la herencia múltiple. De hecho, en algunos casos la herencia puede ser reemplazada por la agregación sin pérdida de significado. Sin embargo, para evitar alguna ambigüedad, se pueden aplicar los criterios sobre los que se basa la herencia y la agregación, es decir, si se trata de una relación generalización/especialización (mediante el uso del término "es un" entre las clases a relacionar) o una relación todo parte (utilizando el término "es parte de" entre las clases relacionadas) [9].

iii) Asociación.

En general, no es fácil reconocer la relación entre las clases de una jerarquía, sobre todo en los casos en que se carece de elementos para considerarla como relación de herencia o agregación. En tales casos se introduce el concepto de asociación entre clases como un

tipo especial de relación entre ellas.

La asociación es una conexión semántica entre dos clases no relacionadas, pero que presentan alguna dependencia entre sí. Tal dependencia semántica considera una relación débil entre las partes. Debido a esto, su uso no es aconsejable, sino que por el contrario, se plantea la posibilidad de redefinirla a lo largo de un diseño para convertirse en otra relación entre objetos más concreta como herencia o agregación [9].

iv) Uso.

Otra relación importante entre clases es el uso. Esta relación se establece entre una clase cliente, la cual hace uso de los servicios de otra clase servidor [9,79].

La relación de uso difiere de la asociación en que esta última denota una conexión semántica bidireccional. De hecho, se puede considerar que la relación de uso es un refinamiento unidireccional de la asociación, si es posible determinar cuál abstracción actúa como cliente y cuál como servidor de algún servicio [9].

En ocasiones, las relaciones de uso resultan demasiado restrictivas ya que sólo permiten al cliente acceder a la interface pública del servidor. Esta situación puede modificarse en casos muy especiales, lo que representa una ruptura en la encapsulación de la abstracción del servidor [9,55].

El uso permite representar la característica de distribución de responsabilidades presente en los sistemas orientados a objetos, es decir, un programa se forma de un conjunto de objetos, los cuales resultan clientes o servidores entre sí.

e) Tipificación.

Los conceptos de tipo y clase tienen un significado semejante en general. Ambos conceptos de la programación orientada a objetos proceden de la teoría de los tipos de datos abstractos. Sin embargo, en forma rigurosa, no se refieren exactamente a la misma idea: una clase se considera la implementación de un tipo [9]. La tipificación tiene entonces una consideración especial dentro de la programación orientada a objetos.

La tipificación se refiere a la capacidad de expresar abstracciones en un lenguaje de programación de tal manera que la implementación pueda utilizarse para reforzar decisiones de diseño, sobre todo en la programación de sistemas grandes, en los que el grado de complejidad es alta. Cada abstracción u objeto se refuerza mediante un tipo, de tal manera que los objetos de diferente tipo no pueden ser intercambiados, a menos que sigan ciertos lineamientos y restricciones.

Es posible distinguir diferentes formas de tipificación. Principalmente, la tipificación puede ser fuerte o débil y, a la vez, estática y dinámica.

La característica de tipificación fuerte o débil se refiere a la flexibilidad de un lenguaje a mezclar tipos de datos. Un lenguaje fuertemente tipificado previene la mezcla abstracciones de diferente tipo. Esto significa que una operación no podrá realizarse en cuanto no se cumpla con la característica de tipo de los valores relacionados, garantizando que las operaciones sean consistentes. En caso de ser necesaria la mezcla de tipos, la mayoría de los lenguajes fuertemente tipificados permiten la conversión de tipos [9].

En contraste, un lenguaje débilmente tipificado no realiza una revisión de la consistencia de tipos, por lo que cualquier operación puede llevarse a cabo si cuenta con valores necesarios. Ofrecen una mayor flexibilidad, responsabilizando al programador directamente respecto a la naturaleza de los argumentos y respuestas obtenidas [79].

Por otro lado, se hace distinción entre la tipificación estática y dinámica. La tipificación estática propone que el tipo de un nombre se declara y reconoce en tiempo de compilación del programa. En contraste se tiene la tipificación dinámica si durante la ejecución de un programa un nombre se asocia con un cierto tipo [55].

f) Persistencia.

La persistencia es la propiedad de un objeto en la cual su existencia trasciende tiempo y/o espacio, es decir, su estado se preserva a lo largo del tiempo y del espacio [9,55,79].

Considerando la persistencia de un objeto en el tiempo, esto quiere decir que el objeto continúa existiendo aún cuando el proceso que lo creó deje de existir. Por otro lado, la persistencia de un objeto en el espacio se

refiere al movimiento que experimenta a partir de la localidad de memoria en que fue creado. La característica de persistencia que se trata con mayor profundidad en aquella relacionada con el tiempo[9].

El introducir la propiedad de persistencia al modelo de objetos da como resultado el concepto de bases de datos orientadas a objetos, las cuales ofrecen al programador la abstracción de una interface orientada a objetos, donde operaciones como la consulta se realizan en términos de objetos cuya existencia es trascendente en el tiempo [9].

g) Concurrencia.

En sus orígenes la programación orientada a objetos fue concebida con la finalidad de simular procesos complejos de la realidad. De esta manera se incluyó la característica de concurrencia entre objetos, de forma que la simulación resultara lo más verídica posible, como se puede encontrar en el apéndice B. Actualmente, la mayoría de los lenguajes orientados a objetos no presentan entre sus características la concurrencia. Es posible lograr una aproximación encapsulando un conjunto de rutinas para el manejo de sincronización y comunicación entre objetos en una clase [55]. Esta solución representa en la actualidad la capacidad de concurrencia de la mayoría de los sistemas orientados a objetos.

Sin embargo, el diseño orientado a objetos promueve arquitecturas descentralizadas de *software* [55], lo que requiere a su vez un control descentralizado de actividades, por lo que el paralelismo aparece como una propiedad importante dentro de los sistemas orientados a objetos.

Los aspectos de comunicación por paso de mensajes de la concurrencia parecen adecuarse fácilmente a los mecanismos básicos de ejecución del modelo de objetos. Una llamada a un método de un objeto puede realizarse perfectamente mediante el envío de un mensaje a dicho objeto, incluyendo posiblemente un conjunto de argumentos [55]. Mientras que la programación orientada a objetos se enfoca en la abstracción, encapsulación y herencia de datos, la concurrencia provee de la abstracción del proceso y sincronización [9].

El objeto es un concepto que unifica la condición de una abstracción de la realidad que posee una existencia propia en tiempo y espacio como una abstracción del proceso. La concurrencia es entonces una propiedad ortogonal a la programación orientada a objetos en niveles de abstracción bajos [9].

B. Características. Ventajas y desventajas.

En las secciones anteriores se ha introducido a la orientación a objetos tratando brevemente las principales definiciones y conceptos que conciernen al modelo de objetos. Sin embargo, la causa más importante de que se considere a la orientación a objetos como una respuesta a los problemas actuales de programación recae más bien en las características que ésta exhibe. En esta sección se hace un análisis de tales características, y de las principales ventajas y desventajas que de uno u otro modo afectan el uso de los objetos en la programación.

1. Características.

A partir de los conceptos expuestos en las definiciones y elementos del modelo de objetos, es posible definir una serie de características de la orientación a objetos. Bertrand Meyer [55] propone las siguientes características de un lenguaje para ser considerado como un lenguaje orientado a objetos.

a) Estructura modular basada en objetos.

"Los sistemas son modularizados en base a sus estructuras de datos" [55]. Este principio se relaciona con el hecho de que los datos deben proveer un criterio estructural fundamental. En un lenguaje orientado a objetos esto ocurre dado el mecanismo de diseño, lo cual resulta en la descomposición de un sistema en una colección de objetos que de manera natural se componen a partir de los datos.

Un programa tradicional consiste en procedimientos y datos. Un programa orientado a objetos consiste sólo en objetos que contienen ambos procedimientos y datos. En otras palabras, los objetos son módulos que contienen tanto datos como las instrucciones que operan sobre esos datos. De esta forma, los objetos son entidades que poseen atributos particulares (datos) y maneras de comportarse (métodos) [79,I.A.1.a].

No es permitida la existencia de datos compartidos o fuera de los objetos. Los datos de un objeto son disponibles para manipulación sólo por parte del objeto que posee los métodos que los operan. Esta característica de los objetos da a la programación orientada a objetos una gran ventaja: la orientación a objetos alienta la modularidad aclarando la creación de límites entre los objetos, haciendo explícita la comunicación entre objetos, y ocultando los detalles de implementación [79].

b) Abstracción de datos.

"Los objetos deben ser descritos como implementaciones de tipos de datos abstractos" [55]. Esto significa que en un lenguaje orientado a objetos, la estructura de un objeto debe corresponder a un tipo de dato abstracto.

La orientación a objetos alienta a los programadores y usuarios a pensar cualquier aplicación en términos abstractos. Comenzando con un conjunto de objetos, el programador puede expresar el comportamiento común y modelar tal comportamiento en clases abstractas. Cada nivel superior de abstracción hace la labor de programar más fácil, debido a que se cuenta con mayor cantidad de código reusable disponible [79].

c) Manejo automático de memoria.

"Los objetos no usados deben ser removidos de la memoria por el sistema, sin intervención del programador" [55]. Esta característica, menos conceptual que las anteriores, se refiere a la forma como los objetos son creados en memoria, y más especialmente, la política de como son eliminados. Se provee de un sistema de soporte al lenguaje el cual cuenta con facilidades para la gestión y manejo del espacio de memoria ocupado por un objeto cuando éste se vuelve inútil, sin intervención del programador.

Esta característica se refiere más precisamente a las propiedades de los sistemas de soporte al lenguaje que al lenguaje en sí. En general, muchos lenguajes orientados a objetos pueden contar o no con tal manejo de memoria, conocido como recolección de basura (*garbage collector*).

d) Clases.

"Cada tipo no simple es un módulo, cada módulo de alto nivel es un tipo" [55]. Los tipos de datos abstractos se implementan mediante clases que determinan la estructura de los objetos como instancias de tipos definidos. Esta característica distingue más claramente a los lenguajes orientados a objetos y basados en objetos de los demás, aún cuando estos cuenten con facilidades para la abstracción de datos y encapsulación.

En programación orientada a objetos, un módulo es mucho más que

una construcción sintáctica que agrupa elementos de programación lógicamente relacionados. Se identifica además con un tipo, que es en sí mismo un elemento más significativo. A la construcción dentro de un lenguaje orientado a objetos que combina los aspectos de módulo y tipo se le conoce como clase [55,II.A.1.a].

Una clase es una descripción de un conjunto casi idéntico de objetos. Una clase consiste de métodos y datos que resumen las características comunes de un conjunto de objetos. La habilidad de abstraer descripciones comunes de métodos y datos de un conjunto de objetos, y expresarlos en una clase es la característica más importante y poderosa de la programación orientada a objetos. Al definir una clase en realidad significa colocar código reusable en un depósito común en lugar de copiarlo una y otra vez. Así, es posible considerar a las clases como los planos para crear objetos. Finalmente, el definir una clase ayuda a aclarar la definición de un objeto: un objeto es una instancia de una clase [79,II.A.1.a].

e) Herencia.

"Una clase puede ser definida como una extensión o restricción de otra" [55]. La característica de herencia de un sistema orientado a objetos surge como consecuencia de la característica anterior, como mecanismo de reusabilidad de clases.

Se plantea la posibilidad que la estructura de una clase dependa de otra clase, de forma que un nuevo tipo pueda ser definido añadiendo o eliminando propiedades de un tipo con mayor nivel de abstracción. A este mecanismo se conoce como herencia, en la cual una nueva clase puede ser declarada como una extensión o restricción de otra clase previamente definida [55,II.A.2.d.i].

La herencia permite el compartir automáticamente métodos y datos entre clases y objetos. Es un mecanismo poderoso que no es posible encontrar en los sistemas procedurales. La herencia permite crear nuevas clases mediante la programación sólo de las diferencias respecto a una superclase. Así, la programación orientada a objetos consiste principalmente en una clasificación de objetos en árboles o jerarquías de clases las cuales mediante la creación de subclases, se vuelven cada vez más específicas. De manera parecida como una clase describe a un objeto, una superclase provee de los planos para la creación de subclases derivadas, o de los objetos de tales clases dentro de una aplicación [79,II.A.2.d.i].

f) Polimorfismo y ligadura dinámica.

"Se debe permitir a las entidades de un programa hacer referencia a objetos de más de una clase, y permitir a las operaciones tener diferentes realizaciones en diferentes clases" [55]. La capacidad de un lenguaje a soportar el polimorfismo permite que dada una entidad de un programa, es posible que haga referencias en tiempo de ejecución a instancias de diferentes clases [II.A.1.c]. Esto se apoya en el concepto de ligadura dinámica, el cual se encarga que en tiempo de ejecución sea posible automáticamente seleccionar la versión de una operación que se adapte a la correspondiente instancia [55,II.A.1.d].

Los objetos actúan en respuesta de los mensajes que reciben. El mismo mensaje puede dar como resultado acciones completamente diferentes cuando se reciben por objetos diferentes. Este fenómeno se conoce como polimorfismo. De esta manera, un mensaje puede ser manejado en forma genérica, dejando los detalles de implementación y comportamiento al objeto que lo recibe [79].

El polimorfismo tiene sus bases en el mecanismo de la herencia. Esto significa que dentro de una jerarquía de clases es posible almacenar funciones con nombres comunes en clases de alto nivel de abstracción, para después heredarlas en diferentes clases utilizando el mismo nombre, pero considerando las variaciones pertinentes, es decir, las variaciones de una función son almacenadas y redefinidas en las clases con nivel de abstracción más bajo, modificando los métodos de la manera más apropiada. Así, los objetos generados a partir de estas clases se encuentran listos para responder de diferente manera al recibir mensajes genéricos que involucren el nombre de la función [79].

La ligadura dinámica es una característica común pero no necesaria de los lenguajes orientados a objetos. La ligadura se refiere al proceso de enlazar los elementos de un programa entre sí, para realizar todas las conexiones que existan entre sus componentes. Convencionalmente, este proceso se efectúa durante el proceso de compilación, con lo cual todas las conexiones se realizan antes de ejecutar el programa. A esto se conoce como ligadura estática o temprana [II.A.1.d]. Por otro lado se tiene la ligadura dinámica, que es utilizada en varios lenguajes orientados a objetos, y que realiza igualmente el proceso de enlazamiento pero durante el tiempo de ejecución del programa [79].

La ligadura dinámica es una consecuencia de la implementación del

polimorfismo. Al ejecutarse un programa orientado a objetos, una serie de mensajes son enviados y recibidos por los objetos. Frecuentemente, un método para el manejo de un mensaje se encuentra almacenado en un nivel alto de abstracción en una jerarquía y, además, es almacenado dinámicamente en memoria cuando se le requiere. En ese momento, se realiza el enlazamiento, es decir, las conexiones entre el método y los datos locales al objeto. La liga ocurre entonces en el último momento posible [79].

g) Herencia múltiple y repetida.

"Debe ser posible declarar una clase como heredera de más de una clase, y más de una vez de la misma clase" [55]. Esto significa que es posible heredar los mismos tipos de datos más de una vez, sin la necesidad de renombrar ningún elemento, lo que puede llevar a confusiones [II.A.2.d.i].

Ciertamente, la herencia múltiple añade potencial de expresividad a un lenguaje, pero a un costo en la complejidad sintáctica, así como un aumento en los tiempos de compilación y ejecución. Las clases son cada vez más complicadas en su diseño, ya que cuentan con un mayor número de líneas de herencia para adquirir métodos y datos con conflictos potenciales [79,II.A.2.d.i].

2. Ventajas.

Considerando sus elementos y características, la programación orientada a objetos cuenta con varias ventajas respecto a la programación procedural. Algunas de las más importantes son:

a) Uniformidad.

La programación orientada a objetos permite desarrollar un programa siguiendo un esquema de pensamiento a partir de una serie de acciones que comprenden fases tales como el análisis y el diseño, hasta la realización del código mismo. Tales acciones se efectúan en general para cualquier aplicación, es decir, la programación orientada a objetos es susceptible a desarrollarse a partir de alguna metodología uniforme que es independiente a la naturaleza de la aplicación [21].

El uso de metodologías uniformes reduce los riesgos inherentes en el desarrollo de sistemas complejos, debido a que tales metodologías se basan

en una integración del desarrollo durante un ciclo de vida, y no ocurren como un solo gran evento. Esto es, las metodologías basadas en el modelo de objetos proveen de una guía en el diseño, lo que reduce los riesgos de desarrollo y aumenta a su vez la confianza en la corrección del diseño [9,52].

Además, la uniformidad conjuntamente con otras ventajas como el entendimiento, la reusabilidad y la flexibilidad, da como resultado un rápido desarrollo de aplicaciones orientadas a objetos [52].

b) Entendimiento.

En los sistemas orientados a objetos, las abstracciones se organizan en un código basado en clases que durante las etapas de análisis y diseño, se procura que correspondan a objetos del dominio de la aplicación, esto es, que el programa represente de una manera clara un modelo del problema [21]. El modelo de objetos apela a la forma de pensar humana, ya que modela en un programa los conceptos del dominio del problema de una forma natural [9].

El modelo de objetos permite al programador, de una manera natural, modelar fenómenos complejos del mundo real. Programar orientado a objetos no es sólo escribir líneas de código, sino desarrollar un modelo basado en clases. Al soportar el diseño modular, la programación orientada a objetos hace posible mantener juntos elementos relacionados y, en particular, los datos y métodos que operan tales datos en un mismo contenedor. Esto hace que el mantenimiento y mejora de los programas se simplifique. No es necesario revisar completamente el código para observar si algún cambio local afecta a otros puntos del sistema [79,52].

c) Flexibilidad y mantenimiento.

Al aplicar la programación orientada a objetos mediante una metodología de desarrollo de *software*, permite el desarrollo independiente y simultáneo de módulos, dada la característica de los objetos de contener métodos y datos independientes entre sí. De esta forma, el desarrollo o modificación de un módulo es posible sin involucrar grandes esfuerzos por parte del programador [21].

Una vez que algunos abstracciones del programa se encuentran definidas mediante clases, el proceso de programación se vuelve incrementalmente más sencillo. Mediante mecanismos como la herencia y

la agregación, el proceso de programación se centra en realizar sólo las diferencias entre las clases derivadas y las clases abstractas definidas [79].

Por otro lado, debido a su característica de modularidad, la orientación a objetos promueve la división del trabajo entre los miembros de un equipo de desarrollo, facilitando la labor independiente para los desarrollos de aplicaciones grandes. Cada objeto puede ser implementado sin realizar una integración explícita de todos los módulos [79].

Además, una aplicación orientada a objetos es más sencilla de modificar y mantener, ya que los objetos pueden ser añadidos o reemplazados a un diseño original sin cambiar la estructura existente. La herencia permite obtener nuevos módulos a partir de los ya desarrollados. Los métodos son más fáciles de modificar y corregir, debido a que cuentan con una sola localización. No es necesario rastrear ni reemplazar funciones que pueden aparecer a lo largo del código [79].

La modularidad es la característica que da un mayor apoyo al mantenimiento, ya que hace más fácil controlar los efectos de modificaciones dentro de un solo módulo que en todo un programa. El polimorfismo reduce el número de procedimientos, y por lo tanto, el tamaño de los programas a mantener. La herencia puede ser utilizada además para documentar los cambios de versiones, representando una historia desde las primeras clases abstractas hasta las clases relacionadas directamente con la aplicación [79].

d) Estabilidad.

Las características inherentes al dominio del problema en un desarrollo orientado a objetos son elementos que en general permanecen estables a través del tiempo [21]. Así, el código se encuentra organizado alrededor de una estructura estable en el tiempo, ya que las formas intermedias utilizadas para la construcción de sistemas orientados a objetos son más renuentes al cambio. De esta forma, tales sistemas tienden a ser utilizados con un mayor tiempo de vida, sin tener que ser abandonados o totalmente rediseñados debido a un cambio en los requerimientos [21,9].

e) Reusabilidad.

La reusabilidad se propone como una de las ventajas más importantes del modelo de objetos, ya que por una parte, al aumentar el grado de entendimiento de un módulo, éste es susceptible de ser reutilizado.

Por otro lado, la programación orientada a objetos contiene mecanismos propios como la herencia, que permiten una realización simple de la reusabilidad [21,19].

Además, el modelo de objetos no sólo permite la reusabilidad de *software* a nivel de código, sino que también permite la reutilización a nivel diseño, enfocándose a la creación de entornos reusables para aplicaciones [9].

Las bibliotecas de clases predefinidas como componentes de lenguajes orientados a objetos maduros aumentan el beneficio de utilizar el modelo de objetos. Al programar, es posible encontrar clases apropiadas que se ajusten a los requerimientos de la aplicación, o cuando menos sean una aproximación la cual puede ajustarse utilizando herencia o agregación [79,19,52,6,57].

f) Tamaño del código.

Se ha observado que los sistemas de programación orientados a objetos son frecuentemente más pequeños que las implementaciones equivalentes no orientadas a objetos. Esto no solo significa un tamaño menor de código por escribir y mantener, sino que también amplía la capacidad de reuso de *software*, lo que a largo plazo se traduce a beneficios en costo y tiempo [9].

Los programas desarrollados en base al modelo de objetos tienen menor número de líneas de código, pocas instrucciones de cambio de control del programa, y módulos más pequeños, que reflejan una relación uno a uno con el modelo conceptual del problema [79].

El mecanismo de herencia es una de las características más importantes en la reducción del código, ya que sólo es necesario programar las diferencias entre las clases abstractas existentes y las clases particulares utilizadas en una aplicación [79].

g) Especificación y verificación.

Finalmente, dado que los objetos se determinan mediante su definición en una clase, es posible considerar los sistemas orientados a objetos a partir de su relación con la teoría de tipos de datos abstractos como susceptibles a ser especificados.

La especificación mediante tipos de datos abstractos tiene la cualidad de poder ser transportada con las consideraciones pertinentes y, de una manera relativamente sencilla, a una aplicación basada en objetos. Esto es, que un objeto representa la implementación de un tipo de dato abstracto, lo cual facilita la especificación de sistemas de *software* utilizando herramientas de lógica y matemáticas. Una especificación formal tiene varias ventajas en el proceso de desarrollo de *software* [16]:

- Es un vehículo para la comunicación precisa entre el cliente y el desarrollador de software.

- Es un estándar contra el cual el código del programa final puede ser verificado.

- Es un modelo a partir del cual es posible comprobar las propiedades, lo que sirve como una ayuda para validar la especificación formal respecto a los requerimientos informales. Algunas propiedades deseables de la especificación son la completez, parsimonia y terminación [4].

Es posible, al contar con una especificación clara del sistema orientado a objetos basada en tipos de datos abstractos, verificar su corrección. La verificación es una manera de medir en qué forma un programa cumple o no con los requerimientos de la aplicación. Toda medición requiere de una referencia. Así, la especificación realizada para un sistema orientado a objetos puede servir a la larga como referencia para comprobar si los requerimientos de la aplicación han sido alcanzados o no [77].

La especificación formal puede mejorar en gran medida el proceso de desarrollo del *software* de varias maneras. Específicamente, el proceso puede mejorarse al crear una especificación formal después que el prototipo ha sido construido utilizando técnicas tradicionales [77].

3. Desventajas.

De igual manera que se mencionan las principales ventajas de la programación orientada a objetos, también es importante considerar las desventajas que el modelo de objetos presenta. De esta manera, se presentan las siguientes desventajas:

a) Complejidad en el diseño.

El problema de complejidad de un sistema parece tener solución mediante la simplicidad del modelo de objetos. Sin embargo, considerar al objeto como la unidad autónoma de conducta y estado es demasiado simple para ser real. En ocasiones, un objeto realmente actúa autónoma y espontáneamente, pero también puede ser el objeto pasivo de una actividad, o como elemento, formar parte de una actividad colectiva. Es verdad que algunos objetos tienen un sentido de coherencia física e integridad espacial, existiendo en un lugar y tiempo precisos. Sin embargo, otros son abstracciones difusas, formando parte de un todo como un solo concepto abstracto. Algunos son visibles y fácilmente comprensibles, pero también existen otros que son abstracciones poco claras o invisibles. Existen algunos más que pueden parecer entidades totalmente independientes, mientras que otros se presentan como conceptos intrincadamente relacionados [45]. Todas estas situaciones representan en cierta forma la complejidad de la realidad a modelar. Esto significa que el modelo a desarrollar en programación debe ser cuando menos tan complejo como tal realidad.

La complejidad se refiere al nivel de dificultad que representa implementar el comportamiento de la realidad, es decir, la complejidad de un sistema es la limitación de la capacidad humana para manejar el conjunto de sus componentes y las relaciones entre ellos. Las partes o elementos de un sistema de *software* pueden interactuar de múltiples maneras teniendo en la realidad poco en común. Tales elementos a considerar en un diseño pueden llegar a un número tal que sea imposible para una persona comprender completamente todos los detalles [9].

La programación orientada a objetos se plantea como una solución al manejo de la complejidad, pero con ciertas limitaciones. Aún dentro de un programa orientado a objetos, si el número de clases involucradas o las relaciones entre tales clases es demasiado grande, puede escapar de la capacidad humana, es decir, continuar siendo complejo a pesar de las facilidades que la orientación a objetos puede ofrecer. Una persona puede comprender perfectamente un sistema en cuanto a sus elementos y funcionamiento y, sin embargo, serle difícil concebir los elementos aislados o poco relacionados. No todo sistema complejo es susceptible a ser modelado mediante objetos, diminuyendo su complejidad notablemente.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

b) Desempeño.

La desventaja más importante que pesa sobre la orientación a objetos se refiere a su desempeño en el tiempo. Un programa desarrollado en un lenguaje orientado a objetos se ejecuta más lentamente que un programa escrito en un lenguaje procedural [9,79,15].

En estudios comparativos del desempeño de lenguajes orientados a objetos y procedurales se ha encontrado algunas características por las cuales los lenguajes orientados a objetos tienden a ejecutarse más lentamente. Estas causas se refieren a las diferencias en la forma de ejecución de un programa procedural y un programa orientado a objetos. Así se han obtenido las siguientes observaciones [15]:

- Los programas orientados a objetos contienen un número menor de saltos condicionales, pero realizan un mayor número de llamadas a procedimientos de gran magnitud, en su mayoría indirectamente.

- Los programas orientados a objetos tienden a tener un mayor número de procedimientos pequeños, los cuales generalmente se alcanzan mediante llamadas indirectas a funciones.

- Los programas orientados a objetos colocan un mayor número de pequeños objetos dinámicamente en memoria, lo que mejora el aprovechamiento del espacio.

- Los programas orientados a objetos realizan un mayor número de operaciones de lectura y escritura de memoria que los programas procedurales.

Por otro lado, a nivel programación el desempeño de un programa orientado a objetos se ve disminuido debido principalmente al costo por envío de mensajes entre objetos. La invocación de un método que no puede ser resuelta estáticamente en una aplicación forza a la implementación a revisar dinámicamente, a fin de hallar el método definido en la clase del objeto que recibe el mensaje. Además, es posible tomar en consideración que otra causa de demora en los programas orientados a objetos es la profundidad de las jerarquías de clases. Una clase con bajo nivel de abstracción puede tener varias superclases, cuyos códigos deben ser incluidos al ligar el programa. De esta forma, el proceso de búsqueda de un método específico aumenta en otra dirección. Para aplicaciones en las cuales el tiempo es un recurso limitado, una gran cantidad de invocación

hacia métodos puede llegar a ser inaceptable [9].

c) Dependencia en etapas de análisis y diseño.

El poder de la programación orientada a objetos para cumplir con características tales como confiabilidad y rápido desarrollo se basan en un ajuste a una metodología de análisis y diseño formal, la cual pueda ser susceptible a verificarse.

Esto tiene como consecuencia que el programador debe, en general, aprender sobre las bibliotecas de clases que provee el lenguaje antes de poder producir un programa orientado a objetos. Como resultado, los lenguajes orientados a objetos dependen aún más de una buenas herramientas para documentación y desarrollo, sobre todo en las etapas de especificación, análisis y diseño. Sin embargo, la gran mayoría de quienes se encargan del diseño de sistemas orientados a objetos no consideran la importancia de tales etapas en el desarrollo de un sistema, sino que por el contrario, procuran evitarlas o en el mejor de los casos las realizan como un simple formalismo del desarrollo, dándoles poca importancia [79].

d) Coherencia y localización.

La orientación a objetos considera al objeto como la unidad de conocimiento, representando dentro de un sistema computacional a un concepto o entidad. Su interface permite comunicación con otros objetos y a la vez oculta los detalles de implementación de los métodos. De esta forma, un objeto puede actuar como cliente o servidor de métodos de otros objetos, es decir, el código de un método que provee la implementación de un objeto puede invocar operaciones como usuario de otros objetos. Esto es funciona siempre y cuando se consideren verdaderas las siguientes afirmaciones [45]:

- Los datos que representan el estado de un objeto ocupan espacios de memoria contiguos.
- Los datos que representan el estado de un objeto se encuentran separados de los datos que representan otro objeto.
- Los datos que representan a un objeto se encuentran siempre en el mismo lugar.
- Los códigos ejecutables que proveen el comportamiento de un objeto son distintos a aquéllos que proveen el comportamiento de otro

objeto.

- Los códigos ejecutables que proveen el comportamiento de un tipo de objetos son distintos de aquéllos que proveen el comportamiento de otro tipo de objetos.

- Los códigos ejecutables que proveen el comportamiento de un (tipo de) objetos se encuentran empacados en un solo programa o aplicación, y ocupan siempre el mismo lugar.

- Los datos que representan el estado de un objeto y los códigos ejecutables que proveen su comportamiento se encuentran empacados en un sola unidad.

Sin embargo, aún cuando tiene sentido definir conjuntos de objetos que tienen varias combinaciones de estas características, ninguna es verdadera para todos los objetos. Mucho tiene que ver con la forma en que los objetos se encuentran implementados en memoria realmente. El problema tiene su origen en el paradigma de despacho (*dispatching paradigm*) definido en los lenguajes de programación. Este paradigma se refiere al mecanismo encargado de inicializar la ejecución de un programa. En los lenguajes procedurales se utiliza el paradigma clásico, esto es, una invocación **p(x)** invoca un procedimiento **p** con el argumento **x**. El comportamiento depende de la naturaleza particular del argumento **x**. Es responsabilidad del programador que cada procedimiento tenga un nombre único y, a su vez, es responsabilidad del usuario llamar únicamente a los procedimientos por su nombre único, logrando que el funcionamiento del sistema sea muy simple [45].

El paradigma clásico orientado a objetos propone un cambio a la invocación de procedimientos. Un objeto **O** contiene su propia versión del procedimiento **p**, lo que da como resultado un comportamiento específico del objeto. La labor del sistema no es buscar el procedimiento **p**, sino más bien al objeto **O**, el cual inicializa por sí mismo su versión del procedimiento. Esto se enfatiza cuando invocamos un procedimiento de la forma **O.p(x)**, lo cual da idea de que al objeto **O** se le envía un mensaje **p** con el argumento **x** [45].

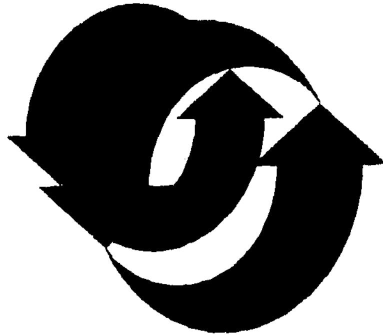
Sin embargo, la mayoría de los sistemas orientado a objetos no proveen a cada objeto de su propia versión de un procedimiento. En lugar de eso, los objetos del mismo tipo pueden compartir una versión del procedimiento. Las suposiciones sobre la coherencia y localización de los

objetos y tipos en memoria son en su mayoría sólo una ilusión. En realidad, el sistema no envía un mensaje al objeto para ser interpretado, sino que busca la versión apropiada de **p** dependiendo del tipo del objeto **O**. Este tipo de manejo hace difícil la ejecución de un objeto dentro de una implementación [45].

e) Costos iniciales.

El inicio de un nuevo proyecto siempre lleva consigo una inversión, tanto en costos como en tiempo y esfuerzo. Ahora bien, iniciar la utilización de la tecnología orientada a objetos en proyectos nuevos también requiere de tal inversión. Esto ha provocado una dificultad de adoptar la orientación a objetos por parte de diversas organizaciones, ya que el uso de la tecnología orientada a objetos requiere la compra de herramientas de desarrollo de *software*.

Por otro lado, una organización que se inicia en el uso de la tecnología orientada a objetos, utilizando por primera vez un lenguaje en particular, no ha establecido una base de *software* ni algún método o política de reutilización. Es necesario iniciar totalmente, o cuando menos, tratando de conectar sus aplicaciones orientadas a objetos con aquellas no orientadas a objetos existentes. En general, el primer intento de adoptar programación orientada a objetos falla debido a la falta experiencia y entrenamiento adecuado. *"Un lenguaje de programación orientado a objetos no es 'solo otro lenguaje de programación' que puede aprenderse en un curso de tres días o leyendo un libro"* [9].



Capítulo III.
Programación Paralela Orientada a Objetos.

"Tres planos gravitacionales distintos se influyen respectivamente en sentido vertical. Tres superficies, sobre cada una de las cuales viven personas, se cortan en ángulo recto. Dos habitantes de mundos distintos no pueden andar sobre el mismo suelo, estar sentados o de pie, ya que no coinciden las ideas que tienen de lo que es horizontal o de lo que es vertical. No obstante, pueden utilizar la misma escalera. Parece imposible que puedan llegar a establecer algún tipo de contacto entre sí: viven en mundos distintos y nada saben de la existencia del otro".
Maurits Cornelis Escher.

La programación orientada a objetos es el paradigma de programación que en la actualidad se está utilizando cada vez más en el desarrollo de sistemas paralelos y distribuidos. Sus características inherentes (abstracción, encapsulación, reutilización, etc.) le hace un excelente candidato para resolver las dificultades encontradas en la programación de tales sistemas. Esto se debe a que este paradigma considera el dominio del problema de una manera más conceptual. Se enfoca en la interacción entre entidades, abstrayendo los detalles de representación e implementación, lo que la hace una metodología más fácil de entender aún para quienes no realizan programación.

