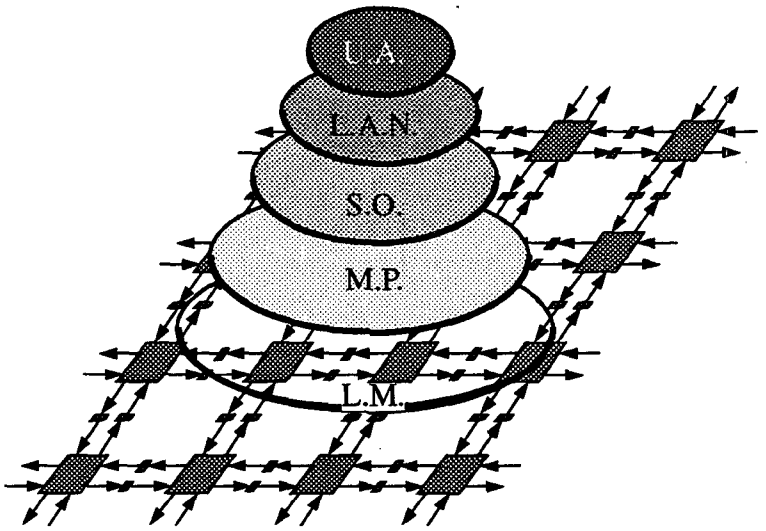
Introducción a la Progamación Concurrente y las Arquitecturas Paralelas



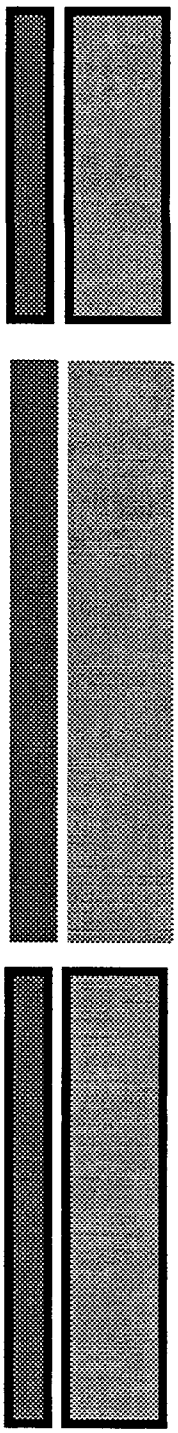
Alvaro Suárez Sarmiento

Departamento de Electrónica, Automática y Telemática

Escuela Técnica Superior de Ingenieros de Telecomunicación

Universidad de Las Palmas de G.C.

Introducción a la Programación Concurrente y las Arquitecturas Paralelas



Alvaro Suárez Sarmiento

Departamento de Electrónica, Telemática y Automática

Escuela Técnica Superior de Ingenieros de Telecomunicación

Universidad de Las Palmas de G.C.

Introducción a la Progamación Concurrente y las Arquitecturas Paralelas

Editado e impreso por el Servicio de Reprografía de la Universidad de Las Palmas de Gran Canaria, Febrero 1996. Campus Universitario de Tafira. Las Palmas de Gran Canaria (35017).

Fecha de impresión: Febrero 1996.

Fotocopiadora:

- Marca: **Rank Xerox**
- Modelo: 5090
- Número de serie: 1104236487

Depósito Legal: GC-180-1996

ISBN: 84-87526-34-9

Indice

Prólogo

En la actualidad existen varios modelos de programación clásicos. Uno de estos modelos de programación es el modelo de programación concurrente. Aunque este modelo se puede utilizar en la programación de un amplio espectro de sistemas reales, simplifica el diseño de software de varios tipos particulares de sistemas de computación que son ampliamente utilizados. Los sistemas distribuidos y tolerantes a fallos son programados naturalmente haciendo uso de este modelo. Algunas aplicaciones reales requieren un tiempo de ejecución pequeño: visualización gráfica realista por computador, control de sistemas en los que se el tiempo de respuesta es crítico, resolución de un gran número de ecuaciones matemáticas, etc. Para obtener este tiempo de ejecución es necesario utilizar arquitecturas con varios procesadores o computadores (arquitecturas paralelas). Aunque en principio la justificación de estas arquitecturas se hizo pensando en este tipo de aplicaciones, cada día son más comunes las máquinas de propósito general que incluyen más de un procesador para resolver cualquier tipo de aplicación.

En este documento se presentan algunas ideas básicas sobre la programación concurrente y las arquitecturas paralelas partiendo del modelo de programación imperativo y de la arquitectura secuencial de Von Neumann respectivamente.

Este documento se estructura en dos temas, en el primero presentamos:

a) Los conceptos básicos de la programación. Se presenta el diseño (desde el punto de vista tradicional) de programas resumidamente.

b) Analizamos varios modelos de programación. Para cada uno de ellos utilizamos el problema del cálculo del factorial de un número entero positivo como ejemplo. Este ejemplo es útil para comparar como se especifica y codifica un problema en diferentes lenguajes de programación.

c) Nos centramos en el modelo de programación concurrente. Simplemente se plantea la idea básica de sincronización y comunicación entre procesos. Analizamos los

requerimientos de ejecución y comunicación entre procesos para el problema del factorial.

d) Por último presentamos algunas aplicaciones potencialmente concurrentes.

En el segundo tema se describen los conceptos básicos de arquitectura de computadores. Se estudian estos conceptos sobre partiendo de la arquitectura de Von Neumann. Se estudia algunas posibles modificaciones de esta arquitectura que conducen hacia arquitecturas de computadores de alto rendimiento. Se presentan las ideas básicas de un amplio abanico de arquitecturas de este tipo y algunas aplicaciones reales en las que son necesarias este tipo de máquinas.

Al final del documento se presentan: unos comentarios breves sobre la bibliografía utilizada para elaborar este documento y varios ejercicios propuestos.

Tema 1. Programación Concurrente

Se presenta una breve introducción a la programación. Se analizan diferentes modelos de programación mediante ejemplos sencillos que ponen de manifiesto algunas de sus características más significativas. Se introduce el modelo de programación concurrente y se presentan varios tipos de sistemas en los que se simplifica el diseño del software (o bien es necesario este tipo de software) mediante el modelo de programación concurrente.

1.1. Conceptos básicos de la programación

La solución de un problema computacional debe ser especificada en términos de secuencias de cálculos, cada uno de los cuales debe ser efectuado efectivamente por un agente humano o por un computador digital. Las notaciones sistemáticas para la especificación de estas secuencias de cálculos se realizan mediante los *lenguajes de programación.* Una especificación de la secuencia de pasos de cálculos en un lenguaje de programación particular es un *programa.* La tarea de desarrollar programas para la solución de problemas computacionales es la *programación.* La persona que está encargada de realizar la programación es el *programador.*

A partir de la especificación de un problema se debe obtener un *algoritmo* de resolución que guíe la solución del problema en una máquina determinada siguiendo unas técnicas de diseño determinadas y un modelo de programación determinado. Una definición de algoritmo es la siguiente: [RaR93, pág.. 1106]

Dados un problema y una máquina, un algoritmo es la caracterización precisa de un método de solución de un problema. En particular, un algoritmo está caracterizado por las siguientes propiedades:

- La aplicación del algoritmo a un conjunto de entradas particulares o una descripción del problema deberá dar un resultado en una secuencia finita de pasos.
- La secuencia de acciones tienen una acción inicial única.
- Cada acción de la secuencia tiene un único sucesor.

- La secuencia termina con una solución al problema o con una condición de error que indica que el problema no tiene solución para el conjunto de datos iniciales o descripción del problema.

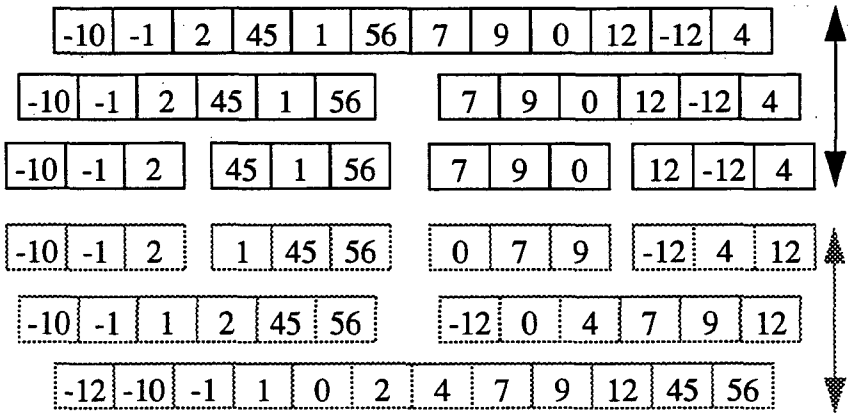
Existen dos tópicos importantes en el estudio de las técnicas de diseño de algoritmos: a) Este estudio es una guía para idear nuevos algoritmos de una forma organizada. Las técnicas de diseño de algoritmos son una guía para crear nuevos algoritmos: existen miles de algoritmos pero pocas técnicas de diseño de algoritmos. b) Este estudio ayuda a categorizar u organizar los algoritmos conocidos tal que éstos se puedan entender mejor.

Algunas técnicas de diseño de algoritmos son: Divide y vencerás, Programación dinámica, Ramificación y acotación, Backtracking (ejecución con vuelta atrás), Métodos heurísticos, etc.

Ejemplo 1.1 Ordenación de vectores mediante la técnica Divide y vencerás

La técnica Divide y vencerás sugiere que un problema debería ser dividido en subproblemas, que de alguna forma es resuelto y entonces las soluciones son combinadas en una solución del problema original. Usualmente los subproblemas son resueltos de la misma forma (dividiéndolos, resolviéndolos y combinando las subsoluciones). Por lo tanto, la solución global se puede obtener recursivamente. Es deseable dividir los subproblemas en problemas del mismo "tamaño" (cantidad de cálculos a realizar, o bien cantidad de datos a procesar).

Un ejemplo ideal para aplicar esta técnica es la ordenación de vectores mediante mezclas. El método trabaja de la siguiente forma: dividir el vector en dos conjuntos de elementos iguales (o muy semejantes en tamaño). Entonces ordenar cada conjunto de elementos individualmente. A continuación mezclar los resultados de las ordenaciones parciales tal que el vector quede ordenado. Esta subdivisión de elementos se puede efectuar varias veces, tal como se muestra en la figura 1.1.



División en subproblemas — Solución y Combinación de resultados

Figura 1.1. Ordenación de vectores mediante la técnica Divide y vencerás

❐

En los libros básicos de programación se puede encontrar una descripción de las técnicas algorítmicas presentadas en este apartado.

Desde el planteamiento del problema hasta que se obtiene el correspondiente algoritmo de resolución instalado en el computador, listo para su uso, se ha de seguir un proceso riguroso que asegure la validez y calidad del programa obtenido. Este proceso está compuesto de varias fases tal como se muestra en la figura 1.2 [AGP90].

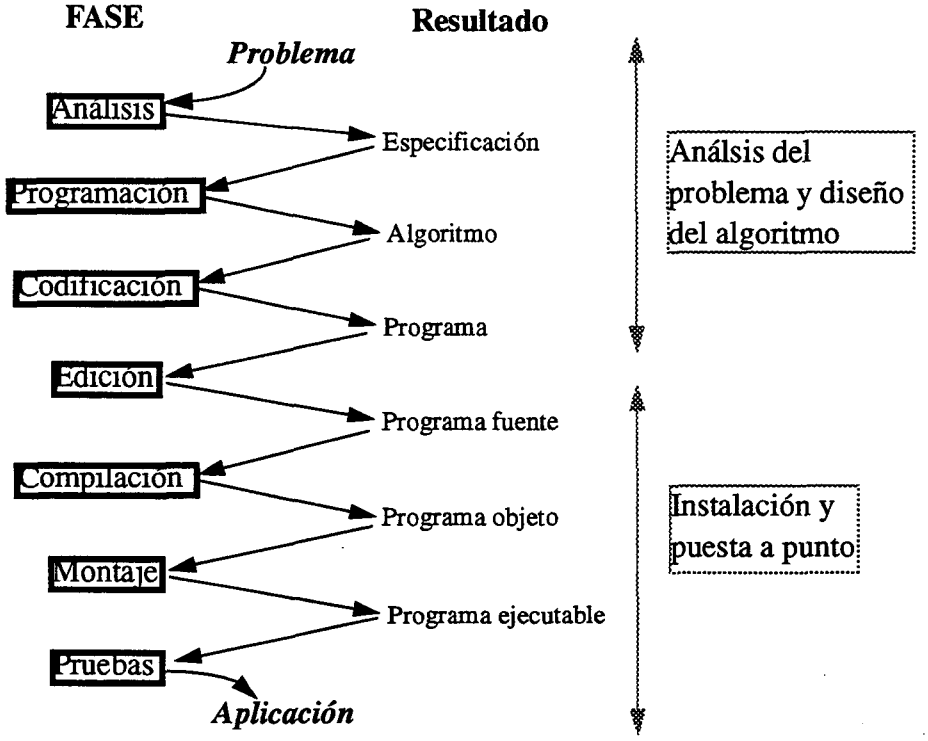


Figura 1.2. Proceso de automatización de un problema

En la fase de análisis se debe estudiar detalladamente el problema con el fin de obtener una serie de documentos (*especificación*), en los que quede totalmente definido el proceso a seguir en la automatización. En la fase de *programación* se obtiene un algoritmo que puede ser representado de diversas formas y utilizando técnicas como el seudocódigo, los organigramas, etc. (notaciones intermedias). La fase de codificación consiste en escribir, en un lenguaje de programación de alto nivel, el algoritmo obtenido en la fase anterior. Es en este punto en el que es crucial elegir un lenguaje de alto nivel adecuado al modelo de programación que se determine para resolver el problema (ver apartado 1.2). En algunos casos es importante tener en cuenta que la técnica de diseño de algoritmos se debería adaptar al modelo de programación elegido. Es importante no confundir el papel del programador con el codificador del algoritmo que simplemente transcribe un algoritmo a un lenguaje de programación de alto nivel.

El programa objeto se obtiene aplicando al programa fuente un compilador. El programa ejecutable se obtiene a partir del programa objeto incluyendo en este programa rutinas internas del lenguaje y haciendo intervenir al sistema operativo. Una vez obtenido este

programa se deberá proceder a hacer las pruebas oportunas para asegurar que el resultado que se obtiene de la aplicación es correcto. Alternativamente se pueden emplear técnicas de comprobación de la correctitud del programa fuente sin necesidad de generar el programa ejecutable. Esto puede dar como resultado el ahorro de gran cantidad de tiempo invertido en las fases de compilación, montaje y pruebas (que en algunos casos puede ser una cantidad de tiempo considerable). Sin embargo estos métodos pueden ser excesivamente complicados si el tamaño del programa fuente es grande. En [Pre93] se puede encontrar un análisis profundo de cada una de estas fases aplicados a ejemplos reales de diseño de software en empresas americanas.

1.1.1. Evolución de la programación

Los primeros programadores de las primeras máquinas digitales tenían que especificar la secuencia de acciones en lenguajes ensambladores primitivos. En estos programas existían muchas sentencias de transferencia del control (**goto**). Lenguajes como el Fortran heredaron este estilo de programación *no estructurada.*

Ejemplo 1.2 Código no estructurado escrito en Fortran

En este ejemplo se muestra un extracto de código escrito en Fortran IV en el que existen sentencias goto.

```
      . . .
IF (A.GT. B) GOTO 20
10    J=3*k
      L=7
GOTO 30
20    K=K+1
      M=2
GOTO 10
30    CONTINUE
      . . .
```

Figura 1.3. Código no estructurado escrito en Fortran IV

En el código de la figura 1.3 seguir la "línea" de ejecución es difícil puesto que tenemos que ir buscando donde están las etiquetas operando de la sentencia goto y a continuación pensar como se realiza la ejecución del programa. En este sencillo ejemplo es fácil averiguar que en el caso de que el valor de la variable A sea mayor que el de la variable B actualizamos las variables K y M y a continuación las variables J y L. Sin embargo; si el valor de A es menor o igual que el de B entonces solo actualizamos a J y L.

En códigos más complejos se complica muchísimo el análisis de los programas y la expresividad del lenguaje es muy pobre puesto que no se interpreta de forma fácil la solución del problema (codificación del algoritmo).

❐

Este estilo de programación se utilizó en el que se reconoce como primer lenguaje de alto nivel, el FORTRAN (finales de los años 50). En 1968 Dijkstra publica en la revista *communications of ACM* una carta al editor titulada "Go to Statement Considered Harmful", y expone su observación de que la facilidad de lectura y comprensión de los listados de los programas es inversamente proporcional al número de sentencias goto (transferencia incondicional del control), que éstos contienen. Con esta carta se da vía a una explosión de interés por la *programación estructurada.*

La programación estructurada se define como un estilo metodológico por medio del cual un programa es construido concatenando o imbricando coherentemente subunidades lógicas que son programas estructurados en sí mismos o bien están constituidos por un conjunto de estructuras de control bien conocidas. Nótese que esta definición es recursiva. En 1964, Bohm y Jacopini [RaR93, pág. 1309] probaron que todo programa, por complicado que fuese, puede ser escrito usando repetida o imbricadamente subunidades de no más de tres tipos diferentes: a) Una secuencia de sentencias ejecutables (**begin...end**), b) una cláusula de decisión (**if-then-else**), c) una construcción de iteración (**while**). Este modelo está influido por la *programación orientada a bloques y procedimientos* introducida por primera vez en el lenguaje Algol 60 en el que se podían declarar bloques de código separados por las palabras claves begin y end. En estos bloques se podían declarar variables o procedimientos cuyo ámbito de existencia era interno al bloque en el que se declaran (cualquier referencia a esos procedimientos o variables fuera del bloque es una causa de error ya que el efecto es el de que esos objetos no existen fuera del bloque).

Ejemplo 1.3 Código estructurado en Algol 60

En este ejemplo se muestra el código del ejemplo 1.2 escrito de forma estructurada en el lenguaje Algol 60.

La lectura del código de la figura 1.4 es más sencilla puesto que se pone de manifiesto directamente la estructura inherente del programa. Además se expresa mejor y más claramente las acciones que se deben realizar en función de los valores de las variables A y B.

