Capítulo 1:
INTRODUCCIÓN

En este capítulo, se describen brevemente diferentes arquitecturas que permiten la ejecución simultánea o concurrente de varias operaciones, centrándose en los mecanismos que éstas ofrecen para explotar esta concurrencia y en las distintas posibilidades que se ofrecen al programador de aplicaciones.

En la sección 1 se introducen los conceptos de segmentación y paralelismo, así como las distintas arquitecturas que utilizan estas técnicas. Las secciones 2 y 3 se centran en las arquitecturas para las cuales se describen y proponen técnicas de paralelización en esta tesis (procesadores vectoriales segmentados y sistemas multiprocesador respectivamente). En la sección 4 se presenta el trabajo realizado y las aportaciones de esta tesis.

1.1 ARQUITECTURA

El aumento en la velocidad de proceso de los computadores, desde la aparición del primer computador digital en 1945 hasta los supercomputadores actuales [Hwan87], se ha debido a dos factores bien diferenciados:

(a) *tecnología:* 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 en la capacidad de encapsulado de los dispositivos en circuitos integrados, están permitiendo reducir el tiempo de cálculo de funciones complejas. El diseñador es quien, en función del estado de la tecnología, decide qué utilizar para obtener la velocidad de cálculo deseada junto con otros parámetros de diseño como tamaño, consumo,...

(b) *concurrencia:* por concurrencia se entiende la posibilidad de efectuar varias operaciones simultáneas en un instante dado. Las técnicas básicas se denominan segmentación y paralelismo [HoJe81].

1.1.1 Segmentación

La técnica de *segmentación* consiste en descomponer una determinada operación en *n* suboperaciones a realizar en fases o etapas distintas. De esta manera, una operación se realiza a medida que la información involucrada en dicha operación atraviesa las *n* etapas. La concurrencia se obtiene a base de ejecutar *n* operaciones simultáneas, aunque cada una de ellas en una etapa distinta.

La técnica de segmentación puede aplicarse (a) a nivel de ejecución de instrucciones o (b) a nivel de operación sobre datos.

Procesadores escalares segmentados

A nivel de ejecución de instrucciones o lenguaje máquina, la técnica de segmentación ofrece la posibilidad de ejecutar varias instrucciones de forma concurrente, dando lugar a los *procesadores escalares segmentados*. La ejecución de la instrucción se divide en *n* fases, de manera que en cada ciclo se

inicia la ejecución de una nueva instrucción, a la vez que el resto de instrucciones ya iniciadas avanzan una fase.

En este tipo de máquinas, el compilador juega un papel muy importante, utilizando técnicas de reordenación de código a fin de mejorar la eficiencia [HeGr83], [GrHe85].

Computadores vectoriales segmentados

La técnica de segmentación también puede aplicarse a nivel de unidad de cálculo o funcional. A este nivel, una misma operación pueda realizarse de forma concurrente, en una misma unidad funcional, sobre varios elementos de una estructura de datos regular (vectores, matrices, …) aunque en etapas distintas. Este tipo de computadores se denominan *computadores vectoriales segmentados.*

Su programación puede abordarse utilizando (a) librerías de funciones ya programadas pensando en la máquina en que van a ser ejecutadas, (b) lenguajes de programación que permitan especificar este tipo de operaciones o (c) utilizando compiladores capaces de vectorizar código secuencial. Estos aspectos son considerados más ampliamente en la sección 1.2.

1.1.2 Paralelismo

Por *paralelismo* se entiende la realización concurrente de cálculos, iguales o diferentes, sobre distintos conjuntos de datos por parte de varias unidades funcionales.

La técnica de paralelismo puede aplicarse (a) a nivel de ejecución de instrucciones (procesadores escalares con varios flujos de ejecución, computadores VLIW y sistemas multiprocesadores) o (b) a nivel de operación sobre datos (procesadores en "array").

Procesadores escalares con varios flujos de ejecución

Algunos procesadores escalares aprovechan la técnica de paralelismo, incorporando en su estructura distintos caminos de datos para las instrucciones que operan con aritmética entera y para las que operan con aritmética en coma flotante. Ambos flujos de instrucciones son tratados de

forma paralela y sólo se sincronizan los flujos cuando se ejecutan instrucciones de conversión de tipos o de transferencia de control.

Algunas arquitecturas sólo separan los flujos en las unidades de ejecución (IBM 360/91 [AnST67]), mientras que otros los separan inmediatamente después de la fase de búsqueda de instrucción (ZS-1 [Smit89]). La búsqueda de un número de bits que en media incluyen más de una instrucción, la detección del tipo de las mismas y su tratamiento posterior por el camino apropiado aumenta el paralelismo.