Este estilo de programación se está diseminando fuertemente entre la comunidad de programadores de sistemas paralelos. Sin embargo, realizar la transferencia de su conocimiento y experiencia a una programación paralela orientada a objetos no es una tarea sencilla, ya que algunos mecanismos desarrollados para la orientación a objetos requieren cierta evolución para ajustarse correctamente al concepto de paralelismo. Además, otra dificultad reside en la inmadurez relativa en el campo del paralelismo, lo

que ha dado como resultado una abundancia de prototipos de lenguajes paralelos orientados a objetos.Sin embargo, es posible aprovechar la experiencia acumulada en el diseño de tales lenguajes, a fin de realizar un análisis cualitativo de esta programación, dando a conocer brevemente sus principales características, modelos, ventajas y aplicaciones. Este es el objetivo del presente trabajo.

A. Objetivo del estudio.

Para el desarrollo de este trabajo es importante presentar de una manera más formal y precisa el objetivo de estudio. Una primera aproximación a la idea principal se basa en una afirmación realizada por Lim y Johnson, citada por G. Booch [9]:

> *"Diseñar características para concurrencia en lenguajes orientados a objetos no es muy diferente a (hacerlo en) otros tipos de lenguajes - la concurrencia es ortogonal a la programación orientada a objetos en los niveles de abstracción más bajos".*

Esto significa que es posible desarrollar una programación paralela orientada a objetos conjuntando elementos del paralelismo y objetos, es decir, un diseño orientado a objetos que se pretende ejecutar en una arquitectura paralela puede desarrollarse en dos partes:

- Una orientada a objetos, que considera elementos como la abstracción. encapsulación, modularidad y jerarquía, obteniendo las ventajas de esta tecnología.

- Una paralela, que se encarga principalmente de la encapsulación, comunicación y sincronización de los módulos (u objetos) de los que consta el programa, permitiendo una mayor velocidad de ejecución respecto a sistemas secuenciales, además de una mayor expresividad entre los elementos del programa.

A partir de esto, se formula el siguiente objetivo de estudio:

Estudiar y analizar una programación paralela orientada a objetos que presente las principales ventajas de ambas tecnologías, y reduzca en cierto grado sus inconvenientes. Evaluar y presentar un diseño paralelo orientado a objetos, esto es, la posibilidad de desarrollo orientado a objetos en una arquitectura paralela, basándose en la ortogonalidad de ambos elementos mediante un ejemplo de aplicación. El reto es lograr un esquema de programación para el paralelismo el cual se mezcle correctamente con la aproximación orientada a objetos, mientras

coincidan con la convincente simplicidad de los conceptos del paralelismo y la orientación a objetos. Se pretende resaltar ventajas tales como la entendibilidad, reusabilidad, desempeño y expresividad del lenguaje.

Para lograr este objetivo, se propone expresar las principales características de los sistemas paralelos como elementos de clases de objetos, es decir, como una implementación de tales características encapsuladas mediante clases que expresen de una forma simple el paralelismo, secuencialidad, no determinismo, comunicación y sincronización entre objetos.

B. La programación paralela y la programación orientada a objetos.

La programación paralela orientada a objetos se puede considerar como una metodología basada en el modelo de objetos, a fin de aprovechar sus ventajas para el desarrollo de *software* de alto nivel y reutilizable, incorporando a su vez las ventajas del procesamiento paralelo. Un programa se describe entonces como un conjunto de módulos autocontenidos los cuales se ejecutan paralela o concurrentemente, intercambiando información entre sí mediante un protocolo de comunicación. De esta manera, es posible descomponer y ejecutar eficientemente un programa en un sistema multiprocesador.

1. ¿Por qué paralelismo orientado a objetos?

La programación orientada a objetos no es un concepto nuevo. El lenguaje Simula 67, desarrollado a fines de los años 60's, contiene todos los conceptos básicos que se encuentran en los lenguajes orientados a objetos modernos. Sin embargo, una parte importante de Simula 67 no se ha incluido en tales lenguajes modernos: el soporte a procesos cuasi-paralelos o corrutinas [1]. Aún cuando la aproximación orientada a objetos ha permanecido básicamente secuencial en los últimos años, la utilización del multiprocesamiento permitiría crear programas consistentes en varios procesos secuenciales operando en paralelo, facilitando el modelado de procesos reales que se efectúan simultáneamente en el tiempo.

Como finalidad propia, el diseño orientado a objetos promueve arquitecturas de *software* descentralizado. Esto implica en extensión que la orientación a objetos requiere un control también descentralizado, es decir, el paralelismo debe soportarse como característica inherente de la orientación a objetos [55].

Es cierto que no todos los objetos pueden ser vistos como procesos. Pero considerar los procesos como objetos es en extremo atractivo. Los procesos cuentan con estados y pueden ejecutar rutinas. Se distinguen de otros objetos en

que tienen un comportamiento prescrito formado por fases sucesivas. Además, los procesos tienen la capacidad de comunicarse con otros procesos sujetándose a alguna condición de sincronización. Sin embargo, esta propiedad de los procesos parece ser cubierta adecuadamente por los mecanismos básicos de la ejecución de un programa orientado a objetos, basándose en la invocación de métodos de un objeto, observándola como el envío de un mensaje a tal objeto. Esta idea sirve como base al **modelo de actores** [51,14,28,63,62,III.A.2.b.*iii*], el cual describe la ejecución concurrente como la generalización del modelo de paso de mensaje [55].

Se han realizado varios intentos para añadir las características de concurrencia a un lenguaje orientado a objetos después de haber sido diseñado [17,18]; sin embargo, en la mayoría de los casos, esto ha dado como resultado un híbrido difícil de usar, que produce programas ineficientes [81,49].

Por otro lado, también se ha intentado aplicar la orientación a objetos en un entorno distribuido de red utilizando computadoras en paralelo [76,44]. Esto ha generado una forma híbrida de comunicación que busca combinar las propiedades de la orientación a objetos y la programación distribuida. El resultado es un sistema fácil de programar y con un desempeño aceptable [76], pero que se soporta principalmente en una comunicación confiable entre las computadoras, que en la realidad es difícil de lograr, lo que demerita el funcionamiento del sistema.

Una tercera propuesta consiste en aprovechar las ventajas de la orientación a objetos en una arquitectura paralela, substituyendo la red de computadoras por una interconexión física de los procesadores dentro de una misma computadora. Como es de esperarse, el resultado debe conjuntar las ventajas de la programación orientada a objetos (reusabilidad de componentes, modularización del problema, etc.), con las ventajas del procesamiento en paralelo (alto desempeño, comunicación simple y confiable entre procesos, etc.) [51]. De hecho, esta propuesta parece ser la más viable para substituir a futuro a los sistemas secuenciales tradicionales, una vez que estos últimos hayan alcanzado su máximo nivel de desempeño y sea impráctico o costoso ampliar sus capacidades [52].

2. El paralelismo orientado a objetos.

a) Objetos y mensajes.

La programación paralela orientada a objetos se basa principalmente en dos conceptos: el objeto, que identifica conocimiento en forma de datos y servicios, y el paso de mensaje como protocolo unificado de comunicación (Figura III.1) en un sistema multiprocesador. A esta aproximación al paralelismo se le conoce como modelo de paralelismo

natural, en el que el objeto encapsula procesos y las llamadas a funciones miembro encapsulan comunicaciones inter-proceso [14,62].

Los objetos representan entidades y conceptos que componen el dominio del problema. No existe un algoritmo global como modelo del problema sino una cooperación de los objetos, los cuales intercambian información mediante mensajes. Su implementación se basa en conceptos útiles del modelo de objetos, como la abstracción (en forma de clases) y la herencia (en forma de subclases), las cuales permiten especificar, clasificar y reutilizar las descripciones de los objetos [14].

El paralelismo es un elemento importante que se incorpora a la programación orientada a objetos, presentándose como la habilidad de expresar simultaneidad de acciones dentro de un programa, lo que aumenta considerablemente la eficiencia en su ejecución. Se representa mediante mecanismos de sincronización y comunicación entre procesos paralelos como es el caso del paso de mensaje. De esta forma, la integración de ambas tendencias da como resultado una programación paralela orientada a objetos, la cual aparece como la generalización natural del paradigma de objetos. Esto permite la creación de objetos con una mayor autonomía en su actividad al exhibir paralelismo.

Es posible expresar el paralelismo a nivel objeto a partir de dos puntos de vista principalmente: interobjeto e intraobjeto. Además, por su nivel de actividad dentro del programa, cada objeto puede ser activo o pasivo.

i) Paralelismo interobjeto e intraobjeto.

El paralelismo interobjeto se refiere a que cada objeto posee una existencia simultánea tanto en tiempo como en espacio. La cooperación entre objetos se realiza mediante mecanismos de comunicación y sincronización entre procesos, basados principalmente en construcciones para envío y recepción de mensajes [I.B.1.d].

El paralelismo intraobjeto se refiere a la actividad que cada objeto realiza sobre sus métodos internamente, lo cual le permite manejar varias acciones y/o mensajes concurrentemente, utilizando un nivel de granularidad más fino [14]. Para esto, se basa principalmente en construcciones del tipo paralela, secuencial y alternativa [I.B.1.a,I.B.1.b,I.B.1.c].

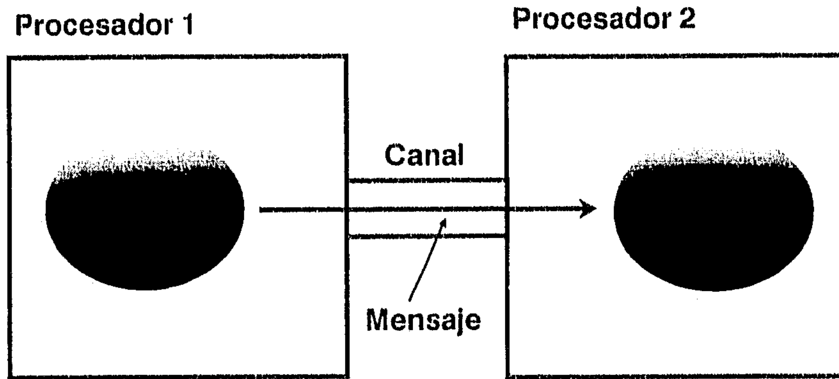


Figura III.1, Objetos y mensajes.

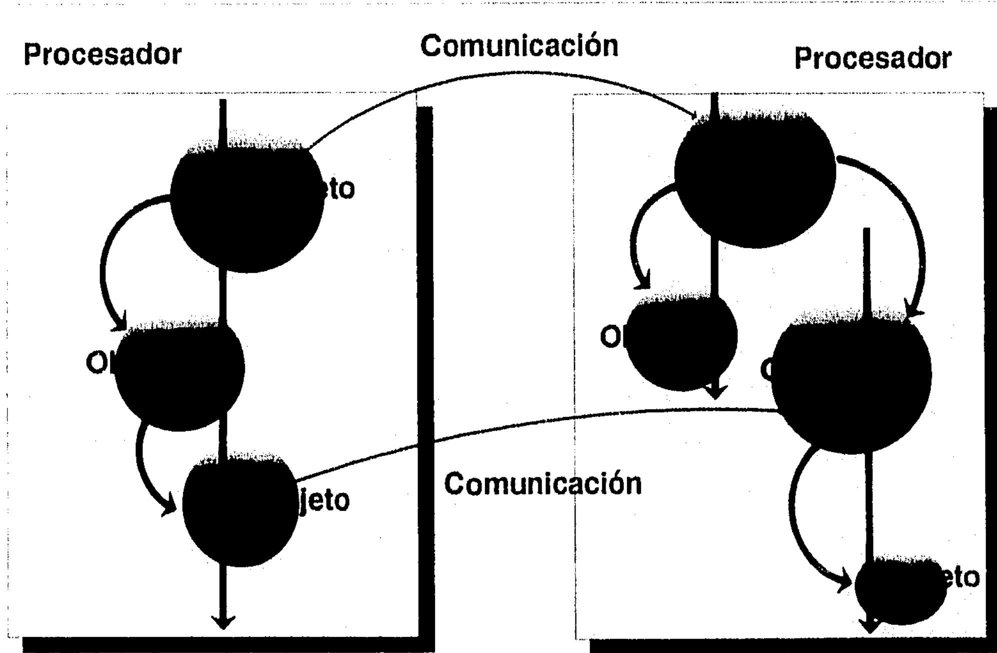


Figura III.2, Tareas paralelas.

Es importante distinguir que a nivel interobjeto, el paralelismo puede realizarse de forma concurrente (como la simultaneidad lógica de actividades) o paralela (como simultaneidad física). Por otra parte, a nivel intraobjeto se preferirá el uso de concurrencia [14].

ii) Objetos activos y pasivos.

En el modelo de objetos, el objeto representa la unidad o módulo básico de un programa. Mediante los elementos del modelo (abstracción, encapsulación, modularidad, jerarquía, etc.) [II.A.2] se permite la creación de programas que son una representación o modelo de la realidad. Tal característica se conserva al conjuntar al modelo de objetos con el procesamiento en una arquitectura paralela.

Un programa paralelo orientado a objetos consiste de un conjunto de objetos con diferentes niveles de acción. El objeto entonces se identifica como la unidad activa dentro del programa. De esta forma, se distinguen dos tipos de objetos: activos y pasivos.

Los objetos activos son instancias capaces cada una de ejecutar sus funciones miembro en paralelo con las de otros objetos activos. Difieren de los objetos pasivos en que adicionalmente a estados y operaciones, encapsulan un proceso, el que permite ejecutar sus funciones miembro en paralelo con el resto del programa [62]. Son una integración de los conceptos del modelo de objetos y el paralelismo, ya que al incluir en su creación un proceso, las labores de sincronización, comunicación y control de ejecución de los métodos de la clase pueden ejecutarse concurrentemente. Esto permite mantener una cierta simplicidad y modularidad, reforzando los conceptos de autonomía y autocontención de los objetos [28].

Los objetos pasivos son muy semejantes a aquéllos creados en programación secuencial. Constan de un conjunto de datos y métodos que pueden ser ejecutados concurrentemente.

En la práctica, en un programa paralelo orientado a objetos típico solo un pequeño número de objetos son activos, formando la estructura paralela del programa. La mayoría de los objetos por lo tanto son pasivos, existiendo como miembros o estructuras de datos manejadas por los objetos activos [62].

b) Modelos de programación.

La programación paralela se relaciona estrechamente con la naturaleza del problema a solucionar. En función de tal naturaleza, un programa se estructura siguiendo un modelo determinado. Son tres los modelos más utilizados: estructuración explícita en tareas paralelas, tareas estructuradas a partir de los datos, y sistemas de objetos comunicantes [25,28].

i) Tareas Paralelas.

El paralelismo de tareas se basa en la creación de hebras de control independientes, que incluyen elementos para su sincronización y comunicación. Un objeto se trata de una estructura que ejecuta el programa como una colección de tareas paralelas controladas de una manera explícita (Figura III.2) [28].

Existe una variedad de estructuras de sincronización y comunicación en diferentes lenguajes orientados a objetos paralelos. Se basan principalmente en el uso de bloques paralelos (*parbegin/parend*), primitivas de espera de un booleano, mecanismos de suspensión (*futures*), serialización del acceso a las instancias de una clase en particular, referencia de objetos y funciones miembros, y uso de primitivas explícitas para paso de mensajes [25].

ii) Paralelismo de datos. El modelo de colección distribuida.

El paralelismo de datos es el término utilizado para describir la aplicación en paralelo de un operador atómico o secuencial sobre un conjunto de datos. Se considera una extensión del concepto de función miembro de la programación orientada a objetos. Esto significa que en el modelo de paralelismo de datos una función miembro perteneciente a una clase tiene la capacidad de ejecutarse en paralelo sobre un conjunto de datos [25,48]. Un objeto realiza en paralelo tareas estructuradas respecto a los datos que manipulan (Figura III.3). Frecuentemente, el compilador utilizado se encarga de realizar los ajustes necesarios para organizar las operaciones sobre los conjuntos de datos definidos [28].

- **El modelo de colección distribuida.**

Partiendo del paralelismo de datos, el modelo de colección distribuida se basa en la creación de una estructura de datos basada en objetos, en forma de tipos de datos abstractos. Tales estructuras se arreglan y distribuyen entre diferentes procesadores. El modelo soporta la característica jerárquica de las abstracciones, por lo que permite la construcción de bibliotecas reusables y la implementación de programas distribuidos [49].

El modelo de colección distribuida tiene en general los siguientes componentes: colección, elemento, representantes de procesadores y distribución. Una colección es un conjunto homogéneo de elementos de una clase, los cuales se agrupan, lo que permite referenciarlos mediante un solo nombre de colección. Cada colección se asocia con una cierta regla de distribución y un conjunto de representantes computacionales activos o representantes de procesadores, los cuales pueden considerarse como procesadores virtuales en tiempo de ejecución. Los elementos de la colección se distribuyen entre los representantes de procesadores utilizando la regla de distribución, que determina la forma como los elementos de la colección se reparten entre los procesadores virtuales [49].

La jerarquía de abstracciones empleada en el modelo de colección distribuida no solo permite que los métodos de una colección puedan ser heredados por otra colección, sino que también es posible que cada elemento de una clase pueda heredar conocimiento de la estructura de la colección. La habilidad de una colección de describir y determinar la abstracción en particular de elementos da como consecuencia una gran flexibilidad en la construcción de bibliotecas genéricas reusables basadas en estructuras de datos distribuidas, como es el caso de las bibliotecas DAPPLE (*DAta-Parallel Programming Library for Education*), basada en C++, y cuya finalidad es operar con arreglos de datos con fines educativos [48,49].

En el modelo de colección distribuida cualquier mensaje enviado a un elemento de una colección es por defecto recibido, procesado y respondido por el procesador que posee ese elemento en particular. Para iniciar una acción paralela, el método de un elemento es invocado por la colección, lo que significa que ese método será aplicado a todos los elementos de la colección siguiendo un esquema

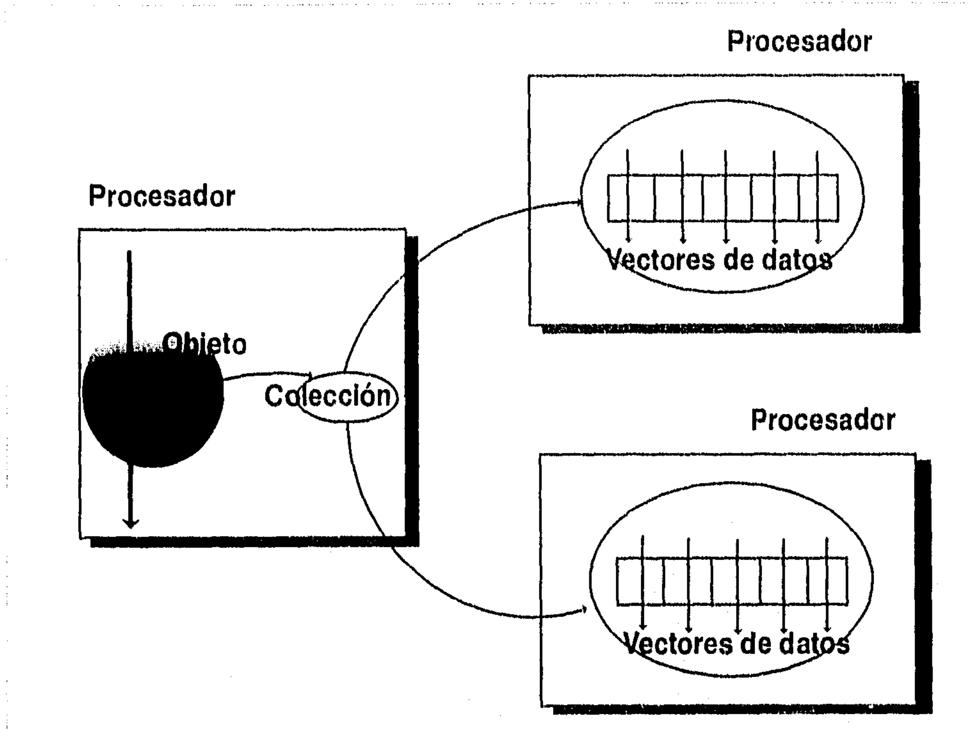


Figura III.3, Paralelismo de datos.

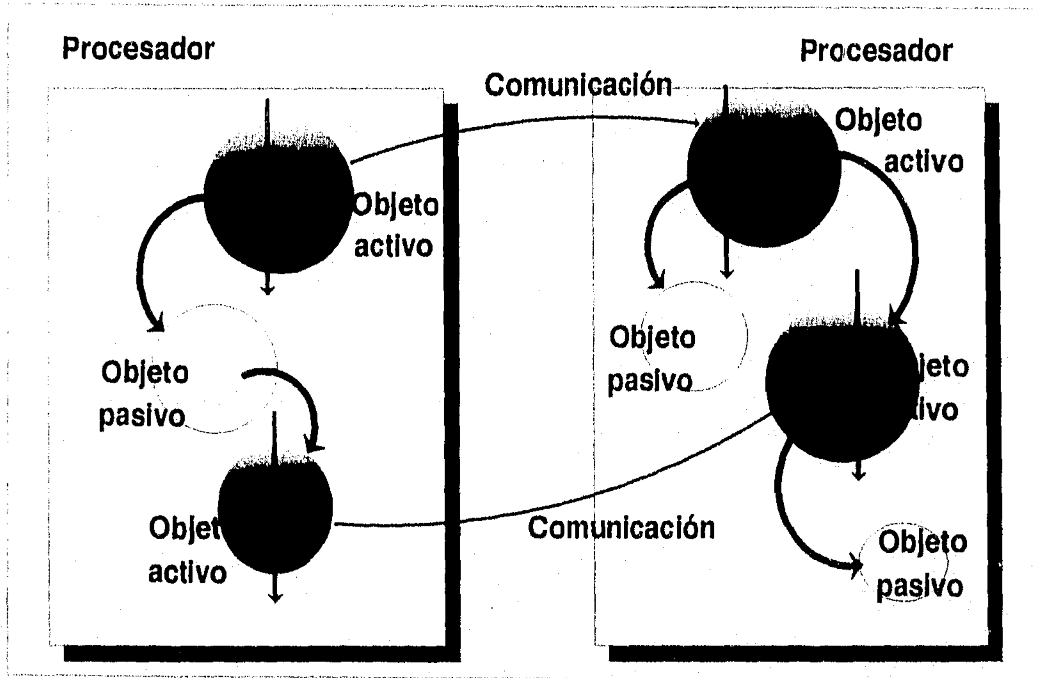


Figura III.4, Objetos comunicantes.

paralelo lógico o concurrente, esto es, que cada procesador aplicará tal método a los elementos que posee. En el caso que la distribución de elementos sea heterogénea entre los procesadores, se han incorporado operadores que permiten dinámicamente reestructurar la distribución en tiempo de ejecución del programa. Así, mientras se mantenga el número de elementos de todas las colecciones, la distribución puede variarse de fase en fase [49].

El paralelismo de datos y su expresión como una colección distribuida ha demostrado ser una herramienta de procesamiento de grandes cantidades de datos eficiente y poderosa. Tiene la ventaja de que su ejecución puede realizarse tanto en máquinas con memoria compartida o distribuida, con o sin esquema vectorial. Sin embargo, debido a la dinamicidad y los cambios que el sistema puede sufrir en tiempo de ejecución, requiere que su programación considere todos y cada uno de los estados del sistema. Además, el manejo del concepto de colección conjuntamente con el de clase dentro de una misma programación, hace que el modelo de datos paralelos sea en ocasiones muy complicado.

iii) Objetos comunicantes. El modelo de actores.

En el modelo de objetos comunicantes cada objeto representa una unidad de procesamiento autónoma. Especialmente, los objetos activos o actores representan la unidad de paralelismo del modelo [63]. Una aplicación se forma de un conjunto de objetos comunicantes que intercambian información activamente entre sí (Figura III.4). Este modelo permite la distinción en una forma más clara de los objetos activos y pasivos [14,28]. Esta organización basada en objetos activos y pasivos permite considerar al modelo de objetos comunicantes en escalas de granularidad media y gruesa [62].

El modelo de objetos comunicantes se basa en una máquina abstracta que consiste en una colección de elementos de procesamiento virtuales, cada uno soportando un objeto activo. Cada objeto activo cuenta con su propia hebra de control individual. El mapeo de los objetos activos a procesadores reales depende directamente del mecanismo de localización propio del sistema [62].

El paso de mensaje es el medio de interacción y coordinación entre objetos. Todo patrón de interacción (informar, requerir alguna

información o proceso, etc.) se somete mediante el concepto de paso de mensaje. Esto significa que al enviar un mensaje, un objeto emisor dispara la activación de un método de otro objeto receptor con los argumentos empacados en el propio mensaje [14].

Partiendo del modelo de objetos comunicantes, se han generado otros modelos de procesamiento paralelo orientado a objetos. Entre los más conocidos se encuentran el modelo de actores [51].

- **El modelo de actores.**

El modelo de actores se considera como la extensión más natural del modelo de objetos. Desde este punto de vista, cada objeto es un programa en miniatura, el cual se comunica con otros objetos mediante paso de mensajes. A cada uno de estos programas se les conoce como *actores*. Los actores u objetos activos son instancias de clases cuya implementación contiene un proceso con la capacidad de ejecutar una o más tareas. Tales tareas se definen dependiendo de los mensajes que el actor pueda recibir, esto es, utilizando una tarea por cada mensaje recibido. Además se definen otras tareas internas al actor, una por cada método activo. De esta forma, un actor es un objeto en movimiento [51].

Basado en el modelo de objetos comunicantes, el modelo de actores se ha desarrollado especialmente considerando la existencia de los dos tipos de objetos: activos y pasivos [III.A.2.a.*ii*]. Su desarrollo se ha planteado mediante una serie de conceptos que parten de los elementos de la programación secuencial tradicional hasta formar un paradigma de programación paralela orientada a objetos. De esta forma, se da la evolución del concepto de actor a partir de conceptos tales como programa, tipos de datos abstractos, objeto, y finalmente, actor [51].

Inicialmente, un programa se considera como un elemento monolítico, consistente en varias estructuras de datos y varios algoritmos [80], siguiendo una forma del tipo:

programa = estructuras de datos + algoritmos

Sin embargo, es notable que esta definición permite una libertad demasiado amplia en la programación, dando como

resultado varios problemas en el desarrollo de programas. Para eliminar cierta cantidad de problemas relacionados con esta definición de programa, se generó posteriormente el concepto de *tipos de datos abstractos (TDA)*, el cual propone mediante el mecanismo de encapsulación una forma de organizar y controlar el acceso a los datos y algoritmos de un programa. De esta forma, un programa basado en tipos de datos abstractos se puede considerar de la siguiente manera [51]:

TDA = encapsulación(estructuras de datos + algoritmos)

La encapsulación se realiza en una variedad de formas dependiendo del lenguaje de programación. En todos los casos, el concepto de ocultamiento de información se utiliza para prevenir el acceso a los datos de un programa por parte de otros programas. Se considera entonces que los datos y algoritmos de un programa son *visibles* para otro programa cuando éste tiene acceso a los datos y algoritmos de aquél. La forma de acceso se determina en una parte del programa conocida como su *interface*. Por otro lado, los datos y algoritmos de un programa son *invisibles* para otro programa cuando no se cuenta con acceso directo o son opacos para cualquier otro programa [78].

La invisibilidad de los datos de un programa, así como la separación de sus partes en interface e implementación, es lo que distingue a los tipos de datos abstractos de los tipos simples de datos. De esta forma, la estructura física de un tipo se oculta de la visión de los usuarios de tal tipo [51,78].

Por otro lado, con el fin de crear conjuntos de procedimientos e implementaciones de tipos de datos, se genera el concepto de herencia como mecanismo de reusabilidad de código [II.A.2.d.*i*]. Al aplicar este mecanismo a una descripción realizada mediante tipos de datos abstractos, se generan los conceptos de clase y objeto [II.A.1.a]. Un objeto es una instancia de las clases que forman la jerarquía, heredando los métodos de una o varias clases. De esta manera, el objeto se representa como una extensión de la idea de tipo de dato abstracto al añadirle la propiedad de herencia [51]:

Objeto = TDA + herencia

El mecanismo de herencia en el modelo de objetos es la clave que permite la reusabilidad de código, ya que un objeto hereda los métodos de su clase cuando se realiza la instanciación, y a su vez, la herencia también puede ser acumulativa si se considera una organización de clases mediante una jerarquía de herencia [II.A.2.d,51].

A partir del concepto de objeto, es más claro introducir el concepto de actor como un objeto al que se añade un proceso. Así, un actor es un objeto en ejecución [51,14].

Actor = ***Objeto*** + ***proceso***

El concepto de clase en el modelo de actores se plantea de igual manera al modelo de objetos: una clase se considera como una agrupación de programas del mismo tipo, es decir, una colección de métodos y tipos de datos encapsulados en forma de objetos que pertenecen a la clase, con el fin de controlar el crecimiento de la complejidad [II.A.1.a]. Un actor es aun una instancia de una clase [62]. Su descripción como objeto se realiza de igual manera que en el modelo de objetos, pero una vez instanciado, el objeto se activa mediante su proceso interno, lo que lo convierte en un actor capaz de realizar acciones propias o en respuesta de mensajes procedentes de otros objetos. Una actor es un objeto en movimiento [51,14].

El modelo de actores se amplía aun más dentro de una arquitectura paralela: un actor puede considerarse como una pequeña computadora completa, con memoria e instrucciones, la cual se encuentra contenida en un procesador físico, ejecutando sus algoritmos paralelamente con otros actores que se encuentran en otros procesadores. Desde este punto de vista, un actor es una extensión al modelo de objetos, ya que representan unidades de programación autocontenidas comunicándose entre sí mediante envío o recepción de mensajes [51]. El concepto de actor en una arquitectura paralela es entonces el resultado de la evolución a partir del concepto inicial de programa, hasta el modelo de actores.

En arquitecturas paralelas, el modelo de actores tiene la capacidad de permitir la ejecución de actores en computadoras con organización de memoria compartida y distribuida. En cierta medida, también es permisible su uso en sistemas computacionales tradicionales de tipo serial [51].

c) Incorporación del paralelismo a la programación orientada a objetos.

Un tema importante para el desarrollo de la programación paralela orientada a objetos es la manera de incorporar las características de paralelismo a la programación orientada a objetos o viceversa. Se reconoce que principalmente son dos los mecanismos que pueden ser utilizados: el paralelismo puede introducirse a un programa a través de bibliotecas que encapsulen operaciones del paralelismo o, por otro lado, a través de la introducción de nuevos elementos o construcciones paralelas al lenguaje, esto es, el paralelismo se incluye ya sea mediante el uso de bibliotecas de clases o como una extensión del lenguaje [25].

A continuación, se analizan las conveniencias e inconveniencias de cada uno de tales mecanismos.

i) Bibliotecas de clases.

El uso de bibliotecas de clases propone la introducción del paralelismo mediante el uso de clases que encapsulen operaciones paralelas. Esto tiene sentido ya que un aspecto importante en el desarrollo de la programación orientada a objetos es facilitar el diseño e implementación de bibliotecas reusables de clases. De hecho, los lenguajes orientados a objetos son un ejemplo importante de aquellos lenguajes con características especialmente desarrolladas para soportar el diseño e implementación de bibliotecas mediante clases [25,55,9].

Por otro lado, se debe tomar en cuenta que los lenguajes orientados a objetos son relativamente nuevos, sujetos actualmente a un proceso de estandarización. Sería prematuro considerar en un momento realizar una extensión de estos lenguajes. Además, aún no se tiene claro un consenso generalizado entre vendedores y usuarios sobre el conjunto de extensiones del lenguaje a realizar, por lo que los vendedores y diseñadores de *software* se encuentran renuentes hacia extensiones de lenguajes, ya que esto los compromete a dar un soporte a largo plazo sin tener claro una retribución [25].

Otra dificultad de las extensiones del lenguaje que llevan hacia el uso de bibliotecas de clases, es el diseño de compiladores para tales extensiones. Los compiladores son en la generalidad programas complejos. Cualquier extensión tiende a aumentar la

complejidad, ya que al añadir características de paralelismo, un nuevo compilador deberá ser compatible con la versión no paralelizada y, además, ser robusto en cuanto a sus nuevas características. En caso contrario, los programadores se encontrarían renuentes en aceptar el nuevo compilador. Aun más, los cambios propuestos en un momento dado como extensiones de un lenguaje orientado a objetos pueden ser inicialmente desarrollados, implementados y probados mediante bibliotecas de clases antes de ser aplicados a un compilador. Una biblioteca es, en general, más fácil de modificar que un compilador [25].

ii) Extensiones del Lenguaje.

En cuanto a las extensiones del lenguaje, éstas proponen la introducción directa de nuevos constructores paralelos en un lenguaje orientado a objetos, basándose principalmente en la idea que el paralelismo y la secuencialidad son dos aspectos de los lenguajes paralelos de igual importancia. Si el lenguaje posee la capacidad de expresar la composición secuencial, se considera importante que también contenga una expresión para la composición paralela. Esta idea se extiende en la programación paralela orientada a objetos, ya que el paralelismo no puede considerarse propiamente como un tipo [25,55].

El uso de extensiones del lenguaje cuenta además con una ventaja en cuanto a la expresividad dentro del modelo de objetos, ya que los programas paralelos realizados en base a construcciones propias del lenguaje son más fáciles de leer y entender que un programa basado en bibliotecas [25]. Es notable que en aquellos lenguajes basados principalmente en bibliotecas, el comportamiento de un programa no puede explicarse directamente dentro del contexto del propio programa, lo que obstaculiza el entendimiento y, por lo tanto, la reusabilidad, ya que un programa poco entendible es susceptible a no ser reutilizado.

El uso de un compilador permite considerar las características de paralelismo como parte del lenguaje, ofreciendo además las ventaja de verificación de tipos y optimización de código durante la compilación, lo que no es posible hacer utilizando bibliotecas de clases [25].

Como conclusión de esta discusión, se puede considerar que de realizarse la incorporación del paralelismo a los lenguajes orientados a objetos en un futuro, ésta observará como un proceso evolutivo, el cual se iniciará mediante el uso de bibliotecas de clases y preprocesadores que incorporen operaciones paralelas y directivas en el código de máquina y, posteriormente, mediante un acuerdo entre usuarios, se incluirán tales características paralelas en compiladores comerciales como extensiones del lenguaje [25].

C. Análisis de ventajas y desventajas.

El objetivo principal de conjuntar los paradigmas de la programación orientada a objetos y las arquitecturas paralelas es, en primera instancia, obtener las ventajas con que ambas tecnologías cuentan, además de intentar reducir mediante su unión las desventajas propias de cada una.

En resumen, las ventajas que aportan ambas tecnologías, cada una por su parte, a la programación paralela orientada a objetos son:

Arquitecturas Paralelas [I.B.2].