❐

Una posible técnica de diseño de programas consiste en comenzar con la definición más general de la función a realizar por el programa y a continuación realizar una serie de diseños por análisis descendente de problemas (el planteamiento coherente de un conjunto de subprogramas conduce al diseño final del programa). Esta técnica (*diseño descendente*) es un aspecto de la programación estructurada y es grandemente mejorada mediante por la

```
    . . .
If A> B then begin
        K=K+1;
        M=2;
end;
J=3*k;
L=7;
    . . .
```

Figura 1.4. Código estructurado escrito en Algol 60

programación modular. El objetivo de la programación modular es dividir una tarea en tareas más pequeñas y más simples de resolver para facilitar la escritura correcta de programas (este es uno de los objetivos que se persiguen). Un programa o módulo se puede definir como una parte lógicamente autocontenida de un programa mayor. Un programa completo puede ser considerado como un conjunto de módulos. Cada módulo debe aceptar una o varias entradas y debe dar una o varias salidas que sean correctas. La programación modular es fundamental para diseñar grandes programas para lo que colaboran un conjunto grande de prorgramadores. Quizás el lenguaje más significativo de la programación modular es el Modula-2 de Niklaus Writh.

Ejemplo 1.4 Código modular escrito en Modula-2

En este ejemplo se muestran las ideas básicas de la construcción de un programa modular utilizando el lenguaje Modula-2. En la figura 1.5 se muestra un código escrito en Modula-2 que codifica el calculo del máximo de un vector de números enteros utilizando una función que calcula el máximo de dos números enteros. Se define un módulo (DEFINITION MODULE) que contiene todos los procedimientos y variables que pueden ser utilizados en otros módulos. La parte de implementación (IMPLEMENTATION MODULE) especifica todo el código que se debe ejecutar. La parte de definición se debe especificar cuando se desea que otros módulos (que pueden ser programas enteros), utilicen las variables y procedimientos especificados en esta parte. La parte de implementación es compilada por separado.

Supongamos que queremos diseñar un programa que utilice el máximo de un vector y el carácter cuyo código hexadecimal es el mayor de un conjunto predefinido para efectuar cálculos. Procederíamos a definir un módulo para el tratamiento de caracteres y utilizaríamos el procedimiento de cálculo del máximo (FROM Vec IMPORT max_vec) para el tratamiento de vectores, tal como se muestra en la figura 1.6.

□

```
DEFINITION MODULE Vec;
    PROCEDURE max_vec (VAR v:ARRAY OF CARDINAL): CARDINAL;
END Vec;
IMPLEMENTATION MODULE Vec;
    PROCEDURE max (VAR a, b: CARDINAL): CARDINAL;
      BEGIN
        IF a> b THEN max:= a;
        ELSE max:= b;
      END max;
    PROCEDURE max_vec (VAR v:ARRAY OF CARDINAL): CARDINAL;
      VAR i, maximo, tmp: CARDINAL;
      maximo:= v[1];
      FOR i=1 TO HIGH(v)-1 BY 2 DO
        tmp:= max (v[i], v[i+1]);
        IF maximo<tmp THEN maximo:= tmp;
      END;
      max_vec:= maximo;
    END max_vec;
END Vec.
```

Figura 1.5. Código modular escrito en Modula-2

La idea de que en los programas se pueda acceder a las variables globales desde cualquier parte del programa tiende a ser producir programas poco manejables, esto llevó a David Parnas en 1970 a defender el concepto de *ocultación de información.* La idea es *encapsular* cada variable global en un módulo junto con un grupo de operaciones (por ejemplo procedimientos y funciones), que son las únicas que tienen acceso a las variables encapsuladas. Otros módulos sólo tienen acceso a esas variables mediante la invocación de las operaciones encapsuladas. Frecuentemente se denomina *objeto* a tal módulo que encapsula datos y operaciones. Por ejemplo, en el código de la figura 1.6 podríamos haber definido una variable dentro del módulo Car que sólo fuese accesible dentro de este módulo; el resto de módulos deberían importar la definición de esta variable para poder utilizarla. Se dice que en este caso esa variable estaría encapsulada con el resto de procedimientos dentro de este módulo (objeto). De esta forma, tenemos pleno conocimiento de cuales son las variables que pertenecen a cada módulo y sabemos que sólo debemos modificar el módulo correspondiente cuando se desee modificar la definición de esa variable.

Se dice que un lenguaje de programación está basado en objetos si soporta objetos como una característica del lenguaje. Se dice que un lenguaje está orientado a objetos si además

```
DEFINITION MODULE Car;
    PROCEDURE mayor_car (VAR s:ARRAY OF CHAR): CARDINAL;
END Car;
IMPLEMENTATION MODULE Car;

        . . .

END Car.
MODULE mi_prog;
    FROM Vec IMPORT max_vec;
    FROM Car IMPORT mayor_car;
    VAR i, j: CARDINAL
    BEGIN

        . . .

END mi_prog.
```

Figura 1.6. Reutilización de Módulos en Modula-2

los objetos pertenecen a clases que pueden ser modificadas mediante un mecanismo de herencia [Wat90]. El concepto de *clase* se introdujo en el lenguaje de programación Simula 67 como una extensión a la estructura de bloque del Algol 60. En la declaración de una clase pueden existir parámetros formales (igual que los procedimientos), e incluye declaraciones de variables, funciones y procedimientos que son locales a la clase, seguida de un "cuerpo" de la clase que es usualmente un bloque. En los lenguajes orientados a objetos, el tipo de los objetos se especifica haciendo referencia a las clases (de forma semejante a como se declaran los tipos de las variables en los lenguajes tradicionales), y se usa para clasificar a los objetos en jerarquías a través de los mecanismos de herencia. El mecanismo de herencia permite compartir código y comportamiento entre objetos. Las nuevas clases heredan las operaciones de su clase "padre" y además pueden añadir nuevas operaciones sobre las variables de la clase.

Una de las diferencias fundamentales entre los procedimientos de Algol y las clases es que el código de declaración de variables, procedimientos y el código ejecutable de las clases no es cargado en memoria después de la compilación; cuando se ejemplariza una clase o un objeto, y se utiliza, entonces se reserva memoria para su utilización.

Ejemplo 1.5 Ejemplo de definición de un objeto y herencia

En este ejemplo mostramos un seudocódigo de declaración de un objeto que es un punto de un espacio plano y mostramos las ideas básicas de la herencia entre objetos.

En la figura 1.7 se muestra un seudocódigo de declaración de un objeto referido a la definición y operaciones que podemos realizar con las coordenadas de ese punto. La

```
Punto: Objeto
interface= (Leer_X: Real, Leer_Y: Real,
Cambiar_x: Real->Real, Cambiar_y: Real-> Real)
x:=0; y:=0; (* Inicializaciones *)
(* operaciones que se realizan con las coordenadas *)
Leer_X: return(x);
Leer_Y: return(y);
Cambiar_x (dx): x:=x+dx;
Cambiar_y (dy): y:=y+dy;
```

Figura 1.7. Seudocódigo de declaración de un objeto

interface de operaciones especifica el tipo de operación y operandos que se pueden utilizar accediendo a este objeto. En este ejemplo Leer_X y Leer_Y son procedimientos que retornan un número real. Cambiar_x y Cambiar_y tienen un parámetro real y retornan (->) un número real.

Una vez definido este objeto podemos definir una clase (punto), y hacer ejemplarizaciones de esta clase definiendo nuevas operaciones sobre las nuevas ejemplarizaciones (subclases), como por ejemplo permitir que los puntos tengan un determinado color, añadiendo operaciones para obtener su color.

❐

La declaración de una clase se parece a la declaración de procedimientos de Algol 60 que fue la base en la que se inspira el mecanismo de *tipos abstractos de datos*. Un tema dominante en el desarrollo de lenguajes y modelos de programación es el desarrollo de herramientas que tratan con las *abstracciones*. Una abstracción es una descripción simplificada, o especificación, de un sistema que se centra sobre alguna estructura esencial o comportamiento de un sistema real o un objeto conceptual. Una buena abstracción es aquella en la que la información importante para el usuario de la abstracción esta enfatizada mientras que los detalles que no son relevantes, en un momento determinado, son suprimidos. Un tipo abstracto de datos es una facilidad de los lenguajes de programación para organizar los programas en módulos usando criterios que están basados en las estructuras de datos del programa. La especificación del módulo debería proporcionar toda la información requerida para usar el tipo, incluyendo valores disponibles de datos y los efectos de las operaciones. Sin embargo; los detalles sobre la implementación, tales como representación de los datos, y los algoritmos de implementación de las operaciones son ocultadas dentro del módulo (ocultación de información). Esta ocultación de la especificación sobre su implementación es la idea fundamental de los tipos abstractos de datos.

Los criterios que se utilizan para organizar los módulos enfatizan la protección de las

estructuras de datos sobre las manipulaciones accidentales o maliciosas en otras partes del programa. Al igual que en la programación estructurada, la programación con tipos abstractos de datos hace énfasis sobre la localidad de los conjuntos de información. E el caso de los tipos abstractos de datos se centra la atención sobre los datos más que en el control y la estrategia es construir módulos que se da más importancia a los tipos de datos que a las operaciones asociadas a esos tipos de datos (definir un tipo de datos que permita abstraer muy bien la realidad del problema que queremos resolver permite que el diseño de las operaciones sea sencillo).

Para diseñar un tipo abstracto de datos, primero especificamos las propiedades funcionales de la estructura de datos y sus operaciones asociadas y entonces implementamos estas operaciones con las construcciones que nos permita un lenguaje determinado.

Ejemplo 1.6 Definición de un tipo abstracto de datos Pila

En este ejemplo definimos un tipo abstracto de datos Pila cuyos elementos tienen un tipo de datos genérico (T). En la figura 1.8 se muestra la definición de este tipo abstracto de datos en seudocódigo.

```
Type Pila (T:Tupe);
specifications
    procedure push (var P: Pila(T); x: T); (* especificación formal del push *)
    procedure pop (var P: Pila(T));  (* especificación formal del pop *)
    function top (var P: Pila(T)); return T; (* especificación formal del top *)
implementation
    (* Declaraciones de los tipos de datos para representar la Pila *)
    (* código de los procedimientos y funciones *)
end Type.
```

Figura 1.8. Seudocódigo de declaración de un tipo abstracto de datos

Habiendo especificado este tipo abstracto de datos, un programador podría ejemplarizar ese tipo con cualquier tipo de datos soportado directamente por el lenguaje o bien por cualquier tipo de datos que se haya definido mediante los constructores básicos del lenguaje. En la figura 1.9 se muestra una ejemplarización del tipo Pila. En esta figura también se muestra como se pueden realizar operaciones con las variables (algunas de ellas han sido ejemplarizadas con un tipo abstracto de datos).

□

A partir de los años 60 los lenguajes modernos como Ada, C++, Concurrent Pascal, LOTOS permiten diseñar tipos abstractos de datos. En los lenguajes orientados a objetos son una parte fundamental del lenguaje: inherente a los objetos pueden tener asociados uno o

```
declare
    X: Pila (integer);
    Y: Pila (real);
    j: integer; r: real;
    ...
push (X, j); r:= top (Y);
```

Figura 1.9. Ejemplarización del tipo abstracto de datos Pila

varios tipos abstractos de datos.

1.2. Modelos de programación

Una vez analizados los conceptos básicos de la programación, en este apartado presentamos algunos modelos de programación [Wat90] más comunes y haremos especial hincapié en el modelo de programación concurrente.

Según el Diccionario de la Real Academia de la Lengua Española, un *modelo* es un esquema teórico de un sistema o realidad compleja, que se elabora para facilitar su comprensión y estudio. Nosotros por *modelo de programación* de computadores entendemos un esquema teórico de especificación de la solución de problemas computacionales que se elabora para facilitar la comprensión y el estudio de la programación de los computadores. Existen lenguajes de programación directamente asociados a un modelo de programación. Estos lenguajes permiten implementar, en el computador, la especificación de la solución a un problema computacional expresado en el modelo de programación.

El modelo de programación *imperativo* (imperare en Latín significa ordenar), está basado en órdenes que actualizan las variables almacenadas en la memoria del computador. Este es el modelo más antiguo y el que se puede implementar más eficientemente (en las arquitecturas secuenciales) puesto que su semántica está muy "próxima" a la implementación tradicional secuencial de los computadores: búsqueda de instrucciones y/o datos, ejecución y almacenado de datos. Este es el modelo de programación, que sigue siendo el más utilizado por los programadores debido, entre otras cosas, a la gran cantidad de software ya escrito y a la facilidad de utilización.

Lenguajes como Cobol, Algol 60, Fortran, Pascal, C, Ada, etc. son representativos de este modelo.

Teniendo en cuenta las técnicas de diseño de programas estructurados y modulares, la ocultación de la información y los tipos abstractos de datos, planteamos la siguiente ecuación:

$$\text{Programa} = \text{Algoritmo} + \text{Estructuras de } \textit{datos}$$

Donde el algoritmo sería el conjunto de órdenes secuenciales que definen cómo se resuelve el problema computacional y las estructuras de datos estarían orientadas a definir la modularidad del programa (para lo cual hemos tenido en cuenta un lenguaje imperativo con posibilidad de especificar ocultación de la información).

Ejemplo 1.7 Cálculo del factorial de un número entero positivo

En este ejemplo se muestra la codificación escrita en el lenguaje Pascal de una función para el cálculo del factorial de un número entero positivo.

```
function factorial (n:integer):integer;
    var f:integer;
    begin
       f:=1;
       while n> 0 begin
          f:= f * n;
          n:= n-1;
       end;
       factorial:= f;
    end; (* factorial *)
```

Figura 1.10. Función Pascal de cálculo del factorial en el lenguaje Pascal

En la figura 1.10 se muestra el código de una función de cálculo del factorial de un número (n). Nótese que existe un bucle iterativo (while), en el que se especifica "cómo" se hace el cálculo del factorial utilizando un conjunto de variables y las operaciones que se deben realizar con estas variables: inicialización de resultados (f=1), actualización del valor del factorial (f=f*n). Se utiliza una variable de control de las iteraciones denominada n que es el número para el cual se desea calcular su factorial.

Supongamos que n=3. Se evaluará la llamada a la función factorial (3) que retornará un número entero. Se reserva memoria para la variable f y se inicializa a un valor aleatorio (var f:integer;). A continuación se producen las siguientes acciones:

- while 3> 0; f:= 1 * 3=3; n:= 3 - 1=2
- while 2> 0; f:= 3 * 2=6; n:= 2 - 1=1
- while 1> 0; f:= 6 * 1=6; n:= 1 - 1=0
- while 0> 0; (FALSO). factorial:= 6 (retorna el valor entero 6).

□

El modelo de programación *orientado a objetos* se deriva directamente del modelo imperativo y está basado en el concepto de objeto y clases de objetos. Este modelo es una

disciplina que cuenta con los objetos para imponer una estructura modular a los programas. El lenguaje más representativo de este modelo es el smalltalk.

La programación orientada a objetos es un método de implementación en el que los programas son organizados como colecciones cooperativas de objetos cada una de las cuales representa una instancia de alguna clase, y cuyas clases son miembros de jerarquías de clases unidas a través de una relación de herencia [AlS95].

Entre las razones para utilizar la programación orientada a objetos están:

- La complejidad del dominio del problema. Si el problema a resolver da como resultado una gran cantidad de módulos es necesaria una buena organización de éstos.
- Dificultad de gestionar el desarrollo de los módulos. La realización de estos módulos se simplifica estructurandolos jerarquicamente.
- La flexibilidad existente en el desarrollo del software. Se pueden reutilizar módulos ya existentes para diseñar otros nuevos.

La herencia es el mecanismo básico para implementar la reusabilidad y extensibilidad de los programas. A través de ella los programadores pueden construir nuevas clases partiendo de una jerarquía de clases ya existentes (comprobadas y verificadas), evitando con ello el rediseño, la recodificación y la verificación de la clase ya implementada. Existen varios tipos de herencia (sólo mencionamos algunos tipos): a) Especialización. Por ejemplo podemos definir un objeto que tenga como variables todas las características de un ser animal (altura, color de los ojos, definición de todos los huesos, etc.). Un objeto que definiese a los seres humanos podría heredar sólo partes de éste (los seres humanos tienen menos huesos que por ejemplo las ballenas). b) Extensión. Por ejemplo un objeto que defina a un empleado de una fábrica podría tener más variables que los de la clase ase que definiría a las personas en general.

En los años 90 este modelo ha tenido mucho éxito y han surgido nuevos lenguajes como C++, Object Pascal, etc.

Un Programa Orientado a Objetos (POO) podría definirse mediante la ecuación:

$$\text{POO} = \text{Algoritmo} + \text{Objetos}$$

Donde se fuerza una determinada estructura del código fuente de alto nivel que se construye en base a objetos y la herencia entre objetos. Nótese que se sigue teniendo un algoritmo imperativo que define la estructura de control de ejecución del programa en el que se especifica cómo se resuelve el problema computacional.

Ejemplo 1.8 Cálculo del factorial de un número en turbo C++

En este ejemplo se muestra el cálculo del factorial de un número entero positivo escrito en

el lenguaje Turbo C++.

En la figura 1.11 se muestra el código de la definición de la clase Factorial (**class** Factorial). En las clases podemos tener funciones (métodos) que pueden ser utilizados por los objetos (**public**). En la función main se define el objeto C (**Factorial C**). Utilizando los métodos de este objeto podemos leer el valor de n, calcular el valor del factorial (por ejemplo C.calcula()) y a continuación escribir el valor del resultado de esta operación.

Supongamos que algún programador requiera calcular el factorial de un número y que dispone de la clase Factorial ya escrita y en una biblioteca de objetos. Simplemente tendría que declarar un objeto de tipo Factorial, y hacer las llamadas a los métodos que se especifican en la función main de este ejemplo. Está reutilizando el código que alguien escribió previamente y habrá diseñado su programa en muy poco tiempo y de forma fiable.