La programación de estas máquinas es convencional aunque el compilador puede ayudar a aumentar el rendimiento aplicando técnicas de reordenación de código.

Computadores VLIW

Los computadores VLIW [FiOd84] (Very Long Instruction Word Machines) están constituidos por varios elementos de proceso, que operan de forma síncrona y bajo control de una única instrucción. La característica básica de estos computadores es que todos los elementos de proceso realizan la búsqueda de instrucciones en paralelo. Esta instrucción está dividida en campos que controlan de forma individual a cada uno de los elementos de proceso, de manera que cada uno de ellos puede realizar una operación distinta.

Es prácticamente imposible programar directamente este tipo de máquinas dado que el número de elementos de proceso suele ser muy elevado, existiendo compiladores capaces de extraer código paralelo a partir de programas secuenciales, como por ejemplo Bulldog que utiliza la técnica de *Trace Scheduling* [Elli86] para dicho fin.

Sistemas multiprocesadores

Otra alternativa son los sistemas multiprocesadores [Ensl77], basados en varios procesadores de uso general que operan de forma asíncrona, y que por lo tanto requieren mecanismos de comunicación y sincronización entre ellos. Existen dos tipos básicos de sistemas multiprocesador: (a) *sistemas con memoria compartida* y (b) *sistemas con memoria distribuida* y comunicación local entre procesadores.

Su programación se aborda utilizando (a) librerías de funciones ya programadas, (b) lenguajes de programación que incluyen construcciones paralelas que permiten especificar el paralelismo o (c) utilizando compiladores capaces de extraer de forma automática partes ejecutables de forma concurrente. Estos aspectos son considerados con más detalle en la sección 1.3.

Actualmente se están utilizando sistemas multiprocesadores donde el elemento de proceso es un procesador vectorial segmentado [Hwan85], permitiendo así los dos niveles de concurrencia descritos dentro del mismo computador.

Procesadores en "array"

Los procesadores en "array" poseen una única unidad de control, de manera que se realiza una misma operación sobre elementos distintos de una estructura de datos regular en unidades funcionales distintas. Por ejemplo, el Burroughs BSP [HoJe81] es un computador de este tipo que incorpora 16 unidades funcionales.

Su programación puede abordarse de la misma manera que los procesadores vectoriales segmentados.

1.2 PROCESADORES VECTORIALES SEGMENTADOS

En esta sección se describen con más detalle los procesadores: vectoriales segmentados, haciendo hincapié en (a) las características que ofrecen estas arquitecturas, (b) cómo los lenguajes de programación aprovechan sus características arquitectónicas y (c) cómo un compilador puede extraer de forma automática operaciones vectoriales a partir de código secuencial.

1.2.1 Arquitectura

Los procesadores vectoriales segmentados disponen de una *unidad escalar* encargada de la ejecución de las instrucciones escalares del programa, y un *controlador vectorial* encargado de controlar la ejecución de las instrucciones vectoriales.

El controlador vectorial gobierna el funcionamiento de las *unidades funcionales segmentadas* y el generador de direcciones o *unidad de*

direccionamiento vectorial, que alimenta a dichas unidades funcionales con la información procedente de estructuras de datos almacenadas en memoria.

La estructura general de un procesador vectorial segmentado se muestra en la figura 1.1.

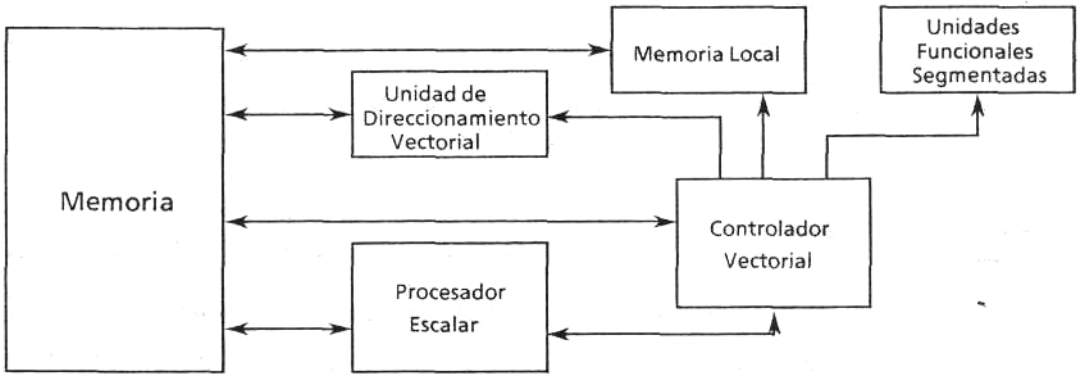

Figura 1.1: Estructura general de un procesador vectorial segmentado.