- Aumento en la velocidad.
- Eficiencia en el procesamiento.
- Simplicidad en la expresión de modelos.
- Aplicaciones.
- Tamaño y costo.

Orientación a Objetos [II.B.2].

- Uniformidad.
- Entendimiento.
- Flexibilidad y mantenimiento.
- Estabilidad.
- Reusabilidad.
- Tamaño del código.
- Especificación y verificación.

Sin embargo, la unión de ambas tendencias ha dado lugar a una serie de ventajas y desventajas complementarias propias de la tecnología orientada a objetos paralela. La presente sección se encarga de considerar y resumir las más importantes que hasta el momento se han presentado en trabajos sobre programación paralela orientada a objetos.

1. Ventajas.

Debido a sus capacidades de abstracción y encapsulación, la programación paralela orientada a objetos permite la descomposición de un programa en pequeños objetos o módulos cooperativos, que pueden ejecutarse eficientemente mediante arquitecturas multiprocesador. Utilizando el paso de mensaje como mecanismo de comunicación entre objetos, hace que sea apropiado para arquitecturas con memoria compartida y distribuida. Tales características dan como resultado ventajas como las siguientes:

a) Alto nivel de abstracción.

La programación se realiza considerando la interacción entre diferentes tipos de entidades conceptuales que describen el problema. Esto significa que el programador enfoca su atención propiamente en el papel que las abstracciones desarrollan relacionándose entre sí para completar una tarea. Se trata de abstracciones de alto nivel que cooperan para lograr un objetivo común, eliminando desde el punto de vista del programador la necesidad de preocuparse por detalles en la implementación [14].

b) Descomposición implícita de actividades paralelas.

Las unidades de programación se encuentran claramente identificadas al descomponer un programa en un conjunto de objetos interactivos. Cada objeto puede representar una unidad autónoma a partir de su ejecución paralela en procesadores diferentes. El paralelismo interobjeto es implícito en este modelo. Por otra parte, mediante la concurrencia a nivel intraobjeto es posible obtener una mayor capacidad, ya que el modelo permite a cada objeto efectuar operaciones concurrentes a nivel interno [14,63].

c) Mayor nivel de encapsulación.

El modelo paralelo de objetos propone la creación de objetos en diferentes procesadores que se comunican entre si exclusivamente mediante mensajes, lo que hace que el uso de los métodos de cada objeto sea más seguro debido a que todo requerimiento al objeto se realiza mediante su interface. Esto asegura aun más la característica de encapsulación de cada objeto. Prácticamente, en un programa paralelo cada objeto se comporta como una unidad autónoma que reconoce la existencia de otros objetos a través de su interface [63].

d) Transparencia en sincronización.

Conjuntamente con el uso de abstracciones de alto nivel se plantea una transparencia en sincronización entre objetos. La sincronización entre actividades paralelas, ya sean operaciones dentro del mismo objeto o en objetos distintos, se considera únicamente durante la etapa diseño de las clases representativas de los objetos involucrados. No es necesario posteriormente para el programador realizar consideraciones de sincronización a bajo nivel [14].

e) Localización y autocontenido.

Los objetos son entidades autocontenidas que conjuntan un cierto grado de conocimiento, acciones y recursos. En el modelo paralelo orientado a objetos, el objeto es la unidad de acción dentro de un programa. Su localización en diferentes procesadores permite su ejecución paralela, dándoles características espacio-temporales, esto es, un actor se convierte en un elemento de procesamiento que cuenta individualmente con un tiempo de ejecución en un procesador determinado [14].

f) Diseño basado en paralelización incremental.

A partir de consideraciones sobre la localización de los objetos que actúan entre sí dentro de un sistema multiprocesador, se desarrolla la característica de diseño basado en paralelización incremental. Esto no es más que la capacidad del sistema de permitir que un diseño realizado en un cierto número inicial de procesadores pueda ser utilizado en configuraciones con mayor o menor cantidad de procesadores. En general se acostumbra realizar el diseño para un solo procesador, migrándolo después a un sistema de múltiples procesadores, lo que aumenta su desempeño [40,41]. Esta forma de diseño puede ser utilizada de igual manera para los programas paralelos orientados a objetos.

La capacidad de paralelización incremental permite a su vez la realización de aplicaciones paralelas a partir de aplicaciones seriales ya existentes, debido a que las aplicaciones paralelas pueden considerarse como variantes de las aplicaciones seriales [63].

g) Dinamicidad.

La característica de dinamicidad en un sistema paralelo orientado a

objetos se refiere a la capacidad de los objetos de ser creados y recuperados dinámicamente. Además, los objetos cuentan con capacidades tales como la adquisición de nuevo conocimiento y cambio de su comportamiento respecto a la información que reciban en tiempo de ejecución. La característica de dinamicidad permite a un conjunto organizado de objetos ser dinámicamente reconfigurado, dando lugar a que nuevos objetos pueden ser creados atendiendo a la necesidad de recursos durante la ejecución de un programa [14].

h) Multigranularidad.

Los objetos de diversas granularidades (en cuanto a tamaño, cantidad de conocimiento, concurrencia interna) pueden coexistir y cooperar entre sí en la misma arquitectura paralela, debido a la uniformidad en los mecanismos de comunicación e interacción y a la encapsulación de los objetos. Los objetos permiten la modelación de entidades de granularidad muy fina (como por ejemplo, neuronas), o grandes componentes de una granularidad media a gruesa (como es el caso de los sistemas multi-agente). Esto significa que la capacidad de multigranularidad permite la descomposición de un problema complejo en varias abstracciones de diferente nivel y tamaño, pero capaces de interactuar entre sí, lo que brinda hasta cierto punto una ayuda al programador durante el diseño de un programa [14,63].

i) Aplicaciones.

La programación concurrente y paralela orientada a objetos tiende a ser aplicada en un número creciente de campos, como son sistemas operativos distribuidos, inteligencia artificial (distribuida), simulación, bases de datos distribuidas, sistemas de información de oficinas, sistemas en tiempo real y control de procesos. Se han logrado resultados significativos en áreas como análisis de texto y música por computadora [14].

En el campo de la inteligencia artificial, el modelo se ajusta especialmente a una nueva área conocida como inteligencia artificial distribuida, la cual propone la resolución de problemas en forma de una colección de *agentes* que coordinan su conocimiento y comportamiento. Los objetos paralelos se presentan como una buena base para la construcción de agentes inteligentes [14].

Otro campo de aplicación se ha desarrollado para soportar modelos y sistemas alternativos de programación. Estos se conocen como modelos de

programación *metafóricos*, y son el resultado de transferir teorías científicas a partir del mundo real para dar solución mediante modelos computacionales alternativos. Desarrollos significativos se han logrado en el campo de las redes neuronales como una abstracción de la neurofisiología. Otros desarrollos incluyen abstracciones de la química, genética, algoritmos, etología, economía, simulación de partículas y, en general, sistemas auto-organizados [14,59].

2. Desventajas.

Aun cuando el modelo paralelo orientado a objetos propone el uso de objetos en una forma natural, a la vez plantea ciertas desventajas propias generadas a partir de la conjunción de ambas tecnologías. Las principales desventajas observadas en trabajos sobre este tema son las siguientes:

a) Dependencia de datos.

Un problema frecuentemente observado en los modelos paralelos orientados a objetos, es el relacionado con la dependencia de datos. Este problema se refiere a que la ejecución paralela de objetos debe limitarse a aquellas aplicaciones donde no existe dependencia de datos, es decir, se requiere la verificación de cualquier dependencia de datos mediante un análisis interprocedural, dado que todos los accesos a los datos toman la forma de procedimientos o métodos. Este análisis es difícil de realizar debido a que los efectos procedurales colaterales forzan un análisis basado en una serie de suposiciones basadas en consideraciones conservadoras, dando en general como resultado información poco precisa sobre la dependencia. Tal es el caso de considerar que una vez logrado el paralelismo, los efectos colaterales de llamar métodos de un objeto son limitados al objeto mismo, y si además que las llamadas a los métodos no utilizan parámetros por referencia, lo que hace posible encapsular completamente la información acerca de las interacciones entre métodos, dando como resultado un análisis interprocedural trivial, y solucionando el problema de la dependencia de datos de una manera parcial [51].

b) Identificación de objetos.

Es posible considerar que la dificultad de identificación de objetos del paralelismo orientado a objetos es un problema "heredado" de la programación orientada a objetos secuencial. Es verdad que una vez determinados los objetos dentro del dominio de un problema, es

relativamente sencillo proponer una forma de paralelización entre ellos. Sin embargo, no existen reglas determinadas para identificar los objetos, lo que deja esa labor como responsabilidad del programador [51,I.B.3.a,II.B.3.a].

El desarrollo de metodologías orientadas a objetos por parte de varios autores han planteado soluciones con el fin de aminorar este problema [81,55,9]. Sin embargo, la dependencia del diseño respecto a tales métodos [II.B.3.c] no ha dado aún una solución definitiva.

c) Comunicación y sincronización asociada al paralelismo.

El principal problema que el paralelismo orientado a objetos "hereda" por parte del paralelismo, es el problema de comunicación y sincronización. Es notable que para poder aprovechar las ventajas respecto a poder computacional que ofrece una arquitectura paralela con memoria distribuida, es necesario la existencia de un *software* eficiente. Sin embargo, la realización de *software* paralelo es un trabajo difícil y laborioso debido a la ausencia de una memoria global entre los procesadores [I.B.3.c,I.B.3.e]. El programador debe distribuir manualmente el procesamiento y los datos entre los procesadores, manejando explícitamente la comunicación entre ellos [5,III.B.1.f]. Aun más, en el paralelismo orientado a objetos, un objeto puede ejecutar sus métodos concurrentemente, comunicándose activamente con otros objetos en el mismo procesador o en otros [III.A.2.a.*ii*], lo que no propone de ninguna manera una resolución definitiva a los problemas de sincronización asociados con el paralelismo [51]. Debido a que los objetos se encuentran distribuidos, el programador aun debe encargarse de la sincronización y compartición de recursos [81]. Esto se complica al tratar de definir especificaciones para la sincronización heredables [III.B.2.g].

d) *Deadlock* recursivo de los sistemas concurrentes

La recursión es una técnica de programación poderosa presente en muchos lenguajes de programación en la actualidad. Desafortunadamente, esta técnica puede provocar un problema de *deadlock* [I.A.1.f] en los sistemas concurrentes orientados a objetos, debido a que en tales sistemas un método que modifica el estado de un objeto al que pertenece no puede llamarse a sí mismo recursivamente. Esto ocurre debido a que el objeto, como emisor, se bloquea esperando que su llamada sea atendida. El método invocado nunca se ejecuta, debido a que el mismo objeto, ahora como receptor, se encuentra bloqueado. En general, el *deadlock* recursivo sucede siempre que un objeto se bloquea esperando un resultado que requiere la invocación de métodos adicionales del mismo objeto [10].

La combinación de tres factores conducen a un *deadlock* recursivo [10]:

- Un objeto debe mantenerse en un estado de espera. En sistemas con, cuando mucho, una sola hebra de control por objeto, cada objeto siempre presenta un estado de espera implícito debido a la concurrencia.

- El objeto realiza una llamada a otro objeto (posiblemente a sí mismo) que lo forza a esperar una respuesta.

- Finalmente, la llamada contiene una componente que requiere al objeto bloqueado que generó la llamada, formando un círculo.

Cuando estas tres condiciones se cumplen, todos los objetos involucrados en el círculo se encuentran bloqueados esperando una respuesta, sin lograr ningún progreso en su ejecución, esto es, se encuentran en *deadlock*.

Se han desarrollado varios trabajos relacionados con este problema, a fin de resolverlo o, cuando menos, detectarlo. Algunos de los más representativos son la recursión directa involucrando un solo objeto, proveer procedimientos no asociados con el objeto en adición de los métodos, el uso de constructores para la aceptación selectiva de mensajes, la especificación de "actores insensitivos" como reemplazo de los objetos en los sistemas de actores, la detección de *deadlock* como se utiliza en los sistemas distribuidos y de bases de datos distribuidas [10].

e) Desempeño.

El desempeño de los sistemas orientados a objetos se presenta como un problema grave, sobre todo en lo que se refiere a sistemas paralelos. Es verdad que el modelo orientado a objetos no hace disponible una potencia de procesamiento considerable, si se compara con un sistema de programación estructurado, esto es, un programa orientado a objetos en general se ejecuta más lentamente [II.B.3.b]. Esta relación se conserva al hacer comparación entre las velocidades de ejecución de un programa paralelo orientado a objetos y un programa paralelo. Desde este punto de vista, no existe una ganancia representativa de incorporar la orientación a objetos a un sistema paralelo [81,63].

Sin embargo, existe otra forma de considerar el desempeño de los sistemas paralelos orientados a objetos. En realidad, la idea y desarrollo de las arquitecturas paralelas surge de una necesidad de mayor velocidad de ejecución que aquella desarrollada por un sistema secuencial [I.B.2.a]. A partir de esto, es posible considerar un punto de vista diferente respecto al desempeño de un sistema paralelo orientado a objetos. Es verdad que el desempeño de un sistema paralelo es mayor que un sistema paralelo que utiliza objetos, pero este último tiene la capacidad de sobrepasar por mucho el desempeño de un sistema secuencial, es decir, comparado con un programa secuencial, un programa paralelo orientado a objetos puede ejecutarse en un tiempo menor.

Aun considerando lo anterior, se han desarrollado trabajos alrededor del aumento de velocidad en el procesamiento de los sistemas paralelos orientados a objetos. Se han propuesto varias soluciones, que van desde la incorporación de un mecanismo de control de recursos [63], hasta considerar aceptable la violación del principio de encapsulación de los objetos paralelos a nivel físico [81], durante la compilación o en tiempo de ejecución, a fin de maximizar el desempeño.

f) Herencia estática y dinámica.

Una ventaja sobresaliente de la programación orientada a objetos es su capacidad de manejar programas reusables y modulares mediante el mecanismo de herencia [II.B.2.e]. El manejo de la herencia puede realizarse desde una aproximación totalmente estática o dinámica.

En herencia estática, el compilador copia el código heredado dentro del código del programa que hereda. Esta aproximación facilita grandemente la labor del programador, ya que soporta una forma sencilla de reuso. Esto es simple y eficiente, pero tiene la desventaja que desperdicia memoria al replicar código [81].

La herencia dinámica es una aproximación más flexible pero su ejecución es más lenta, ya que durante el tiempo de ejecución el sistema determina el método heredado apropiado y genera la hebra de control. La herencia dinámica es especialmente útil cuando se requiere métodos compartidos [II.B.3.d]. Sin embargo, en un ambiente concurrente, la herencia dinámica presenta varios problemas, relacionados justamente con la forma en que el objeto se representa en la memoria. Muchos lenguajes, con el fin de ahorrar espacio en memoria, determinan a un objeto únicamente mediante sus atributos de estado, implementando los métodos

como versiones de procedimientos en un espacio de memoria común entre los diferentes objetos de una clase [II.B.3.d]. En este caso, si dentro de un programa un par de objetos concurrentes pretender hacer uso de un método es necesario que accedan el mismo código como un recurso compartido, lo que aumenta el tiempo de ejecución del programa, además de generar complicaciones aun mayores en la implementación de la herencia múltiple [81].

g) Herencia anómala.

Otro problema relacionado con la herencia, se refiere al uso de la herencia para reutilizar especificaciones de sincronización al incorporar características del paralelismo, ya que no siempre funcionan correctamente. A este problema se le da el nombre de herencia anómala. Su solución, a fin de encontrar un esquema de sincronización expresivo, genérico y reusable, es todavía tema de investigación [14].

Algunos resultados acerca de esta investigación marcan que las especificaciones de sincronización deben ser suficientemente abstractas, fuera de definiciones de métodos, y eficientes, a fin de obtener una sincronización correcta que coordine actividades paralelas para una ejecución eficiente, consistente y predecible. Una coordinación excesiva reduce la concurrencia, pero muy pequeña conduce a efectos indeseables del no determinismo [81,14].

El problema parece surgir en que la herencia complica la sincronización. Cuando una subclase hereda atributos de una clase base, es necesario en ocasiones redefinir las características de sincronización del método heredado. Este único aspecto parece ser la dificultad más grande de integrar paralelismo a lenguajes orientados a objetos [81].

D. Un lenguaje paralelo orientado a objetos: INMOS Parallel C++, de Glockenspiel.

Anteriormente se ha discutido una manera de conjuntar el paralelismo y la programación orientada a objetos, incorporando en un lenguaje orientado a objetos las características principales del paralelismo [I.B.1] mediante bibliotecas de clases [III.A,III.B.2.c.*i*]. Para observar el resultado de esta unión se ha realizado una implementación utilizando como base una arquitectura paralela basada en *transputers*, la cual incluye un lenguaje básico orientado a objetos: el INMOS *Parallel C++* (o INMOS C++ paralelo) de Glockenspiel.

El INMOS C++ paralelo es un conjunto de herramientas de *software* y una biblioteca de clases que realiza las funciones de un compilador orientado a objetos basado en el C++ versión 2.0 de AT&T. Su desarrollo para sistemas basados en *transputers* fue realizado por la compañía Glockenspiel para INMOS [7,22]. Este conjunto de herramientas realizan la traducción de código desarrollado en C++ a código en C, utilizando el compilador ANSI C paralelo del propio INMOS, lo que permite un manejo del paralelismo y comunicación mediante llamadas a bibliotecas de funciones del lenguaje ANSI C paralelo, haciendo posible la ejecución de funciones en paralelo (que pueden ser consideradas como métodos miembros de una clase) y a su vez comunicación entre procesos mediante rutinas de manejo de canales y semáforos. Esto último significa que cuenta también con soporte para comunicación en esquemas de memoria compartida [7,22]. Sin embargo, debido a que se trata de una versión experimental de un lenguaje paralelo orientado a objetos, cuenta con ciertas deficiencias importantes como es el caso de la falta de soporte a la genericidad, polimorfismo, sobrecarga de operadores y manejo de excepciones, que no se encuentran disponibles en esta versión [22].

Utilizando este lenguaje, se proponen nuevas interfaces e implementaciones para expresar las características del paralelismo, ya sea mediante propiedades de la arquitectura paralela o en forma de clases expresadas en C++ paralelo, a partir de las siguientes consideraciones:

- Las interfaces se realizan como parte del modelo de objetos comunicantes u objetos activos. El paralelismo mediante objetos activos es implícita en la semántica del modelo. La concurrencia existe a nivel de métodos de cada objeto y entre objetos que comparten un mismo procesador. La comunicación entre objetos se realiza mediante canales de comunicación [III.B.2.a.*ii*].

- El conjunto de interfaces e implementaciones se expresan en forma de bibliotecas de clases. Las interfaces en esta aproximación se presentan como bibliotecas de clases en C++ [III.B.2.c.*i*]. Una extensión de C++ fue considerada, pero rechazada a favor de las bibliotecas de clases debido a la falta de una versión definitiva que considere un soporte completo a la arquitectura con que se cuenta y, además, se trató de maximizar y conservar la inversión en cuanto a herramientas, código y experiencia acumulada sobre el sistema, sin limitar de manera significativa la expresividad.

- Múltiples niveles de abstracción. Para hacer que un modelo sea reutilizable, su estructura debe considerarse lo más simple posible a fin de que el usuario pueda entender el código a reusar en forma sencilla. Esto significa que una estructura a partir de abstracciones simples aunque diferentes en cuanto a tipo y tamaño resulta atractiva a fin de plantear y expresar relaciones claras entre los

objetos [III.C.1.e,III.C.1.h]. De hecho, esta aproximación contempla la implementación de un modelo que encapsule primitivas de paralelización y cuente, a la vez, con elementos básicos en la construcción de objetos en varios niveles de abstracción. Desafortunadamente, esto puede llevar al desarrollo de interfaces confusas y difíciles de mantener, dependiendo de las abstracciones involucradas [63].

- Paralelización incremental. Para facilitar la generación de aplicaciones paralelas es posible considerar la nueva aplicación como una variante a partir de aplicaciones secuenciales o paralelas ya existentes. Esta actividad es común en el desarrollo de aplicaciones paralelas dentro arquitecturas basadas en *transputer* [III.C.1.f]

Muchas de las decisiones hechas en el desarrollo de estas características involucran consideraciones sobre las necesidades y requerimientos de la arquitectura paralela con la que se cuenta para su desarrollo, y a las preferencias de la comunidad potencial de usuarios. De esta forma, la expresión y manejo del paralelismo en la arquitectura *transputer*, utilizando la versión de C++ paralelo, se basa en el modelo de objetos comunicantes, considerando las características del paralelismo mediante dos modelos incluidos en cada objeto activo, desde los puntos de vista interobjeto e intraobjeto [III.B.2.a.*i*].

1. Paralelismo.

a) Modelo Interobjeto.

Considerando a un objeto activo como una abstracción o modelo encapsulado con una hebra de control propia que se ejecuta dentro de un procesador [III.B.2.b.*iii*], las características de expresión del paralelismo se pueden representar entre objetos dentro de la plataforma paralela basada en *transputers*, como se explica a continuación.

La arquitectura *transputer* cuenta con la capacidad de que su programación puede ser configurada a fin de ejecutarse en un determinado arreglo físico de procesadores. La configuración del programa se refiere a la acción de asignación de unidades de procesamiento independientes que se comunican entre sí dentro de una red específica de procesadores [40,41]. La asignación se realiza a partir de una descripción de la configuración deseada utilizando un lenguaje de configuración mediante el cual se genera un archivo ejecutable que puede ser cargado directamente a la red de *transputers*. De esta forma, los elementos del *software* y *hardware* del programa son independientemente definidos y relacionados en la

configuración a través de un mapeo que se encarga de asignar los módulos de *software* previamente compilados y ligados a procesadores específicos y colocar canales de entrada y salida a las ligas físicas del *transputer* [40].

A partir de esta capacidad de la arquitectura *transputer*, se ha considerado un modelo de objetos activos basado en una máquina abstracta consistente en la descripción de la configuración, es decir, en una colección interconectada de elementos virtuales de procesamiento soportando cada objeto activo, el cual cuenta con una sola hebra de control, comunicándose entre si mediante mensajes [62]. Al mapear cada procesador virtual a cada procesador real se asegura la existencia paralela de cada objeto activo, es decir, se cuenta con un paralelismo físico entre objetos activos en diferentes procesadores. De esta forma, la expresión del paralelismo entre objetos se basa principalmente en el lenguaje de configuración de la arquitectura *transputer*, considerando cada objeto activo como un módulo de *software* propiamente compilado y ligado, como se puede apreciar en el siguiente ejemplo:

```
/* Descripción de los procesadores físicos */
T805 (memory = 1M) master_processor;
T805 (memory = 1M) slave_processor;

/* Conexiones entre los procesadores físicos */
connect master_processor.link[0], host;
connect master_processor.link[2], slave_processor.link[1];

/* Descripción de procesadores virtuales */
process (stacksize = 20k, heapsize = 20k,
     interface(input in, output out, input fromslave,output
toslave)) master;

process (stacksize = 20k, heapsize = 20k,
     interface(input frommaster, output tomaster)) slave;

/* Conexiones lógicas al host */
input from_host;
output to_host;

/* Canales entre procesadores virtuales */
connect master.in, from_host;
connect master.out, to_host;
connect master.fromslave, slave.tomaster;
connect master.toslave, slave.frommaster;


/* Mapeo de elementos virtuales a elementos físicos */
use "master.lku" for master;
use "slave.lku" for slave;
```

```
place master on master_processor;
place slave on slave_processor;

place to_host   on host;
place from_host on host;

place master.in         on master_processor.link[0];
place master.out        on master_processor.link[0];
place master.fromslave on master_processor.link[2];
place master.toslave    on master_processor.link[2];
place slave.frommaster on slave_processor.link[1];
place slave.tomaster    on slave_processor.link[1];
```

En este ejemplo los objetos `master` y `slave` se colocan en dos procesadores reales (`master_processor` y `slave_processor`, respectivamente) lo que permite su ejecución paralela (Figura III.5). La comunicación entre ellos se basa en canales que se introducen en el ANSI C paralelo como un nuevo tipo de dato **`Channel`**, y se mapea dentro de la descripción a ligas físicas de los procesadores. Su creación y control se discute por separado como característica del paralelismo en una sección subsecuente.

Por otro lado, aun cuando la descripción de configuración representa una expresión del paralelismo entre objetos, también presenta limitaciones en cuanto a contar con suficientes elementos físicos de procesamiento para soportar cada elemento virtual de procesamiento de los módulos de *software*. Para solucionar esto, es posible aprovechar la capacidad de repartir y reordenar los módulos de *software* entre los procesadores de la arquitectura, permitiendo una ejecución concurrente entre objetos activos y pasivos dentro del mismo procesador.

b) Modelo Intraobjeto.

El manejo del paralelismo a nivel interno de un objeto activo se expresa como la ejecución concurrente de sus funciones miembro en paralelo. Debido a las características de la arquitectura basada en *transputers*, la ejecución concurrente de tareas paralelas puede realizarse de manera semejante en objetos pasivos, ya que el lenguaje INMOS C++ paralelo cuenta con bibliotecas de control y manejo del paralelismo como parte del lenguaje ANSI C paralelo para la inicialización y ejecución de procesos concurrentes [40,7,22].

De forma semejante a los canales, para el manejo del paralelismo entre funciones se introduce en el ANSI C paralelo un nuevo tipo de dato

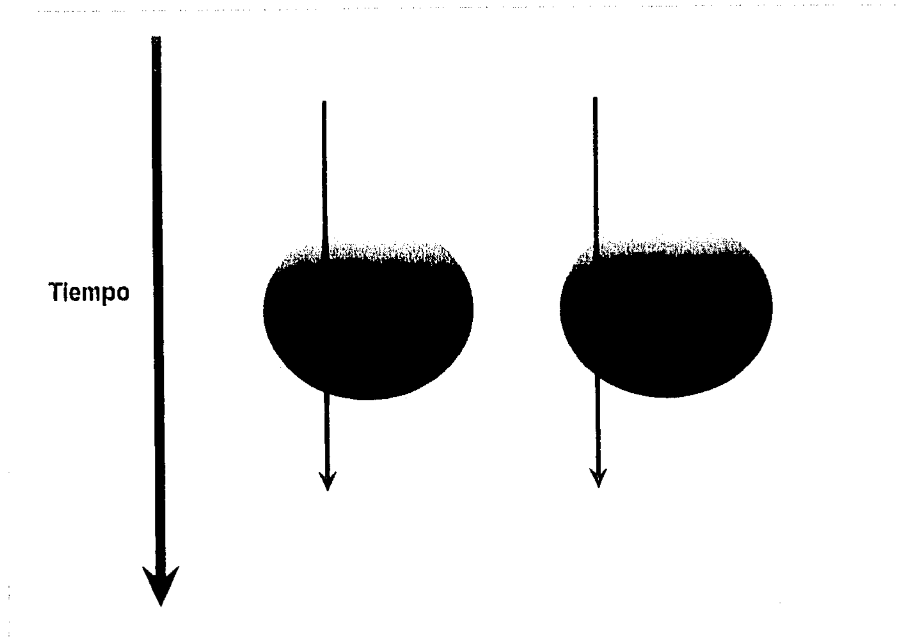


Figura III.5, Paralelismo interobjeto.

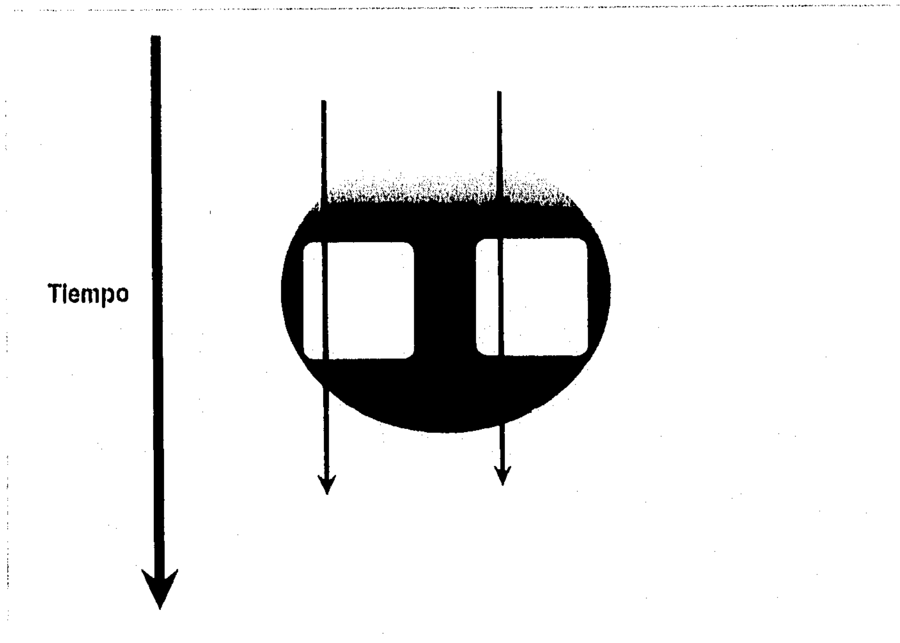


Figura III.6, Paralelismo intraobjeto.

conocido como **Process**. Este no es otra cosa que una estructura que mantiene información acerca de cada función a ejecutarse concurrentemente. Los procesos concurrentes se definen de la misma forma que las funciones regulares de C, excepto que deben contar con un primer parámetro fijo, el cual es declarado como un apuntador a su propia estructura **Process** (Figura III.6). Los parámetros de la función pueden determinarse después del apuntador a proceso en la forma convencional [40].

Considerando esta característica del paralelismo entre funciones, basándose en las bibliotecas del ANSI C paralelo es posible ejecutar funciones concurrentemente, las cuales pueden ser métodos o miembros de una clase [7]. La propuesta de la interface para la clase se presenta como sigue:

```
extern "C" { #include <process.h>}
class Parallel {
      public:
            Parallel() { };
            ~Parallel() { };
            void Par(Process**);
      private:
            Process **p, **q;
}
void Parallel::Par(Process **p) {
      while(**p) {
            q = ProcAlloc(*p,0,0);
            if (q == NULL)
                  exit();
            *p++;
      }
      ProcParList(**q);
};
```

En esta implementación, los procesos son instanciados al llamar la función `ProcAlloc()`, utilizando el nombre de la función como liga a la estructura **Process**. Esta función se encarga de reservar espacio de memoria para el proceso, tomando un apuntador al código de la función, colocando en memoria un *stack* para el proceso, y considerando los parámentros de la función, dando como resultado un apuntador a la estructura **Process**. Los procesos inicializados por `ProcAlloc()` comparten el mismo espacio global de datos y, por lo tanto, tienen acceso a las mismas variables estáticas y externas. Todas las llamadas a `ProcAlloc()` deben ser seguidas por una verificación. Si no es posible la colocación en memoria del proceso, la función da como resultado un apuntador nulo (NULL), a partir del cual se deben tomar las medidas

pertinentes. Todos los procesos a ejecutarse concurrentemente deben ser alojados en memoria antes de utilizarse, de lo contrario se tiene el riesgo de referenciar el mismo espacio de memoria por dos o más procesos [40].

Una vez alojado en memoria, el proceso se ejecuta utilizando la función `ProcParList()`, la cual inicia la ejecución de un grupo de procesos, utilizando como parámetro un apuntador a apuntadores a las estructuras **`Process`** de las funciones a ejecutar, terminando la lista con un apuntador nulo (`NULL`). Una vez que han finalizado todos ellos, la hebra de control del programa regresa al punto en el cual la función fue invocada [40].

A pesar de ser relativamente sencilla en su construcción, la ejecución concurrente de procesos cuenta con varias limitaciones. Primero, un proceso puede ser ejecutado sólo una vez. Si se utiliza un mismo apuntador a proceso como argumento en más de una función, los resultados son impredecibles. Segundo, las funciones que representan los métodos no deben retornan ningún valor, es decir, ser funciones tipo **`void`**. Tercero, debido a la falta de genericidad, es difícil definir métodos con un número variable de parámetros. En caso de requerirse, se aconseja en las clases derivadas extender la definición de los métodos para aceptar diferentes parámetros.

2. Secuencialidad.

a) Modelo Interobjeto.

Al contrario del paralelismo, el concepto de secuencialidad de acciones entre objetos activos independientes no se presenta de una forma simple. Esto se debe a que los objetos cuentan con características de abstracción y encapsulación independientes, lo que hace que su cooperación en forma secuencial sea en cierta medida complicada de comprender y realizar.

A fin de expresar la secuencialidad entre objetos de una forma relativamente simple, se propone una solución en conjunto que depende del no determinismo y comunicación entre objetos mediante canales, con lo que es posible realizar una serialización en las acciones de objetos independientes entre sí. De esta forma, la secuencialidad entre objetos puede ser emulada a partir del intercambio de mensajes entre los objetos que sincronicen en forma serial las acciones que efectúan (Figura III.7). Debido a tal dependencia, es necesario considerar para la secuencialidad la

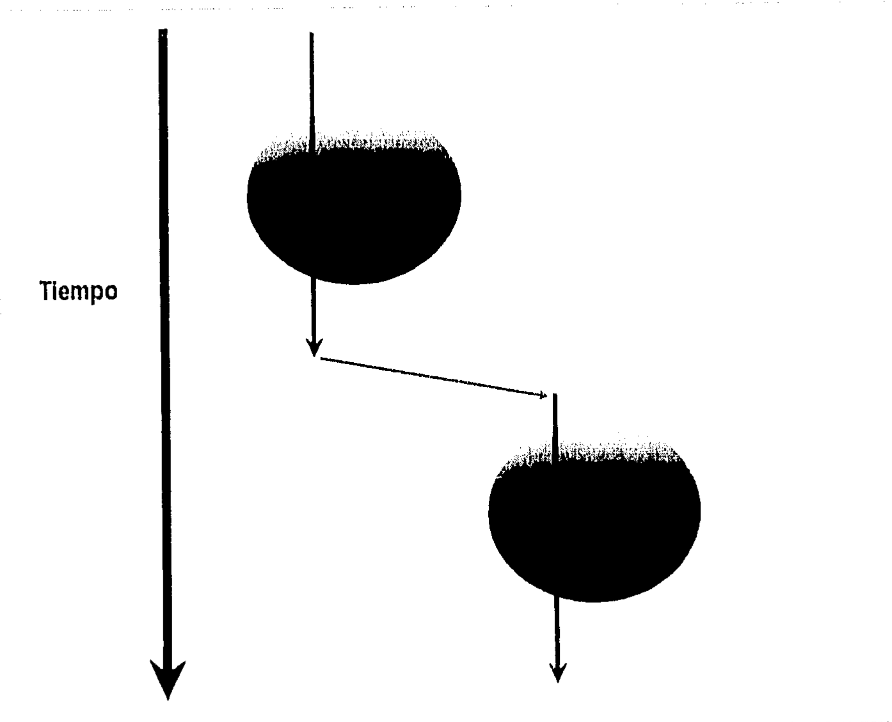


Figura III.7, Secuencialidad interobjeto.