Se puede notar que una clase en Turbo C++ podría interpretarse como un tipo de dato struct con la definición encapsulada de funciones que deben ser definidas por el creador del objeto (por ejemplo, int Factorial::leer_datos(void)).

Supongamos que deseamos calcular el factorial (3). Después de haber compilado el programa, lo ejecutamos. Una vez realizada la ejemplarización del objeto C se reserva memoria para él. En primer lugar se invoca el procedimiento leer_datos que retorna un valor igual a 1 lo que significa que el valor de la variable n (int n) es positivo. Por tanto, se invoca al procedimiento calcula y se producen los mismos pasos que para la función Pascal (aquí resultado es equivalente a la variable f del código Pascal). Se calcula el resultado=6. Finalmente se invoca al procedimiento escribir_resultado que escribe en pantalla: factorial=6. Nótese que a medida que se van invocando estos procedimientos se puede ir cargando en memoria su código con el consiguiente ahorro de memoria. Esto se puede hacer puesto que estos procedimientos no tienen porque ser parte del código compilado y cargado para la función main.

❐

Los lenguajes del modelo imperativo tienen expresiones y funciones pero son una parte muy empobrecida del modelo. El modelo de programación *funcional* está caracterizado por el uso de expresiones y funciones que hacen uso del concepto de funciones de orden superior (higher-order functions) y de la evaluación perezosa (lazy evaluation). Se dice que una función es de orden superior si sus argumentos o sus resultados son funciones [Set92]. El mecanismo de evaluación perezosa consiste en que un argumento de una función es evaluado en el momento en que es utilizado por primera vez, más que en el momento en el que se activa la función. Si el argumento no se utiliza nunca entonces no se evalúa jamás (con el consiguiente ahorro de tiempo de ejecución). De este modo podemos pasar una expresión no evaluada a una función.

Los lenguajes más característicos son el Lisp, ml, Miranda, Haskell y Eiffel.

Ejemplo 1.9 Evaluación perezosa y funciones de orden superior en ml

En este ejemplo mostramos la generación de una lista infinita de números superior a un

```
#include <stdio.h>
class Factorial {
int n ;     // parametros de entrada
int resultado; // resultado
public:
int leer_datos (void);   // Lee a y b. retorna 1--Ok., 0--a o b<=0
void error_datos (void); // Escribe mensaje de error en la entrada de datos
void calcula (void);    // calcula resultado= mcd (a, b)
void escribir_resultado (void);  // escribe mensaje de salida y resultado
}; // Fin de declaración clase. Implementación de funciones.
int Factorial::leer_datos(void)
{
 printf ("valor de n= ");
 scanf ("%d", &n);
 if (n<0) return (-1);
 else return (0);
}
void Factorial::error_datos (void)
{
 printf ("Dato de entrada erroneo\n");
}
void Factorial::calcula(void)
{
 resultado=1;
 while (n>0) {
     resultado= resultado * n;
     n= n-1;
 }
}
void Factorial::escribir_resultado(void)
{
 printf ("factorial=%d\n", resultado);
}
main ()
{
 Factorial C ;
 if (C.leer_datos()==-1) C.error_datos();
 else { C.calcula(); C.escribir_resultado(); }
 return (1);
}
```

Figura 1.11. Cálculo del factorial en el lenguaje Turbo C++

número (n) dado. Además mostramos como calcular el primer número primo que es mayor a un número m dado.

```
fun from (n) = n :: from (n+1)
```

(a)

```
fun primerprimo (n::ns) =
    if prime (n)
    then n
    else primerprimo (ns)
```

(b)

```
primerprimo ( from (m) )
```

(c)

Figura 1.12. a) Lista infinita de números mayores que n. b) Función que calcula el primer número primo. c) Cálculo del primer número primo mayores que m.

En la figura 1.12 se muestra el código de las funciones utilizadas para el cálculo del primer número primo mayores que m. La función *from* genera una lista infinita (en teoría) de objetos (números enteros), sólo limitado por la capacidad de memoria de la máquina (n:: significa que se va a construir una lista que comienza por n y acaba en ::resto). Se basa en el mecanismo de evaluación perezosa mediante llamadas recursivas, generando primero el número que se pasa en el argumento y a continuación activando a la función con el número incrementado en 1.

Explicamos mediante este ejemplo la evaluación perezosa. Para la función from (n)= n :: from (n+1), un intérprete de ml al que se de la orden from (5) haría lo siguiente:

En primer lugar expande la llamada from (5) y sustituye esta definición por el cuerpo de la función, o sea n :: from (n+1). Entonces evalúa n e identifica n igual a 5. El resultado del primer paso de evaluación perezosa será 5 :: from (5+1). Con esto ya tiene como resultado el primer elemento de la lista y por tanto retorna ese valor como resultado de la función (lo imprime en la pantalla). El intérprete fuerza el siguiente paso de evaluación. Este paso consiste en buscar el siguiente elemento de la lista y para ello evalúa from (5+1) de la misma forma que en el paso anterior: identifica n=5+1=6. El resultado de la evaluación será 6 :: from (6+1), retorna el segundo elemento de la lista (6) y el intérprete prosigue forzando pasos sucesivos de evaluación. Así se irán obteniendo todos los elementos de la lista infinita.

Consideremos ahora la definición de primerprimo y la llamada primerprimo (from (m)). El parámetro from (m) no será evaluado hasta que sea estrictamente necesario (cuando se ejecuta la sentencia prime (n)). Debido a que n forma parte del parámetro de la definición de primerprimo (n::ns), entonces necesita identificar n evaluando from (m). A cada paso de evaluación de from (m) comprobará si ha obtenido la identificación del valor de n. En este caso identifica n igual a m y ns igual a from (m+1) en el primer paso de evaluación. Supongamos que m es primo, entonces el resultado de la función será m. En caso contrario el resultado será la evaluación de primerprimo (from (m+1)) y continuará hasta encontrar un valor de n=k que sea primo.

Nótese que la evaluación perezosa significa que sólo (paso de evaluación)

identificaremos un valor de un objeto (en general), única y exclusivamente cuando sea necesario: aunque from genere una lista infinita de valores enteros, sólo consideramos los necesarios (de forma perezosa). Sin embargo, en los lenguajes imperativos (evaluación ansiosa), se intenta ejecutar todo lo que se pueda (sea necesario hacerlo o no); de ahí que en estos lenguajes la función primerprimo no acabaría nunca jamás de ejecutarse (compruébese la dificultad que conllevaría el implementar este ejemplo en lenguaje Pascal).

La función *primerprimo* retorna el primer número primo de una lista de números enteros (que no esté vacía). Nótese que esta definición es recursiva y se basa en una función "prime" que se supone construida. La función priperprimo es de orden superior puesto que en su activación (llamada), utilizamos como argumentos otra función (from). La forma de calcular el primer número primo mayor o igual a m es pasando como parámetro a primerprimo la función from que genera una lista infinita de números mayores o iguales a m. Esta lista siempre permanece parcialmente evaluada: solamente cuando primerprimo selecciona la cola de la lista de números se hace un "poco" más de evaluación (perezosamente) sobre la lista.

□

Un programa es una función (o grupo de funciones), típicamente compuesto de funciones más sencillas. Las relaciones entre las funciones es muy sencilla: una función puede llamar a otra o el resultado de una función puede ser usado como argumento para otra función. Las variables, órdenes y efectos colaterales son eliminados del modelo; en lugar de usar variables y órdenes, el programa puede ser enteramente escrito con las expresiones del lenguaje, funciones y declaraciones.

Este modelo es bastante ineficiente y bastante apropiado para escribir un tipo particular de aplicaciones; por ejemplo existen muchas aplicaciones de procesamiento de imágenes escritas según este modelo, en lenguaje LISP.

En este modelo de programación los programadores proporcionan las funciones, los pasos de parámetros y el retorno de valores entre ellas, mientras que el lenguaje utiliza estos mecanismos para calcular la solución al problema programado. Nótese que el programador expresa qué se debe hacer y no cómo se debe resolver el problema.

Para este modelo de programación podríamos escribir la siguiente ecuación:

$$\text{Programa} = \text{Funciones} + \text{Control}$$

Donde el parámetro Funciones se refiere a las funciones de que consiste el programa y el control se refiere a los mecanismos de implementación de la implementación de como se resuelve el problema aplicando la evaluación perezosa teniendo en cuenta las funciones de orden superior.

Ejemplo 1.10 Cálculo del factorial en ml

En este ejemplo mostramos dos formas diferentes de calcular el factorial de un número positivo entero en el lenguaje ml. En la figura 1.13.a se calcula el factorial reflejando la utilización de las variables n y f de la figura 1.10; la variable n es un parámetro de la función (igual que en el caso del ejemplo 1.7). Ahora la variable f también es un parámetro de la función *factorialloop*.

Una implementación más sencilla recursiva y tradicional es la que se muestra en la figura 1.13.b. Esta nueva definición es la típica definición recursiva que se utiliza en la mayoría de los lenguajes del modelo imperativo.

```
fun factorialloop (n, f)=
    if n>0
    then factotialloop (n-1, f*n)
    else f
fun factorial (n)= factorialloop (n, 1)
```

(a)

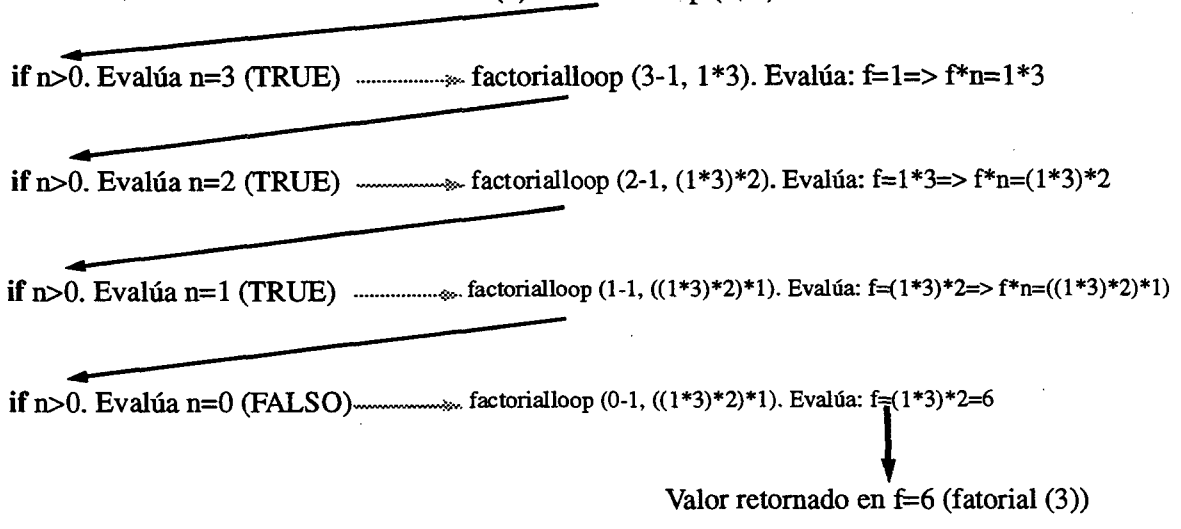
```
fun factorial (n)=
    if n>0
    then n * factotial (n-1)
    else 1
```

(b)

Figura 1.13. a) Función ml de cálculo del factorial utilizando explícitamente las variables n y f. b) Cálculo "tradicional" del factorial.

Supongamos que n=3. Se evaluará factorial (3) que será el resultado de evaluar a factorialloop (3, 1). Entonces se generan pasos de evaluación de la función factorialloop hasta que se produzca un resultado.

Consideremos la definición de factorialloop. En la sentencia **if** n>0 se evaluará el valor de n que se identifica a 3 y se comprueba si 3>0 (CIERTO), entonces se evaluará factorialloop tal como se indica en la siguiente gráfica:



□

Los modelos anteriores están basados en la idea de que un programa es un mapeo (mapping) entre unas entradas y unos resultados que debe producir el programa. En el modelo imperativo el programa es típicamente un conjunto de órdenes que leen entradas y producen resultados basando la solución entre las órdenes y las variables que se van modificando. En el modelo funcional los resultados dependen de las entradas tal que el programa puede ser visto como un mapeo que funcionalmente implementa la relación entre las entradas y los resultados.

El modelo de programación *lógico* está basado en la idea de que el programa implementa una relación más que un mapeo. Se basa en la premisa de que programar con relaciones es más flexible que programar con funciones, debido a que las relaciones tratan de modo uniforme a los argumentos y a los resultados. Este modelo es de más alto nivel ya que las relaciones son más generales que los mapeos. Las relaciones también se conocen como predicados. Los predicados se especifican por medio de reglas que se escriben mediante los constructores del lenguaje. A estas reglas se les suele denominar cláusulas de Horn. Un hecho es un caso especial de una regla que no indica ninguna condición. La programación lógica esta dirigida por consultas a las relaciones. Mediante estas relaciones podemos representar la información. Implementar un lenguaje de programación lógico consiste en construir un demostrador de teoremas que estaría basado en los hechos y las reglas de que consta el programa lógico fuente. Esta demostración se realiza mediante el uso de deducciones. El lenguaje más representativo es el Prolog diseñado en 1972.

De manera informal podemos concluir que los programadores proporcionan los hechos y las reglas mientras que el lenguaje usa la deducción para calcular respuestas a consultas. Kowalski en 1979 ilustra esto mediante la ecuación informal:

$$\text{Programa} = \text{Lógica} + \text{Control}$$

Donde el parámetro lógica se refiere a los hechos y las reglas que especifican lo que realiza el programa, en tanto que el control se refiere a cómo puede implementarse el algoritmo mediante la aplicación de reglas en un orden particular. Los programadores especifican la parte lógica mientras que el lenguaje proporciona el control.

Ejemplo 1.11 Pueblos del archipiélago Canario

En este ejemplo se construye una base de datos basada en hechos y se escriben las reglas que permiten averiguar si un determinado ítem es un pueblo y a que isla del archipiélago Canario pertenece. El código se escribe en el lenguaje Prolog.

En la figura 14.a se programa la base de datos de todos los pueblos que existen en el archipiélago. Esta base de datos está formada por un conjunto de reglas de tipo hecho que lista todos los pueblos (Tiscamanita, Hermigua, etc.). En la figura 14.b se presentan los hechos en los que se relacionan los pueblos y la isla a la que pertenecen. En la figura 14.c se presentan varias cláusulas de Horn mediante las cuales declaramos cuales son los pueblos

```
pueblo (Tiscamanita).
pueblo (Hermigua).
pueblo (Tejeda).
pueblo (Frontera).
pueblo (Teseguite).
pueblo (Garafía).
pueblo (Taganana).
...
```

(a)

```
esde (Fuerteventura, Tiscamanita).
esde (Gomera, Hermigua).
esde (Gran_Canaria, Tejeda).
esde (Hierro, Frontera).
esde (Lanzarote, Teseguite).
esde (La_Palma, Garafía).
esde (Tenerife, Taganana).
...
```

(b)

```
pueblos_provLP (B) :- pueblo(B), esde (Gran_Canaria, B).
pueblos_provLP (B) :- pueblo(B), esde (Fuerteventura, B).
pueblos_provLP (B) :- pueblo(B), esde (Lanzarote, B).
```

(c)

```
pueblo (roska) ?
esde (Tenerife, X) ?
pueblos_provLP (P) ?
```

(d)

Figura 1.14. a) Base de datos de pueblos. b) Base de datos de pueblos relacionados con sus Islas. c) Pueblos que pertenecen a la Provincia de Las Palmas. d) Interogación de la base de datos y relaciones.

que están adscritos a la Provincia de Las Palmas. En la figura 14.d se presentan algunos ejemplos de interrogaciones de la base de datos: 1) Mediante la pregunta pueblo (rosca) ? podemos averiguar si el pueblo rosca es un pueblo del archipiélago. Para ello el motor de inferencia del Prolog (control) compara el argumento rosca con cada uno de los argumentos del predicado pueblo. Si encuentra este argumento nos dará una respuesta afirmativa (yes), en caso contrario la respuesta será negativa (no). 2) Otra forma de preguntar consiste en hacer que nos devuelva una variable (X) con todos los pueblos que pertenecen a la Isla de Tenerife (esde (Tenerife, X) ?). En este caso procederá a comparar el argumento Tenerife con todos los predicados esde y ejemplarizará la variable X con todos los pueblos en los que la comparación produjese un resultado afirmativo (Taganana, ..., en nuestro ejemplo). 3) Para averiguar todos los pueblos de la Provincia de las Palmas preguntamos: pueblos_provLP (P) ?. En este caso, sabemos que un pueblo es de esta provincia si P es un Pueblo y además pertenece a la isla de Gran Canaria o a Lanzarote o a Fuerteventura (que queda expresado en el teorema de la figura 14.c). La búsqueda realizada por el Prolog consiste en buscar los pueblos que pertenezcan a cualquiera de estas tres Islas.

❐

Este modelo de programación se suele utilizar para implementar aplicaciones específicas como la construcción de bases de datos deductivas, sistemas expertos, etc., y no goza de

mucha popularidad debido a su ineficiencia al existir un "salto" semántico muy alto entre el lenguaje máquina (al que se traduce el programa escrito en lenguaje de alto nivel) y los lenguajes de este modelo; y que es difícil de utilizar.

Ejemplo 1.12 Cálculo del factorial en Prolog

En este ejemplo mostramos el cálculo del factorial de un número entero positivo (n) escrito en Prolog.