Registros

En general pueden distinguirse entre (a) procesadores cuyas instrucciones vectoriales obtienen los datos directamente de memoria (por ejemplo eL CDC Cyber 205 [CoDa81]) y (b) procesadores cuyas instrucciones vectoriales operan sobre valores almacenados previamente en registros (por ejemplo el CONVEX C-l [Conv87l o el UNISYSISP [Hode87]).

En el caso de procesadores vectoriales con registros, el tamaño máximo de los vectores con que se puede operar viene determinado por el número de registros disponible (por ejemplo 128 elementos en el CONVEX C-l). Vectores de mayor tamaño deben partirse en secciones y operar con dichas secciones de forma secuencial.

Registros de control

Para permitir la ejecución condicional de instrucciones vectoriales, estas arquitecturas poseen registros de control cuyos bits indican qué elementos del vector están realmente involucrados en la operación vectorial. Los dos mecanismos básicos consisten en (a) obtener y almacenar únicamente aquellos elementos indicados en el registro de control o (b) realizar la operación sobre

todos los elementos y posteriormente almacenar los resultados de los elementos indicados por el registro de control.

Encadenamiento

Otra característica importante es la capacidad de encadenamiento de operaciones vectoriales cuando se dispone de varias unidades funcionales (por ejemplo el CRAY X-MP [Chen85]). Esta característica arquitectónica permite que una instrucción vectorial pueda utilizar directamente el resultado de una instrucción vectorial previa, aunque esta última no haya finalizado todavía su ejecución.

Tiempo de Ejecución

Las características que determinan el tiempo de ejecución de las operaciones vectoriales en estos computadores son (a) el *tiempo de carga* t_c o tiempo que se tarda en obtener el primer resultado de una unidad funcional y (b) el *tiempo de etapa* t_e de cada una de las etapas de la unidad funcional.

A partir de ellos se obtiene la velocidad máxima (cuando se obtiene un nuevo resultado a cada t_e), la longitud de los vectores $n_{1/2}$ con la cual el procesador va a la mitad de su velocidad máxima y la longitud de los vectores por debajo de la cual es mejor trabajar en modo escalar.

El compilador debe conocer estos parámetros a fin de decidir cuando debe generar código, vectorial o secuencial.

1.2.2 Lenguajes de programación

La alternativa más utilizada en el desarrollo de lenguajes de programación para este tipo de máquinas ha sido la de extender los lenguajes secuenciales convencionales existentes, incorporando construcciones y primitivas orientadas a soportar operaciones vectoriales. Estas extensiones pretenden obtener el máximo provecho de las características que ofrecen estas arquitecturas, algunas de las cuales se han descrito brevemente en el apartado anterior.

A continuación se describen algunas de las extensiones incluidas en el FORTRAN-8X [ANSI87] y el CEDAR FORTRAN [Guzz87] y que serán utilizadas a lo largo de este trabajo. Estas extensiones permiten, entre otras

posibilidades, (a) seleccionar elementos de un array sobre los cuales realizar operaciones de asignación, aritméticas o lógicas, así como llamadas a funciones y (b) ejecutar operaciones vectoriales de forma condicional.

Selección de elementos

Una *sección* o conjunto de elementos de un vector se especifica mediante

var(i:f[:p])

siendo *var* una variable estructurada tipo vector, *i* el índice del elemento inicial, *f* el índice del último elemento y *p* el paso o número de elementos del vector entre dos consecutivos de la sección. Así por ejemplo la siguiente operación vectorialespecifica una operación vectorial sobre secciones

a(1:10:3) = b(2:8:2)· f(c(l:4))

y equivale a la secuencia de operaciones escalares

```
a(4)  =  b(4)·f(c(2))
a(7)  =  b(6)·f(c(3))
a(10) = b(8)·f(c(4))
```

Otra construcción que permite especificar los elementos de una estructura multidimensional de datos, sobre la cual realizar una determinada operación vectorial, es FORALL. Así por ejemplo, el siguiente fragmento de programa

```
FORALL(¡ = 1 : 10)
    a(i,i) = b(i)*c(¡)
ENDFORALL
```

realiza la operación vectorial indicada almacenando el resultado en la diagonal de la matriz.