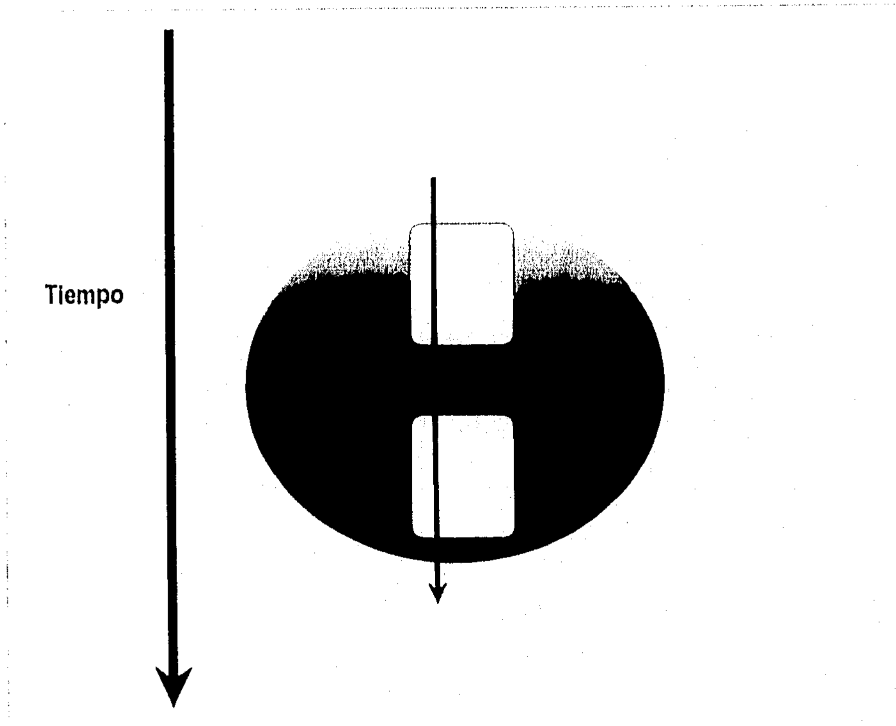


Figura III.8, Secuencialidad intraobjeto.

existencia previa de las características de no determinismo y comunicación entre objetos.

b) Modelo Intraobjeto.

La característica de secuencialidad se incluye en la mayoría de los lenguajes de programación como una construcción en la cual un proceso sigue a otro [41]. En el caso de lenguajes tales como Pascal, C y C++, la secuencialidad de acciones en una misma hebra de control se expresa mediante el uso del carácter `;` al final de cada instrucción [1,80]. De esta forma, la característica de secuencialidad de acciones a nivel interno de cada objeto cuenta con una expresión directa en el lenguaje, por lo que no es necesario incluir ninguna construcción o clase adicional para su realización (Figura III.8).

3. Comunicación y Sincronización.

a) Modelo Interobjeto.

La comunicación y sincronización dentro del ANSI C paralelo se realiza mediante operaciones de envío y recepción de mensajes a través de canales, los cuales se incluyen como un nuevo tipo de datos **`Channel`**, contando con funciones para crear, inicializar, reinicializar canales, así como soportar la entrada y salida de datos mediante un protocolo de comunicación [40,41].

Para extender esta característica al modelo paralelo orientado a objetos, haciendo posible la comunicación y sincronización entre objetos, se propone una aproximación que encapsule las funciones de comunicación entre objetos mediante una clase, a fin de apoyar la entendibilidad y reusabilidad del tipo **`Channel`** mediante el mecanismo de herencia. De esta forma, se considera al tipo **`Channel`** como una característica común entre objetos que se pretende comunicar entre sí, los cuales pueden ubicarse en el mismo procesador o en diferentes procesadores (Figura III.9).

b) Modelo Intraobjeto.

Internamente, dado que un objeto activo también puede presentar un nivel de paralelismo respecto a su conjunto de funciones miembro, se puede considerar una comunicación interna entre tales funciones, las cuales se ejecutan concurrentemente. Sin embargo, aun cuando los métodos concurrentes en el interior de un objeto comparten el mismo espacio global

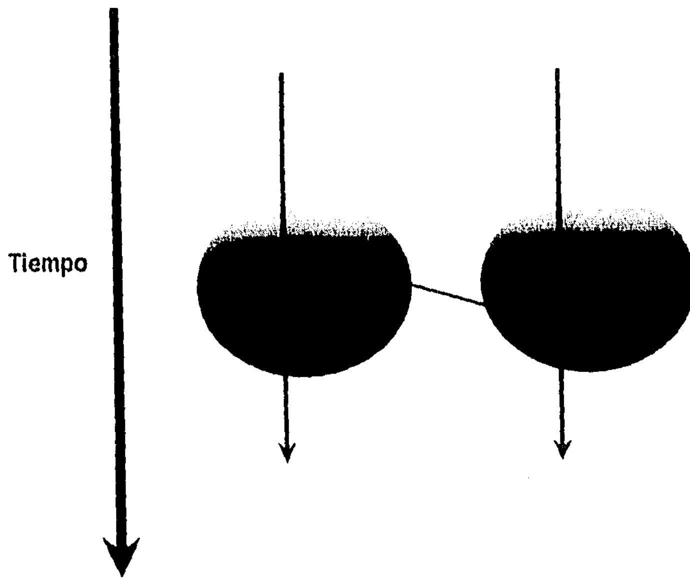


Figura III.9, Comunicación interobjeto.

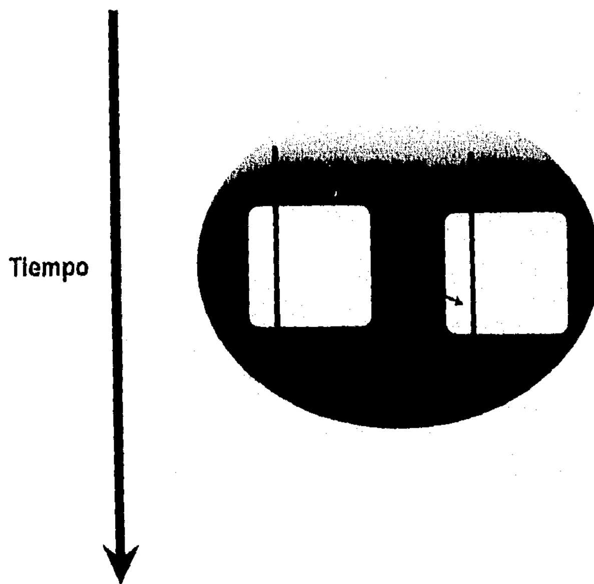


Figura III.10, Comunicación intraobjeto.

de datos, la aproximación propuesta pretende utilizar canales como vías de comunicación entre métodos, introduciendo tipos de dato **Channel** internos entre ellos, esto con el fin de ser consistentes en cuanto al modelo de objetos comunicantes y aprovechar de alguna forma una misma implementación para comunicación mediante canales tanto interna como externamente al objeto (Figura III.10).

Es importante prestar atención especial a que los métodos que se comunican entre sí mediante un canal se ejecuten en forma concurrente dentro del objeto, ya que una distracción al respecto puede generar situaciones de ejecución complicadas, como es el caso especial del *deadlock* recursivo [III.C.2.d].

Debido al uso común de canales como elementos de comunicación y sincronización entre objetos e intraobjeto, y considerando la falta de las propiedades de genericidad y polimorfismo en el INMOS C++ paralelo [22], la interface de los objetos comunicantes como clase dependiendo del tipo de dato que se envía o recibe a través de un canal se propone de la siguiente forma:

```
extern "C" {     #include <channel.h> }
class Comunicacion {
     public:
          Comunicacion() { };
          ~Comunicacion() { };

          void SendInt (int, Channel*);
          int ReceiveInt (Channel*);
          void SendChar (char, Channel*);
          char ReceiveChar (Channel*);
          void SendStr (char*, Channel*);
          char* ReceiveStr (Channel*);
          void SendFloat (float, Channel*);
          float ReceiveFloat (Channel*);
          void SendDouble (double, Channel*);
          double ReceiveDouble (Channel*);

     private:
          Channel*   from;
          Channel*   to;
          int        i;
          char       ch;
          char*      str;
          float      f;
          double     d;
};
```

```
void Comunicacion :: SendInt (int i, Channel *to) {
      ChanOutInt (to, i);
}

int Comunicacion :: ReceiveInt (Channel *from) {
      i = ChanInInt (from);
      return i;
}

void Comunicacion :: SendChar (char ch, Channel *to) {
      ChanOutChar (to, ch);
}

char Comunicacion :: ReceiveChar (Channel *from) {
      ch = ChanInChar (from);
      return ch;
}

void Comunicacion :: SendStr (char* str, Channel *to) {
      ChanOut (to, &str, sizeof(str));
}

char* Comunicacion :: ReceiveStr (Channel *from) {
      ChanIn (from, &str, sizeof(str));
      return str;
}

void Comunicacion :: SendFloat (float f, Channel *to) {
      ChanOut (to, &f, sizeof(f));
}

float Comunicacion :: ReceiveFloat (Channel *from) {
      ChanIn (from, &f, sizeof(f));
      return f;
}

void Comunicacion :: SendDouble (double d, Channel *to) {
      ChanOut (to, &d, sizeof(d));
}

double Comunicacion :: ReceiveDouble (Channel *from) {
      ChanIn (from, &d, sizeof(d));
      return d;
}
```

La operación de cada método depende del tipo de dato que se pretende enviar por un canal. Es importante mencionar que el tipo canal puede comunicar

cualquier tipo de dato a través de él. En general los métodos se basan en una parejas de funciones complementarias que deben ser ejecutadas por los objetos comunicantes, como por ejemplo `SendInt()` y `ReceiveInt()`, las cuales realizan las operaciones básicas para el paso de valores de tipo entero por un canal [40]. Su utilización se muestra en el siguiente ejemplo:

```
extern "C" { #include <misc.h>}
#include "slave.h"

int main (void) {
      Channel*    from_master;
      Channel*    to_master;
      int         i;
      char        ch;
      float       imag;

      from_master = (Channel *) get_param (1);
      to_master   = (Channel *) get_param (2);

      Slave my_slave = Slave();

      i = my_slave.ReceiveInt(from_master);
      my_slave.SendInt(i, to_master);

      ch = my_slave.ReceiveChar(from_master);
      my_slave.SendChar(ch, to_master);

      imag = my_slave.ReceiveFloat(from_master);
      my_slave.SendFloat(imag, to_master);

      exit_terminate(0);
}
```

4. No determinismo.

a) Modelo Interobjeto.

Dado que la arquitectura *transputer* considera al canal como la única forma de comunicación entre procesos, el tratamiento al no determinismo se realiza a partir de un conjunto de funciones alternativas dentro de las bibliotecas del ANSI C paralelo para el manejo del no determinismo, considerando canales como custodias de recepción de mensajes [40].

Para obtener una versión orientada a objetos entendible y reutilizable del tratamiento al no determinismo, se propone utilizar las principales

funciones alternativas en una clase a fin de aprovechar el mecanismo de herencia. Como se mencionó anteriormente, esta aproximación tiene la limitante de considerar únicamente como custodias la recepción de mensajes por parte de un canal (Figura III.11).

b) Modelo Intraobjeto.

Debido también a que por parte de la comunicación entre los métodos de un objeto se ha considerado igualmente al canal como único medio de comunicación, también la implementación propuesta de un clase para el manejo del no determinismo puede referirse para ser utilizada internamente en el objeto (Figura III.12).

De esta forma, en general la vista exterior de un objeto en C++ paralelo se define a partir de una clase con la siguiente sintaxis:

```
extern "C" { #include <process.h> }
class Alternativa {
      public:
            Alternativa() { };
            ~Alternativa() { };

            int Alt(Channel**);

      private:
            Channel**  from;
            int        i;
};

int Alternativa :: Alt (Channel **from) {
      i = ProcAltList(**from);
      return i;

}
```

Al utilizar la función `ProcAltList()` en la implementación, se asegura que la acción del objeto se suspende hasta que uno de los canales se encuentra listo para recibir un mensaje. De esta forma, el método responde con un índice dentro de la lista de canales que establece cuál custodia se ha verificado. A fin de mantener cierta flexibilidad en cuanto al código a ejecutar en el caso de la recepción por un canal específico (o verificación de una custodia) se propone utilizar inmediatamente después un constructor de selección `switch()`, como se muestra en el siguiente ejemplo:

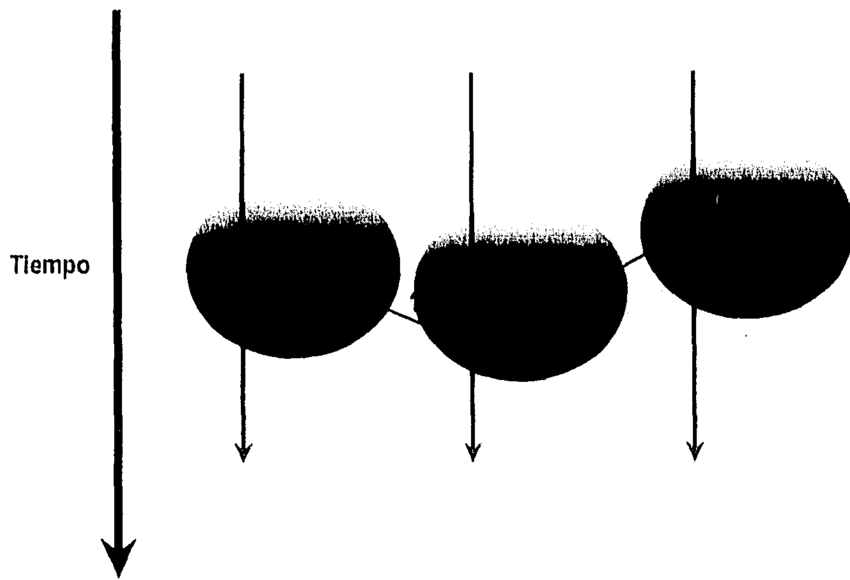


Figura III.11, No determinismo interobjeto.

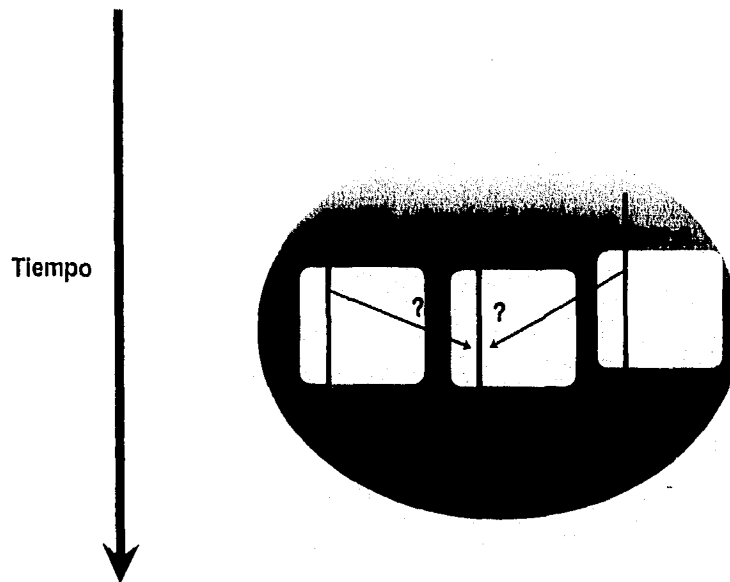


Figura III.12, No determinismo intraobjeto.

```
int i;
Channel *c0, *c1, *c2;
Object *my_object;
i = my_object.Alt(c0,c1,c2);
switch(i) {
      case 0: //Canal 0 listo
      ...// Código a ejecutar para recepción en el canal 0
      break;
      case 1: //Canal 1 listo
      ...// Código a ejecutar para recepción en el canal 1
      break;
      case 2: //Canal 2 listo
      ...// Código a ejecutar para recepción en el canal 2
      break;
      default: // Opción de manejo de error
      error_handler();
      break;
}
```

5. Temporizadores.

Adicionalmente a los elementos y clases que permiten la expresión del paralelismo entre objetos, es igualmente importante considerar la encapsulación en una clase de aquellos elementos que permitan medir el tiempo de ejecución de un programa paralelo, conocidos como temporizadores.

La acción de los temporizadores se basa en rutinas del ANSI C paralelo, con capacidades como retraso, suspensión, reordenamiento o terminación de la ejecución de procesos, conjuntamente con funciones que hacen uso de los relojes internos del *transputer*, a fin de tener una base de tiempo real de ejecución [40,41]. Se propone encapsular estas rutinas en una clase a fin de contar con los temporizadores como base de la medida del desempeño en la programación paralela orientada a objetos. De esta forma, se propone la siguiente sintaxis para la clase de temporizadores:

```
extern "C" { #include <time.h> }
class Timer {
      public:
            Timer() { }; // Constructor de la clase
            ~Timer() { }; // Destructor de la clase

            clock_t GetTime ();// Operadores de la
            clock_t TimePlus(clock_t, clock_t);    // clase
            clock_t TimeMinus(clock_t, clock_t);
            float TimeInSeconds(clock_t);

      private:

            clock_t         time0, time1, time2;
```

```
            float       seconds;
};
```

La implementación de los métodos de esta clase, que encapsulan funciones del ANSI C paralelo, se propone de la siguiente forma:

```
clock_t Timer::GetTime () {
      time0 = clock();
      return time0;
}

clock_t Timer::TimePlus(clock_t time2, clock_t time1) {
      time0 = time2+time1;
      return time0;
}

clock_t Timer::TimeMinus(clock_t time2, clock_t time1) {
      time0 = time2-time1;
      return time0;
}

float Timer::TimeInSeconds(clock_t time0) {
      seconds = ((float)time0/(float)CLOCKS_PER_SEC);
      return seconds;
}
```

Estos métodos permiten el acceso a los atributos privados de la clase que representan medidas en un contador de tiempo del reloj interno del *transputer*, como número de pulsaciones (*ticks*) por segundo. Para cuantificar estos datos, se utiliza el tipo `clock_t` para pulsaciones. De esta forma, `GetTime()` permite conocer el valor de pulsaciones que representan en determinado momento el estado del contador. Los métodos `TimePlus()` y `TimeMinus()` tienen el propósito de realizar las operaciones de adición y substracción de pulsaciones que representan tiempos, utilizando operaciones de módulo aritmético debido a que el contador del reloj interno es cíclico. Finalmente, la función `TimeInSeconds()` permite obtener una representación de tipo punto flotante de las pulsaciones, la cual puede presentarse directamente en pantalla. Un ejemplo del uso de esta clase puede ser el siguiente programa:

```
extern "C" { #include <stdio.h>
      #include <stdlib.h>
      #include <misc.h>
}
#include "timer.h"

int main () {

      clock_t tiempo, tiempo1, tiempo2;
      float segundos;
```

```
    Timer local = Timer();
    tiempo1 = local.GetTime();
    printf("Texto para medir tiempo de impresion en
          pantalla\n\n");
    tiempo2 = local.GetTime();
    tiempo = local.TimeMinus(tiempo2,tiempo1);
    segundos = local.TimeInSeconds(tiempo);
    printf ("Time in seconds: %10.6f \n", segundos);
    exit_terminate(EXIT_SUCCESS);
}
```

6. Objetos activos paralelos.

Finalmente, se presenta un árbol de herencia propuesto para la creación de clases de objetos activos paralelos como resultado de la combinación de las características expresadas como clases del paralelismo (Figura III.13), utilizando el método de Booch [9].

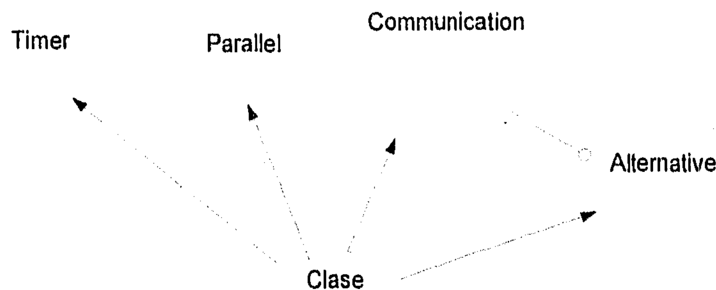


Figura III.13, Clase de objetos paralelos.

De esta forma, combinando este diseño con una configuración en la arquitectura *transputer*, es posible expresar y realizar una clase de objetos activos que permitan paralelismo interobjeto e intraobjeto.

Capítulo IV. Ejemplo de Aplicación: La Transformada Rápida de Fourier.

"Lo que sí poseía era una misteriosa capacidad de ver la aplicación de una teoría; de ver el modo de ponerla en práctica. En un frío bloque de mármol de estructura abstracta podía ver, sin aparente dificultad, el complicado diseño de un invento maravilloso. El bloque se partía a su contacto, y quedaría el invento".
Isaac Asimov.

Tradicionalmente, un programa se define como un método o algoritmo que da una solución específica a un problema planteado. Sin embargo, aún cuando actualmente esta afirmación se considera correcta, no expresa en forma clara la finalidad del concepto de programa. Otra definición, expresada desde un punto de vista orientado a objetos, considera al programa como un modelo del mundo real que tiene como finalidad representar los elementos del dominio del problema mediante objetos, representando sus relaciones y comportamiento. Esta última aproximación permite observar la aplicación que la tecnología orientada a objetos presenta en la solución de problemas computacionales reales. Para poder representar esto, en el presente capítulo se propone el desarrollo de una aplicación práctica programada mediante objetos cuya ejecución se realice en una arquitectura paralela. De esta forma, se propone una aplicación práctica para la evaluación de la programación paralela orientada a objetos, como un método de modelación de problemas reales, como es el caso de la Transformada Rápida de Fourier.

A. Procesamiento de señales e imágenes.

Dos áreas dentro de la computación han tomado relevancia en los últimos años, debido a su utilidad práctica y complejidad operacional: el procesamiento de señales y el procesamiento de imágenes. Ambos procesamientos se utilizan de alguna manera con los mismos propósitos [30,69]:

- Para mejorar la apariencia visual de las señales e imágenes a la vista humana.
- Para preparar a las señales e imágenes para la medición de sus características estructurales.

Las técnicas utilizadas para lograr esto no siempre son únicas, sino que se utilizan en forma sobrepuesta, es decir, una tras otra hasta obtener un efecto en particular [30,24]. Estas técnicas pueden clasificarse respecto al tipo de respuesta o modificación que se obtiene de cada una de ellas: técnicas de procesamiento morfológicas, topológicas, de transformación, etc. [24].

1. Técnicas de transformación en el procesamiento de señales e imágenes.

Se considera como técnicas de transformación a aquellas operaciones que en general convierten a las señales o imágenes en una nueva estructura. En ocasiones, tal estructura es de nuevo una señal o una imagen [24]. Las razones más comunes de la aplicación de las técnicas de transformación a una señal o imagen son, entre otras, para facilitar su manejo o su comprensión. De hecho, es frecuente que por conveniencia se utilicen técnicas de transformación antes de realizar otro tratamiento o procedimiento matemático sobre la señal o imagen involucrada [30,69,24].

Las técnicas de transformación se caracterizan por la forma en que son empleadas, ya que involucran no solo un proceso de transformación por sí mismo, sino también un correspondiente proceso de transformación inverso. Una tarea a realizar sobre una señal o imagen utiliza una transformación para convertirla en otra estructura que es la representación de la señal o imagen en un nuevo entorno o dominio, donde es posible obtener el resultado deseado realizando un procedimiento más simple que aquél requerido en el dominio original. Es evidente que al finalizar la salida de este procedimiento se encuentra aún en el nuevo dominio, por lo que para completar la solución al problema original se propone la utilización de una transformación inversa que mapea tal salida a su representación en el dominio original [24].

Existen una gran variedad de técnicas de transformación de señales e imágenes, en general relacionadas con la aplicación de una expresión matemática. Entre las más conocidas se encuentran la transformada de Laplace, la transformada Z, y la transformada de Fourier [24,67].

2. Transformada Rápida de Fourier.

La Transformada de Fourier es de fundamental importancia en el análisis matemático, y ha sido objeto de gran cantidad de estudios. El número de sus aplicaciones es enorme, entre las que se le considera como la base de varias manipulaciones fundamentales, en especial en problemas tales como la solución de ecuaciones diferenciales, procesamiento de señales y procesamiento de imágenes [70,27]. Se define para una función en el tiempo x(t) mediante la siguiente expresión:

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi ft}\, dt$$

Por otro lado, uno de los algoritmos aritméticos basados en la Transformada de Fourier utilizado más ampliamente en la actualidad es la Transformada Rápida de Fourier (*Fast Fourier Transform* o FFT), que se basa en el cálculo de la transformada de Fourier convolucionando los datos de entrada con un número de frecuencias diferentes y considerando un número finito de muestras de entrada. La expresión de la transformada de Fourier se reduce entonces a un equivalente discreto, conocido como a Transformada Discreta de Fourier (*Discrete Fourier Transform* o DFT). La FFT se trata de un caso especial de la DFT, y representa como algoritmo un ejemplo del uso eficiente de las matemáticas en la computación. Sus aplicaciones se extienden a campos tales como la teoría de control, identificación de señales, teoría de codificación y procesamiento de señales e imágenes [68,70].

a) Señales. FFT en una dimensión.

Para el tratamiento de señales, es suficiente evaluar la FFT respecto a una sola dimensión o variable. Así, la Transformada de Fourier de una señal discreta no periódica o DFT está definida por la ecuación [24,67,68,27]:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{\frac{-j2\pi kn}{N}}$$

Esto significa que dado un conjunto de valores de una función compleja x(k), k = 0, 1, ..., N-1, la DFT genera un nuevo conjunto de valores complejos X(k), donde $j = \sqrt{-1}$.

Por otro lado, la transformada inversa discreta de Fourier tiene la siguiente forma [68,27]:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\, e^{\frac{j2\pi kn}{N}}$$

Como se puede observar, ambas transformaciones tienen expresiones matemáticas tan parecidas que es posible considerarlas mediante una sola expresión genérica. Para el caso de la transformada directa, se tiene [67,27]:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{kn}$$

donde $W_N = e^{-j2\pi/N}$. La inclusión del signo negativo en la exponencial para expresar la transformada directa es arbitrario. Para el caso de la transformada inversa, tal signo puede considerarse positivo, y multiplicar toda la expresión por 1/N.

Se considera a la FFT como un caso particular de la DFT en donde el valor N es tomado como una potencia de 2. No es propósito de este trabajo realizar completamente el proceso matemático para obtener la FFT, sino que a partir de la expresión para la DFT, considerando la secuencia de datos x(n) de entrada a un circuito discreto formada por dos secuencias par x(2n) e impar x(2n+1) en términos de W_N, y simplificando dado que el valor W_N tiene la característica de periodicidad, se tienen las siguientes expresiones:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)\, W_{\frac{N}{2}}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\, W_N^{kn}$$

$$X(k) = X(k + \frac{N}{2})$$

De esta forma, se puede observar que cada término del lado derecho de la primera ecuación se trata de una DFT directa. La segunda ecuación obedece a la utilización de resultados parciales como lo sugieren Cooley y Tukey para la evaluación de la FFT. Ambas ecuaciones se puede reescribir como [68,27,51]:

$$X(k) = X_{par}(k) + W_N^k X_{impar}(k)$$

$$X(k + \frac{N}{2}) = X_{par}(k) - W_N^k X_{impar}(k)$$

Así, a fin de calcular la DFT a partir de ambas expresiones X(k) y X(k+N/2) es necesario evaluar dos DFT's, $X_{par}(k)$ y $X_{impar}(k)$, de longitud media y por lo tanto, a la mitad del costo computacional. Ahora bien, dado que N es potencia de 2, el proceso puede aplicarse recursivamente dividiendo aún más el problema hasta obtener adiciones de longitud 1. En este punto, el algoritmo de la FFT consiste en combinar pares de resultados parciales partiendo de los valores originales de la función.

Describir completamente el proceso de la FFT algebraicamente tiende a complicarse grandemente. Sin embargo, utilizando la teoría de gráficas, la FFT se puede expresar mediante una gráfica (Figura IV.1) [67,68,27,51].

b) Imágenes. FFT en dos dimensiones.

En este caso de imágenes, la DFT se expresa a partir considerar una imagen como una función bidimensional como una matriz de *m* por *n* con la forma:

$$f = \begin{pmatrix} f(0,0) & f(0,1) & \cdots & f(0,n-1) \\ f(1,0) & f(1,1) & \cdots & f(1,n-1) \\ \vdots & \vdots & & \vdots \\ f(m-1,0) & f(m-1,1) & \cdots & f(m-1,n-1) \end{pmatrix}$$

La DFT de una imagen como esta se define como otra matriz con la siguiente forma:

$$F = \begin{pmatrix} F(0,0) & F(0,1) & \cdots & F(0,n-1) \\ F(1,0) & F(1,1) & \cdots & F(1,n-1) \\ \vdots & \vdots & & \vdots \\ F(m-1,0) & F(m-1,1) & \cdots & F(m-1,n-1) \end{pmatrix}$$

donde el valor *F(p,q)* para cada elemento de la matriz es dado por la

expresión [24]:

$$F(p,q) = \frac{1}{mn}\sum_{r=0}^{m-1}\sum_{s=0}^{n-1} f(r,s)e^{-j2\pi(\frac{rp}{m}+\frac{sq}{n})}$$

Sin embargo, debido a la propiedad de separabilidad, la DFT para funciones de dos dimensiones puede ser obtenida en dos pasos mediante la evaluación sucesiva de dos DFT's en una dimensión. Tal propiedad se aplica de igual forma a la inversa. Así, la DFT en dos dimensiones para una matriz cuadrada (m=n=N) se puede expresar como:

$$F(p,q) = \frac{1}{N}\sum_{r=0}^{N-1} F(r,q)e^{\frac{-j2\pi pr}{N}}$$

donde:

$$F(r,q) = N[\ \frac{1}{N}\sum_{s=0}^{N-1} f(r,s)e^{\frac{-j2\pi qs}{N}}\ ]$$

Esto es, para cada valor de r, la expresión dentro de los corchetes es una DFT unidimensional. Por lo tanto, la función bidimensional $F(r,q)$ se obtiene de considerar la transformada a lo largo de cada fila de la matriz f, y multiplicando el resultado por N. El resultado total $F(p,q)$ se obtiene al tomar la transformada de cada columna. De esta forma, es posible aplicar dos veces el algoritmo de la FFT en una dimensión para evaluar una FFT en dos dimensiones [30].

B. Análisis y diseño orientado a objetos.

Es posible considerar como resultado de un análisis orientado a objetos a una jerarquía de clases expresada en forma de un diagrama de clases que se desarrolla en base a listar objetos, agruparlos en clases y finalmente, agregar las relaciones existentes entre ellos. Cada clase es un módulo separado con estructuras de control y datos propios. El modelo inicial se va perfeccionando iterativamente. Por otro lado, el resultado de un diseño orientado a objetos es la implementación de la jerarquía de clases obtenida en el análisis mediante un lenguaje de programación [9,79].

Los elementos iniciales de un análisis orientado a objetos son los propios objetos. Posteriormente, a medida que se identifican aspectos comunes, los objetos se van agrupando en clases, que a su vez serán subclases de clases más abstractas [79].

En esencia, el análisis y diseño orientado a objetos consta de cuatro pasos fundamentales:

- Identificación y definición de objetos y clases.
- Organización de relaciones entre clases.
- Extracción de estructuras en una jerarquía de clases.
- Diseño y construcción de bibliotecas de clases reutilizables.

Como punto final, el diseño orientado a objetos es cíclico. Se inicia con un conjunto de clases, las cuales a su vez se amplían, modifican y encajan formando un prototipo de la aplicación. La interacción con los usuarios finales hace que el prototipo sea revisado, a fin de reorganizar y redefinir las clases, buscando cada vez un mayor nivel abstracción.

1. Identificación y definición de objetos y métodos.

No existen reglas estrictas para identificar objetos. Sin embargo, existe un acuerdo general sobre algunas directrices que ayudan a identificar y definir objetos y métodos [79]:

- Modelar con objetos los elementos que ocurren en forma natural en el problema.
- Cada objeto debe corresponder a una abstracción de datos significativa.
- Diseñar métodos de finalidad única.
- Diseñar un nuevo método al encontrar una oportunidad de ampliar uno existente.
- Evitar métodos extensos.

El análisis y el diseño orientados a objetos se inician a partir de obtener los objetos involucrados. Encontrarlos es quizá el principal problema. Se han utilizado varios métodos para identificar objetos, entre los que se encuentra la inspección gramatical de documentos. Este método es idea de G. Booch [9], y sugiere comenzar con una descripción del sistema deseado identificando con los sustantivos a los objetos potenciales y con los verbos a los métodos. La lista resultante de objetos (sustantivos) y métodos (verbos) se utiliza para comenzar el proceso de análisis [79].

Booch propone comenzar por una definición del problema y una descripción de la solución. Considerando el caso de estudio de la FFT, y partiendo de los detalles expresados en la sección anterior, se tiene lo siguiente:

- *Definición del problema:* Desarrollar una implementación de la FFT unidimensional, basada en la programación orientada a objetos que se ejecute en una arquitectura paralela basada en *transputers*.

- *Descripción de la solución:* El elemento básico de la FFT es un número de operaciones de suma, resta y multiplicación entre dos valores complejos que forman el vector de datos. Inicializa sus datos propios como vectores de factores de peso. Evalúa, permuta y combina sus elementos de una forma determinada para formar la FFT de 2^N puntos del vector de datos.

A partir de esta descripción, y mediante un proceso de análisis de la solución se consideraron finalmente los siguientes objetos y métodos potenciales:

Objetos	**Métodos**
FFT	Inicializa, evalúa (mediante operaciones de suma, resta y multiplicación), permuta y combina sus elementos.

2. Organización de interfaces entre objetos.

La definición de interfaces entre objetos es el siguiente paso de la metodología de Booch, una vez mas mediante una descripción escrita involucrando las clases y métodos definidos en el paso anterior. Como se mencionó anteriormente [III.B.2.a], para el caso de objetos ejecutándose en una arquitectura paralela, se considera importante que la interface de cada objeto cuente con las operaciones necesarias para comunicarse con los demás mediante paso de mensajes a través de canales.