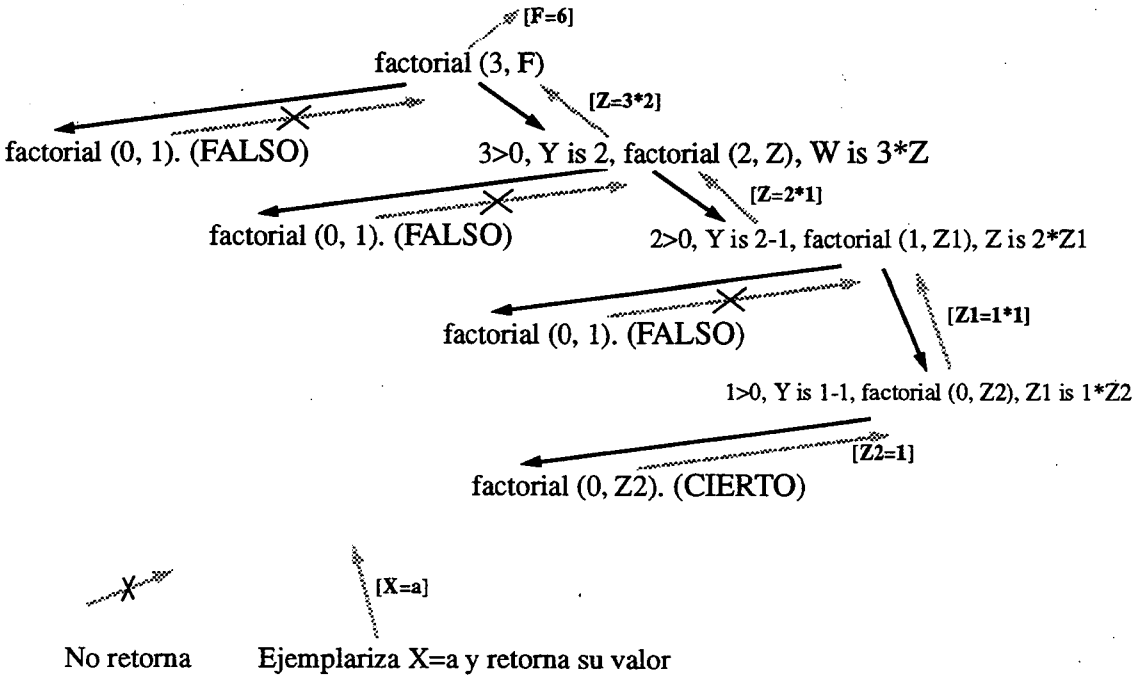
```
factorial (0, 1).                                    factorial (5, 9) ?
factorial (n, W) :- n>0, Y is X-1, fact (Y, Z), W is n*Z.   factorial (5, F) ?
```

(a) (b)

Figura 1.15. a) Relaciones para el cálculo del factorial en Prolog. b) Interrogación de la base reglas.

En la figura 1.15 se declara que el factorial de 0 y 1 es 1, y además el factorial de un número entero positivo (n>0), se puede calcular recursivamente multiplicando: $n \times (n-1) \times (n-2) \ldots 2 \times 1$. Nótese que podemos consultar si el factorial de un número n es factorial (n) (factorial (5, 9) ?), o bien calcular el factorial de un número y que se retorno el resultado en una variable (factorial (5, F) ?).

Supongamos que queremos obtener el factorial (3). Interrogamos factorial (3, F) ?. En la siguiente gráfica mostramos como trabaja el motor de inferencia del Prolog para obtener el resultado en la variable F:



Nótese que se hacen llamadas recursivas al predicado factorial (que tiene dos

definiciones: un hecho y un teorema). Si encuentra que uno de las definiciones es cierta entonces retorna el valor instanciado de las variables que forman parte de la definición del predicado; y no comprueba las otras definiciones. La variable Z se ha renombrado como Z1 y Z2 para distinguir que se trata de la misma variable en distintas llamadas recursivas.

□

1.2.1. El modelo de programación concurrente

Podemos definir la *programación concurrente* como el área de la ciencia de la computación que trata sobre las metodologías, lenguajes, técnicas y herramientas de programación necesarias para la construcción de programas reactivos. Los programas reactivos, frente a los programas transformacionales, interaccionan continuamente con el entorno intercambiando información; por el contrario, los programas transformacionales tienen el objetivo de producir un resultado final a partir de unos datos de entrada.

Este funcionamiento de los programas concurrentes o reactivos se diferencia de los programas imperativos tradicionales en que en estos últimos un programa consta de una serie de instrucciones que son ejecutadas secuencialmente. Un *programa concurrente* se puede entender como un conjunto de programas secuenciales ordinarios que son ejecutados en lo que llamaremos *concurrencia abstracta* (abstraemos un modelo independiente de la máquina). Utilizaremos el término *proceso* para referirnos a la ejecución de cada uno de estos programas. La concurrencia es abstracta porque no se requiere que exista un procesador físico separado para cada proceso. Incluso si el programa es ejecutado compartiendo un solo procesador, la programación concurrente nos puede facilitar el entendimiento de determinados problemas suponiendo que el programa está compuesto por un conjunto de procesos.

Los programas transformacionales también se codifican mediante la concurrencia abstracta sincronizando y comunicando procesos que pueden ejecutar subconjuntos o cálculos del programa original.

Ejemplo 1.13 Especificación concurrente abstracta del cálculo del factorial

Para el problema del cálculo del factorial especificamos tres funciones (cuya ejecución serán tres procesos), que calculen los siguientes productos: $N1 = (n-1) \times (n-2) \times \ldots \times (n-k)$, $N2 = (n-k-1) \times (n-k-2) \times \ldots \times (n-l)$ y $N3 = (n-l-1) \times (n-l-2) \times \ldots \times 2 \times 1$. Donde n>k>l respectivamente. Finalmente calculamos $factorial\,(l) = N1 \times N2 \times N3$ en cualquiera de los procesos anteriores o en otro proceso adicional (función).Este último proceso debe conocer los valores calculados previamente de N1, N2 y N3. Además estos valores se pueden calcular al mismo tiempo.

Intuitivamente se puede hacer notar que se necesita comunicar datos desde los tres primeros procesos al proceso que calcula el factorial y para ello estos procesos deben sincronizar la realización de estos cálculos.

□

A continuación analizamos una definición de procesamiento concurrente y una breve introducción histórica a la programación concurrente.

El procesamiento concurrente es una forma eficaz de tratamiento de la información que favorece la explotación de los sucesos concurrentes en el proceso de computación. La *concurrencia* implica [HBr88]:

1) Paralelismo,

2) Simultaneidad y

3) Solapamiento.

Los *sucesos paralelos* son los que pueden producirse en diferentes recursos durante el mismo intervalo de tiempo. Los *sucesos simultáneos* son los que pueden producirse el mismo instante de tiempo. Los *sucesos solapados* son los que pueden producirse en intervalos de tiempo superpuestos. Estos sucesos concurrentes pueden darse en un sistema computador en varios niveles de procesamiento. El procesamiento paralelo exige la ejecución concurrente en el computador de muchos programas y es un medio coste-efectivo para mejorar el rendimiento del sistema mediante la realización de actividades en el computador.

En el ejemplo 1.13 los productos que se calculan en las variables N1, N2 y N3 podrían efectuarse en paralelo (diferentes procesadores). Simultáneamente (en uno o varios procesadores diferentes). Solapadamente (en un mismo procesador ejecutando varias fases de la ejecución de una instrucción en intervalos superpuestos. En la figura 1.16 se muestra una gráfica de la evolución en el tiempo de estos tipos de sucesos.

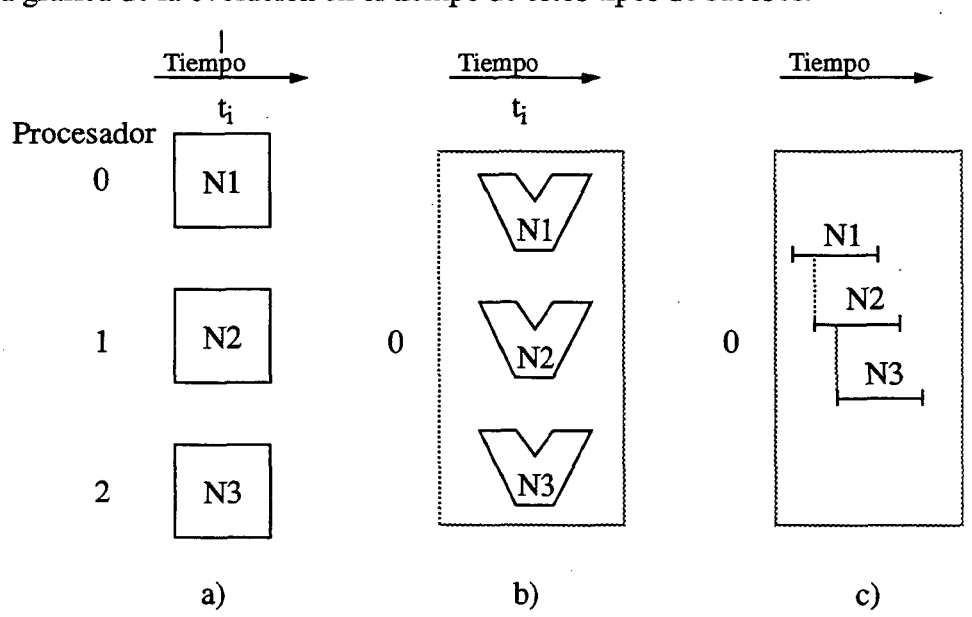


Figura 1.16. Evolución en el tiempo de las posibles formas de calcular N1, N2 y N3. a) En paralelo. b) Simultaneamente. c) Superpuestamente.

Históricamente la concurrencia fue introducida en los sistemas de computación para mejorar el rendimiento [Wat90]. Los primeros computadores eran máquinas secuenciales

que procesaban datos (un bit por cada unidad de tiempo). Una mejora que se introdujo fue el procesamiento de datos almacenados en palabras de 32 bits al mismo tiempo. Mejoras adicionales en el rendimiento supusieron el efectuar las operaciones de Entrada/Salida y las operaciones de Cálculo al mismo tiempo (en paralelo). Esto introdujo rápidamente la programación concurrente. Para explotar esto, se idearon esquemas de buffereado (buffering) con el ánimo de desacoplar el progreso de un cálculo tan pronto como fuera posible a las restricciones temporales impuestas por la necesidad de sincronizar la CPU (Unidad Central de Proceso) con los dispositivos de Entrada/Salida [Dic87]. Se introdujeron los sucesos paralelos (varios bits procesados al mismo tiempo), simultáneos (Entrada/Salida y CPU trabajan al mismo tiempo). Este es el nivel más bajo en el que se introduce la concurrencia (a nivel de Hardware).

El incremento de la complejidad que suponía esta técnica de programación era muy grande como para que el programador pudiese encargarse de manejarla completamente y fue en este contexto en el que se introdujeron los sistemas operativos que manejaban eficientemente este solapamiento de operaciones. Por tanto los sistemas operativos son el arquetipo de los programas concurrentes. Los sistemas *multiprogramados* intentan utilizar los recursos que en otro caso estarían siendo utilizados no eficientemente, ejecutando varios trabajos concurrentemente. En los sistemas modernos orientados a ventanas, los sistemas operativos que manejen eficientemente el acceso multiple de varios programas de uno o varios usuarios, a los recursos de computación son fundamentales. Estos sistemas operativos se organizan de forma natural como un conjunto de procesos que se ejecutan de forma concurrente. Usualmente estos procesos se ejecutaban en un procesador *intercalando* la ejecución de los distintos procesos. En este nivel se aprovechan sucesos que ocurren a nivel de Sistema Operativo y del Hardware.

Aunque en un principio la evolución de la programación concurrente se vio muy ligada a la evolución de los sistemas operativos, conforme han ido apareciendo nuevos sistemas de computación más complejos ha sido necesario emplear técnicas de programación concurrente que permitan aprovechar mejor estos sistemas con un bajo coste y con alto rendimiento. Con este sistema se pueden explotar eficientemente los sucesos paralelos (varias CPUs).

A continuación presentamos algunos de estos sistemas y las ideas básicas de su programación.

1.2.2. Sistemas inherentemente concurrentes

En [Pre93] se define *sistema* basado en computadores como un conjunto u ordenación de elementos organizados para llevar a cabo algún método, procedimiento o control mediante el procesamiento de información. Existen sistemas en los que se producen sucesos concurrentes de forma "natural". A modo de ejemplo presentamos tres tipos de sistemas: a) los sistemas distribuidos, b) los sistemas de tiempo real y c) los sistemas tolerantes a fallos.

a) Sistemas distribuidos

Un sistema distribuido es una colección autónoma de computadores unidos por una red de comunicación, con software diseñado para producir un método de computación integrada

[CDK94].

En la figura 1.17 se muestran los componentes de un sistema distribuido en el que éstos están conectados mediante una red de área local.

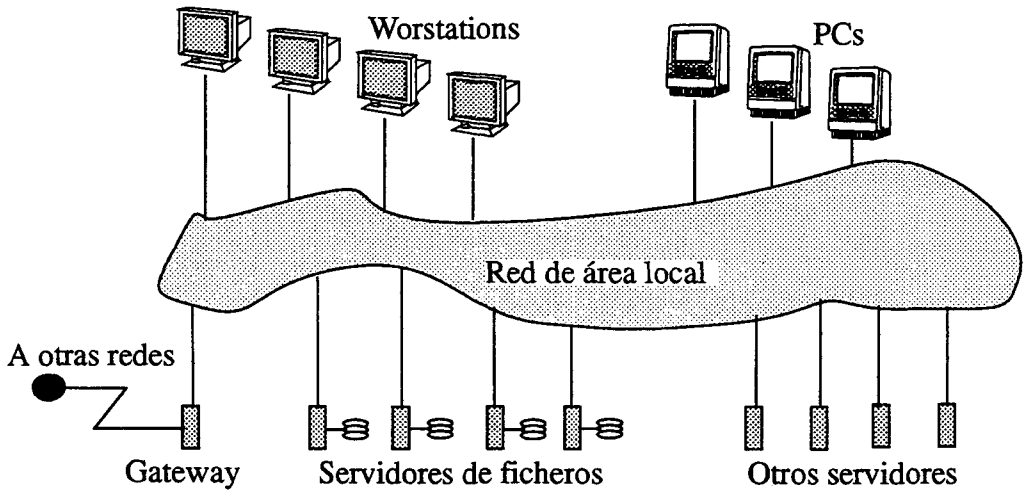


Figura 1.17. Componentes de un sistema distribuido conectado en una red de área local

Este ejemplo de sistema provisto del software necesario puede soportar las necesidades de computo de un conjunto de usuarios cumpliendo un papel similar al de un sistema multiusuario centralizado. Sin embargo; este tipo de sistemas introduce mejoras en rendimiento, fiabilidad, escalabilidad y soportan aplicaciones que implican comunicaciones de datos y de señales.

Cada usuario trabaja con un computador para ejecutar programas. Existen *servidores* que proporcionan el acceso a los recursos compartidos (discos, impresoras, CPUs, etc.). Los programas que se ejecutan en los computadores de los usuarios actúan como *clientes* de los servidores accediendo a los recursos que le han pedido a los servidores. Por ejemplo, a menudo se suelen incluir varios servidores de ficheros, incluso en pequeños sistemas distribuidos debido al frecuente acceso a los ficheros por parte de los clientes. A este esquema de trabajo o cooperación entre los distintos programas que se ejecutan en las diferentes máquinas (procesos), se le denomina *modelo Cliente-Servidor*, y es la base de muchos de los sistemas distribuidos diseñados.

Claramente se puede intuir la cantidad de sucesos que ocurren en estos sistemas y la gran cantidad de procesos reactivos que se pueden estar ejecutando en un momento dado. Estos procesos pueden estar o no colaborando para solucionar un determinado problema computacional. En este sistema se nota claramente la necesidad de disponer de técnicas de programación concurrente para solucionar eficientemente los problemas computacionales (varios programas secuenciales ejecutándose concurrentemente).

a) Sistemas de tiempo real

Existen muchas interpretaciones del término exacto sistema de tiempo real, sin embargo; todas esas interpretaciones tienen en común la noción de tiempo de respuesta (tiempo que

tarda el sistema en dar una respuesta frente a una entrada asociada con esa respuesta). En [BWe90] se presentan dos definiciones que según estos autores cubren un amplio rango de sistemas que se podrían adaptar a esas definiciones. La primera definición (según el diccionario de Computación de Oxford) es: Un sistema de tiempo real es cualquier sistema en el que el tiempo en el que se produce es significativo. Esto es debido a que la entrada se corresponde con algún cambio del mundo físico y la salida tiene que relacionarse con ese cambio. El tiempo que transcurre entre que se produce la entrada y la salida debe ser suficientemente pequeño como para que sea aceptable. Young en 1982 estableció la siguiente definición: Un sistema de tiempo real es un sistema de procesamiento de información que tiene que responder a estímulos de entrada generados externamente dentro de un periodo de tiempo finito y especificado.

En la figura 1.18 se muestra un sistema de control de procesos distribuido que debe dar respuestas en tiempo real.

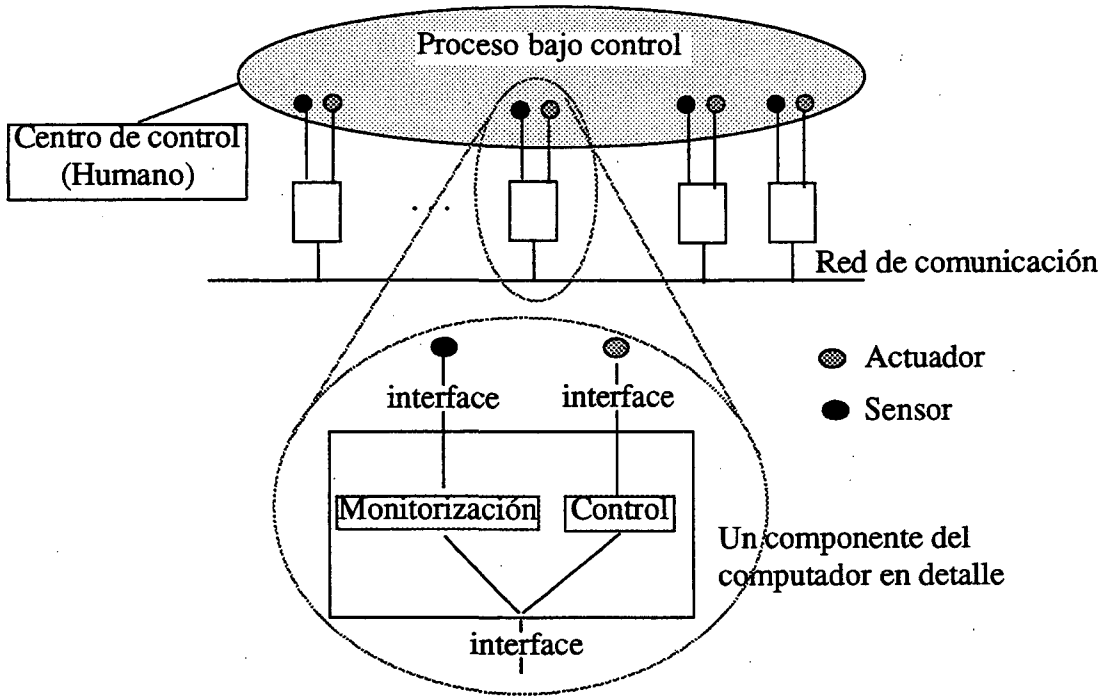


Figura 1.18. Sistema de control de procesos distribuido

En este ejemplo se trata de recoger una gran cantidad de datos de temperatura o presión e implicar varios componentes de un computador (que pueden ser procesadores especializados) para que realicen un proceso de refinamiento y control de la seguridad del sistema a controlar. Los datos se recogen cada cierto periodo de tiempo y se debe dar una respuesta del estado del sistema en intervalos de tiempo pequeños (del orden de milisegundos o microsegundos). Otra posibilidad podría ser que los datos se recogiesen a intervalos de tiempo no periódicos. En cualquier caso las tareas a realizar por los componentes son de dos tipos: a) monitorización que implica la recogida de datos, y el b) control que implica el afinado regular o irregular de datos por si se debe poner en marcha un sistema de alarma. Es importante que si ocurre una situación de alarma entonces el sistema

reaccione inmediatamente.

En este tipo de sistemas es importante emplear técnicas de programación concurrente debido a que existen muchos pequeños sistemas de computación para controlar una planta industrial relativamente grande. Cada uno de estos sistemas de computación puede tener tareas propias de control y juntos pueden colaborar en el control total de la planta: cada uno puede estar ejecutando su propio programa reactivo e intercambiar datos con los otros sistemas.

a) Sistemas tolerantes a fallos

Un sistema de proceso de información es tolerante a fallos si procesa la información satisfactoriamente en presencia de fallas en el sistema. En este contexto, las fallas son desviaciones del funcionamiento correcto del sistema que pueden ser causadas por: anomalías en los dispositivos, errores de diseño, o incluso errores debidos a los operadores del sistema [Ren84].

Existen varios métodos para proporcionar servicios tolerantes a fallos. En general, un servicio tolerante a fallos puede ocasionalmente fallar temporalmente, pero se diseña para recuperarlo después de un fallo sin pérdida de datos. En un sistema distribuido, la mayoría de las aplicaciones tolerantes a fallos pueden estar basadas en transacciones. Los servicios que están relacionadas con transacciones pueden fallar ocasionalmente seguidas de un procedimiento de recuperación relativamente largo. En los sistemas de tiempo real, las aplicaciones basadas en control de procesos tolerantes a fallos se caracterizan por tener que recuperar el fallo en un intervalo de tiempo relativamente pequeño.

Una forma natural y directa de proporcionar tolerancia a fallos es replicar los componentes de un sistema. Por ejemplo, en un sistema distribuido se pueden replicar los nodos servidores de ficheros teniendo varias copias de un mismo fichero en varios discos y actualizando simultaneamente las diferentes copias. En estos casos, la forma en la que se puede solucionar un error es por medio de un sistema de votación: cada componente replicado emite un diagnóstico (voto), que representa el buen funcionamiento de la o las operaciones que ha efectuado y a continuación estos componentes se deben poner de acuerdo en cual es el que ha fallado al operar.

En la figura 1.18 se presenta un sistema distribuido tolerante a fallos.

Un grupo de nodos especiales donde se procesa la información se conectan a través de nodos de conmutación que tienen una estructura similar a un bus redundante en trípode pero que pueden hacer de conmutadores de circuitos para proporcionar tolerancia a fallos y reencaminar los datos por diferentes caminos físicos. Estos nodos especiales pueden: estar compuestos de varios procesadores (1), ser un único procesador (2) o incluso ser un procesador tolerante a fallos (3). Cada nodo tiene su reloj local que sincroniza a los procesadores del nodo; los relojes de los distintos nodos no están sincronizados. El hardware de votación se implementa a través del sistema. En los nodos en los que se dispone de varios procesadores el sistema de votación es directo porque los procesadores están sincronizados por un reloj y ejecutan programas idénticos. El voto emitido (por triplicado), entre los nodos con distinto reloj requiere operaciones de sincronización,

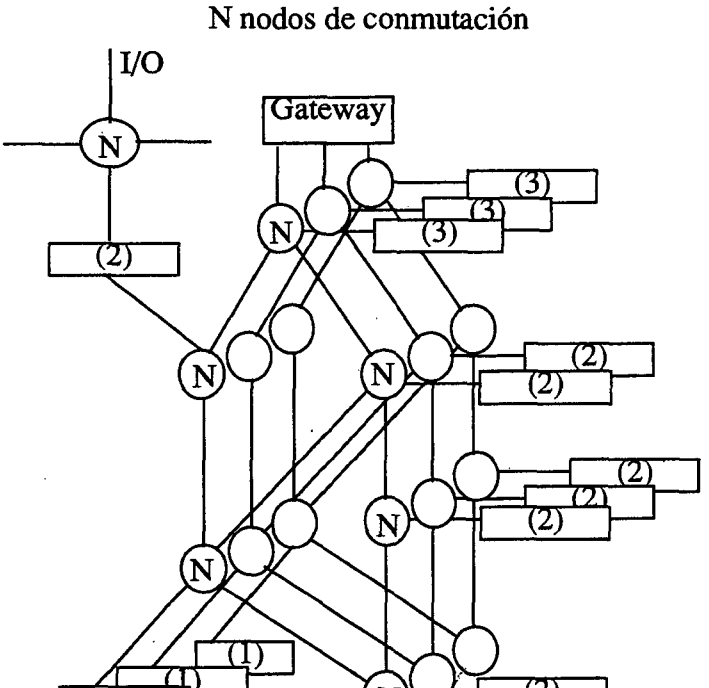


Figura 1.19. Sistema distribuido tolerante a fallos

aunque las derivas que pueden existir en los datos (debido a que los relojes son diferentes), puede ser solucionado eficientemente por el hardware. En los sistemas reales, no todas las aplicaciones son lo suficientemente importantes como para requerir que se duplique todo el sistema, reduciendo bastante la complejidad final del sistema de votación y del sistema.

En los sistemas tolerantes a fallos en los que existen varias máquinas y aplicaciones ejecutándose existe la necesidad de programar las aplicaciones mediante técnicas de tolerancia a fallos. En la práctica se han desarrollado muchas técnicas de programación concurrentes para diseñar aplicaciones tolerantes a fallos y es un área que está en constante evolución por la importancia crucial que implica el disponer de sistemas tolerantes a fallos.

1.2.3. Aplicaciones potencialmente concurrentes

En este apartado presentamos algunas aplicaciones en las que se puede explotar eficientemente los posibles sucesos concurrentes, haciendo uso de la programación concurrente, para obtener tiempos de ejecución pequeños con un coste reducido. Algunos motivos para usar la programación concurrente para la implementación de estas aplicaciones son:

- Que tienen una gran cantidad de datos para ser procesados.
- Que existen requerimientos de tiempo real para obtener el resultado de la aplicación.
- Que existe hardware disponible para ejecutar concurrentemente la aplicación.

- Que tienen una gran cantidad de cálculo a realizar.

Algunas de las áreas a las que pertenecen estas aplicaciones son:

- Ingeniería del Software [Gom93]: compilación de programas con un gran número de módulos.
- Simulación de procesos físicos y visualización gráfica en tiempo real: "raytracing", "rendering", etc.
- Inteligencia artificial: reconocimiento de voz e imágenes [SuF95], sistemas expertos, procesamiento del lenguaje natural, matemática, etc.
- Matemática, física, biología: álgebra lineal y diferencial, elementos finitos, matemática discreta, métodos de optimización, etc.

En los últimos años se ha trabajado mucho en las aplicaciones que requieren gran cantidad de cálculos para procesar una gran cantidad de datos, utilizando arquitecturas de computadores en las que se utilizan varios procesadores. En el siguiente apartado presentamos la evolución de la arquitectura de computadores secuencial hacia estas arquitecturas.

Tema 2. Arquitectura de Computadores

Se presentan los conceptos básicos de Arquitectura de Computadores secuencial aplicando una definición de Arquitectura de Computadores. Mediante un ejemplo sencillo se demuestra la ineficiencia de esta arquitectura y se presentan innovaciones arquitectónicas clásicas que son apropiadas para la ejecución de programas que requieren un tiempo de ejecución pequeño. Finalmente se presentan varias aplicaciones reales de estas últimas arquitecturas.

2.1. Introducción

Existen varias definiciones del concepto Arquitectura de un Computadores. Al parecer, el término de arquitectura de un computador fue acuñado por IBM en 1964 en el diseño del IBM 360. Definían arquitectura como la estructura de un computador que debe conocer un programador en lenguaje máquina para poder escribir programas en esa máquina.

Esta definición se podía adaptar a la visión clásica del computador en la que se distinguían dos niveles marcadamente distintos: hardware y software. Esta visión se mantuvo durante el desarrollo de los primeros computadores, debido a la reducida complejidad de los mismos. Sin embargo, la creciente complejidad de los computadores ha obligado a organizar su estudio de forma estructurada. Esta tendencia nos ha llevado a ver el computador como una jerarquía de niveles interrelacionados.

Cada uno de los niveles constituye una máquina virtual en la que se omiten los detalles pertenecientes a los niveles inferiores. Cada nivel ofrece su propio lenguaje mediante el cual el usuario (usuarios especializados en el caso de algunos de los niveles) puede especificar el algoritmo que debe ser ejecutado por el sistema. En cualquier caso, las prestaciones de un nivel están soportadas por los niveles inferiores.

Por tanto, cualquier operación especificada en un determinado nivel, debe ser reformulada utilizando los elementos propios del nivel inferior en la jerarquía. Esta reformulación puede realizarse según dos estrategias diferentes: traducción o interpretación. Por traducción se entiende el cambio de representación de un programa especificado en un nivel i a su representación funcionalmente idéntica de un nivel inferior. Igualmente, diremos

que un nivel i interpreta a otro nivel superior j, cuando el nivel i ejecuta directamente los programas codificados en el lenguaje del nivel j.

Una posible descomposición en niveles es la propuesta en [Tan90][Fer94]:

1) *Circuitos digitales.* Este nivel está constituido por los circuitos digitales que componen el computador. En este nivel se estudian los componentes básicos de diseño lógico (puertas, biestables, ...) y las técnicas de diseño de sistemas combinatorios y secuenciales. Con estos componentes básicos se construye el computador. Lógicamente, este es el nivel que, en última instancia ejecuta el programa del usuario.

2) *Microprogramación.* El objetivo principal de este nivel es el diseño de la unidad de control del procesador. Este nivel no siempre está presente en el computador. Muchos procesadores son suficientemente sencillos como para que el control pueda ser cableado. Sin embargo, cuando la complejidad del procesador es elevada, la introducción del nivel de microprogramación facilita el diseño de la unidad de control, a expensas de una pérdida de velocidad en la ejecución de los programas.

El control microprogramado permite modificar la visión del nivel superior (lenguaje máquina) para una misma realización digital. En general, basta con modificar el contenido de la unidad de control.

2) *Lenguaje máquina.* El lenguaje máquina es habitualmente el nivel más bajo desde el cual los usuarios tienen acceso a un computador. Constituye, en consecuencia, el primer nivel de programación. Habitualmente los usuarios no trabajan directamente en lenguaje máquina, sino con lenguaje ensamblador que es una representación simbólica directa del mismo.

Como lenguaje de programación, el lenguaje máquina dispone de elementos para guardar datos, de instrucciones para operar con ellos y mecanismos para indicar en que orden se han de ejecutar las instrucciones. Dado que el lenguaje máquina es directamente interpretado por el computador, estos mecanismos han de ser lo suficientemente sencillos para que esta interpretación se realice eficientemente.

3) *Sistema operativo.* Este nivel aparece a raíz de la necesidad de gestionar y proteger los recursos del computador, especialmente cuando el computador está compartido por varios usuarios. También facilita el uso del computador escondiendo al usuario la dificultad de su utilización desde el nivel inferior.

4) *Lenguajes de alto nivel.* El objetivo de este nivel es facilitar la utilización del computador. A este nivel, el usuario del computador puede comunicarse con el mismo utilizando un lenguaje cómodo, cercano al utilizado en la descripción de los algoritmos que el usuario utilice. De esta forma, el usuario puede desentenderse de detalles tediosos y que son resueltos desde niveles inferiores.

Actualmente existen numerosos lenguajes de programación de alto nivel. Algunos pensados para codificar algún tipo especial de aplicaciones, otros de tipo más general. En cualquier caso, estos lenguajes de alto nivel pueden adaptarse completamente a un modelo de programación o aprovechar distintos elementos de varios modelos con el fin de permitir una abstracción más poderosa de la programación de alto nivel. Habitualmente este nivel

está soportado mediante un proceso de traducción al nivel Lenguaje Máquina. Durante esta traducción, además se añade el código necesario para adaptar los servicios ofrecidos por el sistema operativo a los definidos en el lenguaje (entradas/salidas, gestión de memoria, solapamiento de la Entrada/Salida con los Cálculos, etc.).

5) *Usuario y aplicación.* Este es el nivel superior de la jerarquía. En este nivel, el usuario se comunica con el sistema a través de aplicaciones ya programadas. El objetivo de este nivel es crear un entorno agradable y eficaz al usuario del computador.

En la figura 2.1. se muestra un gráfico de estos niveles.

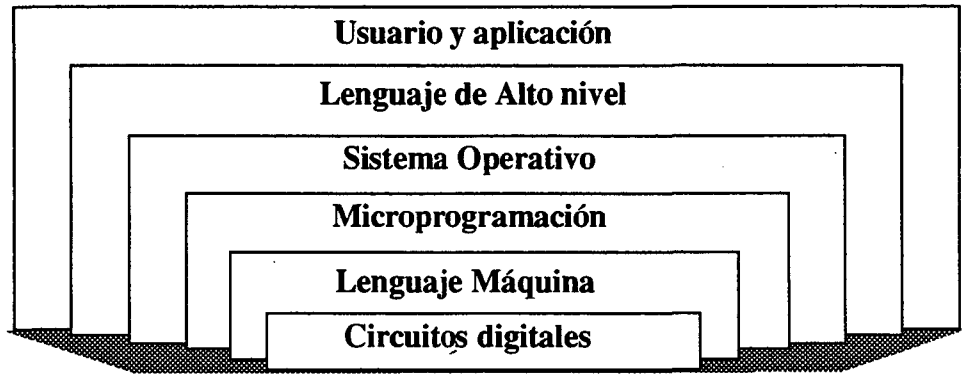


Figura 2.1. Jerarquía de niveles de los computadores modernos

Teniendo en cuenta estos niveles podemos emitir una nueva definición, más apropiada, de Arquitectura de Computadores:

Arquitectura de Computadores es la visión que tiene del computador el programador de lenguaje máquina, el generador de código de los compiladores, y el programador del núcleo del sistema operativo, es decir, el estado del computador, el formato y operación de las instrucciones, los métodos de direccionamiento, los formatos de los datos y la comunicación con los sistemas de Entrada/Salida.

La primera definición y ésta que acabamos de enunciar están hechas desde el punto de vista funcional. Otra definición (o apunte adicional), es: la Arquitectura de Computadores engloba la estructura, organización, realización y evaluación. *Estructura* es la interconexión de los distintos componentes del nivel de circuitos digitales. *Organización* es la interacción dinámica y gestión de los componentes. *Realización* es el diseño de bloques específicos. *Evaluación* es la medida de las prestaciones del sistema.

Esta última definición da una idea de la arquitectura estructural. Se puede entender el término arquitectura como la suma de estas dos últimas definiciones.

Nosotros hemos definido la arquitectura de un computador atendiendo al modelo de niveles presentado. Haciendo una extrapolación del término de arquitectura podríamos estudiar la arquitectura de un nivel determinado. Por ejemplo podríamos estudiar la arquitectura del nivel de sistema operativo: su estructura (interconexión de los distintos componentes software o programas que componen el sistema operativo), organización (interacción dinámica entre los programas del sistema operativo), realización (diseño de

estos programas), evaluación (medidas de las prestaciones del sistema operativo). A esto tendríamos que añadir la visión del usuario o programador del sistema operativo. Por otro lado, la arquitectura del nivel de lenguaje máquina está descrita en el manual de lenguaje máquina. En él se describe todo lo que es necesario saber para comunicarse con el sistema desde este nivel.

2.2. Arquitectura Von Neumann

La descripción de los conceptos básicos sobre Arquitectura de Computadores se enmarcan dentro del modelo Von Neumann presentado en el año 1946. Este modelo ha constituido una línea tradicional en el diseño de computadores que se ha mantenido vigente hasta nuestros días.

La estructura básica de una máquina Von Neumann se muestra en la figura 2.2. Las líneas continuas corresponden a caminos por los que se transmiten datos y las líneas discontinuas corresponden a caminos por los que se transmiten señales de control.