Para el caso de la clase FFT, se propone que tenga la capacidad de comunicación mediante canales. Una FFT completa se calcula mediante un número determinado de FFT's parciales mediante la división de sus datos entre objetos activos, cada uno realizando funciones sobre sus elementos: inicializar, evaluar, permutar, combinar e intercambiar resultados con otros objetos.

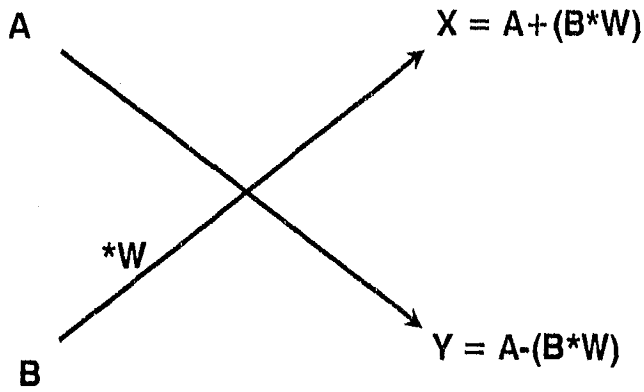


Figura IV.1, Gráfica del flujo de datos en una FFT.

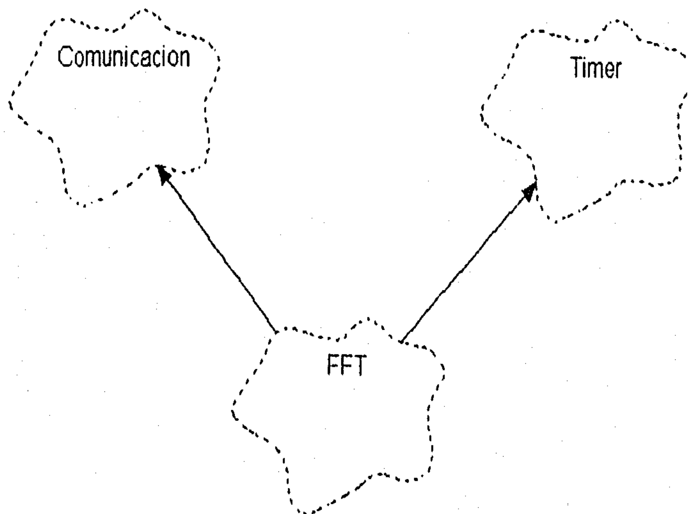


Figura IV.2, Jerarquía de clases para la FFT paralela.

3. Definición y organización de una jerarquía clases.

Una vez definidos los objetos, el paso siguiente consiste en reunir definiciones comunes de objetos en una abstracción de clase, y observar las relaciones de las clases entre sí. Para el caso de la FFT, se requiere que ésta presente ciertas características de comunicación entre objetos mediante canales, así como de elementos de temporización para la medición del desempeño del sistema. Para lograr esto, se propone que la clase FFT herede tales características de las clases Comunicacion y Timer expuestas en el capítulo anterior. Es importante hacer notar que debido a la manera en la que se pretende calcular la FFT, todos los objetos activos creados durante la ejecución paralela comparten una estructura semejante, pero operan sobre segmentos de datos diferentes, realizando cada uno las operaciones necesarias de evaluación y comunicación con los demás. Así, la jerarquía de clases para el cálculo de la FFT queda como se muestra en la figura IV.2.

C. Análisis y diseño paralelo.

El análisis paralelo consiste básicamente en determinar y describir los elementos del programa que son susceptibles a ejecutarse en paralelo y definir las comunicaciones entre ellos, en general considerando los elementos del paralelismo requeridos para una aplicación determinada [I]. El diseño paralelo por su parte consiste en proponer una arquitectura tanto de *hardware* como de *software* que soporte los procesos y comunicaciones considerados en el análisis.

1. Identificación de paralelismo potencial.

Como se mencionó anteriormente [IV.A.2.a], el procedimiento básico para el cálculo de la FFT es efectuar un número de operaciones complejas, representadas mediante una mariposa (Figura IV.1). Cada mariposa consiste en una adición, una sustracción y dos multiplicaciones sobre valores complejos. La mariposa es el caso más sencillo de una FFT sobre dos puntos únicamente. Al combinar las operaciones de cada mariposa de una manera determinada, una FFT de 2^N puntos puede ser creada. La representación en forma de mariposa es utilizada principalmente para determinar el orden en el cual las operaciones deber realizarse en una máquina secuencial [68].

Por otro lado, es posible considerar a partir de la representación en forma de mariposa el paralelismo potencial entre operaciones. Para ilustrar esto, se considera a continuación el caso en que N = 2. Como se puede observar en la figura IV.3, en donde se realizan las operaciones sobreescribiendo los valores en el vector de datos original, que el primer paso de la transformación puede ser calcular

$g_0(0,1)$ y $g_1(0,1)$ independientemente de $g_0(1,1)$ y $g_1(1,1)$. De forma similar, el segundo paso puede descomponerse, calculando $g_0(0,2)$ y $g_1(0,2)$ independientemente de $g_1(0,2)$ y $g_3(0,2)$. De esta forma, es posible descomponer el cálculo de una FFT de 2^N puntos en mariposas que realizan sus operaciones simultáneamente, y calcular transformadas individuales separadamente en un sistema multiprocesador, lo que parece ser la ruta más sencilla hacia el paralelismo. Sin embargo, debe hacerse consideraciones especiales de la forma en que los elementos del vector serán repartidos entre los procesadores, manteniendo un tamaño razonable de granularidad al asegurar que las transformaciones individuales sean de una longitud apropiada, y balanceando el problema de modo que los recursos disponibles sean totalmente explotados [27].

2. Definición de la configuración de la arquitectura.

En este punto del análisis paralelo se considera la configuración de la arquitectura como las características del *hardware* y el *software* que comprende la ejecución paralela. Esto implica inherentemente considerar la disponibilidad de elementos de procesamiento, su capacidad de interconectividad en una topología en particular, y el tamaño y distribución de los objetos activos que se alojan en ellos. A fin de limitar el problema y permitir realizar un análisis y diseño propiamente del *software*, se realizan las siguientes consideraciones:

- El número de procesadores depende directamente del requerimiento de la topología definida, esto es, se considera la cantidad de nodos necesaria para representar una topología en especial.

- Las topologías de conexión dependen del esquema de comunicación definido a partir de la aplicación, esto es, considerando las relaciones entre los objetos dentro del dominio del problema. Para el caso especial de la FFT se proponen principalmente las topologías lineal (*pipeline*) e hipercubo [I.A.2.b].

- Se propone un esquema de granularidad gruesa, debido a que los objetos se plantean como elementos de programación de granularidad gruesa o mediana. Esto significa que la cantidad de procesamiento realizado por un objeto es mayor que la cantidad de comunicaciones entre ellos.

- El balance de trabajo depende directamente del tamaño y número de elementos de procesamiento con los que se cuente. Se considera un balance de trabajo de tipo uniforme para el caso específico de la FFT, dado que cada objeto realiza en forma aproximada las mismas operaciones durante la ejecución del programa.

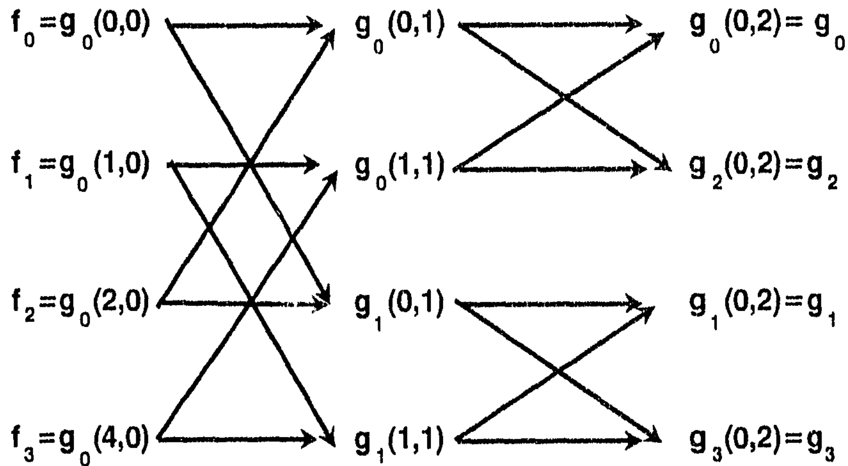


Figura IV.3, FFT para N=4 con sobreescrituras.

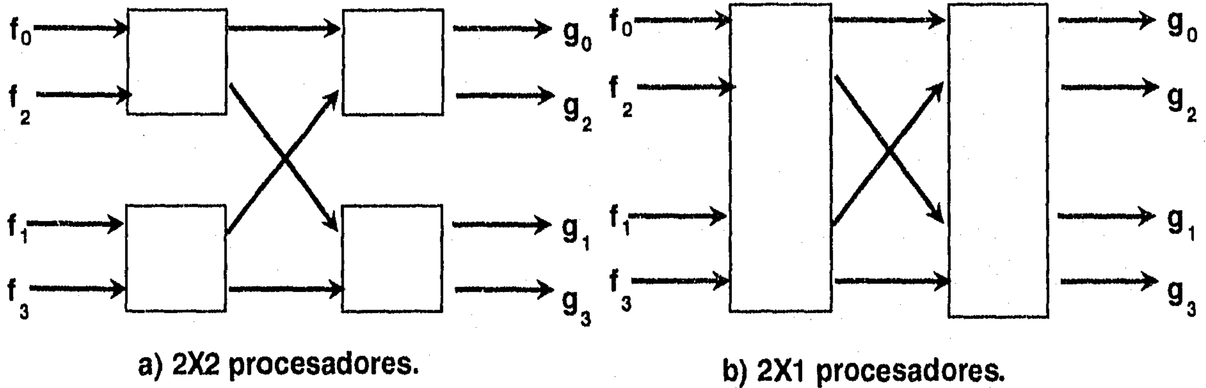


Figura IV.4, FFT para N=4 en arreglos de procesadores.

Partiendo de esto, se proponen las siguientes configuraciones para la arquitectura paralela de procesamiento de una FFT mediante objetos.

a) Lineal (*pipeline*).

El cálculo de una FFT cuyo tamaño es potencia de dos se puede realizar en forma de flujo de datos conectando muchas mariposas en forma de *pipeline*. La red de *transputers* se configura entonces en una topología lineal, de tal forma que el *hardware* coincide perfectamente con el *pipeline* de mariposas [51,68].

Como ejemplo, supóngase el caso más simple en el que cada paso del *pipeline*, formado por varias mariposas, se asigna a un solo *transputer*. Cada mariposa requiere de dos canales de entrada, conectados a una mariposa anterior del *pipeline*, y dos canales de salida para pasar los resultados de sus operaciones a la siguiente mariposa en el *pipeline*. Ahora bien, dado el paralelismo potencial entre las varias operaciones [IV.C.1], una FFT completa puede calcularse a partir de la colección de mariposas que se ejecutan concurrentemente en cada paso del *pipeline*. Para el caso de un vector de 8 elementos, el *pipeline* cuenta con tres pasos, cada uno consistente en cuatro procesos concurrentes [51].

Ahora bien, es posible considerar que las colecciones de mariposas de cada paso se ejecuten en arreglos de procesadores manteniendo el efecto de paralelismo entre las mariposas, y reduciendo el paralelismo local. De hecho, pueden utilizarse para lograr esto diversos arreglos de procesadores. Para el caso de 4 elementos, se pueden agrupar las mariposas en arreglos de procesadores de 2×2 ó 1×2 (Figura IV.4).

Sin embargo, la solución de la FFT mediante un *pipeline* tiene la desventaja que al aumentar el número de procesadores del arreglo se tiende a producir una gran cantidad de comunicaciones, lo que provoca retrasos debido a tiempos de espera en la comunicación. Es necesario tomar en cuenta previamente esta cantidad de comunicaciones entre procesadores, para determinar su impacto sobre el tiempo de ejecución del programa [68].

b) Hipercubo.

En general, una aplicación que se ajusta a un hipercubo consiste en un objeto maestro (*master*) que se ejecuta en un solo procesador, y uno o más objetos esclavo (*slave*) que se ejecutan a su vez en otros procesadores del hipercubo. Ambos objetos, maestro y esclavos se ejecutan

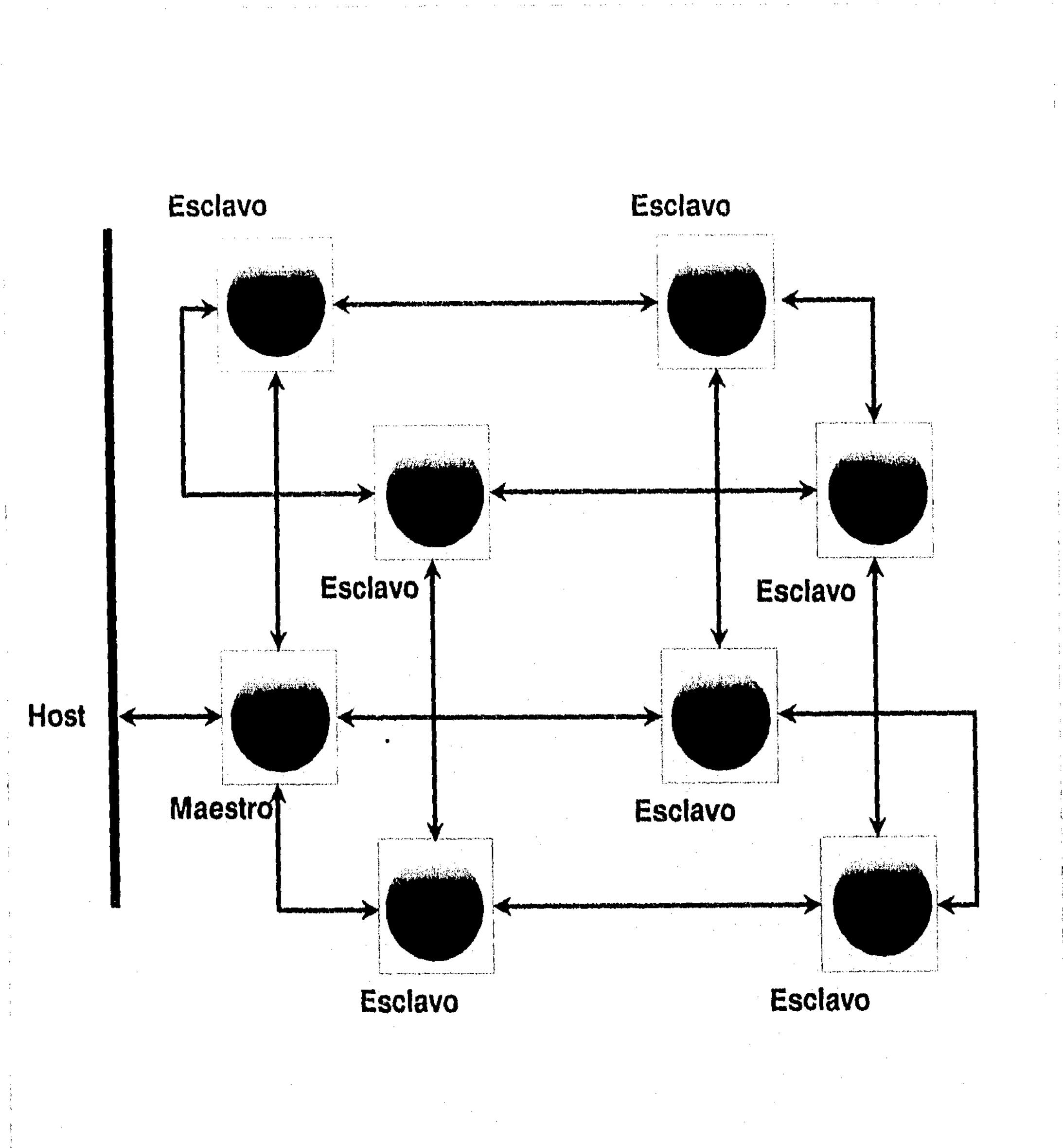


Figura IV.5, FFT en hipercubo para N=8.

independientemente como una aplicación convencional, con la diferencia de que el maestro se encarga de inicializar el procesamiento, comunicación con el usuario y proporcionar los datos a los nodos esclavos. Cada objeto, sea maestro o esclavo, realiza el mismo procesamiento, intercambiando información con los otros programas cuando se requiere, y al final, envía sus resultados al maestro. Es debido a esta característica que una aplicación en hipercubo se realiza en un conjunto de procesadores que no comparten memoria, sino que cada uno cuenta con su propia memoria local y se comunican entre sí mediante paso de mensajes.

En el caso particular de la FFT en una dimensión, cada objeto opera sobre una pieza local de datos, ejecutando una FFT local, seguida por un número de operaciones de combinación, intercambiando datos con sus nodos vecinos. Las operaciones son las mismas que aquellas realizadas en una topología lineal. Sin embargo, la diferencia consiste en la forma en que se programan y realizan tales operaciones: en el *pipeline*, cada objeto u objetos efectúan un paso de las operaciones para el cálculo de la FFT sobre los datos, a diferencia del hipercubo, que inicialmente divide los datos entre los objetos de sus nodos, calculando cada objeto una FFT local, y reportando sus resultados al objeto maestro. Las ventajas que presenta el hipercubo sobre el *pipeline* son principalmente:

- Disminuye la cantidad de comunicaciones entre los procesadores. Cada nodo que se comunica sólo requiere comunicarse con sus nodos vecinos.

- Permite un mejor balance de trabajo. La división de los datos entre los procesadores para el caso de la FFT se realiza en una forma más uniforme, debido a la característica de tamaño de los vectores de datos, que corresponde a un valor que es potencia de 2. Por otro lado, considerando que se utiliza un conjunto de procesadores semejantes, cada procesador realiza aproximadamente las mismas operaciones en el mismo tiempo, lo que disminuye a su vez los tiempos de espera durante el intercambio de datos entre ellos.

- Utiliza una granularidad más gruesa, lo que permite adecuarlo de una forma más clara al modelo de objetos.

D. Diseño paralelo orientado a objetos.

Una característica de diseño de los sistemas orientados a objetos es su tendencia a utilizar un vocabulario en el análisis relativo al espacio del problema, lo que permite

representar la solución como una máquina virtual que paraleliza la abstracción en entidades clave del problema. A partir de las consideraciones realizadas por cada una de las partes, se conjuntan los elementos del paralelismo con las abstracciones de la orientación a objetos en un diseño, a partir del cual es posible observar en forma práctica la medida en que se han realizado las expectativas de la programación paralela orientada a objetos. Utilizando las consideraciones y clases previamente determinadas que expresan el paralelismo en un entorno orientado a objetos [III.D], se definen para el ejemplo de la FFT los métodos para su cálculo en una clase que además, debe contar con la capacidad de ejecutarse en múltiples procesadores, comunicar los objetos entre sí, y medir los tiempos de ejecución del proceso total. Se propone que las implementaciones se ejecuten en una topología de hipercubo, cuyo número de nodos puede ser 2, 4 u 8, debido a limitaciones en cuanto al número de ligas entre los procesadores. Es importante mencionar que la implementación de la clase FFT como se ha planteado en el análisis orientado a objetos no presenta ninguna modificación, esto es, es independiente al número de objetos y procesadores utilizados. La clase FFT queda como sigue:

```
extern "C" {     #include <stdio.h>
                 #include <stdlib.h>
                 #include <misc.h>
                 #include <math.h>
                 #include <channel.h>
                 #include <time.h>
}
#include "timer.h"     // Inclusión de la clase Timer
#include "comunica.h" // Inclusión de la clase Comunicacion

class FFT : public Timer, public Comunicacion {
     public:
          FFT();  // Constructor de la clase FFT
          ~FFT() { }; // Destructor de la clase FFT

          // Operadores de la clase FFT
          int bitrev(int,int);
          void permute(int,int,float*,float*,int*);
          void initialize(float*,float*,int*,int,int);
          void combine(int,int,int,int,float*,float*,float*,float*);
          void evaluate(float*,float*,float*,float*,int,int,int);

          // Operadores auxiliares de la clase FFT
          void generate_signal(float*,float*,int);
          void restore(float*,float*,float*,float*,int);
          void back(float*,float*,float*,float*,int);

     private:
          float *ra,*ia,*c,*s,*RTemp,*ITemp;
          int   *rbit,signal;
};
```

```
FFT::FFT() {
// Constructor de la clase, donde se definen los vectores sobre los
//que se realiza la transformación.
      ra = (float*)malloc(256*sizeof(float));
      ia = (float*)malloc(256*sizeof(float));
      c = (float*)malloc(256*sizeof(float));
      s = (float*)malloc(256*sizeof(float));
      RTemp = (float*)malloc(256*sizeof(float));
      ITemp = (float*)malloc(256*sizeof(float));
      rbit = (int*)malloc(256*sizeof(int));
};

int FFT::bitrev(int j,int p) {
// Obtiene una versión invertida de p bits, tomando como argumento el
// bit j.
      int j1,j2,k0,q;
      j1 = j;
      k0 = 0;
      for (q = 1; q <= p; q++) {
            j2 = j1/2;
            k0 = k0*2 + (j1 - j2 - j2);
            j1 = j2;
      };
      return k0;
 };

void FFT::permute(int n,int p,float *ra,float *ia,int *rb) {
// Obtiene la permutación de las versiones invertidas de n puntos de
// los arreglos ra,ia.
      int   i,k;
      float u,v;
      p = p;
      for (k = 0; k < n; k++)      {
            i = *(rb+k);
            if (i >= k) {
                  u = *(ra+k);
                  v = *(ia+k);
                  *(ra+k) = *(ra+i);
                  *(ia+k) = *(ia+i);
                  *(ra+i) = u;
                  *(ia+i) = v;
            };
      };
};

void FFT::initialize(float *c,float *s, int *br, int n, int p)     {
// Inicializa los vectores c y s de factores de peso, y el vector br
// de índices de bits invertidos.
      int i;
      float arg, Pi2on;
      Pi2on = 2.0*3.1415926/n;
      for (i = 0; i < n; i++)      {
            arg = i*Pi2on;
            *(c+i) = cos(arg);
```

```
            *(s+i) = sin(arg);

            *(br+i) = FFT::bitrev(i,p);
      }
};


void FFT::combine(int nlocal,int arg0,int arg_inc,int Stage,float
*ra,float *ia,float *c,float *s) {
// Combina entre sí los resultados para formar el vector de la FFT
// local utilizando aritmética de apuntadores.
      int arg, i, k;
      float r_temp, i_temp;
      arg = arg0;
      if (Stage == 0) {
            for (i = 0; i < nlocal; i++) {
                  k = i + nlocal;
                  i_temp = (*(ia+k)) * (*(s+arg));
                  r_temp = (*(ra+k)) * (*(c+arg));
                  i_temp = i_temp + r_temp;
                  *(ra+i)  = *(ra+i) + i_temp;
                  i_temp = (*(ia+k))*(*(c+arg));
                  r_temp = (*(ra+k))*(*(s+arg));
                  i_temp = i_temp - r_temp;
                  *(ia+i)  = *(ia+i) + i_temp;
                  arg = arg + arg_inc;
            };
      };
      if (Stage == 1) {
            for (i = 0; i < nlocal; i++) {
                  k = i + nlocal;
                  i_temp = (*(ia+i))*(*(s+arg));
                  r_temp = (*(ra+i))*(*(c+arg));
                  i_temp = i_temp + r_temp;
                  *(ra+i)  = *(ra+k) - i_temp;
                  i_temp = (*(ia+i))*(*(c+arg));
                  r_temp = (*(ra+i))*(*(s+arg));
                  i_temp = i_temp - r_temp;
                  *(ia+i) = *(ia+k) - i_temp;
                  arg = arg + arg_inc;
            };
      }
};

void FFT::evaluate(float *ra,float *ia,float *c,float *s, int ntotal,
int nlocal, int plocal) {
// Realiza una FFT local de una dimensión para nlocal puntos de los
//arreglos ra,ia.
      int   I,ii,k,j,l,LE,LE1,IP,ilim;
      float a,u,v,cc,ss;
      for (l = 1; l <= plocal; l++) {
            LE = 1;
            for (j = 1; j <= l; j++) LE = 2*LE;
            LE1 = LE >> 1;
```

```
            k = 0;
            for (j = 0; j < LE1; j++) {
                  cc = *(c+k);
                  ss = *(s+k);
                  I = j;
                  ilim = nlocal/LE + j;
                  for (ii = j; ii < ilim; ii++) {
                        IP = I + LE1;
                        u = cc*(*(ra+IP));
                        a = ss*(*(ia+IP));
                        u = u + a;
                        v = cc*(*(ia+IP));
                        a = ss*(*(ra+IP));
                        v = v - a;
                        *(ra+IP) = *(ra+I) - u;
                        *(ia+IP) = *(ia+I) - v;
                        *(ra+I) = *(ra+I) + u;
                        *(ia+I) = *(ia+I) + v;
                        I = I + LE;
                  }
                  k = k + ntotal/LE;
            }
      }
};

void FFT::generate_signal(float *ra,float *ia,int ntotal) {
// Inicializa a los vectores ra,ia. E·n este caso se utililiza una
// función senoidal real.
      for (int i = 0; i < ntotal; i++) {
            *(ra+i) = 10*sin(i*8*3.1415926/ntotal);
            *(ia+i) = 0.0;
      };
};

void FFT::restore(float *ra,float *ia,float* RTemp,float *ITemp,int
nlocal) {
// Realiza una copia de los vectores RTemp,ITemp sobre los vectores
// ra,ia, respectivamente.
      for(int i = 0;  i < nlocal; i++) {
            *(ra+nlocal+i) = *(RTemp+i);
            *(ia+nlocal+i) = *(ITemp+i);
      };
};

void FFT::back(float *RTemp,float *ITemp,float *ra,float *ia,int
nlocal) {
// Realiza una copia de los valores de ra,ia sobre los vectores
// RTemp,ITemp respectivamente.
      for(int i = 0; i < nlocal; i++) {
            *(RTemp+i) = *(ra+nlocal+i);
            *(ITemp+i) = *(ia+nlocal+i);
      };
};
```

Como puede observarse, la mayoría de los métodos de la clase FFT se determinan de las observaciones obtenidas a partir del análisis orientado a objetos del problema. A estos métodos, se añaden algunos otros auxiliares que durante el diseño se ha observado son operaciones comunes de la clase que se ejecutan frecuentemente al calcular una FFT.

Por otra parte, en cuanto a los datos se ha preferido para la implementación el uso de dos vectores reales para representar un vector complejo, tanto en los datos como en los resultados de la FFT, a fin de reducir las operaciones a realizar sobre vectores complejos a funciones entre sus elementos como números reales. La misma consideración se hace para el vector complejo de factores de peso, que igualmente se representa mediante dos vectores reales.

Para ilustrar el uso de la clase FFT, a continuación se ejemplifica su uso en una arquitectura paralela, variando el número de objetos activos creados y el número de procesadores del hipercubo donde se ejecuta.

1. Configuración utilizando dos objetos.

El hipercubo más sencillo que puede construirse consiste en dos objetos de la clase FFT que se reparten la tarea de calcular una FFT, Considerando un diseño incremental [III.C.1.f], inicialmente se realiza una implementación que se ejecute concurrentemente en un procesador, a fin de simplificar problemas en cuanto a la sincronización y comunicación entre ambos objetos. A partir de esta implementación, se realiza la configuración para ejecutarse en dos procesadores (Figura IV.5).

Como se puede observar, los objetos *masterfft* y *workerfft* se ejecutan paralelamente en dos procesadores. El código de cada uno refleja las acciones que cada uno representa sobre sus datos, y el envío y recepción de datos entre sí. A fin de mantener una cierta flexibilidad respecto al hipercubo, se ha preferido no tratar de encapsular la acción de los objetos en algún método, ya que al variar el número de objetos también varían las acciones que cada uno toma en el cálculo de la FFT. Para poder observar esto último, se presenta a continuación la implementación del cálculo de la FFT mediante cuatro objetos ejecutándose en una arquitectura paralela.

2. Configuración utilizando cuatro objetos.

Una vez que el diseño ha sido probado en un solo procesador, es transportable a una configuración de más procesadores. Es importante hacer notar que la definición de la clase FFT se reutiliza independientemente del número de objetos o el número de procesadores de la topología hipercubo en la que se ejecuta.

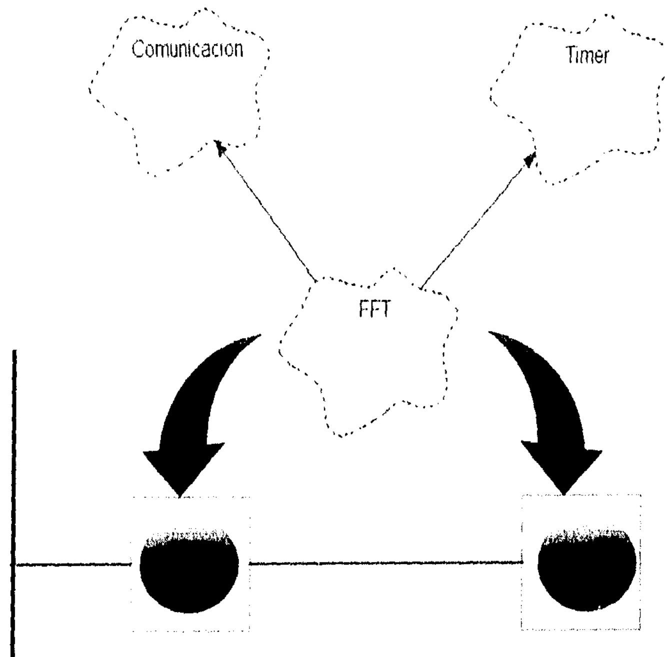


Figura IV.6, Configuración en dos procesadores.

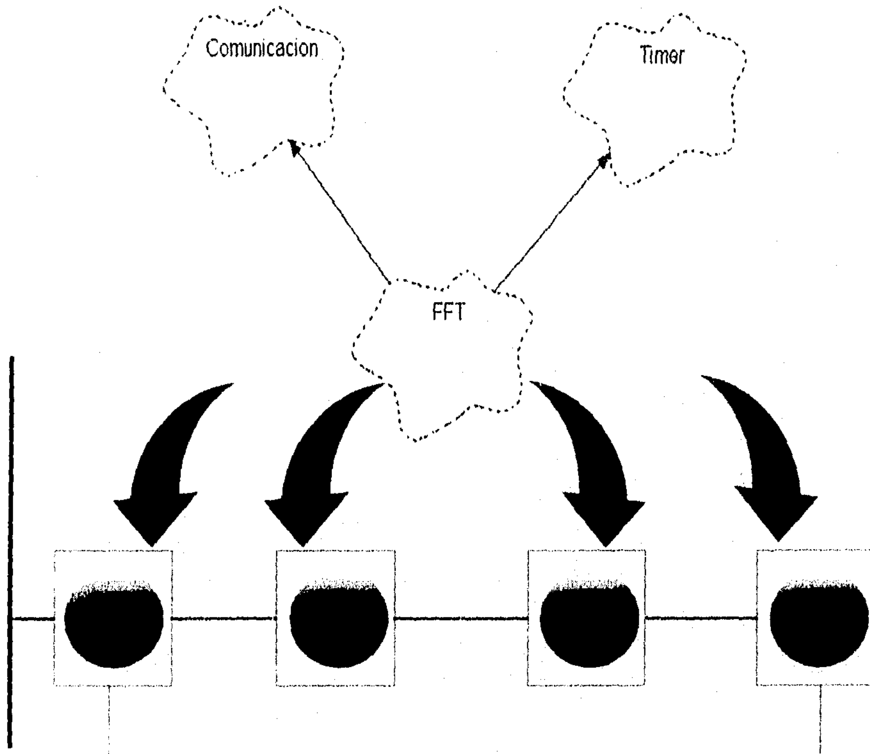


Figura IV.7, Configuración en cuatro procesadores.

Como se menciona en la sección anterior, la clase definida para la FFT es reutilizable independientemente del número de objetos involucrados, siendo únicamente necesario modificar el código que representa la actividad de tales objetos. Esto se puede observar en el caso de cuatro objetos (Figura IV.6)

En este caso, el código que representa las acciones y comunicaciones de un objeto de la clase FFT tiende a ser más complejo, debido a que en este caso cada objeto debe sincronizar su actividad selectivamente respecto a los demás objetos con los que se comunica, lo que complica la forma del código.

E. Métricas y observaciones.

A partir de la ejecución del programa FFT en uno, dos y cuatro procesadores conectados en hipercubo se obtienen las siguientes métricas a fin de evaluar la programación paralela orientada a objetos.

1. Métricas del paralelismo.

Como se mencionó anteriormente [I.A.2.d], la principal medida que se considera de un sistema de procesamiento paralelo es la velocidad con que ejecuta un programa en particular. En este punto de la evaluación y para el caso de estudio de la FFT, se ha utilizado como dato de entrada un vector real de la función seno unitario de 256 elementos. Como referencia, se utilizan las mismas medidas realizadas para el cálculo de la misma FFT mediante un programa estructurado en el INMOS ANSI C Paralelo, realizado por Dyke Stiles[1], que también se ejecuta en una topología de hipercubo.

a) Tiempo de ejecución.

Los tiempos de ejecución [I.A.2.d.*i*.] para el cálculo de a FFT mediante programación estructurada se muestran en la tabla IV.1.

		Procesos		
		1	2	4
Procesadores	1	7.68 ms	21.632 ms	24.896 ms
	2	-	10.984 ms	13.76 ms
	4	-	-	9.856 ms

Tabla IV.1, Tiempos de ejecución (ANSI C Paralelo).

[1]**North American Transputer Users Group.** Dyke Stiles, Electrical Engineering Department, Utah State University. Email dyke@opus.ee.usu.edu.

Para el caso del cálculo de la FFT utilizando objetos en procesadores paralelos, se presentan los tiempos en la tabla IV.2.

		Objetos		
		1	2	4
Procesadores	1	13.632 ms	24.192 ms	22.592 ms
	2	-	13.696 ms	19.648 ms
	4	-	-	8.768 ms

Tabla IV.2, Tiempos de ejecución (ANSI C++ Paralelo).

A partir de estos valores, es posible graficar la variación del tiempo de ejecución respecto al número de procesadores involucrados para ambos casos. Las gráficas se presentan en la figura IV.7.