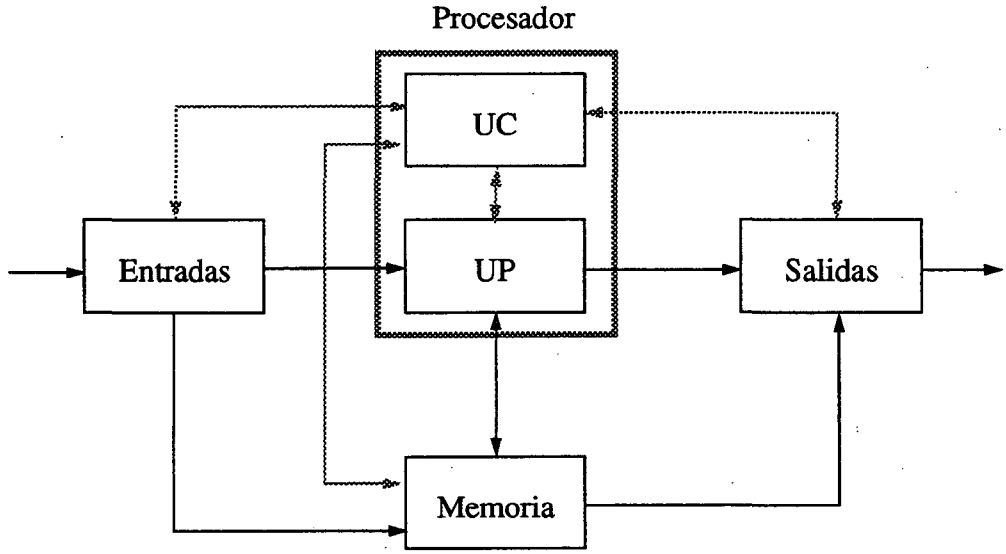


Figura 2.2. Estructura de la máquina Von Neumann

En la estructura básica de una máquina Von Neumann se distinguen tres unidades básicas:

- **Memoria.** Es la unidad encargada de almacenar instrucciones y datos.

  Desde los primeros computadores la comunicación entre el procesador y la memoria ha sido el cuello de botella del modelo Von Neumann. El problema a resolver es cómo diseñar el sistema de memoria que pueda suministrar y recibir datos a la velocidad requerida por el procesador a un coste razonable.

- **Procesador.** Es la unidad encargada de la ejecución de instrucciones. A su vez se descompone en dos subunidades: a) Unidad de Proceso (UP) o camino de datos, que contiene las unidades de cálculo, registros y los caminos de conexión entre estos elementos, y b) Unidad de Control (UC), que es la encargada de interpretar las instrucciones generando las señales de control para coordinar el funcionamiento

del resto de unidades del computador.

El objetivo fundamental de toda arquitectura es dar soporte a los programas que debe ejecutar. Normalmente estos programas están codificados en un lenguaje de alto nivel, existiendo un desnivel semántico entre éste y el lenguaje máquina que ofrece la arquitectura.

En el diseño de un procesador se ha de considerar: a) qué tipo de datos ha de soportar directamente, b) qué repertorio de instrucciones ha de ser capaz de ejecutar, c) qué modos de direccionamiento o algoritmos de cálculo de la dirección efectiva de memoria ha de disponer, y d) cómo se realiza el control de secuenciamiento de las instrucciones.

Por otro lado, podemos ver el procesador como un bloque secuencial complejo. De este bloque secuencial, la unidad de control es la parte más compleja. Existen dos técnicas para diseñar unidades de control:

- Cableada. El diseño se hace a partir de sistemas secuenciales a partir de grafos de estados.

- Microprogramada. Este diseño se basa en una memoria en la que están pregrabadas las señales de control y un secuenciador que determina el orden en que se generan.

- **Entradas y Salidas**. Su función es transferir información entre la memoria y el exterior. Este subsistema contiene los controladores que permiten gobernar los dispositivos periféricos. Un elemento importante a tener en cuenta es el diseño de los controladores (generalmente en lenguaje máquina), para hacer efectivo el mecanismo de comunicación entre el procesador y el exterior.

Las características más relevantes del modelo Von Neumann a nivel de arquitectura son las siguientes:

- Organización lineal de las palabras de memoria, de tamaño fijo y dentro de un espacio de direcciones unidimensional. No existe distinción explícita entre instrucciones y datos almacenados en memoria.
- El procesador es responsable de ejecutar una a una las instrucciones del programa. Para ello, debe leer la instrucción de memoria, determinar el tipo de instrucción a realizar, localizar los operandos, calcular el resultado y almacenarlo.
- El secuenciamiento de las instrucciones es implícito y viene determinado por el orden en que han sido almacenadas en memoria. Dicho secuenciamiento sólo puede ser modificado por la ejecución de instrucciones de ruptura de secuencia.
- Dispone de un Lenguaje Máquina de bajo nivel, basado en operaciones simples sobre datos elementales.

Ejemplo 2.1 Multiplicación de una matriz por un vector

En este ejemplo presentamos la ejecución de un programa escrito en un lenguaje de alto

nivel imperativo secuencial para ser ejecutado en una arquitectura Von Neumann. Para entender mejor la traducción a lenguaje ensamblador presentaremos el mismo programa traducido a un lenguaje ensamblador ficticio que es lo suficientemente sencillo como para entender el objetivo de este ejemplo. Este objetivo es demostrar que la arquitectura Von Neumann no es eficiente para ciertas aplicaciones en las que se requiere un tiempo de ejecución pequeño.

Se trata de realizar la operación vectorial: $c = a \times b + c_o$.

Donde c es un vector resultado de m elementos, b es un vector de entrada de n elementos y a es una matriz de n filas y m columnas de elementos enteros, y c_o es un vector de n elementos que podrían ser 0.

El siguiente fragmento de la solución del problema en el lenguaje de alto nivel es un par de bucles imbricados:

```
do  i =1, n
    do j = 1, m
       c(i) = c(i) + a(i. j) * b(j)
    enddo
enddo
```

Este código especifica la ejecución secuencial del código: para cada fila (i=1..n) se realiza un producto escalar de todos los elementos de esa fila de a (j=1..m), por todos los elementos del vector b y el resultado se va almacenando en el vector c.

Para poder obtener el programa ensamblador (traduciendo el programa de alto nivel), haremos algunas suposiciones de simplicidad:

- La máquina posee, como mínimo, las instrucciones aritméticas de suma y multiplicación, y tardan el mismo tiempo en ejecutarse.
- La máquina tiene un juego de registros suficientemente grande. Todos son de propósito general, tienen longitud suficiente y un tiempo de acceso nulo.
- La máquina tiene instrucciones de incremento y decremento unitario de los registros. Estas operaciones afectan al flag de cero.
- Inicialmente se encuentran cargados en memoria el vector b, la matriz a y el valor inicial que puede dársele al vector resultado, si se va a usar éste como acumulador (si no fuese así, el vector resultado estará inicializado a cero). El vector b está almacenado (en posiciones consecutivas) a partir de la posición de memoria INIVEC, la matriz a está almacenada por filas a partir de la dirección de memoria INIMAT. El vector c se almacenará a partir de la dirección de memoria INIRES.
- El valor de n se almacena dos posiciones antes de la matriz y el valor m en la posición anterior a ésta.

El programa traducido a lenguaje ensamblador se muestra en la figura 2.3. Además para

cada instrucción del lenguaje ensamblador se expresa el tiempo de ejecución (muy simplificado) para el que se ha considerado los dos siguientes términos: a) t_r. Tiempo de acceso a la memoria, y b) t_c. Tiempo de cálculo del procesador.

```
                                                  Tiempo de ejecución
        LD    R4, (@INIMAT-2)     ; valor de n      2tr
        LD    R1, @INIMAT         ; Dir. In. de a   2tr
        LD    R3, @INIRES         ; Dir. Inicio c   2tr
BUCFIL: LD    R2, @INIVEC         ; Dir. Inicio b   2tr
        LD    R5, (@INIMAT-1)     ; valor de m      2tr
        LD    R6, (R3)                              2tr
BUCCOL: LD    R8, (R1)                              2tr
        LD    R7, (R2)                              2tr
        MUL   R7, R8 ,R9                            tr+tc
        ADD   R6, R9, R6                            tr+tc
        INC   R1                                    tr+tc
        INC   R2                                    tr+tc
        DEC   R5                                    tr+tc
        JNZ   BUCCOL                                tr+tc
        ST    (R3), R6                              2tr
        INC   R3                                    tr+tc
        DEC   R4                                    tr+tc
        JNZ   BUCFIL                                tr+tc
```

Figura 2.3. Programa ensamblador matriz por vector

Todas las instrucciones tardan, al menos t_r. Las instrucciones LD (Load) y ST (Store) tienen otro acceso a memoria para cargar o almacenar el dato (t_r).

Las instrucciones MUL (Multiplicación), ADD (Suma), DEC (Decremento), INC (Incremento) y JNZ (Salto de programa si los operandos son iguales), tendrán un ciclo de cálculo (t_c).

Teniendo en cuenta este modelo sencillo sobre la ejecución del programa, el valor del tiempo de ejecución es: $(t_r + t_c)\ (n\,(6m+3)) + 2t_r\,(n\,(2m+4))$.

Después de haber calculado el tiempo de ejecución en función del valor de n, m, t_r y t_c discutiremos el tiempo de ejecución en función de los valores de estos parámetros.

- Si $n=m=3*10^3$ (memoria de aproximadamente 9 Mbytes, 32 bits), t_r=60 nsg. y t_c=20 nseg. Obtenemos un Tiempo de ejecución de 6.48 seg.
- Si $n=m=10^3$ (memoria de aproximadamente 4 Mbytes, y números codificables en 32 bits). Con t_r=60 nsg. y t_c=20 nseg. Obtenemos un Tiempo de ejecución de 0.72 seg.

- Si $n=m=10^3$ (memoria de aproximadamente 4 Mbytes, 32 bits), t_r=3 msg. y t_c=20 nseg. Obtenemos un Tiempo de ejecución de **8.3 HORAS.**

□

Después de analizar el ejemplo anterior, queda claro que la Arquitectura Von Neumann no ejecuta rápidamente el producto de una matriz por un vector cuando el número de filas (columnas) de la matriz es grande. En la práctica existen muchos algoritmos de álgebra lineal en los que se deben realizar una gran cantidad de productos de matriz por vector (o cálculos muy semejantes). Un ejemplo bastante claro es el cálculo de un producto de matrices, cuyo código secuencial en lenguaje de alto nivel imperativo se presenta en la figura 2.4.

```
do  i =1, n
    do j = 1, m
       do k = 1, m
          c(i, j) = c(i, j) + a(i.k) * b(k, j)
    enddo
enddo
```

Figura 2.4. Programa ensamblador matriz por matriz (c = a * b)

En este producto de matrices se efectúan n (supongamos que n=m) productos de matriz por vector. ¡ Por lo tanto se multiplica el tiempo de ejecución anterior por un factor de n !.

Está claro que no podemos estar esperando a resultados de un producto de matrices durante horas. Por tanto es necesario estudiar la modificación de la arquitectura de Von Neumann para permitir que el problema analizado, y muchos otros, se ejecuten en un tiempo razonable. Este es sólo un ejemplo sencillo; pero en la práctica existen otros muchos tipos de aplicaciones que aconsejan esta modificación.

En el siguiente apartado introduciremos las ideas básicas de las modificaciones a esta arquitectura que permiten obtener arquitecturas de alta velocidad.

2.3. Evolución de la arquitectura

El aumento en las prestaciones a nivel de velocidad por parte de los computadores puede realizarse mejorando cualquiera de los niveles de la jerarquía. Así por ejemplo, es posible mejorar la velocidad de una aplicación utilizando algoritmos que requieran menos operaciones. En esta sección, sin embargo, estamos interesados en ver las soluciones arquitectónicas utilizadas para mejorar el rendimiento de los computadores basados en el modelo Von Neumann.

Las mejoras conseguidas respecto al rendimiento y velocidad, desde la aparición del primer computador digital en la década de los cuarenta, hasta los supercomputadores

actuales [Hbr88], se debe a diversos factores que pueden agruparse en:

- Tecnológicos. Los avances de la tecnología, tanto en materiales utilizados para la construcción de dispositivos de conmutación (Si, AsGa, ...), capacidad de integración (SSI, MSI, LSI, VLSI, ...) así como la capacidad de encapsulado de los dispositivos en circuitos integrados, están permitiendo reducir el tiempo de cálculo en funciones complejas. Es el diseñador quien, en función del estado de la tecnología, decide qué utilizar para obtener la velocidad de cálculo deseada junto con los parámetros de diseño como consumo, coste, ... Esta técnica permite disminuir el tiempo de cálculo (t_c) en el procesador y el tiempo de respuesta de la memoria (t_r), con el consiguiente ahorro del tiempo de ejecución (desde el punto de vista del nivel de circuitos digitales).
- Traspaso de funciones a niveles inferiores. Funciones utilizadas con mucha frecuencia son realizadas directamente en hardware. Por ejemplo, ciertas funciones de manejo de memoria podrían ser ejecutadas directamente por el hardware. Otro ejemplo sería implementar directamente en hardware el cálculo de la Transformada Rápida de Fourier (FFT) para el proceso de señales por computador.
- Crear nuevas arquitecturas que exploten eficientemente las características de los algoritmos a ejecutar. En muchos casos se suelen definir arquitecturas específicas que están especializadas en la ejecución de uno o varios algoritmos determinados. Un ejemplo de estas arquitecturas son los procesadores ASICs (Applied Specific Integrated Circuit), o los procesadores sistólicos.
- Adaptación del Hardware a diferentes tipos de problemas (programación del Hardware a nivel muy bajo). Se trata de construir máquinas que en tiempo real (microsegundos o incluso nanosegundos si fuera posible), adapten componentes del nivel de "circuitos digitales" a los problemas que requieren alta velocidad de ejecución. Un tipo de componente apropiado para adaptarse a la implementación de diferentes tipos de problemas a alta velocidad son las FPGAs (Full Programmable Gate Arrays). Mediante estos componentes se podrían construir máquinas que reconfiguran "el Hardware" para realizar diferentes tipos de operaciones eficientemente.
- Innovaciones arquitectónicas para mejorar cierto nivel. Como por ejemplo, la incorporación de una memoria cache, sistemas de gestión de memoria virtual o la aplicación de técnicas de ejecución concurrente.

De estas últimas innovaciones arquitectónicas nos centraremos en las que se obtienen aplicando técnicas de ejecución concurrente. A continuación se detallan las técnicas básicas de concurrencia:

a) segmentación y

b) replicación.

Así como los distintos niveles a los que pueden ser aplicadas.

2.3.1. Segmentación

La técnica de segmentación consiste en descomponer una determinada tarea (T) en n subtareas (T_i) a realizar en fases o etapas sucesivas en el tiempo y explotar la ocurrencia de sucesos superpuestos. De esta manera, una tarea se realiza a medida que la información involucrada en dicha tarea atraviesa las n etapas. La concurrencia se obtiene a base de ejecutar n subtareas de forma superpuesta o simultánea, aunque cada una de ellas, se encuentra en una etapa diferente.

De los resultados del ejemplo 1.13 podemos resumir las siguientes conclusiones directas: a) La arquitectura de Von Neumann no es apropiada para ciertos tipos de problemas que tienen una gran cantidad de cálculos a realizar. b) El Bus de memoria-procesador está francamente inutilizado. Cuando el procesador está calculando, la memoria está inactiva, y viceversa, cuando la memoria está accediendo a un dato para comunicarlo al procesador éste está inactivo. El resultado de todo esto es que el procesador está inactivo la mayor parte del tiempo. La técnica de segmentación permite solventar estos problemas de la arquitectura de Von Neumann.

La técnica de segmentación puede aplicarse a nivel de:

1) Ejecución de instrucciones (*procesadores escalares segmentados*). La técnica de concurrencia se aplica a nivel de lenguaje máquina o instrucciones. La ejecución de la instrucción se divide en fases, tales como búsqueda de la instrucción, decodificación, lectura de operandos, ejecución y escritura de resultados. En cada ciclo se inicia la ejecución de una nueva instrucción, a la vez que el resto de instrucciones ya iniciadas avanza una fase.

En estos procesadores los compiladores de los lenguajes de alto nivel traducen los programas a lenguaje ensamblador tal que éstos últimos programas se ejecutan haciendo uso eficiente de la segmentación a nivel de instrucciones.

En la actualidad no se construye ningún procesador que no haga un uso intensivo de la segmentación [HeP93]. Estamos pues ante una nueva introducción de la concurrencia a nivel de hardware; esta vez a nivel de ejecución dentro del procesador permitiendo la explotación de sucesos simultáneos o superpuestos.

2) Operaciones sobre datos (*procesadores vectoriales segmentados*). La característica más relevante de estas máquinas es la inclusión de instrucciones de alto contenido semántico (operaciones sobre vectores), los cuales especifican una misma operación sobre múltiples datos. Normalmente estas instrucciones son operaciones aritméticas sobre números en coma flotante o bien operaciones de acceso a memoria. Disponen de varias unidades funcionales segmentadas profundamente segmentadas. La velocidad de estas máquinas está limitada, en parte, por la cadencia a la que se pueden alimentar las unidades funcionales segmentadas con nuevos datos.

En estas máquinas, muchos problemas se pueden compilar a lenguaje ensamblador tal que se utilice eficientemente la segmentación a nivel de datos. A estos compiladores se les denomina compiladores vectorizantes. Sin embargo existen problemas computacionales en los que el usuario debe asistir al compilador vectorizando manualmente el código del programa de alto nivel o de nivel de lenguaje máquina.

Ejemplos de estas arquitecturas son el CRAY 1, el IBM 3090/VF, el CYBER 205, etc. [MiA1995].

En la figura 2.5 se muestra una gráfica de la segmentación a nivel de instrucciones y a nivel de datos.