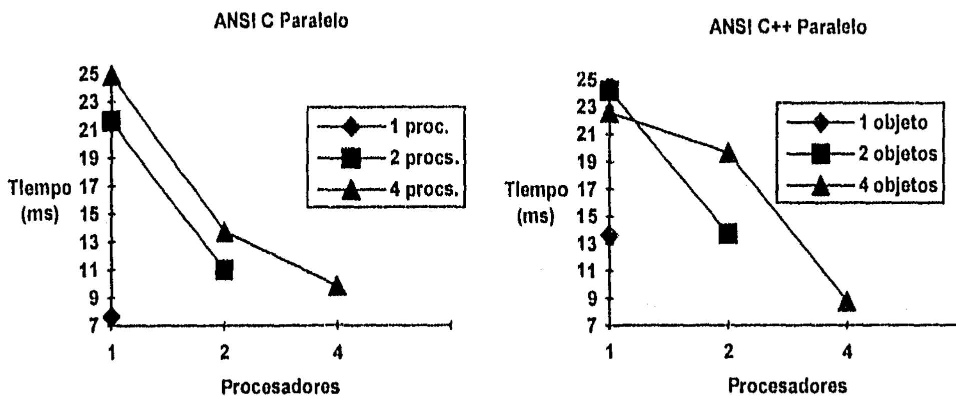


Figura IV.8, Tiempos de ejecución para el cálculo de la FFT.

A partir de las gráficas, se puede observar que en general los tiempos de ejecución del cálculo de la FFT al variar el número de procesadores son menores para la programación estructurada que los de la modelación en objetos. Sin embargo, también se observa que conforme va aumentando el número de procesadores involucrados, la diferencia entre los tiempos de ejecución en ambos casos tiende a disminuir y estabilizarse. Extrapolando los resultados a un número mayor de procesadores (8, 16, 32, etc.), es posible considerar una tendencia de estabilidad en los tiempos de ejecución.

Por otro lado, el comportamiento de las curvas para 2 y 4 elementos de procesamiento en ambos casos mantienen una semejanza a pesar de la diferencia en los tiempos. De esta forma, el uso del modelo de objetos únicamente ha impactado en el diseño en el aumento en los tiempos de ejecución del programa.

b) *Speedup.*

Tomando como base los datos de las tablas IV.1 y IV.2, se realiza el cálculo del *speedup* [I.A.2.d.*ii*] para este ejemplo. Los resultados se muestran en las tablas IV.3 y IV.4.

		Procesos		
		1	2	4
Procesadores	1	1	1	1
	2	-	1.969	1.81
	4	-	-	2.526

Tabla IV.3, *Speedup* (ANSI C Paralelo).

		Objetos		
		1	2	4
Procesadores	1	1	1	1
	2	-	1.766	1.15
	4	-	-	2.576

Tabla IV.4, *Speedup* (ANSI C++ Paralelo).

Las gráficas correspondientes del *speedup* respecto al número de procesadores involucrados se muestran en la figura IV.8.

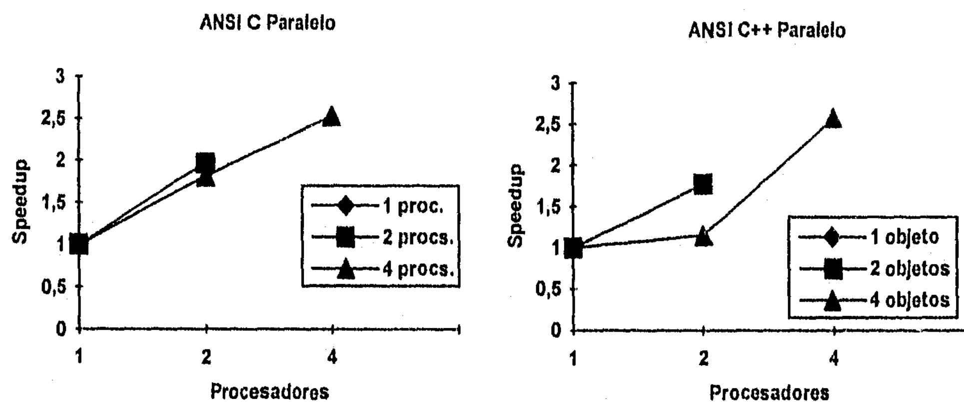


Figura IV.9, *Speedup* para el cálculo de la FFT.

De las gráficas se observa que los valores del *speedup* para los casos estructurado y orientados a objetos mantienen una semejanza que se acentúa al utilizar esta métrica. Aun cuando los valores obtenidos mediante programación estructurada se conservan por encima de aquellos obtenidos en la ejecución de los programas orientados a objetos, el comportamiento del *speedup* respecto al número de procesadores presenta una tendencia a

incrementarse para alcanzar los valores propuestos de la programación estructurada, mejorando su desempeño conforme aumenta el número de procesadores.

c) Eficiencia.

Utilizando la información de los puntos anteriores, se procede al cálculo de la eficiencia [I.A.2.d.*iii*] para cada caso de ejecución del cálculo de la FFT. La eficiencia de cada caso se presenta en las tablas IV.5 y IV.6.

		Procesos		
		1	2	4
Procesadores	1	1	1	1
	2	-	0.883	0.575
	4	-	-	0.644

Tabla IV.5, Eficiencia (ANSI C Paralelo).

		Objetos		
		1	2	4
Procesadores	1	1	1	1
	2	-	0.9845	0.905
	4	-	-	0.631

Tabla IV.6, Eficiencia (ANSI C++ Paralelo).

De igual forma que en las métricas anteriores, se presentan en la figura IV.9 las gráficas de la eficiencia obtenida para programación estructurada y orientada a objetos.

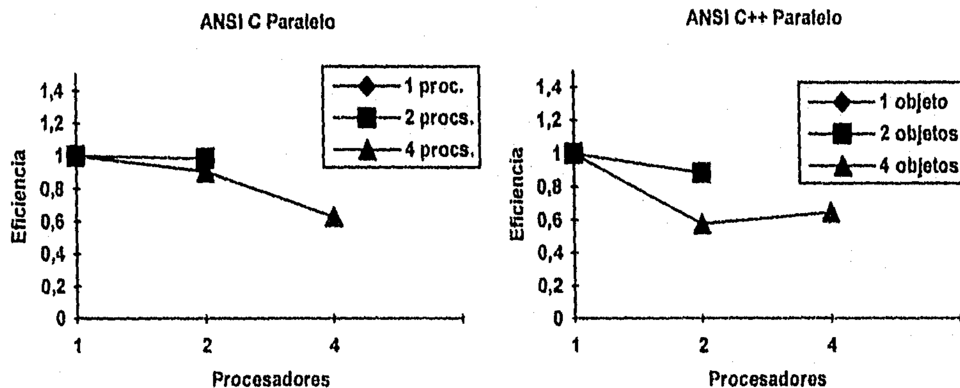


Figura IV.10, Eficiencia para el cálculo de la FFT.

Comparando ambas gráficas de eficiencia, es posible observar el hecho que conforme se va aumentando el número de procesadores involucrados, la eficiencia del programa estructurado disminuye, y a la vez, la eficiencia del programa orientado a objetos aumenta, tendientes ambos al mismo valor. De esta forma, conforme se van aumentando el número de procesadores, es posible que, como las anteriores métricas, la eficiencia tienda a estabilizarse para ambos casos.

d) Fracción serial.

Como una medida complementaria, a la vez se realiza el cálculo de la fracción serial [I.A.2.d.*iv*] para cada caso de procesamiento. Los valores obtenidos para las programaciones estructurada y orientada a objetos pueden observarse en las tablas IV.7 y IV.8, respectivamente.

		Procesos		
		1	2	4
Procesadores	1	-	-	-
	2	-	0.883	0.575
	4	-	-	0.644

Tabla IV.7, Fracción serial (ANSI C Paralelo).

		Objetos		
		1	2	4
Procesadores	1	-	-	-
	2	-	0.9845	0.905
	4	-	-	0.631

Tabla IV.8, Fracción serial (ANSI C++ Paralelo).

La figura IV.10 presenta las gráficas obtenidas para la fracción serial en ambos casos de programación. Se consideran únicamente los casos para 2 y 4 procesadores, debido a que el cálculo de la fracción serial para un solo procesador presenta una indeterminación.

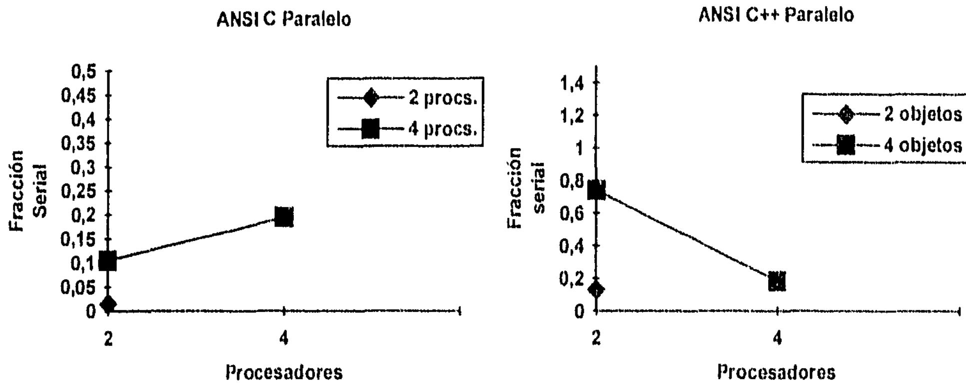


Figura IV.11, Fracción serial para el cálculo de la FFT.

Desafortunadamente, debido a la escasez de datos para el caso de estudio actual, la fracción serial provee muy poca información para la evaluación de ambos procesamientos, ya que su criterio se basa la variación de su comportamiento al incrementar el número de procesadores.

2. Métricas de la orientación a objetos.

A diferencia de las métricas del paralelismo que basa su criterio principalmente en valores objetivos como el tiempo de ejecución de un programa, las métricas de la orientación a objetos se presentan de una manera más subjetiva. Sin embargo, se ha tratado de hacer más objetivas tales métricas, siguiendo en particular algunos criterios aplicados de ingeniería de software. Así, en este punto, se pretende presentar una evaluación para el ejemplo de aplicación de la FFT utilizando las métricas propuestas por G. Booch [9, II.A.1.e]. De esta forma, se presenta la siguiente evaluación.

a) Métodos por clase.

Considerando la relación entre los métodos existentes por clase, para el caso de la clase FFT se han definido un total de 10 métodos. Durante la ejecución del programa, cada objeto realiza llamadas a tales métodos. Para los casos de 1, 2 y 4 objetos, se presenta en la tabla IV.9 el número de veces que cada llamada aparece en el código.

Clase FFT	Objetos		
	1	2	4
Masterfft	7	9	7
Workerfft	-	5	5

Tabla IV.9, Métodos ponderados por clase para el cálculo de la FFT.

Se observa a partir de los valores de la tabla que el análisis e implementación de los métodos se basa principalmente en un sistema basado en 2 objetos, el cual fue modificado para cubrir los requerimientos de los casos para 1 y 4 objetos, que no utilizan algunos métodos auxiliares. Sin embargo, en cuanto al uso extensivo de los métodos no auxiliares, la clase FFT para todos los casos es utilizada en su totalidad, debido en gran medida a que tales métodos se han implementado basándose en las definiciones y elementos propuestos para el cálculo de la FFT.

b) Profundidad del árbol de herencia.

La figura IV.2 presenta la jerarquía de herencia múltiple propuesta para el caso de la FFT. De hecho, este árbol de herencia presenta sólo dos niveles, en los que se cuenta con un hijo único para dos clases, lo que coloca a la aplicación dentro de los límites propuestos empíricamente por Booch respecto a la estructura del árbol de clases: tres clases en total, dispuestas en forma balanceada, con anchura y altura igual a dos clases.

c) Acoplamiento entre objetos.

Para el caso de estudio de la FFT, la característica de acoplamiento entre objetos se considera principalmente para los casos de 2 y 4 objetos que se comunican entre sí.

De cada ejemplo se puede observar que las acciones que los objetos deben considerar para comunicarse entre sí, conforme el número de objetos aumenta, se complican, debido a que cada uno debe seleccionar e intercambiar datos con el vecino correcto dentro del hipercubo.

Sin embargo, el acoplamiento entre los objetos se puede considerarse como bajo o, en un caso extremo, medio, ya que el intercambio de datos se realiza sólo cuando es necesario durante cada iteración. Por otro lado, debido a la ejecución simultánea sincronizada de los objetos, en la cual cada uno realiza en cada iteración aproximadamente el mismo procesamiento sobre los datos, existe en realidad poco tiempo de espera entre ellos, lo que permite considerar cada uno como una unidad de procesamiento independiente.

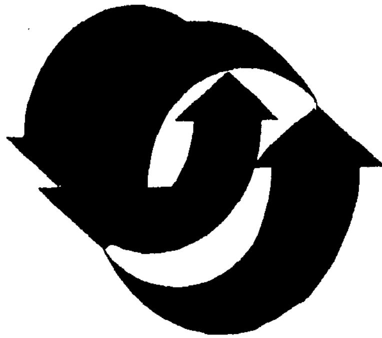
d) Cohesión en métodos.

Debido a que la FFT es un problema matemático ampliamente

estudiado y desarrollado, durante las fases de análisis del problema los métodos se definen directamente a partir del modelo matemático. Las funciones que definen los métodos de la clase FFT se realizan a partir de las acciones necesarias para el cálculo, y se implementan reutilizando en varios casos un código diseñado con anterioridad. Esto permite que su ejecución por parte del objeto se realice relacionando fuertemente actividades entre sí. De esta forma, se puede considerar que exite una alta cohesión entre los métodos de la clase FFT.

3. Observaciones.

Es importante observar que el ejemplo de aplicación de la FFT de este capítulo sirve únicamente como caso de estudio de un análisis y diseño orientado a objetos en una arquitectura paralela. Aun cuando la FFT es un algoritmo artimético, el ejemplo muestra la factibilidad de desarrollar aplicaciones utilizando objetos activos que cooperan entre sí como unidades de procesamiento dentro de un sistema multiprocesador. Aun cuando su desempeño no es mejor que el de un sistema paralelo tradicional, tiene la ventaja de contar con los elementos del modelo de objetos [II.A.2] que facilitan en gran medida el desarrollo de la programación en términos de entendimiento y, por lo tanto, de reusabilidad.



Capítulo V. Conclusiones generales y orientaciones.

"Lo que tiene de grande el hombre es el ser puente y no fin; lo que puede amarse en el hombre es el ser tránsito y hundimiento"
Federico Nietzsche.

No todas las áreas de aplicación se verán afectadas por la tecnología orientada a objetos. Serán favorecidas aquellas en las que las relaciones entre sus elementos y datos llevan la información clave, como es el caso de las aplicaciones técnicas y de ingeniería que manejan complejos tipos de datos y capturan la estructura de los mismos.

A. Conclusiones.

En un entorno de cálculo paralelo las técnicas orientadas a objetos facilitan la cooperación entre sus elementos, ya que estos se presentan como módulos que contienen ambos procedimientos y datos, lo que los hace intrínsecamente mejor dotados para la cooperación. Además, los métodos orientados a objetos permiten la construcción de aplicaciones que contienen objetos activos, los cuales ejecutan tareas bien definidas tales como cálculos y clasificaciones, con un desempeño mayor que los sistemas uniprocesador actuales. Sin embargo, esto no justifica aún su uso en la actualidad en sistemas de procesamiento en tiempo real.

La arquitectura tiene un papel muy importante en el desempeño de un sistema de cómputo, en especial en los sistemas paralelos. Actualmente, las arquitecturas paralelas aún presentan una gran dependencia a sistemas secuenciales que les sirven como interface, dando como resultado que el sistema paralelo actúe únicamente como un

"coprocesador" poderoso y dependiente del sistema secuencial. Esto se debe principalmente a la falta de experiencia en el desarrollo de arquitecturas paralelas independientes, las cuales tienen la posibilidad de desarrollarse y controlarse mediante metodología y técnicas orientadas a objetos, es decir, utilizando un arquetipo paralelo orientado a objetos donde los objetos activos sean los elementos básicos de programación. De esta forma, es posible a la vez considerar la capacidad de los sistemas paralelos potencialmente como las arquitecturas orientadas a objetos.

La principal desventaja que los sistemas orientados a objetos presentan en el desarrollo de sistemas multiprocesador es la poca mejoría en la definición en cuanto a los mecanismos de sincronización y comunicación. El modelo de objetos permite facilitar su uso al dar una mayor expresividad a los métodos de comunicación. Sin embargo, no presenta una solución directa a los problemas de complejidad en la comunicación entre elementos paralelos, ya que estos dependen directamente de la topología y cantidad de mensajes entre los procesadores.

Por otro lado, actualmente se ha desarrollado un fuerte interés en la ingeniería de software particularmente en el desarrollo y soporte de sistemas paralelos, a fin de contar con productos de *software* correctos, utilizables y costo efectivos para arquitecturas multiprocesador. Se proponen la aplicación y extensión de métodos de análisis y diseño orientados a objetos tradicionales que consideren la expresión del paralelismo y secuencialidad, así como el uso del modelo de objetos activos o actores "versátiles", con capacidades de acción sobre los objetos pasivos dentro de un ambiente paralelo o distribuido. Gran parte de los proyectos de investigación en ingeniería de software han basado su desarrollo considerado a C++ como lenguaje paralelo orientado a objetos. Sin embargo, también se presentan ejemplos de esta tendencia en otros lenguajes basados en Eiffel y Smalltalk.

B. Orientaciones.

Debido a que se trata de un campo nuevo, la programación paralela orientada a objetos está generando grandes esfuerzos en investigación, siguiendo principalmente las siguientes directivas:

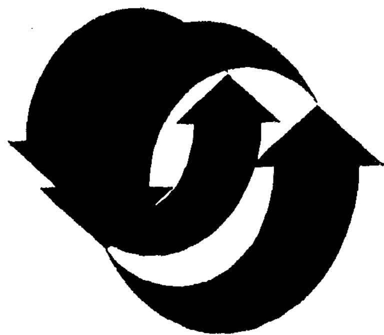
- Inicialmente, la orientación a objetos se ha definido más conceptualmente que formalmente. Esto puede presentarse como una área de investigación para proveer descripciones más formales que permitan expresar el comportamiento y comunicación entre objetos activos.

- El mecanismo de herencia es uno de los elementos más atractivos del modelo de objetos. Sin embargo, la reutilización de las especificaciones de sincronización entre objetos mediante herencia tiende a dar resultados

complejos en la programación. Encontrar especificaciones de sincronización más expresivos, genéricos y reutilizables es también un área de investigación a desarrollar, a fin de obtener esquemas de comunicación suficientemente abstractos, independientes de métodos y eficientes.

- Otra importante área de investigación es la creación y desarrollo de ambientes de programación para satisfacer la necesidad de visualizar y controlar la actividad y cooperación entre objetos en una arquitectura paralela. Se han realizado algunos trabajos sobre aspectos específicos de visualización y compilación, pero aun se requiere un buen conjunto integrado de herramientas genéricas.

Para concluir este trabajo, es importante hacer notar que las fronteras entre lenguajes de programación, sistemas operativos y arquitecturas se hacen cada vez más difíciles de definir. Existe una gran actividad de investigación sobre este tema, con la finalidad de obtener una óptima integración entre estas perspectivas, distintas pero complementarias, de la computación.



Apéndice A.
Antecedentes del paralelismo.

Utilizar una gran número de procesadores para aumentar el poder de procesamiento parece una solución simple para varios problemas de cómputo; sin embargo, esto no significa que siempre es posible mejorar el desempeño de una arquitectura de procesamiento, añadiendo cada vez más procesadores, o que es fácil encontrar la manera de dividir un problema en partes, de tal manera que pueda ser expresado en términos de procesos paralelos. En términos reales, sí existe un límite para el paralelismo. Desde su inicio, la programación paralela ha sido tema de discusión para varios autores, quienes han presentado sus trabajos apoyando o criticando al paralelismo. De aquí, han surgido algunos comentarios importantes respecto al desempeño paralelo de sistemas. Entre los más conocidos se encuentran la *Ley de Amdahl* y la *Ley de Gustafson-Barsis.*

La Ley de Amdahl.

En 1967, Gene Amdahl, diseñador de la compañía IBM, hizo la siguiente declaración:

> *"Por más de una década, los 'profetas' han clamado que la organización de computadoras de un solo procesador ha alcanzado sus límites, y que verdaderos y significativos avances sólo pueden lograrse mediante la interconexión de una multiplicidad de computadoras de tal manera que permita una solución cooperativa. La naturaleza de un aumento en el tiempo de ejecución (en paralelo) parece ser secuencial, por lo que no parece ser manejable por técnicas de procesamiento en paralelo. Tal aumento en el tiempo de ejecución por sí mismo coloca un límite superior, en una relación de cinco a siete veces de la velocidad de un procesamiento*

secuencial, aun cuando las acciones de mantenimiento se realizaran en un procesador por separado. En cualquier punto en el tiempo, es difícil prever cómo los cuellos de botella anteriores en una computadora secuencial serán efectivamente superados" [51].

Conocida como la ***Ley de Amdahl***, esta afirmación ha sido cuantificada mediante una fórmula matemática. Supóngase una computadora paralela de N procesadores interconectados por un esquema cualquiera. El propósito de esta máquina es ejecutar un programa que consiste en partes naturalmente paralelas y seriales. De hecho, supóngase como β a la fracción serial del programa, y como ($1-\beta$) la fracción paralela.

Ahora bien, sea S la relación entre los tiempos de ejecución de un procesador y N procesadores. Entonces, S se define como sigue:

$$S = \frac{T(1)}{T(N)}$$

donde $T(j)$ es el tiempo requerido para resolver un problema utilizando j procesadores.

La parte serial del programa puede calcularse en el tiempo mediante el producto $\beta T(1)$, y la parte paralela, como $(1-\beta)T(1)/N$, suponiendo la idea que *"N trabajadores deben hacer el trabajo en una fracción 1/N del tiempo que le lleva a un solo trabajador"*[51]. De esta manera, el tiempo de ejecución en N procesadores queda de la siguiente manera:

$$T(N) = T(1)\beta + \frac{T(1)(1-\beta)}{N}$$

Sustituyendo en la expresión de S (conocida como *speedup* del sistema), tenemos lo siguiente:

$$S = \frac{1}{\beta} + \frac{(1-\beta)}{N}$$

$$S = \frac{N}{\beta N + (1-\beta)}$$

(Ley de Amdahl)

La *Ley de Amdahl* establece una relación extremadamente pesimista entre los tiempos de procesamiento, basado principalmente en la relación fraccional entre las

partes que se procesa en serie y en paralelo del programa. Según esto, se establece un máximo del 50% de mejora utilizando 10 procesadores en lugar de uno solo [51]. Económicamente hablando, tal mejora en el desempeño no compensaba el precio del *hardware* de los años 60. De esta manera, esta afirmación provocó que durante muchos años, el paralelismo fuera evitado en el diseño de computadoras, pues aun el más sencillo programa paralelo contiene una cierta cantidad de código serial.

La Ley de Gustafson-Barsis.

En 1988, John Gustafson y Ed Barsis, de los Sandia Laboratories, realizaron una importante observación, replanteando la *Ley de Amdahl*:

> "*La expresión y gráfica (de la Ley de Amdahl) contienen la suposición implícita que (1-β) es independiente de N, lo cual virtualmente nunca es el caso*"[51].

La clave es entonces, la dependencia entre β y N. De nuevo, es necesario iniciar con la expresión del *speedup* S.

$$S = \frac{T(1)}{T(N)}$$

La interpretación de Gustafson y Barsis fue que si se utiliza un solo procesador, éste debe calcular ambas partes serial y paralela, de tal manera que:

$$T(1) = \beta + (1 - \beta)N$$

Pero, si se utilizan N procesadores, el problema puede ser escalado para que todos los N procesadores paralelos ejecuten la parte serial y paralela del programa, uno tras otro. Así, al sumar las partes serial y paralela, se tiene la unidad.

$$T(N) = \beta + (1 - \beta) = 1$$

Substituyendo ambas expresiones en la definición de *speedup* S, se tiene que:

$$S = N - (N - 1)\beta$$

(Ley de Gustafson-Barsis)

Esta fórmula fue desarrollada exactamente igual que la *Ley de Amdahl*, excepto que T(N) es reducido a 1, lo que significa que el problema es escalado especialmente para ajustarse a una computadora paralela. Por otro lado, y contrariamente, T(1) es el tiempo para calcular las partes paralela y serial de un programa en un solo procesador.

Obviamente, la diferencia entre la *Ley de Amdahl* y la *Ley de Gustafson-Barsis* es debida a que esta última considera el problema escalable para ajustarse al paralelismo del problema.

Consideraciones como éstas sobre el desempeño del paralelismo fueron las tendencias que tuvieron una mayor influencia sobre la creación de sistemas paralelos comerciales durante varias décadas, basando sus observaciones en suposiciones sobre el comportamiento de los sistemas paralelos, y no en la factibilidad de su desarrollo. Sin embargo, a pesar de las críticas, un grupo de investigadores continuaron los trabajos sobre concurrencia y paralelismo. De estos últimos, se presenta un breve análisis de aquellos que se han considerado como los más importantes a nivel tecnológico y científico. Tales trabajos han servido como la base sobre la cual se han escrito varios libros sobre temas de programación concurrente, paralela y distribuida.

1. Concurrencia.
Dijkstra, Hoare y Brinch-Hansen.

E.W. Dijkstra.
"Co-operating Sequential Processes"[23].

En 1968, E. W. Dijkstra presenta un artículo, con el cual realiza una primera propuesta del tratamiento de la concurrencia en un lenguaje de programación, inicialmente utilizando elementos de la programación secuencial, y finalmente mediante nuevos elementos de programación, utilizando una notación semejante al lenguaje de programación **ALGOL60** (*Algoritmic Language*).

En este artículo, Dijkstra añade algunos conceptos nuevos a la programación secuencial, a fin de extenderla en una *programación concurrente*. Define elementos tales como las *corrutinas* o procesos concurrentes (utilizando las palabras reservadas **parbegin** y **parend**, interpretándose como la ejecución paralela del programa), la *exclusión mutua*, la *sincronización de eventos*, *secciones críticas*, y la definición de nuevos tipos de datos llamados *semáforos*. Además, Dijkstra hace especial mención de un nuevo problema que se genera al intentar ejecutar procesos concurrentes, refiriéndose con nombre de *abrazo mortal* (*deadly embrace* o *deadlock*).

La idea central del artículo de Dijkstra se basa en el intercambio de información entre procesos secuenciales independientes que se ejecutan a diferentes velocidades, mediante variables compartidas, las cuales se

modifican utilizando operaciones indivisibles de inspección y asignación. Los objetivos de la programación de tales procesos, para Dijkstra, son la claridad del código y la seguridad de la información compartida.

Dijkstra hace un análisis de los nuevos problemas que surgen a partir de la paralelización de procesos. Inicialmente, los programadores de sistemas secuenciales sólo debían resolver dos problemas principalmente: el hecho que el programa tuviera una finalización en el tiempo, y que su funcionamiento presentara resultados correctos. Sin embargo, la nueva programación presenta problemas totalmente nuevos, como la expresión del paralelismo, el no determinismo, la exclusión mutua y la sincronización entre eventos. A partir de estos problemas, Dijkstra propone una serie de algoritmos para dar con una solución satisfactoria, utilizando para esto elementos de la programación secuencial. Es así como da a conocer un algoritmo que soluciona los problemas de exclusión mutua y sincronización entre procesos [23]. Tal algoritmo se atribuye al matemático Th. J. Dekker. Sin embargo, a pesar de haber hallado tal respuesta a los problemas que surgen de la programación concurrente, Dijkstra prosigue su análisis con el fin de hallar una solución aun mejor, ya que la solución de Dekker presenta el inconveniente de ser demasiado compleja.

De esta manera, propone la creación de un nuevo tipo de datos de propósito especial, los *semáforos*, que permiten controlar el acceso a los datos compartidos, conocido como *sección crítica* del programa. La conclusión del trabajo de Dijkstra gira alrededor del uso de los semáforos para solucionar algunos problemas que en la actualidad se consideran clásicos, como los algoritmos de productores-consumidores, el barbero dormido y del banquero.

La importancia del trabajo de Dijkstra radica en que propone las bases sobre las que se soportan los conceptos de la programación paralela y distribuida en la actualidad. En especial, ha sido mencionado como un antecedente en los trabajos de otros investigadores como Habermann [33], Peterson [65], Hoare, Brinch-Hansen y otros quienes han propuesto nuevas técnicas para la solución de los problemas de exclusión mutua y sincronización, alrededor de la idea de los semáforos.

C.A.R Hoare.
"Towards a theory of Parallel Programming"[35].

Aun cuando el trabajo de Dijkstra sobre procesamiento concurrente marca un inicio en el desarrollo de una programación paralela, otros autores

han continuado y contribuido a tal desarrollo. Este es el caso de Hoare, que en 1972 presenta un trabajo basado en los conceptos expuestos por Dijkstra.

En su artículo, Hoare realiza un análisis de las características de una programación paralela, proponiendo un nuevo mecanismo para el manejo de la exclusión mutua y sincronización, conocido como *regiones críticas*, basadas en el concepto de semáforos de Dijkstra, en las que se regula la entrada a una sección de código, que funciona como el recurso común de un grupo de procesos que cooperan entre sí.

Además, en el mismo trabajo introduce el concepto de *comandos custodiados* o *custodias*, a fin de permitir un control sobre la entrada a la región crítica. Finalmente, hace mención de algunas definiciones formales a fin de probar la corrección de la programación, como son las *invariantes*, *precondiciones* y *postcondiciones*.

Es importante mencionar que en este artículo Hoare presenta una primera propuesta de las características de una programación paralela para el diseño de un lenguaje de alto nivel. Los objetivos principales de esta propuesta son la **seguridad contra errores**, la **eficiencia**, la **simplicidad conceptual** y la **amplitud de aplicaciones**. A partir de estas consideraciones llega a la siguiente conclusión:

"El diseño de lenguajes de programación de alto nivel que simultáneamente satisfagan estos cuatro objetivos es uno de los mayores retos para la invención, imaginación e intelecto de los científicos de la computación".

P. Brinch-Hansen.
"Structured Multiprogramming"[11].

Además de Hoare, otra figura dentro del procesamiento concurrente ha sido Brinch-Hansen. En un artículo publicado también en 1972, propone conceptos semejantes a los expuestos por Hoare para el manejo de la exclusión mutua y sincronización. Sin embargo, Brinch-Hansen considera las regiones críticas condicionales de Hoare como un caso especial de las regiones críticas que él propone.

Brinch-Hansen utiliza los conceptos de procesos concurrentes mediante la notación **cobegin ... coend** de Dijkstra, ya que esto simplifica la comprensión y verificación de un programa de una manera considerable.

Aún cuando los conceptos considerados en este trabajo son muy semejantes a los expuestos por Hoare, como es el caso de los *procesos disjuntos*, la exclusión mutua mediante regiones críticas, y la comunicación entre procesos concurrentes, el trabajo de Brinch-Hansen además propone una primera idea para la implementación de rutinas de espera, utilizando para esto una estructura de cola. Esto se utilizó posteriormente en algunos lenguajes de programación que incluyen facilidades para el manejo de la concurrencia.

Finalmente, menciona en su conclusión que los conceptos tratados en este trabajo son el fundamento para la implementación de procedimientos *monitores*, tratados por él mismo y por Hoare más adelante.

C.A.R. Hoare.
"Monitors: An operating system structuring concept"[36].

Para el año de 1975, Hoare publica otro artículo en el que desarrolla el concepto de Brinch-Hansen de *monitor*, como un método para el desarrollo de procesos concurrentes dentro de un sistema operativo. La exclusión mutua y sincronización entre procesos queda definida a través de una colección de datos locales, funciones y procedimientos, los cuales en su conjunto forman un *monitor*. Tal estructura como recurso compartido, puede implementarse mediante semáforos, utilizando las clásicas operaciones de *wait* y *signal* para regular el uso del código compartido.

El concepto de monitor va más allá de considerar únicamente la acción de procesos concurrentes sobre datos compartidos. Se trata más bien de localizar los procedimientos y funciones que actúan sobre tales datos, restringiéndose la invocación de tales procedimientos y funciones mediante semáforos. Haciendo una comparación con las regiones críticas podemos considerar que éstas restringen el acceso sobre datos (que actúan como recurso compartido), mientras que los monitores restringen el acceso a los procedimientos y funciones de datos locales, como recursos compartidos entre los procesos concurrentes.

Finalmente, Hoare propone en la conclusión de este trabajo una serie de observaciones sobre el concepto de monitor, con la finalidad de facilitar la implementación de los mismos dentro de un sistema operativo.

P. Brinch-Hansen.
"The programming language Concurrent Pascal"[12].

También en 1975, Brinch-Hansen publica un trabajo en el cual propone a *Pascal Concurrente* como un nuevo lenguaje de programación para sistemas operativos. Basado en el lenguaje Pascal [80], y en los conceptos de procesos concurrentes y monitores, utiliza una notación semejante a las clases de **Simula 67** para mostrar una definición informal de Pascal Concurrente. Según el propio Brinch-Hansen:

"La principal contribución del Pascal Concurrente es extender el concepto de monitor mediante una jerarquia explícita de derechos de acceso a estructuras de datos compartidos, que pueden ser declarados en el texto del programa y verificados por un compilador".

2. Paralelismo. Hoare y CSP.

El trabajo de Hoare ha sido fundamental para el desarrollo de la programación paralela en los últimos años. Su principal contribución es la definición de un lenguaje de especificación formal de algoritmos paralelos, conocido como CSP (*Communicating Sequential Processes*). Este lenguaje es definido en el siguiente artículo.

C. A. R. Hoare
Communicating Sequential Processes.[39]

En este artículo, Hoare inicia realizando un análisis de estructuras básicas utilizadas en la programación: asignación, secuenciación, repetición, y alternativa. Comenta nuevas estructuras para expresar la concurrencia y comunicación entre procesos, dentro de una arquitectura multiprocesador.

Las características principales de un lenguaje paralelo, las expresa con las siguientes propuestas:

- Utilizar los conceptos de *comandos custodiados*, introducidos por Dijkstra, como un medio para el control del no determinismo.

- Considerar un comando paralelo basado en el **parbegin...parend** de Dijkstra, el cual especifique la ejecución paralela de los comandos o procesos que lo componen. Se considera que todos los procesos se inician simultáneamente, y el comando paralelo

finaliza únicamente cuando todos sus procesos constituyentes terminan.

- Uso de formas simples de entrada y salida para la comunicación entre procesos.

- La comunicación entre procesos se establece únicamente en el caso en el que un proceso invoca a otro como salida y además, éste invoca a aquél como entrada. En tal caso, el valor de salida es copiado del proceso emisor como un valor de entrada al proceso receptor. No existe ningún almacenamiento temporal durante la comunicación, es decir, los procesos que invocan a otro en una operación de entrada o salida, detienen su ejecución hasta que este último puede enviar o recibir los datos. El retraso es invisible para ambos procesos.

- Los comandos de entrada de datos pueden aparecer como custodias. Un comando custodiado por una entrada se ejecuta sólo cuando el proceso asociado a la entrada ejecuta el comando de salida. Si varias custodias de entrada se encuentran en esta situación, sólo una es ejecutada y las otras no tienen efecto; tal elección es arbitraria.

- Los comandos de repetición también pueden utilizar custodias. El comando repetitivo termina cuando todas las llamadas a él terminan.

- Se propone la utilización de un elemento de correspondencia de patrones para discriminar la estructura de una entrada. Tal elemento se usa para inhibir la entrada de mensajes que no presenten el patrón especificado.

Estas características determinan en gran medida el comportamiento del lenguaje propuesto: primero, se trata de un lenguaje estático, ya que puede utilizarse tanto en una máquina convencional de almacenamiento único, o en una red fija de procesadores conectados entre sí mediante canales de entrada/salida; segundo, y debido al primer punto, no se permite la característica de recursión, presente en otros lenguajes de programación; y tercero, el lenguaje se restringe al mínimo necesario, a fin de obtener una implementación de aplicaciones más flexible.

Aun cuando este artículo de Hoare es una base para el desarrollo de la programación paralela, tiene la desventaja de no aportar un método de prueba que asista a la construcción y prueba de programas correctos. En realidad, tal método es expuesto en detalle en un libro del propio Hoare sobre el mismo tema [13].

3. Distribución. Brinch-Hansen y RPC's.

Al mismo tiempo, con el advenimiento de la comunicación entre computadoras a través de una red, Brinch-Hansen desarrolla un tipo de paralelismo, utilizando los procesadores de las computadoras conectadas a una red, en el siguiente trabajo.

Per Brinch-Hansen
Distributed Processes: A concurrent programming concept.[46]

La propuesta de Brinch-Hansen para la programación paralela, se basa propiamente en la intercomunicación de sistemas monoprocesador, mediante un sistema de red de cómputo. Su trabajo introduce el concepto de *distribución* como una forma de programación concurrente. Se enfoca principalmente en la programación en tiempo real, la cual según su opinión, cuenta con las siguientes características:

1. La programación en tiempo real interactúa con un ambiente en el que se suscitan eventos simultáneos a una gran velocidad.

2. Un programa en tiempo real debe responder a una serie de entradas *no determinísticas* del ambiente, es decir, no es posible predecir el orden en que las entradas serán efectuadas, pero debe estar preparado para responder en un límite de tiempo.

3. Un programa en tiempo real controla una computadora con una configuración fija de procesadores y periféricos, realizando en la mayoría de los casos un número fijo de tareas concurrentes en su ambiente.

4. Un programa en tiempo real nunca termina, sino que continúa sirviendo a su ambiente mientras la computadora se encuentre funcionando.

A partir de estas consideraciones, y basándose en el lenguaje Pascal Concurrente propuesto por él mismo, analiza y propone las propiedades de un nuevo lenguaje de programación para aplicaciones en tiempo real. Estas son:

- Un programa en tiempo real consiste en un número fijo de procesos concurrentes que se inician simultáneamente. Cada proceso cuenta con variables propias; sin embargo, no deben existir variables comunes entre los procesos.

- Un proceso puede hacer una *llamada a procedimientos comunes* definidos en otro proceso. Esta es la única forma de comunicación entre procesos.

- Los procesos se sincronizan mediante instrucciones no determinísticas llamadas *comandos custodiados o custodias.*

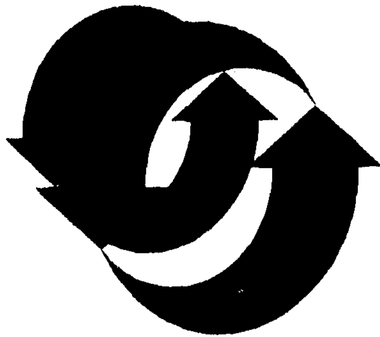
- Los procesos pueden ser utilizados como módulos de programación en un sistema multiproceso, con una *memoria compartida o distribuida.*

- Para satisfacer los requerimientos de la programación en tiempo real, cada procesador del sistema se dedica a un solo proceso.

Estos conceptos principalmente han dado origen a lo que en la actualidad se conoce como **llamadas a procedimientos remotos (o RPC's)**, que forma la base de la programación distribuida actual.

El artículo continúa con una descripción de un lenguaje distribuido basado en Pascal Concurrente, ejemplos sobre algunas implementaciones y algunas ideas sobre la implementación. Finalmente, a manera de conclusión, menciona que las propiedades de la programación distribuida son muy semejantes a la propuesta hecha por Hoare en su artículo sobre CSP.

La importancia de este trabajo recae no solamente en el uso actual de los RPC's, sino principalmente en las consideraciones hechas sobre la programación en tiempo real.



Apéndice B.
Antecedentes de la orientación a objetos.

A fin de dar un panorama de la manera como se ha desarrollado la programación orientada a objetos, es necesario hacer mención a algunos de los principales lenguajes orientados a objetos: Simula, Smalltalk, Eiffel y C++. Se considera una serie de artículos publicados en 1992, conmemorando el 25 aniversario de la orientación a objetos. Recopilado por Ron Kerr bajo el título de *"Objects strike silver"* [47], la serie de artículos presenta a los autores de los lenguajes orientados a objetos antes mencionados. A continuación, se presenta un resumen de cada uno de los artículos.

1. Simula. O. Dahl, K. Nygaard.

Hasta ahora, existen más de 100 diferentes lenguajes *basados en objetos* y *orientados a objetos*. Sin embargo, se coincide que el antecesor común de tales lenguajes es **Simula**, desarrollado en los años 60's por un grupo de trabajo del Centro de Cómputo Noruego, bajo las órdenes de Ole-Johan Dahl y Kristen Nygaard.

Simula se basa principalmente en **ALGOL**, añadiendo conceptos de encapsulación y herencia. Sin embargo, la principal importancia de Simula radica en que introdujo la disciplina de escribir programas que reflejan directamente el vocabulario del dominio del problema [9].

Kristen Nygaard, "To program is to understand"[60].

Simula tiene sus orígenes en la simulación de procesos físicos e industriales. Inicialmente, también se consideró como un lenguaje para la descripción de procesos complejos mediante una computadora.

La idea inicial fue intentar dar solución a un gran número de problemas que aparecieron con el desarrollo de los sistemas operativos, interfaces y sistemas de comunicación. Para esto, era necesario la creación de un lenguaje basado en un número reducido de conceptos poderosos. Según las propias palabras de Nygaard: *"un nuevo lenguaje que provea de una nueva perspectiva, una nueva manera de describir y entender el mundo"* [60].

Para la creación de la primera versión de Simula (conocido como **Simula 1**), se tomaron en cuenta una gran cantidad de conceptos contenidos en el lenguaje estructurado **ALGOL 60**. Cinco años después aparecen los conceptos de *objetos* y *clases*, en **Simula 67**.

Desde el punto de vista de Simula, la modelación del mundo real se realiza mediante una perspectiva, que consiste esencialmente en dos partes: la primera, es un filtro que separa las características que se desean modelar del total existente, ya que la cantidad de información del mundo real es extremadamente grande y compleja para considerarla por completo. La segunda parte se forma de un conjunto de conceptos en términos de los cuales sea posible interpretar y entender lo que se ha filtrado.

"Simula es una perspectiva del mundo" [60]. Emplea *objetos* para describir los fenómenos (y no precisamente las cosas) y categorías o *clases* para describir conceptos. Permite, por lo tanto, reflejar ambos mundos: el físico y el mental. Para la programación orientada a objetos, el mundo real no es más que una serie de modelos.

Por otra parte, para lograr un efecto aun más realista, Simula incluye el concepto de *corrutina*, tratando a la larga de soportar la concurrencia real. Este concepto sería más adelante abandonado por la mayoría de los lenguajes que sucedieron a Simula, como es el caso de Smalltalk y C++.

2. Smalltalk. A. Kay.

Smalltalk fue creado por miembros del Centro de Investigación de Xerox en Palo Alto, Ca., como parte del proyecto **Dynabook** bajo la dirección de Alan Kay.

Smalltalk representa tanto un lenguaje como un ambiente de desarrollo. Toma sus principales ideas de Simula, del lenguaje **FLEX**, y de otros trabajos. Se le considera un lenguaje orientado a objetos puro, dado que en su interior todo se considera como una clase o un objeto. Su importancia dentro del desarrollo de la programación orientada a objetos, fue el añadir interfaces gráficas de usuario (*Graphic User Interfaces* o *GUI*), lo cual se ha reflejado en la mayoría de los lenguajes que le sucedieron [9].

Alan Kay, "The natural history of objects"[43]

"Simula fue difícil de aprender; utilizaba un lenguaje extraño, debido a que los autores pensaron en Simula como un lenguaje de simulación de eventos discretos" [43]. De esta manera Alan Kay se refiere a su primer experiencia en objetos. Más adelante, él mismo aceptaría que Simula trataba de ser algo más que trabajar con tipos de datos abstractos, visión que se ha perdido en nuestros días.

El problema al que los desarrolladores de *software* se han enfrentado en los últimos años, se relaciona con la construcción de sistemas complejos, con diferentes equipos de *hardware*. Debido a esto, ninguna aplicación diseñada puede considerarse terminada al entregarse. Por esto último, se propone que un sistema de programación de tipo más modular y "biológico" sería ideal [43]. Un lenguaje orientado a objetos cumple con ambas características.

Smalltalk surge como el resultado (por parte del *software*) del proyecto **Dynabook**, cuya principal característica fue introducir una interface gráfica al usuario con ventanas. Combina los conceptos de objeto y las clase al estilo de Simula, añadiendo otras características. La primera versión comercial se conoce como **Smalltalk 80.** Es tal vez esta versión la que ha contribuido de una forma definitiva en la manera en que el usuario final se encuentra frente a la computadora. Gran cantidad de las interfaces gráficas de usuario que en la actualidad se utilizan, han sido influidas por Smalltalk.

Sin embargo, el trabajo relacionado con un diseño orientado a objetos no es sencillo. *"Tratar* (el problema) *con fineza, en lugar de soluciones directas, cuando se trata de hacer software. La naturaleza usa una estrategia fina, y nosotros debemos hacer lo mismo".* Con esto, Kay resume la idea que el modelo de objetos persigue.

3. Eiffel. B. Meyer.

Eiffel surge como resultado del trabajo de Bertrand Meyer, no sólo como un lenguaje orientado a objetos, sino también como una herramienta para la Ingeniería de Software.

Aun cuando Eiffel se encuentra influenciado por Simula, su diseño desde un principio tenía la finalidad de ser un lenguaje orientado a objetos independiente, así como un ambiente de desarrollo. Son varias las ventajas con las que Eiffel ha contribuido al modelo orientado a objetos: *ligadura dinámica, fuerte tipificación, clases parametrizadas, manejo de excepciones,* y *uso de precondiciones y postcondiciones* [9].

Bertrand Meyer, "Eiffel, the legacy of Simula" [56].

En este artículo, Bertrand Meyer menciona que su primer contacto con la programación orientada a objeto fue durante su periodo estudiantil, al utilizar Simula. Más adelante, realiza contactos en el Centro Noruego de Cómputo, desarrollando algunos proyectos experimentales y operacionales.

Por ese tiempo presenta dos desarrollos experimentales basados en Simula: un sistema de transformación de programas, y un sistema de preparación y manipulación de especificaciones, conocido como **ZAIDE** [56]. Esto le permite entrar a la Asociación de Usuarios de Simula (ASU), donde tiene contacto con Dahl, Nygaard, Kerr, y otros.

En 1985, tiene la oportunidad de participar en un proyecto de un ambiente de desarrollo en ingeniería de software interactiva. Sin embargo, para ese entonces reconoce que *"Simula no era una solución viable en las máquinas que teníamos en mente"* [56]. Por esto último, decide diseñar **Simula 85**, una versión más moderna de Simula que elimina algunas características del lenguaje consideradas como innecesarias, y añade otras a partir del criterio obtenido con su experiencia adquirida durante sus años de trabajo con Simula. Es **Simula 85** el lenguaje que más adelante se conocería como Eiffel.

El lenguaje Eiffel incorpora principalmente cuatro mejoras al modelo original de Simula:

- *Genericidad.* De manera semejante al lenguaje **Ada**, se utiliza la genericidad como un complemento de la herencia.

- *Herencia múltiple*. Permitiendo la reutilización de varias clases a la vez.

- *Aserciones*. Debido a la conexión entre la noción de clase y la teoría de los tipos de datos abstractos, se consideró necesario incluir una capacidad axiomática al *software*, esto es, adicionar equivalentes a axiomas, *precondiciones* y *postcondiciones* a la propia descripción de las clases.

- *Tipificación*. Simula presenta una tendencia relajada hacia los tipos de datos, mediante una aproximación dinámica. Esto cambia radicalmente en Eiffel, que utiliza primordialmente una fuerte tipificación en sus datos.

Además de estas características, Eiffel proporciona un conjunto extensivo de bibliotecas estándar que la ofrecida por Simula.

Por otro lado, en el diseño de Eiffel también se consideraron las características de portabilidad y compatibilidad del *software*. Por esto último, el compilador de Eiffel no es "nativo", es decir, se encuentra implementado en otro lenguaje de programación portable: el lenguaje C.

Es importante mencionar que en este artículo, Meyer menciona que *"La parte principal de Simula que no fue retenida por Eiffel es el mecanismo de cuasi-paralelismo o corrutinas"*. Esto se debe a que se consideró esta característica como "no esencial" para las aplicaciones estándar. Sin embargo, Meyer considera que el uso de corrutinas en un lenguaje de programación orientado a objetos se ha subestimado, ofreciendo en un futuro proveer de tal característica a las siguientes versiones de Eiffel.

4. C++. B. Stroustrup.

C++ fue diseñado por Bjarne Stroustrup en los laboratorios de AT&T Bell. Recibe la influencia de Simula a través de su inmediato sucesor, el lenguaje C con clases, diseñado por el mismo Stroustrup.

C++ es por mucho, un superconjunto de C. Sin embargo, desde un punto de vista cualitativo, se trata de un "C mejorado", incluyendo además de las nociones de *clase, objeto, herencia*, las de *fuerte tipificación*, *sobrecarga de funciones*, y otras mejoras [9].

Bjarne Stroustrup, "Tracing the roots of C++"[75]

En este artículo, Stroustrup examina la herencia de Simula, mencionando en especial algunas áreas de interés para él, como son el uso de

clases y herencia, la concurrencia, la aproximación de diseño basado en un modelo, las librerías de clases en lugar de funciones, y el establecimiento de un balance entre los *tipos dinámicos* y *tipos estáticos*.

Es esta última característica de Simula la que mayor importancia le confiere, pues su sistema de tipos flexible y extensible le permite combinar los beneficios de lenguajes con tipos de datos estáticos y dinámicos: un sistema de tipos estático provee de detección temprana de errores y un grado importante en cuanto a la eficiencia durante el tiempo de ejecución; por su parte, un sistema de tipos dinámico ofrece flexibilidad en el uso de los datos.

"C++ fue diseñado como una deliberada y reconocida imitación de la aproximación de Simula, utilizando el lenguaje de programación de sistemas más popular, C, como base" [75]. Con estas palabras, Stroustrup establece la relación entre C++ y Simula. De hecho, menciona que el primer C++ era simplemente un "C con clases". De no haber existido C, hubiera utilizado cualquier otro lenguaje con los beneficios que aporta C. Propone como un ideal el unir las características de un lenguaje popular, eficiente y "tradicional" como C, con los mecanismos de estructuración de Simula.

El diseño de C++ como un lenguaje orientado a objetos fue realizado considerando tres objetivos principalmente [75]:

- Proveer la expresividad y flexibilidad del sistema de tipos de Simula.
- Incrementar el grado de restricción de los tipos de datos estáticos, a semejanza de Simula y **ALGOL 68**.
- Hacer que las ideas de diseño iniciadas por Simula fueran alcanzables en áreas de aplicación dominadas por C. Este último se considera como el objetivo más importante de C++.

Para Stroustrup, Simula no solamente significa el origen de la programación orientada a objetos, sino también del diseño orientado a objetos. Considera que este estilo de programación y diseño se enfoca en la modelación directa de los objetos del mundo real en un programa. Simula fue el primer lenguaje en soportar este modelo, mediante la definición de clases de objetos, relaciones entre tales clases (herencia), y manipulación de tales objetos.

Para complementar la descripción del desarrollo de la orientación a objetos, se añade un artículo de Peter Wegner en el que realiza un análisis sobre el presente y futuro de la orientación a objetos.

Peter Wegner, "Dimensions of Object-Oriented Modeling"[78].

En este artículo, Peter Wegner analiza la programación orientada a objetos y la programación lógica en relación con la concurrencia y distribución, en términos de encapsulación y reactividad.

En general, Wegner define la programación orientada a objetos como un paradigma de modelación que complementa el poder de los *objetos* con la flexibilidad que introduce el manejo de *clases* y *herencia*. Por otro lado, define a la programación lógica como un paradigma de razonamiento que se enfoca principalmente en el proceso de deducción, y de manera secundaria en los procesos de cómputo. La forma en que ambos paradigmas presentan la concurrencia y distribución, y su grado de encapsulación y reactividad los distinguen entre sí [78].

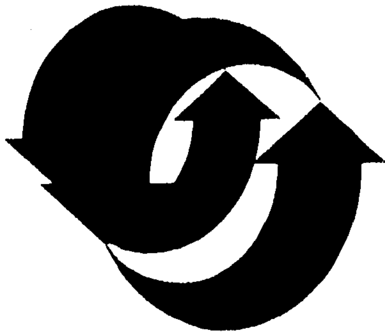
Primeramente, la concurrencia y distribución son formas de mejorar el tiempo de proceso y el poder del modelo, describiéndolo en términos de múltiples procesos o simulación secuencial en un procesador (en el caso de la concurrencia), o definiéndolo dentro de un sistema de procesos que se ejecutan en varios procesadores que no comparten memoria primaria, pero que cooperan mediante envío y recepción de mensajes a través de una red de comunicación (en el caso de la distribución).

Por otro lado, la encapsulación y reactividad son mecanismos del *software* que le permiten ser modificado y extendido [78]. La reactividad (o interactividad) es la capacidad de responder a estímulos externos, modificando su estado y emitiendo una respuesta. La encapsulación, desde un punto de vista modular, permite la creación de un número de componentes que pueden ser estáticamente extendidos por un agente externo, como es el caso de un programador.

A partir de las consideraciones sobre concurrencia, distribución, encapsulación y reactividad, Wegner considera al paradigma orientado a objetos como una programación donde la encapsulación se presenta en forma de *objetos*, y la reactividad como su comportamiento. La concurrencia y distribución son entonces, elementos que permiten acentuar más tales características: por una parte, Wegner considera la distribución como una forma fuerte de encapsulamiento, mientras que la concurrencia contribuye a aumentar la reactividad de un programa.

Wegner define la estructura del paradigma orientado a objetos basado en los conceptos de *objetos*, *clases* y *herencia*. Analiza cómo un lenguaje orientado a objetos es encapsulado y reactivo mediante la definición de estos conceptos, y lo contrasta con la programación lógica, que es por naturaleza no encapsulada y no reactiva.

Para Wegner, la encapsulación, distribución y concurrencia representan las dimensiones del modelo orientado a objetos, dado que se consideran independientes entre sí. Sin embargo, menciona el caso especial en el que se combinan entre sí lo que se conoce como *procesos secuenciales distribuidos* [78]. Se trata de una red de procesadores secuenciales, con múltiples objetos en cada procesador, lo que permite un modelado simple y natural del mundo real.



Apéndice C.
Códigos fuente de los programas para el ejemplo de aplicación de la FFT.

En este apéndice se presentan los ejemplos de aplicación de la FFT utilizados para evaluar la programación paralela orientada a objetos para los casos de 2 y 4 procesadores en una topología de hipercubo. Estos programas pueden servir como referencia para el desarrollo en un número mayor de procesadores. Los comentarios sobre su funcionamiento se presentan directamente sobre el código.

Como se menciona anteriormente [IV.C.2.b], la topología en hipercubo cuenta en general con dos tipos de objetos: el maestro (*master*) y el esclavo (*slave*). De esta forma, cada caso de estudio presenta los códigos de ambos objetos, así como los archivos de configuración para diferente número de procesadores.

1. Programación para dos objetos.

a) Programa Maestro (`master.cxx`).

```
#include "fft.h"

// Se crea un objeto FFT maestro
FFT my_masterfft = FFT();

int main(void) {
      // Declaración de parámetros
      int    count,i,ntotal,nlocal,ptotal,plocal,limit;
      clock_t proc1, proc2, proc_time;
      float  time_seconds;


      // Declaración de canales de comunicación
      Channel      *Node0_to_Node1, *Node1_to_Node0;
```

```
// Declaración de apuntadores a los arreglos de datos
float  *real,*imag,*c,*s,*RTemp,*ITemp;
int    *rbit;

// Asignación a los canales declarados en la configuración
Node1_to_Node0 = (Channel *) get_param(3);
Node0_to_Node1 = (Channel *) get_param(4);

printf("\nEnter upper limit of log of number of points      to
       process (< 9): ");
scanf("%d", &limit);
printf("\nFFT timing tests; two processor version.\n");

// Envío de límites de operación al esclavo
my_masterfft.SendInt(limit, Node0_to_Node1);

// Se verifica si el esclavo está listo
i = my_masterfft.ReceiveInt(Node1_to_Node0);
if (i != limit)      printf("\n Node1 not initialized
       properly! %d \n",i);

// Inicio de la operación
for (count = 2; count <= limit; count++) {
       ptotal = count;
       plocal = ptotal - 1;
       ntotal = 1;
       for (i = 1; i <= ptotal; i++) ntotal = 2*ntotal;
       nlocal = ntotal/2;

       // Envía los parámetros al esclavo
       my_masterfft.SendInt(ptotal,Node0_to_Node1);
       my_masterfft.SendInt(plocal,Node0_to_Node1);
       my_masterfft.SendInt(ntotal,Node0_to_Node1);
       my_masterfft.SendInt(nlocal,Node0_to_Node1);

       // Se genera el arreglo de datos para la FFT
       my_masterfft.generate_signal(real,imag,ntotal);

       // Se calculan los factores de peso en los arreglos c y s, y
       // se inicializa el vector de bit reversal rbit
       my_masterfft.initialize(c,s,rbit,ntotal,ptotal);

       // Se toma el tiempo inicial para realizar la FFT local
       proc1 = my_masterfft.GetTime();

       // Se permutan los datos a partir del vector rbit
       my_masterfft.permute(ntotal,ptotal,real,imag,rbit);

       // Se hace una copia de los arreglos
       my_masterfft.back(RTemp,ITemp,real,imag,nlocal);

       // Envío de datos al esclavo
       my_masterfft.SendFloat(*(real+nlocal),Node0_to_Node1);
       my_masterfft.SendFloat(*(imag+nlocal),Node0_to_Node1);

       // Evalúa la transformada local
       my_masterfft.evaluate(real,imag,c,s,ntotal,nlocal,plocal);

       // Intercambio de datos con el esclavo
       *(real+nlocal) = my_masterfft.ReceiveFloat(Node1_to_Node0);
       *(imag+nlocal) = my_masterfft.ReceiveFloat(Node1_to_Node0);
       my_masterfft.SendFloat(*real,Node0_to_Node1);
       my_masterfft.SendFloat(*imag,Node0_to_Node1);

       // Se reestablecen los datos para continuar el cálculo
       my_masterfft.restore(real,imag,RTemp,ITemp,nlocal);
```

```
        // Se combinan los resultados de dos mitades para
        // obtener la transformada.
        my_masterfft.combine(nlocal,0,1,0,real,imag,c,s);

        // Recibe los resultados del esclavo
        *(real+nlocal) = my_masterfft.ReceiveFloat(Node1_to_Node0);
        *(imag+nlocal) = my_masterfft.ReceiveFloat(Node1_to_Node0);

        // Se reestablecen los datos originales
        my_masterfft.restore(real,imag,RTemp,ITemp,nlocal);

        // Se toma la medida del tiempo final, se obtiene la
        // diferencia con el tiempo inicial y se transforma a
        // segundos.
        proc2 = my_masterfft.GetTime();
        proc_time = my_masterfft.TimeMinus(proc2,proc1);
        time_seconds = my_masterfft.TimeInSeconds(proc_time);
    };

    // Impresión en pantalla de los vectores real e imaginario de la
    // FFT, el módulo al cuadrado y el tiempo total en segundos para
    // realizar la transformada de los puntos establecidos.
    for (i = 0; i <= nlocal; i++) {
        printf("\n %7d %10.3f %10.3f ", i, *(real+i)/ntotal,
            *(imag+i)/ntotal);

    printf("%10.3f",(((*(real+i))*(*(real+i)))+((*(imag+i))*(*(imag+i))
        ))/ntotal/ntotal);
    };
    printf("\n %6d points; Total time: %10.6f s. \n", ntotal,
        time_seconds);

    exit_terminate(EXIT_SUCCESS);
};
```

b) Programa Esclavo (slave.cxx).

```
#include "fft.h"

// Se crea un objeto FFT esclavo
FFT my_workerfft = FFT();

int main(void) {

    // Declaración de parámetros
    int     ptotal,plocal,ntotal,nlocal,limit,count;

    // Declaración de canales de comunicación
    Channel       *Node0_to_Node1, *Node1_to_Node0;

    // Declaración de apuntadores a los arreglos de datos
    float *real,*imag,*c,*s,*RTemp,*ITemp;
    int    *br;

    // Asignación a los canales declarados en la configuración
    Node0_to_Node1 = (Channel *) get_param(1);
    Node1_to_Node0 = (Channel *) get_param(2);

    // Recepción de límites de operación del maestro
    limit = my_workerfft.ReceiveInt(Node0_to_Node1);

    // Se verifica que el esclavo está listo
    my_workerfft.SendInt(limit,Node1_to_Node0);
```

```
        // Inicio de la operación
        for(count = 2; count <= limit; count++) {

                // Recibe los parámetros del maestro
                ptotal = my_workerfft.ReceiveInt(Node0_to_Node1);
                plocal = my_workerfft.ReceiveInt(Node0_to_Node1);
                ntotal = my_workerfft.ReceiveInt(Node0_to_Node1);
                nlocal = my_workerfft.ReceiveInt(Node0_to_Node1);

                // Se calculan los factores de peso en los arreglos c y s, y
                // se inicializa el vector de bit reversal rbit
                my_workerfft.initialize(c, s, br, ntotal, ptotal);

                // Recepción de datos del maestro
                *(real+nlocal) = my_workerfft.ReceiveFloat(Node0_to_Node1);
                *(imag+nlocal) = my_workerfft.ReceiveFloat(Node0_to_Node1);

                // Evalúa la transformada local
                my_workerfft.evaluate(real,imag,c,s,ntotal,nlocal,plocal);

                // Intercambio de datos con el maestro
                my_workerfft.SendFloat(*real,Node1_to_Node0);
                my_workerfft.SendFloat(*imag,Node1_to_Node0);

                *(real+nlocal) = my_workerfft.ReceiveFloat(Node0_to_Node1);
                *(imag+nlocal) = my_workerfft.ReceiveFloat(Node0_to_Node1);

                // Se combinan los resultados de dos mitades para
                // obtener la transformada.
                my_workerfft.combine(nlocal,0,1,1,real,imag,c,s);

                // Se envían los resultados al maestro
                my_workerfft.SendFloat(*real,Node1_to_Node0);
                my_workerfft.SendFloat(*imag,Node1_to_Node0);
        };
        exit_terminate(EXIT_SUCCESS);
}
```

c) Configuraciones.

i) Un procesador (`fft2-1.cfs`).

```
/******************
*
*                          T805
*
*                   ------------------
*                  |  (memory = 8M)   |
*                 0|                  |
*       Host ------| master_processor |
*                  |                  |
*                  |                  |
*                   ------------------
*
*
******************/

/*{{{  Descripción del hardware */
T805 (memory = 8M) master_processor;

connect master_processor.link[0], host;
/*}}}*/
```

```
/*{{{ Descripción del software */
process (stacksize = 20k, heapsize = 20k,
        interface(input in, output out, input fromslave, output
        toslave)) master;

process (stacksize = 20k, heapsize = 20k,
        interface(input frommaster, output tomaster)) slave;

input from_host;
output to_host;

connect master.in, from_host;
connect master.out, to_host;
connect master.fromslave, slave.tomaster;
connect master.toslave, slave.frommaster;
/*}}}*/

/*{{{ Mapeo a la red */
use "master.lku" for master;
use "worker.lku" for slave;

place master on master_processor;
place slave on master_processor;

place to_host   on host;
place from_host on host;

place master.in       on master_processor.link[0];
place master.out      on master_processor.link[0];
/*}}}*/
```

ii) Dos procesadores (`fft2-2.cfs`).

```
/*******************
*
*                        T805                        T805
*
*                 ------------------           ------------------
*                |   (memory = 1M)   |         |   (memory = 1M)   |
*               0|                   |2       1|                   |
*       Host ------| master_processor |--------|  slave_processor  |
*                |                   |         |                   |
*                |                   |         |                   |
*                 ------------------           ------------------
*
*
*
*******************/

/*{{{ Descripción del hardware */
T805 (memory = 1M) master_processor;
T805 (memory = 1M) slave_processor;

connect master_processor.link[0], host;
connect master_processor.link[2], slave_processor.link[1];
/*}}}*/

/*{{{ Descripción del software */
process (stacksize = 20k, heapsize = 20k,
        interface(input in, output out, input fromslave, output
        toslave)) master;

process (stacksize = 20k, heapsize = 20k,
        interface(input frommaster, output tomaster)) slave;
```

```
input from_host;
output to_host;

connect master.in, from_host;
connect master.out, to_host;
connect master.fromslave, slave.tomaster;
connect master.toslave, slave.frommaster;
/*}}}*/

/*{{{ Mapeo a la red */
use "master.lku" for master;
use "worker.lku" for slave;

place master on master_processor;
place slave on slave_processor;

place to_host   on host;
place from_host on host;

place master.in        on master_processor.link[0];
place master.out       on master_processor.link[0];
place master.fromslave on master_processor.link[2];
place master.toslave   on master_processor.link[2];
place slave.frommaster on slave_processor.link[1];
place slave.tomaster   on slave_processor.link[1];
/*}}}*/
```