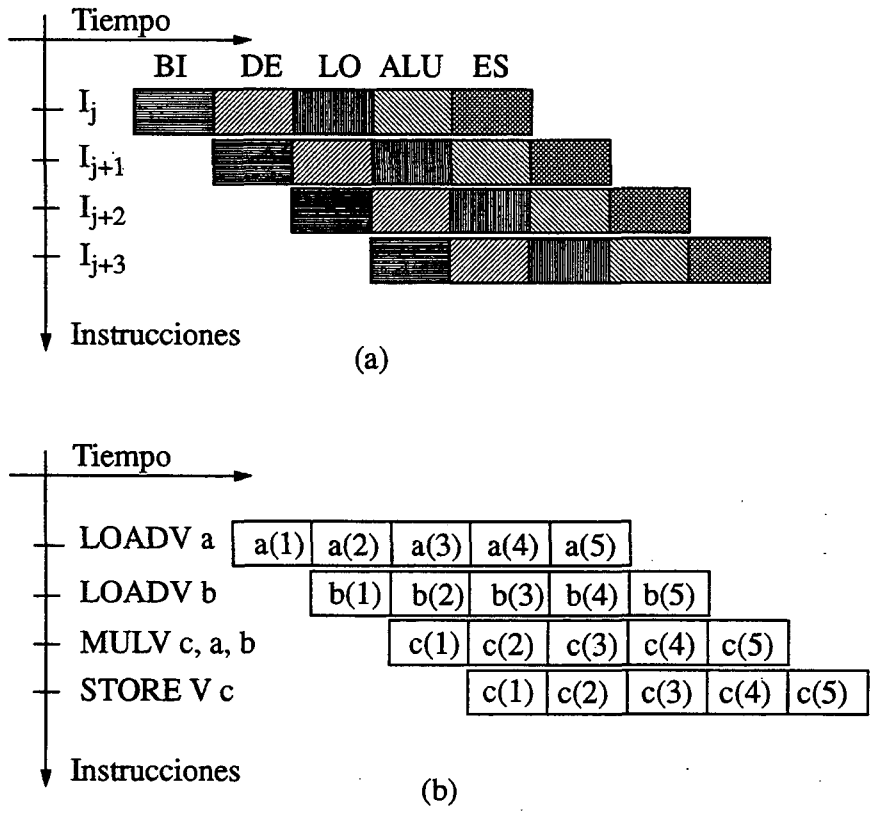


Figura 2.5. Técnica de segmentación: a) Instrucciones, b) datos

En la segmentación a nivel de instrucciones, en el caso que los operandos sean leídos de memoria (LO) y los resultados sean escritos en memoria (ES) es necesario disponer de dos buses memoria-procesador separados: uno para lectura y otro para escritura. Se ha de tener en cuenta a que en un instante de tiempo determinado pueden ocurrir sucesos en la fase de: ALU en la I_j, LO en la I_{j+1}, DE en la I_{j+2}, BI en la I_{j+3}. En la segmentación a nivel de datos hemos de disponer de tres buses memoria-procesador: dos de lectura y un bus de escritura en memoria (por ejemplo para escribir c(2) al mismo tiempo que leemos a(5) y b(4) simultaneamente). El efecto es que en cada ciclo leemos una pareja de elementos de los vectores a y b, calculamos un elemento del vector c y almacenamos otro elemento del vector c.

Nótese que en estos casos estamos utilizando eficientemente los buses de memoria y tanto el procesador como la memoria están continuamente trabajando. Esto conduce a un ahorro sustancial en el tiempo de ejecución de los problemas computacionales (haciendo intervenir los niveles de circuitos digitales y de lenguaje máquina).

2.3.2. Replicación

La otra técnica básica para realizar varias operaciones concurrentemente es replicar las unidades funcionales de manera que cada una de ellas trabaje con datos distintos en un instante determinado.

La replicación puede aplicarse a nivel de:

a) Operaciones sobre datos (*procesadores vectoriales matriciales*). Los procesadores vectoriales matriciales poseen una única unidad de control que genera las mismas señales de control que se distribuyen a todas las unidades funcionales. De esta manera, se realiza una misma operación sobre elementos distintos de una estructura de datos regular (vectores o matrices) que están distribuidos entre las unidades funcionales replicadas.

Estas máquinas están especialmente diseñadas para trabajar con aplicaciones numéricas muy regulares (algoritmos matriciales), o problemas sobre modelado de sistemas físicos en los que existe un paralelismo inherente muy grande. Sin embargo, no son muy adecuadas para el procesamiento de propósito general.

La programación de estas máquinas suele hacerse mediante lenguajes en los que se expresan las acciones secuencialmente y existen compiladores bastante optimizados que paralelizan el código fuente para aplicaciones que tratan estructuras de datos regulares. Sin embargo; en muchos casos se debe asistir manualmente al compilador paralelizante para generar un código paralelo eficiente.

En [HBr88] se puede encontrar la descripción de procesadores vectoriales matriciales históricos: ILLIAC IV, PEPE, STARAN, MPP, etc.

En la figura 2.6 se muestra un diagrama de ejecución de instrucciones en este tipo de arquitecturas. Nótese que de forma ideal podemos acceder en un sólo ciclo y bajo el control

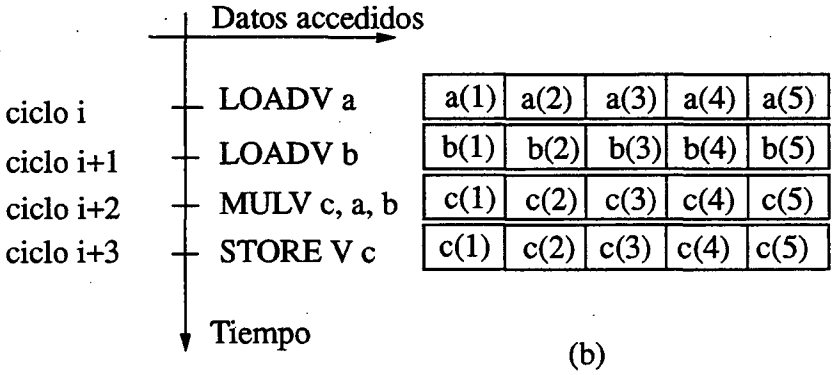


Figura 2.6. Técnica de replicación a nivel de datos

de la unidad de control a todos los elementos de un vector (a), en el siguiente ciclo podemos multiplicar (en tantos multiplicadores como sean necesarios idealmente), dos vectores (a, b) y obtener el resultado en otro vector (c), y en el último ciclo podemos almacenar todos los elementos del vector resultado en la memoria.

a) Ejecución de instrucciones. Algunas arquitecturas que se adaptan a este modelo, entre

otras, son:

1) *Procesadores con varios flujos de ejecución o superescalares.* Los procesadores superescalares disponen de varias unidades funcionales, por ejemplo una para tratar con enteros y otra para tratar con números en coma flotante, y la capacidad de poder lanzar a ejecutar varias instrucciones en paralelo, una para cada unidad funcional. De esta manera, se pueden conseguir velocidades medias de más de una instrucción por ciclo. Evidentemente en estas máquinas se requiere un ancho de banda con memoria mayor que en los procesadores segmentados.

Para poder obtener rendimientos buenos, es muy importante que el compilador efectúe una reordenación adecuada de las instrucciones. Por ejemplo entrelazando instrucciones que operan con números en coma flotante, con instrucciones sobre enteros (cálculo de direcciones). En la actualidad no existen buenos compiladores que aprovechen eficientemente la arquitectura; pero la programación secuencial sigue siendo la forma en la que se puede programar estas arquitecturas.

Ejemplos de este tipo de máquinas son el RS 6000 de IBM, el intel i80960, y el Alpha AXP de Digital y el PowerPC de IBM que es capaz de lanzar a ejecutar hasta 6 instrucciones. En la actualidad la mayoría de los fabricantes de procesadores apuestan por la fabricación de este tipo de procesadores.

En la figura 2.7 se muestra de forma ideal la ejecución de varias instrucciones en un procesador superescalar. En el ejemplo de la figura las instrucciones I_k e I_{k+1} se ejecutan en el mismo ciclo.

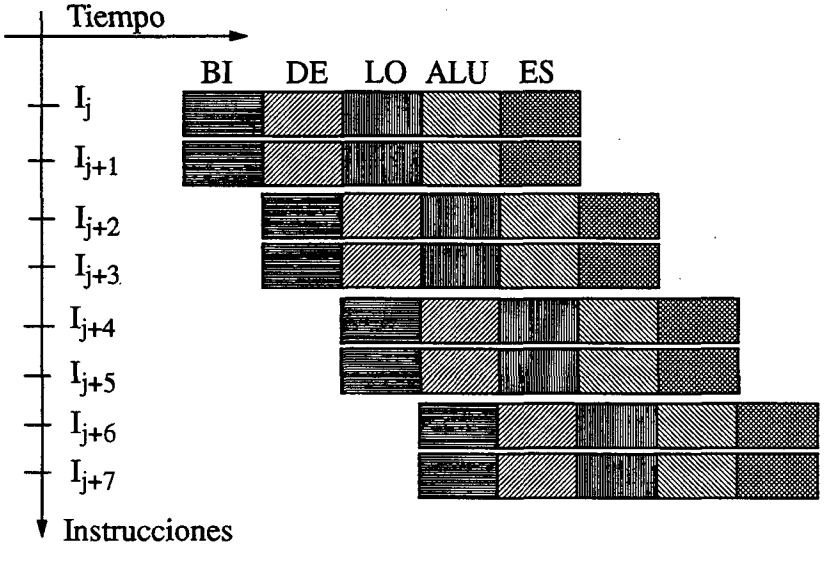


Figura 2.7. Técnica de replicación de instrucciones

Nótese que esta ejecución consiste simplemente en aumentar el número de instrucciones que se ejecutan en un ciclo. Debido a que se ejecutan varias instrucciones por ciclo, se necesita mejorar el bus memoria-procesador para que pueda soportar el ancho de banda necesario para acceder a varios datos simultáneamente.

2) *Multiprocesadores*. Los Multiprocesadores son computadores constituidos por varios Elementos de Proceso (EPs), que cooperan de forma asíncrona en la ejecución de un programa (no existe una unidad de control que supervise a todos los EPs). La comunicación y sincronización entre EPs se realiza a través de un único espacio de memoria compartido.

Los multiprocesadores comerciales actuales difícilmente superan los 16 EPs, uno de las últimas máquinas de CRAY el CRAY Y-MP C90 puede tener hasta 16 EPs en su configuración máxima, aunque se han construido multiprocesadores experimentales con más de 16 EPs.

La programación de los multiprocesadores se puede hacer secuencialmente y existen compiladores paralelizantes que dan buenos resultados para aplicaciones que exhiben ciertos patrones regulares de accesos a las estructuras de datos que trata el programa.

3) *Multicomputadores*. Los multicomputadores están constituidos por varios nodos (cada nodo está formado por un EP, memoria y varios enlaces de comunicación), que cooperan de forma asíncrona y se comunican a través de enlaces de comunicación.

En estas máquinas existe la posibilidad de incluir un mayor número de EPs, por ejemplo los multicomputadores basados en Transputers pueden tener un número elevado de EPs, en junio de 1991, la firma alemana Parsytec anunció el modelo Parsytec GC-5, que en su configuración máxima puede tener hasta 16384 EPs (T9000).

La programación de estas arquitecturas es difícil y se debe realizar, en muchos de los casos, manualmente debido a que los compiladores paralelizantes están en un estado de construcción primitivo.

4) *Agrupaciones de estaciones de trabajo*. Una agrupación de estaciones de trabajo está formada por una serie de estaciones de trabajo conectadas, generalmente, en un área reducida mediante una red de comunicación. Estas arquitecturas son atractivas porque su coste económico es pequeño y técnicamente ofrecen una funcionalidad equivalente o mayor a las de los multiprocesadores o los multicomputadores.

La computación paralela sobre redes de estaciones de trabajo es útil para las aplicaciones que tienen un elevado grado de paralelismo. En la actualidad existen entornos de programación paralela para redes de comunicación de baja velocidad. La ventaja es que se puede aprovechar máquinas de propósito general para hacer cálculo en paralelo. Sin embargo, la alta latencia de las comunicaciones es un handicap importante, aunque con el avance de las técnicas de transmisión este handicap es cada vez menos importante. Utilizando el Modo de Transmisión Asíncrono (MTA) se puede alcanzar velocidades de comunicación relativamente altas (2 Mbits/sg.).

En la actualidad existen algunos intentos de construcción de compiladores paralelizantes que generen código paralelo para su ejecución en estas arquitecturas; sin embargo, lo más común es la existencia de librerías de comunicación que facilitan la construcción de los programas paralelos.

En la figura 2.8 se muestra un esquema de la ejecución ideal teórica de instrucciones en el tiempo para este tipo de arquitecturas.

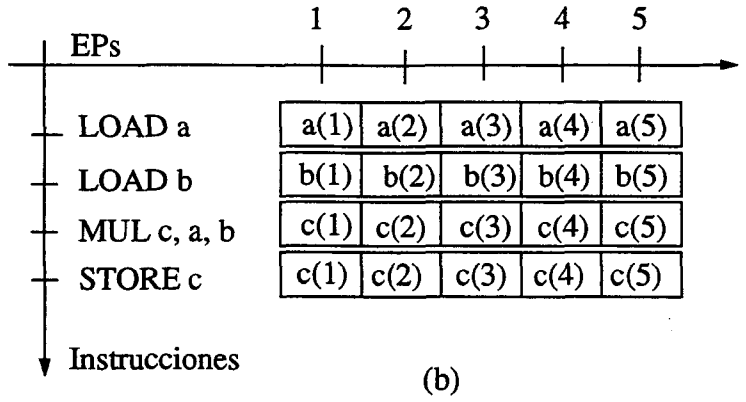


Figura 2.8. Técnica de replicación a nivel de instrucciones

Nótese que por motivos de comparación con la técnica de replicación de datos hemos supuesto que estas arquitecturas ejecutan las mismas instrucciones (cada EP ejecuta la misma instrucción LOAD, MUL y STORE), pero con distintos datos. Además hemos supuesto que no existe ningún tipo de sincronización y no hemos especificado en que posiciones de la memoria están almacenados los datos.

2.4. Aplicaciones de las Arquitecturas Paralelas

Existen muchas áreas de aplicación de las arquitecturas paralelas [Los94]. Mientras que una descripción exhaustiva requeriría muchas hojas, simplemente explicaremos las características principales de algunas de las aplicaciones. En concreto, analizamos las siguientes aplicaciones:

a) Aplicaciones que modelan de sistemas físicos.

b) Biología y vida artificial.

c) Gráficos, visualización y realidad virtual.

2.4.1. Aplicaciones que modelan de sistemas físicos

Las propiedades de los fenómenos físicos se explican mediante las leyes universales de la Física. Esas leyes, tales como las leyes de la conservación de la masa, la conservación de la energía o la conservación del momento se pueden usar para describir la naturaleza de muchos fenómenos. Esos fenómenos pueden ser modelados para predecir el comportamiento de situaciones del mundo real: Flujos de fluidos, predicción del tiempo atmosférico, diseño de cristales, etc.

Todos los modelos que se plantean para diferentes fenómenos Físicos tienen en común que se pueden describir mediante ecuaciones diferenciales parciales. Desde el punto de vista de las arquitecturas paralelas esto es interesante puesto que se pueden emplear métodos de computación eficientes para resolver estas ecuaciones. Las arquitecturas paralelas son necesarias para resolver estas ecuaciones debido a que: a) la cantidad de cálculos que se debe efectuar para resolver estas ecuaciones es relativamente grande (cuanto mayor sea el número de procesadores y cantidad de memoria mayor número de cálculos podremos efectuar). b) Cuanto más precisión queramos obtener en el modelo mayor número de

cálculos debemos realizar. Por lo tanto para obtener una aproximación lo suficientemente buena deberíamos hacer una cantidad de cálculos apropiada a la resolución deseada.

A continuación analizamos un ejemplo de modelado de un sistema Físico y un método de solución de ecuaciones que son apropiados para ser resueltos en las arquitecturas paralelas con el objetivo de que se pueda permitir un nivel de aproximación con gran cantidad de cálculos.

Ejemplo 2.2 Conducción de calor

En este ejemplo analizamos el problema Físico de la conducción de calor. La conducción de calor es el mecanismo a través del cual el calor es distribuido desde una fuente específica en un área cerrada. Un ejemplo de esta forma de transferencia de calor es la que se produce desde una habitación calurosa a un vaso de soda fría que está en esa habitación. A medida que pasa el tiempo el frío se vuelve más caliente. La razón de esto es que el calor de la habitación se distribuirá uniformemente en todos los objetos de la habitación.

En un sistema de coordenadas de tres dimensiones nosotros podemos medir la difusión del calor como una función de la temperatura y la conductividad de los materiales a través de los cuales fluirá el calor. La ecuación utilizada para describir la difusión del calor en ausencia de fuentes de calor es la ecuación de Laplace. Formalmente se escribe de la siguiente forma:

$$\frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\frac{\partial T}{\partial z}\right) = 0$$

Donde: T es la temperatura, k representa la conductividad del material, y las variables x, y, z son las tres coordenadas de direcciones espaciales. Además, un conjunto de condiciones de contorno describe las restricciones que existen en los límites del sistema.

□

Las ecuaciones diferenciales que modelan los sistemas Físicos pueden ser resueltas mediante el método de los elementos finitos. Este método requiere una discretización: transformación un conjunto de puntos de un espacio continuo a un espacio discreto. Para muchos problemas de interés la geometría de los objetos implicada puede hacer que encontrar esa discretización (en un nuevo conjunto de puntos que se denominan rejilla), pueda ser difícil o fácil. La región se divide en un número finito de elementos que representan una porción del espacio continuo. De esta forma se puede analizar el comportamiento del sistema en una porción, estudiando un conjunto de elementos finitos.

Teniendo discretizado el sistema, se procede a construir un sistema de ecuaciones lineales para cada porción del sistema. A continuación se procede a ensamblar todos los conjuntos de ecuaciones lineales para estudiar el comportamiento de todas aquellas porciones de interés. El resultado de este ensamblado es un sistemas de ecuaciones lineales que junto con las condiciones de contorno son ensambladas de forma matricial en una matriz

que generalmente suele ser dispersa (el número de elementos igual a cero es bastante mayor que el de elementos distinto de cero). Las filas de esta matriz representan a las variables del sistema Físico y cada columna representa a una ecuación diferente.

Por lo tanto si queremos analizar una gran cantidad de variables del sistema Físico y tener una gran cantidad de ecuaciones (una región grande o discretizar con mucha precisión), tenemos matrices que tienen muchos elementos; por lo tanto necesitamos resolver el sistema de ecuaciones mediante una arquitectura con potencia de cálculo y memoria grandes. Una ventaja que presenta la solución de estos problemas es que se pueden resolver paralelamente varias partes diferentes del sistema de ecuaciones (varias regiones diferentes del sistema físico: diferentes partes del vaso de soda), suponiendo que el sistema es lineal. Un paso final consiste en acoplar las diferentes soluciones obtenidas por separado (descomposición de dominios y datos).