2. Programación para cuatro objetos.

a) Programa Maestro (master.cxx).

```
extern "C" { #include <stdio.h>
             #include <math.h>
             #include <stdlib.h>
             #include <channel.h>
             #include <misc.h>
             #include <time.h>
}
#include "fft.h"
#define log_max_pts  8 /* log 2 of slize_size_max */
#define log_max_procs 3 /* log 2 of max parallel splits  */

// Se crea un objeto FFT maestro
FFT my_masterfft = FFT();

int main(void) {

      // Declaración de apuntadores a los arreglos de datos
      float        *ra,*ia,*c,*s,*RTemp,*ITemp;
      int          *rbit;

      // Declaración de parámetros
      int          i,j,k,temp,ntotal,nlocal,ptotal,plocal;
      int          np,  /* Número de objetos paralelos
                          (debe ser potencia de 2) */
      int          lnp; /* log base 2 del número de objetos */
      int          ThisSlice = 0; /* identificador del número de parte */
      int          ThisChan = 0;  /* El maestro se comunica con el host
                                      mediante el canal 0 */
      int          depth;
      int          twiddle_0,twiddle_step,sw,group_size;
      int          two_to_the[log_max_pts+1];
      int          p_count,lnp_count,n_limit,p_limit,np_limit,lnp_limit;
      clock_t      time1, time2;
```

```
clock_t       Sec;

// Declaración de canales de comunicación
Channel       *To_Node[log_max_procs+1],
Chyannel      *From_Node[log_max_procs+1];

// Asignación a los canales declarados en la configuración
From_Node[2] = (Channel *) get_param(3);
To_Node[2] = (Channel *) get_param(4);
From_Node[3] = (Channel *) get_param(5);
To_Node[3] = (Channel *) get_param(6);

// Se lee y verifica el número total de puntos de la transformada
p_limit = 0;
while (p_limit < 1 || p_limit > log_max_pts) {
      printf("\nEnter upper limit of log of number of pts to
             process (< %1d): ",(log_max_pts+1));
      scanf("%2d",&p_limit);
};

// Se asignan identificadores y el número de puntos por objeto,
two_to_the[0] = 1;
for (i = 1; i <= p_limit; i++) two_to_the[i] = two_to_the[i-1] * 2;
n_limit = two_to_the[p_limit];

printf("\nGenerating %1d points ...\n", n_limit);

// Verificación entre el número de objetos y datos
lnp_limit = 2;
if (lnp_limit > p_limit ) printf("\n** Error **  cannot have more
             processes than data items\n");

// Inicializando comunicaciones
np_limit = two_to_the[lnp_limit];
printf("\nAttempting to set up %1d parallel fft/combine worker(s)
      ...\n",np_limit);

// Verificación de comunicaciones con objetos vecinos
for (i = 1; i <= lnp_limit; i++) {
      printf("Inside loop\n");

      // Envía identificador del esclavo
      my_masterfft.SendInt(two_to_the[i-1],To_Node[4-i]);
      printf("Test point\n");

      // Envía señal de verificación
      my_masterfft.SendInt(lnp_limit,To_Node[4-i]);
      printf("Second Test point\n");
}

// Verificación si los esclavos están listos
printf("\nNode 0 boss-worker is running\n");
for (k=ThisChan+1; k <= lnp_limit; k++) {
      temp = my_masterfft.ReceiveInt(From_Node[4-k]);
      printf("Worker %1d initialized.\n",temp);
}

for (j = 1; j <= lnp_limit-ThisChan-1; j++)
for (i = 1; i <= two_to_the[j-1]; i++)
for (k=ThisChan+1; k <= (lnp_limit-j); k++) {
      temp = my_masterfft.ReceiveInt(From_Node[4-k]);
      printf("Worker %1d initialized.\n",temp);
}

// Inicio de la operación
printf("\nFFT timing tests; spectrum calculated for real data.\n");
for (p_count = 2; p_count <= p_limit; p_count++) {
```

```
for (lnp_count = 0;lnp_count < lnp_limit; lnp_count++) {
      ptotal = p_count;
      ntotal = two_to_the[ptotal];
      lnp    = lnp_count;
      np     = two_to_the[lnp];
      nlocal = ntotal/np;
      plocal = ptotal - lnp;

      // Envía parámetros a esclavos vecinos más cercanos.
      for (i = 1; i <= lnp; i++) {

            // Envía identificador del esclavo
            my_masterfft.SendInt(two_to_the[i-1],
                                  To_Node[4-i]);
            my_masterfft.SendInt(ptotal,To_Node[4-i]);
            my_masterfft.SendInt(plocal,To_Node[4-i]);
            my_masterfft.SendInt(ntotal,To_Node[4-i]);
            my_masterfft.SendInt(nlocal,To_Node[4-i]);
            my_masterfft.SendInt(lnp,To_Node[4-i]);
      }

      // Se genera el arreglo de datos para la FFT
      my_masterfft.generate_signal(ra,ia,ntotal);

      // Se calculan los factores de peso en los arreglos c
      // y s, y se inicializa el vector de bit reversal rbit
      my_masterfft.initialize(c,s,rbit,ntotal,ptotal);

      // Checa para ver si los esclavos están listos
      for (k=ThisChan+1; k <= lnp; k++) {
            temp = my_masterfft.ReceiveInt(From_Node[4-k]);
      }
      for (j = 1; j <= lnp-ThisChan-1; j++)
      for (i = 1; i <= two_to_the[j-1]; i++)
      for (k=ThisChan+1; k <= (lnp-j); k++) {
            temp = my_masterfft.ReceiveInt(From_Node[4-k]);
      }

      // Se toma el tiempo inicial para realizar la FFT
      // local
      time1 = my_masterfft.GetTime();

      // Se permutan los datos a partir del vector rbit
      my_masterfft.permute(ntotal,ptotal,ra,ia,rbit);

      // Envía parte de los datos a los esclavos
      // NOTA: Debido a la configuración utilizada, y a que
      // es difícil reslizar una transmisión simultánea, los
      // protocolos de comunicación entre objetos tiende a
      // complicarse demasiado!!
      for (i = 1; i < np; i++) {

            // Para cada objeto
            j = i;
            k = lnp;
            while (j > ThisSlice) { // Decide cual canal
                                    // utilizar.
                  k--;
                  if (j >= (ThisSlice+two_to_the[k]))
                        j -= two_to_the[k];
            }
            k++;

            // Identifica la parte de datos a enviar
            my_masterfft.SendInt(i,To_Node[4-k]);
            my_masterfft.SendFloat(*(ra+(i*nlocal)),
                                    To_Node[4-k]);
```

```
        my_masterfft.SendFloat(*(ia+(i*nlocal)),
                                        To_Node[4-k]);
}

// Evalúa la transformada local
my_masterfft.evaluate(ra,ia,c,s,ntotal,nlocal,plocal);

// Intercambia datos con los objetos vecinos en el
// hipercubo.
twiddle_step = np / 2;
group_size = 1;
for (depth = 1; depth <= lnp; depth++) {

        // 1 para el conjunto de datos inferiores
        // 0 para el conjunto de datos superiores
        sw = ((ThisSlice / group_size)+1) % 2;
        twiddle_0 =(ThisSlice % group_size) *
                        twiddle_step * nlocal;
        if (sw) {
                // Recibe primero, luego envía
                *RTemp = my_masterfft.ReceiveFloat
                                (From_Node[4-depth]);
                *ITemp = my_masterfft.ReceiveFloat
                                (From_Node[4-depth]);
                my_masterfft.SendFloat(*ra,
                                To_Node[4-depth]);
                my_masterfft.SendFloat(*ia,
                                To_Node[4-depth]);
        }
        else {
                // Envía primero, luego recibe
                my_masterfft.SendFloat(*ra,
                                To_Node[4-depth]);
                my_masterfft.SendFloat(*ia,
                                To_Node[4-depth]);
                *RTemp = my_masterfft.ReceiveFloat
                                (From_Node[4-depth]);
                *ITemp = my_masterfft.ReceiveFloat
                                (From_Node[4-depth]);
        }

        // Combina dos mitades de la transformada
        my_masterfft.combine(nlocal,twiddle_0,
                        twiddle_step,sw,ra,ia,c,s);

        twiddle_step /= 2;
        group_size *= 2;
}

// Recibe los resultados parciales de los esclavos
// NOTA: De nuevo, debido a la configuración y a la
// dificultad de realizar una transmisión simultánea,
// los protocolos de comunicación entre objetos tiende
// a complicarse demasiado!!
for (i = 1; i < np; i++) {

        // Para cada objeto
        j = i;
        k = lnp;
        while (j > ThisSlice) { // Decide cual canal
                                // utilizar
                k--;
                if (j >= (ThisSlice+two_to_the[k]))
                        j -= two_to_the[k];
        }
        k++;
        // Identifica la parte de datos a enviar
```

```
                        temp = my_masterfft.ReceiveInt(From_Node[4-k]);
                        *(ra+(temp*nlocal)) = my_masterfft.ReceiveFloat
                                                    (From_Node[4-k]);
                        *(ia+(temp*nlocal)) = my_masterfft.ReceiveFloat
                                                    (From_Node[4-k]);
                    }

                    // Se toma la medida del tiempo final y se obtiene la
                    // diferencia con el tiempo inicial.
                    time2 = my_masterfft.GetTime();
                    Sec = my_masterfft.TimeMinus(time2,time1);
            }
        }

        // Impresión en pantalla de los vectores real e imaginario de la
        // FFT, el módulo al cuadrado y el tiempo total en segundos para
        // realizar la transformada de los puntos establecidos.
        printf("\n");
        for (i = 0; i <= ntotal/2; i++) {
              printf("\n %7d %10.3f %10.3f ",
                        i,*(ra+i)/ntotal,*(ia+i)/ntotal);
              printf("%10.3f",
              ((*(ra+i))*(*(ra+i))+(*(ia+i))*(*(ia+i)))/ntotal/ntotal);
        }
        printf("\n");
        printf(" %6d points; %1d parallel split(s); Total time: %9.6f
                seconds.\n",ntotal, np, ((float) Sec /(float)
                CLOCKS_PER_SEC));

        exit_terminate(EXIT_SUCCESS);
}
```

b) Programa Esclavo (slave.cxx).

```
extern "C" { #include <stdlib.h>
             #include <misc.h>
             #include <math.h>
             #include <channel.h>
             #include <process.h>
}
#include "fft.h"
#define log_max_procs 3 /* log 2 of max parallel splits  */

// Se crea un objeto FFT esclavo
FFT my_workerfft = FFT();

int main(void) {

        // Declaración de apuntadores a los arreglos de datos
        float       *ra,*ia,*c,*s,*RTemp,*ITemp;
        int         *br;

        // Declaración de parámetros locales del esclavo
        int         i,j,k,m,temp,ptotal,plocal,ntotal,nlocal,lnp,np;
        int         two_to_the[log_max_procs+1];
        int         ThisSlice,ThisChan, no_comms;
        int         depth, group_size, sw, twiddle_0, twiddle_step;

        // Declaración de arreglos de canales de comunicación
        Channel     *To_Node[log_max_procs+1],*From_Node[log_max_procs+1];

        // Asignación a los canales declarados en la configuración.
        From_Node[2] = (Channel *) get_param(1);
        To_Node[2] = (Channel *) get_param(2);
        From_Node[3] = (Channel *) get_param(3);
```

```
To_Node[3] = (Channel *) get_param(4);

// Calcula los identificadores de los objetos vecinos
two_to_the[0] = 1;
for (i = 1; i <= log_max_procs; i++)
      two_to_the[i] = two_to_the[i-1] * 2;

// Obtiene la señal de iniciación del maestro. Esta señal puede
// llegar por cualquier canal, dependiendo de la localización del
// esclavo en la red. Debido a la falla en la ejecución del
// ProcAlt(), el cual examina las ligas y cancela cualquiera no
// conectada, se desarrolla una implementación "casera" del ALT, la
// cual no requiere conexión física entre las ligas.
no_comms = 1;
while (no_comms) {
      ThisChan = 3;
      no_comms = ChanInTimeFail(From_Node[3], (char *) &ThisSlice,
            sizeof(int), ProcTime()+50);

      if (no_comms) {
            ThisChan = 2;
            no_comms = ChanInTimeFail(From_Node[2],
                  (char *) &ThisSlice, sizeof(int),
                  ProcTime()+50);
      }
}

ThisChan = 4 -ThisChan;
// Envío para esparcir la señal
lnp = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);

// Pasa la señal de inicio a los esclavos vecinos si es necesario
k = ThisChan+1;
while ((ThisSlice + two_to_the[k-1]) < two_to_the[lnp]) {
      my_workerfft.SendInt(ThisSlice+two_to_the[k-1],
                                    To_Node[4-k]);
      my_workerfft.SendInt(lnp,To_Node[4-k]);
      k++;
}

// Responde a la señal de inicio con OK
my_workerfft.SendInt(ThisSlice,To_Node[4-ThisChan]);

// Pasa la respuesta de la señal de inicio a los vecinos si es
// necesario
for (k=ThisChan+1; k <= lnp; k++) {
      temp = my_workerfft.ReceiveInt(From_Node[4-k]);
      my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
}

for (j = 1; j <= lnp-ThisChan-1; j++)
for (i = 1; i <= two_to_the[j-1]; i++)
for (k=ThisChan+1; k <= (lnp-j); k++) {
      temp = my_workerfft.ReceiveInt(From_Node[4-k]);
      my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
}

// Realiza tantas FFT's locales como sean necesarias. NOTA: Para
// una ejecución subsecuente, el esclavo debe ser reinicializado.
while (1) {

      // Recibe los parámetros del maestro
      ThisSlice = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
      ptotal    = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
      plocal    = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
      ntotal    = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
      nlocal    = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
```

```
lnp        = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);

np         = two_to_the[lnp];

// Pasa los parámetros a los esclavos vecinos si es
// necesario
k = ThisChan+1;
while ((ThisSlice + two_to_the[k-1]) < two_to_the[lnp]) {
      my_workerfft.SendInt(ThisSlice+two_to_the[k-1],
                                     To_Node[4-k]);
      my_workerfft.SendInt(ptotal,To_Node[4-k]);
      my_workerfft.SendInt(plocal,To_Node[4-k]);
      my_workerfft.SendInt(ntotal,To_Node[4-k]);
      my_workerfft.SendInt(nlocal,To_Node[4-k]);
      my_workerfft.SendInt(lnp,To_Node[4-k]);
      k++;
}

// Se calculan los factores de peso en los arreglos c
// y s, y se inicializa el vector de bit reversal rbit
my_workerfft.initialize(c, s, br, ntotal, ptotal);

// Envía al objeto anterior una señal de OK
my_workerfft.SendInt(ThisSlice,To_Node[4-ThisChan]);

// Pasa las señales de OK de otros vecinos, si es necesario
for (k=ThisChan+1; k <= lnp; k++) {
      temp = my_workerfft.ReceiveInt(From_Node[4-k]);
      my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
}
for (j = 1; j <= lnp-ThisChan-1; j++)
for (i = 1; i <= two_to_the[j-1]; i++)
for (k=ThisChan+1; k <= (lnp-j); k++) {
      temp = my_workerfft.ReceiveInt(From_Node[4-k]);
      my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
}

// Recibe los datos del objeto anterior. Los datos sirven
// para este objeto y los objetos vecinos, si es necesario.
for (m = 1; m <= two_to_the[lnp-ThisChan]; m ++) {
      temp = my_workerfft.ReceiveInt(From_Node[4-ThisChan]);
      if (temp == ThisSlice) {

            // Retiene los datos
            *ra = my_workerfft.ReceiveFloat
                                    (From_Node[4-ThisChan]);
            *ia = my_workerfft.ReceiveFloat
                                    (From_Node[4-ThisChan]);
      }
      else {

            // Pasa los datos a un vecino por el canal
            // apropiado.
            *RTemp = my_workerfft.ReceiveFloat
                                    (From_Node[4-ThisChan]);
            *ITemp = my_workerfft.ReceiveFloat
                                    (From_Node[4-ThisChan]);
            j = temp;
            k = lnp;
            while(j > ThisSlice) {
                  k--;
                  if (j >= (ThisSlice+two_to_the[k]))
                        j -= two_to_the[k];
            }
            k++;
            // Envía el identificador y los datos
            my_workerfft.SendInt(temp,To_Node[4-k]);
```

```
                my_workerfft.SendFloat(*RTemp,To_Node[4-k]);
                my_workerfft.SendFloat(*ITemp,To_Node[4-k]);
        }
}

// Evalúa la transformada local
my_workerfft.evaluate(ra,ia,c,s,ntotal,nlocal,plocal);

// Intercambia datos con sus vecinos en el hipercubo
twiddle_step = np / 2;
group_size = 1;
for (depth = 1; depth <= lnp; depth++) {

        // 1 para el conjunto de datos inferiores
        // 0 para el conjunto de datos superiores
        sw = ((ThisSlice / group_size)+1) % 2;
        twiddle_0 =(ThisSlice % group_size)
                        * twiddle_step * nlocal;
        if (sw) {

                // Recibe primero, luego envía
                *RTemp = my_workerfft.ReceiveFloat
                                        (From_Node[4-depth]);
                *ITemp = my_workerfft.ReceiveFloat
                                        (From_Node[4-depth]);
                my_workerfft.SendFloat(*ra,To_Node[4-depth]);
                my_workerfft.SendFloat(*ia,To_Node[4-depth]);
        }
        else {

                // Envía primero, luego recibe
                my_workerfft.SendFloat(*ra,To_Node[4-depth]);
                my_workerfft.SendFloat(*ia,To_Node[4-depth]);
                *RTemp = my_workerfft.ReceiveFloat
                                        (From_Node[4-depth]);
                *ITemp = my_workerfft.ReceiveFloat
                                        (From_Node[4-depth]);
        }

        // Combina dos mitades de la transformada
        my_workerfft.combine(nlocal,twiddle_0,
                                twiddle_step,sw,ra,ia,c,s);

        twiddle_step /= 2;
        group_size *= 2;
}

// Envía sus resultados al objeto anterior
my_workerfft.SendInt(ThisSlice,To_Node[4-ThisChan]);
my_workerfft.SendFloat(*ra,To_Node[4-ThisChan]);
my_workerfft.SendFloat(*ia,To_Node[4-ThisChan]);

// Pasa los resultados de los objetos vecinos, si es
// necesario
for (k=ThisChan+1; k <= lnp; k++) {
        temp = my_workerfft.ReceiveInt(From_Node[4-k]);

        *RTemp = my_workerfft.ReceiveFloat(From_Node[4-k]);
        *ITemp = my_workerfft.ReceiveFloat(From_Node[4-k]);

        my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
        my_workerfft.SendFloat(*RTemp,To_Node[4-ThisChan]);
        my_workerfft.SendFloat(*ITemp,To_Node[4-ThisChan]);
}
for (j = 1; j <= lnp-ThisChan-1; j++)
for (i = 1; i <= two_to_the[j-1]; i++)
for (k=ThisChan+1; k <= (lnp-j); k++) {
```

```
                    temp = my_workerfft.ReceiveInt(From_Node[4-k]);
                    *RTemp = my_workerfft.ReceiveFloat(From_Node[4-k]);
                    *ITemp = my_workerfft.ReceiveFloat(From_Node[4-k]);
                    my_workerfft.SendInt(temp,To_Node[4-ThisChan]);
                    my_workerfft.SendFloat(*RTemp,To_Node[4-ThisChan]);
                    my_workerfft.SendFloat(*ITemp,To_Node[4-ThisChan]);
            }
        }
        exit_terminate(EXIT_SUCCESS);
}
```

c) Configuraciones.

i) Un procesador (`fft4-1.cfs`).

```
/*******************
*
*                              T805
*
*                       ------------------
*                      |  (memory = 8M)   |
*                    0 |                  |
*         Host ------|        Tram        |
*                      |                  |
*                      |                  |
*                       ------------------
*
*
*******************/

/* Descripción del hardware */
T805 (memory = 8M) Tram;

connect Tram.link[0] to host;

/* Descripción del software */
process (stacksize = 2k, heapsize = 50k,
        interface (input in, output out, input a, output b,
                   input c, output d)) test1;
process (stacksize = 2k, heapsize = 50k,
        interface (input p, output q, input r, output s)) test2[3];

input hostin;
output hostout;

connect test1.in, hostin;
connect test1.out, hostout;
connect test1.a, test2[1].q;
connect test1.b, test2[1].p;
connect test1.c, test2[0].s;
connect test1.d, test2[0].r;
connect test2[0].p, test2[2].q;
connect test2[0].q, test2[2].p;
connect test2[1].r, test2[2].s;
connect test2[1].s, test2[2].r;

/* Mapeo del software al hardware   */

use "master.lku" for test1;
place test1 on Tram;

rep i=0 for 3
{
      use "worker.lku" for test2[i];
```

```
        place test2[i] on Tram;
}

place hostin on host;
place hostout on host;
```

ii) Dos procesadores (`fft4-2.cfs`).

```
/******************
*
*                             T805                               T805
*
*                    ------------------              ------------------
*                   |  (memory = 1M)   |            |  (memory = 1M)   |
*                 0 |                  | 2        1 |                  |
*      Host ------|     Tram[0]      |----------|     Tram[1]      |
*                   |                  |            |                  |
*                   |                  |            |                  |
*                    ------------------              ------------------
*                            3 |                              3 |
*                              |                                |
*                               --------------------------------
*
******************/

/* Descripción del hardware */
T805 (memory = 8M) Tram[2];

connect Tram[0].link[0] to host;
connect Tram[0].link[2] to Tram[1].link[1];
connect Tram[0].link[3] to Tram[1].link[3];

/* Descripción del software */
process (stacksize = 2k, heapsize = 50k,
        interface (input in, output out, input a, output b,
                    input c, output d)) test1;
process (stacksize = 2k, heapsize = 50k,
        interface (input p, output q, input r, output s)) test2[3];

input hostin;
output hostout;

connect test1.in, hostin;
connect test1.out, hostout;
connect test1.a, test2[1].q;
connect test1.b, test2[1].p;
connect test1.c, test2[0].s;
connect test1.d, test2[0].r;
connect test2[0].p, test2[2].q;
connect test2[0].q, test2[2].p;
connect test2[1].r, test2[2].s;
connect test2[1].s, test2[2].r;

/* Mapeo del software al hardware   */

use "master.lku" for test1;
place test1 on Tram[0];

use "worker.lku" for test2[0];
place test2[0] on Tram[0];

use "worker.lku" for test2[1];
place test2[1] on Tram[1];

use "worker.lku" for test2[2];
```

```
place test2[2] on Tram[1];

place hostin on host;
place hostout on host;

place test1.in on Tram[0].link[0];
place test1.out on Tram[0].link[0];
place test1.a on Tram[0].link[2];
place test1.b on Tram[0].link[2];
place test2[0].p on Tram[0].link[3];
place test2[0].q on Tram[0].link[3];
```

iii) Cuatro procesadores (`fft4-4.cfs`).

```
/*******************
*
*                  T800                       T800
*
*
*                -----------------          -----------------
*               |  (memory = 1M)  |        |  (memory = 1M)  |
*              0|                 | 2    1 |                 |
*     Host -----|    Tram[0]      |--------|    Tram[2]      |
*               |                 |        |                 |
*               |                 |        |                 |
*                -----------------          -----------------
*                        3 |                        2 |
*                          |                          |
*                          |                          |
*                 T800     |                 T800     |
*                        3 |                        1 |
*                -----------------          -----------------
*               |  (memory = 1M)  |        |  (memory = 1M)  |
*               |                 | 1    2 |                 |
*               |    Tram[1]      |--------|    Tram[3]      |
*               |                 |        |                 |
*               |                 |        |                 |
*                -----------------          -----------------
*
*
*
*
*******************/

/* Descripción del hardware */
T800 (memory = 1M) Tram[4];

connect Tram[0].link[0] to host;

connect Tram[0].link[2] to Tram[2].link[1];
connect Tram[0].link[3] to Tram[1].link[3];
connect Tram[1].link[1] to Tram[3].link[2];
connect Tram[2].link[2] to Tram[3].link[1];

/* Descripción del software */
process (stacksize = 2k, heapsize = 50k
        interface (input in, output out, input a, output b,
                   input c, output d)) test1;
process (stacksize = 2k, heapsize = 50k,
        interface (input p, output q, input r, output s)) test2[3];

input hostin;
output hostout;

connect test1.in, hostin;
connect test1.out, hostout;
connect test1.a, test2[1].q;
```

```
connect test1.b, test2[1].p;
connect test1.c, test2[0].s;
connect test1.d, test2[0].r;
connect test2[0].p, test2[2].q;
connect test2[0].q, test2[2].p;
connect test2[1].r, test2[2].s;
connect test2[1].s, test2[2].r;

/* Mapeo del software al hardware    */

use "master.lku" for test1;
place test1 on Tram[0];

rep i=0 for 3
{
        use "worker.lku" for test2[i];
        place test2[i] on Tram[i+1];
}

place hostin on host;
place hostout on host;
```

3. Compilación.

Finalmente, a fin de completar esta explicación se proporciona una guía de compilación general para los programas expuestos anteriormente. Es importante mencionar que se utilizó el compilador INMOS C++ paralelo en una plataforma PC, con cuanto más 4 procesadores tipo T800.

Se considera un caso general, en el que los archivos `master.cxx`, `slave.cxx` y `fft.cfs` representan cualquiera de los programas maestro, esclavo y configuración respectivamente. Por simplicidad, se omiten los mensajes de información y advertencia del compilador. De esta forma, los pasos de compilación son los siguientes.

1. Para compilar el programa maestro:

```
C:\>iccxx master.cxx -t805
C:\>ilink master.tco libcxx.lib /t805 /f startup.lnk
```

2. Para compilar el programa esclavo:

```
C:\>iccxx slave.cxx -t805
C:\>ilink slave.tco libcxx.lib /t805 /f startrd.lnk
```

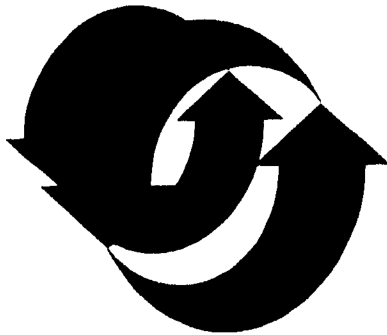
3. A partir de los pasos 1. y 2., se obtienen las unidades ligadas de los programas maestro y esclavo como `master.lku` y `slave.lku`, los cuales son invocados por el archivo de configuración. Para obtener una versión ejecutable de este último, se utilizan los siguientes pasos:

```
C:\>icconf fft.cfs
C:\>icollect fft.cfb
```

4. Del paso anterior, se obtiene una versión ejecutable `fft.btl` del programa, el cual se ejecuta mediante la siguiente línea de comando:

```
C:\>iserver /sb fft.btl
```

Es importante considerar durante los pasos 1. y 2. la diferencia entre los pasos de compilación propiamente y ligado al utilizar guiones (-) y diagonales (/) . Además, es también importante notar que debido a que el programa maestro se encarga de la comunicación con el *host*, se liga utilizando el archivo indirecto `startup.lnk`, mientras que el código de los esclavos que no tienen comunicación con el *host* utilizan el archivo indirecto reducido `startrd.lnk`. Los demás pasos son generales para desarrollar un archivo ejecutable.



Bibliografía.

[1] Akerbaek, T., *C++, Coroutines, and Simulation.* The C Users Journal, march 1993.

[2] Andrews, G.R., *Concurrent Programming, principles and practice.* The Benjamin Cummings Publishing Company Inc., 1991.

[3] Andrews, G.R., Schneider, F.B., *Concepts and Notations for Concurrent Programming.* Communications of the ACM, 1983.

[4] Antoy, S., *Systematic Design of Algebraic Specifications*, ACM SIGSOFT, Software Engineering Notes, January 1989.

[5] Banerjee, P., Chandy, J. A., Gupta, M., Hodges, E.W., Holm, J.G., Lain, A., Palermo, D.J., Ramaswamy, S., Su, E.,*The Paradigm Compiler for Distributed-Memory Multicomputers*, Computer, IEEE Computer Society, october 1995.

[6] Batory, D., Singhal, V., Sirkin, M., Thomas, J., *Scalable Software Libraries.* ACM SIGSOFT, Software Engineering Notes, December 1993.

[7] Bjorkholm, T,. *Summary of Transputer Compiler Survey*, Abo Akademi, University, Finland, october 1993.

[8] Bond, J., *Parallel-processing concepts finally come together in real systems.*Computer Design. PennWell Publications. June 1, 1987.

[9] Booch, G., *Object Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[10] Brewer, E.A., Waldspurger, C.A., *Preventing Recursion Deadlock in Concurrent Object-Oriented Systems*, Technical Report MIT/LCS/TR-526, January, 1992.

[11] Brinch-Hansen, P., *Structured Multiprogramming*.Communications of the ACM, Volume 15, Number 7, July 1972

[12] Brinch-Hansen, P., *The programming language Concurrent Pascal*. IEEE Trans. Software Eng. 1, 2, June 1975.

[13] Brinch-Hansen, P., *Distributed Processes: A Concurrent Programming Concept*. Communications of the ACM, Volume 21, Number 11, November 1978.

[14] Briot, J.P., *Object-Oriented Concurrent Programming: Introducing a New Programming Methodology*, Proceedings of the 7th International Meeting of Young Computer Scientist (IMYCS'92), Topics in Computer Science Series, Gordon & Breach, 1993.

[15] Calder, B., Grunwald, D., Zorn, B., *Quantifying Behavioral Differences Between C and C++ Programs*. Journal of Programming Languajes, Vol 2, Num 4, 1994.

[16] Carrington, D., Duke, D., *Deriving Modular Designs from Formal Specifications*. ACM SIGSOFT, Software Engineering Notes, May 1993.

[17] Chandra, R., Gupta, A., Hennesy, J., *COOL: An Object-based languaje for Parallel Programming*. Computer, IEEE Computer Society, august 1994.

[18] Chandy, M., Kesselman, C., *CC++: A declarative Concurrent Object-Oriented Programming notation*. Research Directions in Concurrent Object-Oriented Programming. The MIT Press, Cambridge, Mass., 1993.

[19] Chou, H., Wang, F.L., *Software Reuse Based on a Large Object-Oriented Library*. ACM SIGSOFT, Software Engineering, January 1993.

[20] Clements, A., *Multiprocessor Systems*. Electronics & Wireless World, 1990.

[21] Coad, P., Nicola, J., *Object Orientad Programming*. Yourdon Press Computing Series, Prentice-Hall 1993.

[22] Computer Associates International, Inc., *CA-C++ version 2 Compiler Manual*, 1993.

[23] Dijkstra, E.W., *Co-operating Sequential Processes*. F. Genuys, Ed. Programming Languages, Academic Press, New York, 1968.

[24] Dougherty, E.R.,Giardina, C.R., *Matrix Structured Image Processing.* Prentice-Hall, Inc., 1987.

[25] Drakos, N., *Summary of Workshop on Parallel Programming in C++*, May of 1993, Los Angeles California.

[26] Flatt, H.P., Kennedy, K., *Performance of parallel processors.* Parallel Computing 12, Elsevier Science Publishers B.V., North-Holland 1989.

[27] Freeman, T.L., Phillips, C., *Parallel Numerical Algorithms.* Prentice-Hall International Series in Computer Science, 1992.

[28] Furmento, N., Roudier, Y., Siegel, G., *Parallélisme et Distribution en C++. Un revue des langages existants*, Rapport de Recherche I3S RR95-02.

[29] Galletly, J., *Occam 2.* Pitman Publishing, 1990.

[30] González, R.C., Wintz, P., *Digital Image Processing.* Addison-Wesley Publishing Company, 1987.

[31] Green, S., *Parallel Processing for Computer Graphics.* The MIT Press, Pitman Publishing, 1991.

[32] Gelernter, D., *Getting the Job Done.* Parallel Processing, BYTE, November 1988.

[33] Habermann, A.N., *Synchronization of communicating processes.* Communications of the ACM, Volume 15, Number 3, March 1972.

[34] Harland, D.M., *Concurrency and Programming Languages.* Ellis Harwood Ltd., John Wiley and Sons, 1986.

[35] Hoare, C.A.R., *Towards a theory of Parallel Programming.* In Operating Systems Techniques. Academic Press, New York, 1972.

[36] Hoare, C.A.R., *Monitors: An operating system structuring concept.* Communications of the ACM, Volume 17, Number 10, October 1974.

[37] Hoare, C.A.R., *Communicating Sequential Processes.* Communications of the ACM, Volume 21, Number 8, August 1978.

[38] Hoare, C.A.R., *Communicating Sequential Processes.* C.A.R. Hoare Series Editor, Prentice-Hall, 1985.

[39] Howe, C.D., Moxon, B., *How to program parallel processors.* IEEE Spectrum, September 1987.

[40] INMOS Ltd., *ANSI C Toolset User Manual,* August 1990.

[41] INMOS Ltd., *Occam 2 Reference Manual.* C.A.R. Hoare Series Editor, Prentice-Hall, 1988.

[42] Karp, A.H., Flatt, H.P., *Measuring Parallel Processor Performance.* Communications of the ACM, Volume 33, Number 5, May, 1990.

[43] Kay, A., *The natural history of objects.* Happy 25th Anniversary Objects! Special Silver Anniversary Supplement. SIGS Publications 1992.

[44] Kaiser, G., Hseush, W., Popovich, S.,Wu, S., *Multiple Concurrency Control Policies in an Object-Oriented Programming System.* Research Directions in Concurrent Object-Oriented Programming. The MIT Press, Cambridge, Mass., 1993.

[45] Kent, W., *The State of Object Technology.* Database Techonology Department, Hewlett-Packard Laboratories, Palo Alto, California.

[46] Kernigan, B.W., Pike, R., *El entorno de la programación UNIX.* Prentice-Hall, 1987.

[47] Kerr, R., *Objects strike Silver.* Happy 25th Anniversary Objects! Special Silver Anniversary Supplement. SIGS publications 1992.

[48] Kotz, D., *A DAta-Parallel Programming Library for Education (DAPPLE),* Technical Report PCS-TR94-235, November 7, 1994.

[49] Kuen Lee, J., Gannon, D., *Objetc Oriented Parallel Programming Experiments and Results.* Department of Computer Science, Indiana University.

[50] Lester, B.P., *The Art of Parallel Programming.*Prentice-Hall, 1993.

[51] Lewis, T.G., El-Rewini, H., *Introduction to parallel computing.* Prentice-Hall, 1992.

[52] Lewis, T.G., *Where is computing headed?* Computer, IEEE Computer Society august 1994.

[53] Lipschutz, S., *Matemáticas para computación.* McGraw-Hill, 1992.

[54] Little, M.J., Grinberg, J., *The Third Dimension.* Parallel Processing, BYTE, November 1988.

[55] Meyer, B., *Object-Oriented Software Construction.* C.A.R. Hoare Series Editor, Prentice-Hall, 1988.

[56] Meyer, B., *Eiffel, the legacy of Simula.* Happy 25th Anniversary Objects! Special Silver Anniversary Supplement. SIGS Publications 1992.

[57] Moorman Zaremski, A., Wing, J.M., *Signature Matching: A Key to Reuse.* ACM SIGSOFT, Software Engineeering Notes, December 1993.

[58] Neumann, J. von, *The Computer Brain.* Yale University Press, 1958.

[59] Norton, C.D., Szymanski, B.K., Decyk, V.K., *Object-Oriented Parallel Computation for Plasma Simulation,* Communications of the ACM, october 1995.

[60] Nygaard, K., *To program is to understand.* Happy 25th Anniversary Objects! Special Silver Anniversary Supplement. SIGS Publications 1992.

[61] Obermeier, K.K., *Side by side.* Parallel Processing, BYTE, November 1988.

[62] O'Brien, T., Roberts, G., Wei, M., Winder, R., *UC++ v1.0: Language Definition and Semantics.* Department of Computer Science, UCL. May, 1994.

[63] Parkes, S., Chandy, J.A., Banerjee, P., *A Library-based Approach to Portable, Parallel, Object-Oriented Programming: Interface, Implementation and Application.* Supercomputing '94, November 1994.

[64] Perihelion Software Ltd., *The Helios Parallel Operating System.* Prentice-Hall, 1991.

[65] Peterson, G.L., *Myths about the mutual exclusion problem.* Information processing letters, Volume 12, Number 3, 13 June 1981.

[66] Pountain, D., May, D., *A tutorial introduction to Occam Progamming.* INMOS, BSP Professional Books, Oxford 1987.

[67] Pšenicka, B., *Procesamiento Digital de Señales, Primera Parte, Filtros Digitales.* UNAM DIEEC, Facultad de Ingeniería, México 1994.

[68] Roebbers, H., Welch, P., Wijbrans, K., *A generalized FFT algorithm on transputers.* ESPRIT Parallel Computing Action proj. nr. 4122.

[69] Russ, J.C., *The Image Processing Handbook.* The CRC Press,1992.

[70] Sedgewick, R., *Algorithms in C++.* Addison-Wesley Publishing Company, 1992.

[71] Stein, R.M., *T800 and Counting*. Parallel Processing, BYTE, November 1988.

[72] Stevens, W.R., *UNIX Network Programming*. Prentice-Hall, 1990.

[73] Stroustrup, B., *The C++ Programming Language*. AT&T Bell Laboratories. Addison-Wesley Publishing Co., 1991.

[74] Stroustrup, B., Ellis, M.A., *The Annotated C++ Reference Manual*. AT&T Bell Laboratories. Addison-Wesley Publishing Co., 1991.

[75] Stroustrup, B., *Tracing the roots of C++* . Happy 25th Anniversary Objects! Special Silver Anniversary Supplement. SIGS Publications 1992.

[76] Tannenbaum, A., Kaashoek, F., Bal, H., *Parallel Programming using shared Objects and Broadcasting*. Computer, IEEE Computer Society, august 1992.

[77] Terwillinger, R.B., Maybee, M.J., Osterweil, L.J., *An Example of Formal Specification as an Aid to Design and Development*. ACM SIGSOFT, Software Engineering Notes, March 1989.

[78] Wegner, P., *Dimentions of Object Oriented Modeling*. Computer, IEEE Computer Society, october 1992.

[79] Winbald, A.L., Edwards, S.D., King, D.R., *Software orientado a objetos*. Addison-Wesley/Díaz de Santos, 1993.

[80] Wirth, N., *The programming language PASCAL*. Acta Informatica 1, 1, 1971.

[81] Wyatt, B., Kavi, K., Hufnagel, S., *Paralelism in Objetc-Oriented languajes: a survey*. IEEE Software, november 1992.