Otros modelos de los sistemas Físicos podrían ser: Flujo de fluidos, Análisis estructural, Diseño de aviones, Análisis de choques en los automóviles, predicción del tiempo, modelos de los océanos, etc.

2.4.2. Biología y vida artificial

Uno de los objetivos de la ciencia de la computación es la utilización de los computadores para comprender mejor el funcionamiento del cuerpo humano. Bien modelando el funcionamiento del cuerpo o bien creando software de inteligencia artificial que intente crear modelos de aprendizaje semejante al aprendizaje del ser humano.

Uno de los campos de la biología en los que intervienen activamente los computadores es la bioquímica computacional. Uno de los problemas de la bioquímica computacional es el problema del conocimiento de las proteínas. Las proteínas constan de cadenas de aminoácidos que forman objetos tridimensionales. El perfil de las proteínas es crucial para entender como trabajan. El conocimiento de una proteína específica es importante debido a que ayuda a aprender como puede funcionar una molécula que esté compuesta de varias de esas proteínas. Además el conocimiento de las proteínas da luz para la construcción de nuevos medicamentos sintéticos que ayuden a combatir una enfermedad.

Aunque es relativamente sencillo determinar la secuencia de aminoácidos que forman una proteína, es muy difícil predecir como una cadena particular de aminoácidos se constituyen en una nueva proteína determinada. Existe un número pequeño de proteínas para las cuales se ha determinado su perfil tridimensional; esas proteínas han sido incluidas en una base de datos. Cuando se intenta determinar el perfil de una proteína se debe comparar con la base de datos de proteínas para encontrar un determinado perfil (en definitiva consiste en buscar una determinada secuencia de aminoácidos y la relación espacial que existe entre ellos). Si se encuentra un determinado perfil en la base de datos entonces su estructura se extrae directamente de la base de datos, en caso contrario entonces se deben hacer búsquedas de aquellas proteínas que se parezcan en la mayor medida de lo posible. Las proteínas que se encuentren en esta segunda búsqueda serán utilizadas para modelar la proteína buscada.

El tamaño de la base de datos de proteínas podría ser enorme (cada año se incrementa en

un orden de 10). En este caso, es imprescindible disponer de mucha potencia de cálculo y memoria para poder obtener buenas prestaciones del sistema de comparación de proteínas. Fijémonos que este es un problema sencillo (simples comparaciones), de datos (secuencias de números enteros que codifican los aminoácidos), sin embargo, necesitamos tener una base de datos distribuida entre las memorias de varios procesadores y gran cantidad de procesadores para poder obtener buenas prestaciones. Este ejemplo es sustancialmente diferente al de los modelos de sistemas Físicos y el de visualización en los cuales se tratan con datos en coma flotante y se necesita una potencia de cálculo interna de los procesadores bastante grande.

Un ejemplo de vida artificial son los conocidos virus informáticos que pueden infectar el software, se reproducen y se pueden comunicar a otras máquinas. Los procesos evolutivos son utilizados para estudiar la forma en que individuos de un determinado mundo evolucionan atendiendo a su interacción con su entorno. Cada proceso puede: aprender cosas sobre su entorno, cambiar su entorno, mutar, envejecer, moverse por su entorno, "razonar" y morir o ser asesinado por otros procesos. Un sistema de este tipo puede ser utilizado para modelar diferentes leyes Físicas de la biología de un ecosistema que evoluciona dinámicamente en el tiempo. Estos sistemas son inherentemente paralelos y en un caso real en el que se quiere modelar un ecosistema con muchas leyes Físicas y una abundante cantidad de individuos sólo puede ser estudiados haciendo uso de las arquitecturas paralelas. Pensemos que podríamos simular la evolución de un millón de microbios en un trozo de materia viva en la que nos interesan magnitudes como la temperatura, presión, estructura atómica de la materia, etc. (podríamos pensar en un centenar de magnitudes Físicas); si además cada microbio puede mutar y recombinarse con sus congéneres rápidamente podría dispararse el número de microbios. Si cada microbio es representado mediante un proceso podríamos estar tratando simultáneamente millones de procesos lo cual es sólo factible con arquitecturas paralelas.

2.4.3. Gráficos, visualización y realidad virtual

El campo de los gráficos por computador está emparejado a muchas áreas de generación y representación de imágenes sintéticas por computador. Estos campos incluyen: librerías de usuario tales como las X-windows, modelado tridimensional, animación por computador, video interactivo, realidad virtual, etc. Las técnicas de síntesis realista por computador son computacionalmente complejas y deben producir imágenes realistas en tiempo real. Esto sería imposible sin arquitecturas paralelas que puedan resolver estos problemas en una cantidad de tiempo razonablemente pequeño. En este sentido es importante poder visualizar los resultados de la simulación de un sistema Físico previamente modelado mediante ecuaciones diferenciales parciales (por ejemplo la evolución de la temperatura en el vaso de soda); algunos autores se refieren a este campo de los gráficos por conmutador como visualización de datos científicos.

A continuación analizamos: a) una técnica de formación de imágenes realistas denominada traceo de rayos (ray tracing), b) visualización científica y c) realidad virtual atendiendo a sus necesidades de computación.

a) La idea básica de la técnica de traceo de rayos es considerar todas las posibles fuentes

de luz y como interaccionan con cada objeto de un espacio de objetos. Los rayos son traceados a partir de cada fuente de luz a través de cada píxel del plano y dentro del espacio de objetos. El primer objeto que es alcanzado por el rayo es el que aparece como superficie visible. Cada vez que se alcanza un objeto se modelan un número determinado de fenómenos físicos: En primer lugar, cualquier objeto que está detrás de una superficie por el que pasa un rayo queda en la zona oscura de la imagen y no se ilumina. En segundo lugar, si el material de la superficie es reflectivo o refractivo entonces se poyectan rayos secundarios a partir de esta superficie. Estos rayos contribuyen a la iluminación de otros objetos.

El cálculo de las imágenes mediante traceo de rayos está basado en la intersección de los rayos de un conjunto particular de fuentes de luz con los objetos. Cualquier punto x de un rayo puede ser descrito mediante la ecuación: $x = P + \lambda D, (\lambda \geq 0)$. Donde P es el punto de inicio del rayo y λ es la distancia a lo largo del rayo, y D es una dirección expresada en función de las coordenadas del espacio.

El objeto más simple que se puede modelar, una esfera, puede ser descrito en términos de el centro de la esfera y su radio. Cualquier punto x de la superficie de la esfera puede ser descrito mediante la ecuación: $|x - v| = r$, donde v es el punto central de la esfera y r es el radio.

Sustituyendo la primera ecuación en la segunda y simplificando obtenemos la siguiente ecuación cuadrática: $\lambda^2 + (2((P - v) \times D)\lambda + (P - v)^2 - r^2) = 0.$

La solución a esta ecuación son todos los puntos del rayo que intersectan la superficie de la esfera. Analizando esta ecuación se puede revelar que se necesitan: 17 sumas o restas, 17 multiplicaciones y una raíz cuadrada para poder solucionarla. Un espacio de objetos típico puede contener miles de esferas y un número grande de fuentes de luz. Además podemos suponer que estas esferas son reflectivas (lo que incrementa el número de rayos). Suponiendo que tenemos una superficie de 1000x1000 píxeles, 1000 esferas y una media de 5 rayos por píxel, el número de operaciones para determinar las intersecciones de rayos con objetos son:

$$1000 \times 1000 \times 5 \times 1000 \times (17 + 17 + 1)$$

operaciones en coma flotante. Esto es,

175 BILLONES

de operaciones en coma flotante.

Es por tanto necesario disponer de una máquina con gran potencia de cálculo para poder obtener imágenes en un tiempo razonable. Una forma de hacer factible la implementación de esta técnica es repartir diferentes partes de la imagen a diferentes procesadores que podrían hacer los cálculos en paralelo.

b) La visualización científica utiliza las técnicas de gráficos para representar de forma gráfica la información resultante del modelado de los sistemas Físicos. La mayoría de los sistemas de modelado de sistemas Físicos incluyen una etapa de postproceso mediante la

cual se puede ver de forma gráfica la evolución del sistema representando una o varias magnitudes físicas de interés.

c) La realidad virtual es una técnica de representación gráfica mediante la cual se permite a una persona que se integre en un entorno tridimensional generado por computador en el que la persona puede experimentar con un mundo que es virtual. Este mundo virtual puede ser enriquecido mediante sensores visuales, táctiles, etc.

Quizás estos sistemas interactivos son los más complejos: requieren un tiempo real bastante cercano al tiempo de reacción de los seres humanos se requiere el proceso de información de sensores y actuadores sobre el mundo virtual, además se deben representar distintos objetos volumétricos y poder desplazar objetos en el mundo virtual (*sistema de tiempo real*). Existen algunos ingenios de computación que integran *sistemas distribuidos* de computación especializados mediante redes de comunicación de muy alta velocidad. Estos sistemas se combinan con *arquitecturas paralelas* que procesan los gráficos de distintos objetos en paralelo. Y además se conectan a procesadores especialmente construidos para la representación gráfica de los objetos del mundo así como de la evolución de la persona dentro del mundo virtual. Opcionalmente, en sistemas que simulen un proceso de información crítico (una central nuclear, por ejemplo), se puede replicar parte del sistema de computación para tolerar fallos (*sistemas tolerantes a fallos*)

Notas Bibliográficas

En [RaR93] podemos encontrar un índice general sobre conceptos de la informática. Se pueden encontrar las definiciones de muchos de los conceptos básicos comentados. En [AGP90] se muestra la idea fundamental de la programación como una sucesión encadenada de fases en el desarrollo de un proyecto software (que se explica más profundamente en [Pre93]).

En [Set92] y [Wat90] se presentan los modelos de programación y una revisión general de la evolución de la programación comenzando desde la programación estructurada hasta la programación orientada a objetos. Además se explican muchos ejemplos escritos en diferentes lenguajes de programación. Hay un capitulo dedicado al estudio del modelo de programación concurrente desde el punto de vista del diseño de lenguajes. En [AIS95] se presenta una clasificación de las metodologías de programación interesante en la que no se reconoce el modelo de programación concurrente como tal debido a que los parámetros que utiliza el autor no son exactamente los que se utilizan en [Wat90].

En [Dic87] se presentan ejemplos en los que se maneja el solapamiento de operaciones con la entrada y salida de datos en una arquitectura secuencial. En [Gom93] se presentan métodos de ingeniería de software especialmente apropiados para los sistemas de tiempo real. En [CDK90] se puede encontrar una buena introducción a los sistemas distribuidos. En [DWe90] se presentan los sistemas en tiempo real de forma bastante clara y se explican diferentes lenguajes de programación concurrentes aplicados a estos tipos de sistemas. En [Ren84] se presenta una introducción muy breve a los sistemas tolerantes a fallos.

En [Tan90] se puede encontrar la explicación de los niveles de abstracción de la arquitectura de computadores que es muy clásica. En [Fer94] se puede encontrar una introducción didáctica a los distintos niveles (se introducen los principales conceptos de los distintos niveles). En [MiA95] se puede encontrar una introducción muy sencilla a las arquitecturas de alta velocidad. En este libro se explican algunas arquitecturas vectoriales ya clásicas. En [HeP93] se puede encontrar una buena introducción a las arquitecturas monoprocesador y métodos de evaluación de la ejecución de los programas secuenciales en estas arquitecturas. También se introducen las arquitecturas de segmentación de instrucciones y de datos.

En [HBr88] se define el procesamiento concurrente y se explican las arquitecturas segmentadas con un nivel profundo. Sirve como introducción a las arquitecturas más modernas.

En [Los94] se puede encontrar un amplio espectro de aplicaciones de la computación de alta velocidad. Es muy sencillo comprender las aplicaciones que se explican si bien el estudio profundo de estas aplicaciones sería más complicado. En [SuF95] se puede encontrar un estudio sobre la aplicación de las arquitecturas paralelas en el procesamiento de señales.

Biliografía

[AGP90] E. Alcalde, M. García, S. Peñuleas, *Informatica Básica*, McGraw Hill, ISBN 84-7615-241-8, 1990. pp. 125-141.

[AlS95] Alonso Amo, F., Segovia Pérez, F.J., *Entornos y metodologías de programación*, Paraninfo, ISBN 84-283-2164-7.

[CDK94] Colouris Goerge, Dollimore Jean, Kinderberg Tim, *Distributed Systems. Concepts and Design*, Addison-Wesley, ISBN 0-210-62433-8.

[Dic87] Whiddett Dick, *Concurrent Programming for Software Engineers*, Ellis Horwood Limted Publishers (Jonh Willey&Sons), ISBN 13-161746-X.

[DWe90] Burns Alan, Wellings Andy, *Real-Time systems and their programming languages*, Addison-Wesley international Computer Science series, ISBN 0-201-17529-0.

[Fer94] Fernández Gregorio, *Conceptos básicos de arquitectura y sistemas operativos*, Sistemas y servicios de comunicación S.L., ISBN 84-605-0522-7.

[Gom93] Gomasa, Hassan, *Software design methods for concurrent and Real-Time systems*, Addison-Wesley, ISBN 0-201-52577-1.

[HBr88] Hwang Kai, Briggs Faye, *Arquitecturas de Computadores y Procesamiento paralelo*, McGraw-Hill, ISBN 968-422-344-7.

[HeP93] Hennessy J.L. y Patterson D.A., *Arquitectura de Computadores. Un enfoque cuantitativo*, McGraw-Hill, 1993

[Los94] Loshin David, *High performance Compunting Demystified*, AP Profesional, ISBN 0-12-455825-9.

[MiA95] De Miguel Pedro, Angulo José M., *Arquitectura de computadores. Fundamentos e introducción al paralelismo*, Cuarta Edición, Paraninfo, ISBN 84-283-1883-2.

[Pre93] Pressman Roger S., *Ingeniería del Software. Un enfoque práctico*, McGraw-Hill, ISBN 84-481-0026-3.

[RaR93] Ralston A., Reilly E.D., *Encyclopedia of Computer Science*, IEEE press, ISBN 0-442-27679-6.

[Ren84] Rennels David A., *Fault Tolerant computing. Conceps and examples.* IEEE Transactions on Computers, vol. C-33, no. 12, Dic. 1984.

[Set92] Sethi Ravi, *Lenguajes de programación. Conceptos y constructores*, Addison-Wesley, ISBN 0-201-51858-9.

[SuF95] Suárez Alvaro, De la Puente, Fernando, *Arquitecturas paralelas para el procesamiento digital de imágenes*, Servicio de Reprografía de la Universidad de Las Palmas de Gran Canaria, ISBN GC-408-1995.

[Tan90] Tanembaum A.S., *Structured Computer Organization*, Prentice Hall, 3a edición, 1990.

[Wat90] Watt A. , *Programming Language concepts and Paradigms*, C.A.R Hoare series Editor, Prentice Hall, ISBN 0-13-728866-2.

Ejercicios

Ejercicio 1. Algoritmo de ordenación

Escribir un programa recursivo en el lenguaje imperativo que desee, que ordene los elementos de un vector de elementos reales (elija el tipo de ordenación), mediante la técnica divide y vencerás (ordenación por mezcla). Para resolver los subproblemas, los subvectores han de tener como máximo p elementos (el vector original tiene N elementos).

Ejercicio 2. Sistemas distribuidos, de tiempo real y tolerantes a fallos

Describir (a nivel de diagrama de bloques), un sistema de control aéreo en el que existan varias máquinas que realicen el control del despegue y aterrizaje de aviones comerciales.

Para centrar la descripción, se debe suponer que las prestaciones son las siguientes:

- Existirá un pequeño sistema informático (c<6 Computadores Personales y un Servidor de bases de datos de aviones), dedicado a controlar si un determinado avión tiene permiso para despegar o aterrizar. Además en este servidor se almacena los nombres de los pasajeros (nombres, D.N.I., sexo, asiento que ocupa, etc.) que viajan en el avión. En cada Computador Personal existirá un operador que lleva la contabilidad de los pasajeros que despegan y aterrizan en los aviones. Se tendrá informada a la Policía en caso necesario de si una persona sospechosa de haber cometido un delito ha llegado o salido del aeropuerto.
- Existirán dos pantallas de radar con operadores que llevan el registro de la información gráfica del plan de vuelo de los aviones controlados por ese centro de control aéreo. Estas pantallas están conectados a un radar situado dentro del espacio aéreo del centro de control. Además están conectadas mediante computadores especializados (a través de un radio enlace) a la torre de control por si fuese necesario modificar la trayectoria de los aviones o el momento de aterrizaje o despegue. La torre de control es la que envía estas órdenes a los aviones mediante transmisión por radio.

  Desde los computadores Personales se podría acceder a esta información gráfica mediante un servidor de información gráfica de radares.
- Existirán un computador de supervisión del funcionamiento (monitorización) de todas las máquinas anteriores que en caso de fallo envían señales a una consola donde está el supervisor general de la instalación de control aéreo informático.

Describir cualitativamente y brevemente (en lenguaje natural) los programas que deberían ser usados en la instalación y qué partes del sistema deberían ser tolerantes a fallos. ¿Qué lenguaje o lenguajes elegirías para la programación del sistema y porqué?. ¿Sería necesaria una máquina paralela para implementar la base de datos de los aviones y los pasajeros?. ¿Podría una red de comunicación vía radio en caso de que el sistema fallase sin posibilidad de ser recuperado para comunicar el centro de control aéreo con la torre de control?. Razonar brevemente las respuestas.

Ejercicio 3. Procesadores escalares y vectoriales segmentados

Calcular el tiempo de ejecución del problema matriz por vector en un procesador

segmentado con 5 fases de ejecución y en un procesador vectorial segmentado. Suponer que existen suficientes buses memoria-procesador como para permitir que se hagan tantas lecturas y escrituras simultáneas como sean necesarias.

Ejercicio 4. Modificación de la arquitectura Von Neumann mediante replicación

Hacer el diagrama de tiempos para las arquitecturas paralelas que se obtienen a partir de la arquitectura Von Neumann mediante la técnica de replicación. Dar un orden de magnitud del tiempo de ejecución que se ahorra suponiendo que existen buses memoria procesador o procesadores suficiente como para hacer todas las operaciones en paralelo que sean necesarias.