Ejecución condicional

Estos lenguajes también permiten especificar instrucciones vectoriales a ejecutar de forma condicional bajo control de una condición, que también puede ser definida de forma vectorial. Así por ejemplo, la sentencia

```
WHERE(a(1 : 100) .LT. 0)
    a(1 : 100) = 0
ENDWHERE
```

es equivalente al siguiente fragmento de código secuencial

```
DO I = 1,100
    IF(a(I).LT.O)
    a(I) = 0
    FI
ENDO
```

La construcción FORALL descrita anteriormente también permite especificar una condición bajo la cual ejecutar una secuencia de instrucciones vectoriales. Así por ejemplo, el siguiente fragmento de programa

```
FORALL(i = 1 : 10, a(i, i) .LT. 0) a(i,i) = 0
```

pone a cero todos aquellos elementos de la diagonal de la matriz que sean

1.2.3 Compiladores

La otra alternativa que se presenta a la hora de programar estos computadores es la de utilizar un lenguaje imperativo secuencial y disponer de un compilador capaz de extraer operaciones vectoriales a partir del código secuencial de usuario.

Se denomina *vectorización* al proceso de generación automática de instrucciones vectoriales a partir del código secuencial de un bucle. Esta técnica se basa en la posibilidad de reorganizar el código de un bucle, de tal forma que, cada sentencia se ejecute para todos los valores del rango de iteración, antes de iniciar la ejecución de la siguiente sentencia del bucle. Así por ejemplo, para el siguiente bucle secuencial

```
    DO I = 1,100
Si:     a (I) = b (I) * c (I)
S2:     b(I) = 2*c(I)
    ENDO
```

el compilador se daría cuenta de que, sin violar la semántica del programa, es posible ejecutar de forma vectorial Si para todo el rango de iteraciones y a continuación S2 del mismo modo, generando el siguiente código

```
Si:     a(1:100) = b(1:100) *c(1:100)
S2:     b (1:100) = 2* c (1:100)
```

Observar que el compilador debe determinar en que orden han de ejecutarse las sentencias vectoriales, a fin de preservar la semántica del programa. Este orden queda determinado por las relaciones de precedencia de las operaciones realizadas sobre los datos en el programa secuencial. Las técnicas de análisis de estas precedencias se describen en el siguiente capítulo.

Las ventajas que esto supone son: (a) posibilitar la recompilación de programas secuenciales ya existentes, evitando así su reprogramación manual y (b) liberar al programador de la tarea de expresar la concurrencia del algoritmo. Sin embargo, esta es un área de investigación en la que se está trabajando actualmente y todavía no se disponen de técnicas que permitan obtener de forma automática-un alto grado de concurrencia.

En la actualidad existen algunos compiladores experimentales o comerciales que realizan esta tarea, de los cuales cabe destacar el PFC [AIKe87] de la Rice University, Parafrase [KKLW80] de la Universidad de Illinois o el UFTN [Cole87] para el computador UNISYS ISP, entre otros.

1.3 SISTEMAS MULTIPROCESADORES

En esta sección se describen con más detalle los sistemas multiprocesadores con memoria compartida, haciendo hincapié en (a) sus características arquitectónicas y mecanismos de sincronización que deben ofrecer, (b) cómo los lenguajes de programación permiten expresar la concurrencia en este tipo de máquinas y (c) cómo puede un compilador generar de forma automática tareas a ejecutar de forma concurrente.

1.3.1 Arquitectura

Los sistemas multiprocesadores son computadores constituidos por varios elementos de proceso, que cooperan de forma asíncrona en la ejecución de un programa.

Los elementos de proceso en un multiprocesador con memoria compartida se comunican a través de la memoria, a la cual pueden acceder mediante una red de interconexión [VSLL83]. La estructura general de este tipo de máquinas se muestra en la figura 1.2.

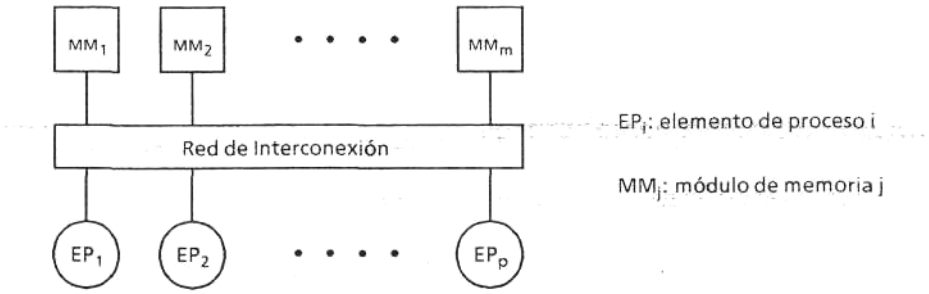

Figura 1.2: Estructura general de un sistema multiprocesador con memoria compartida.

Con la finalidad de reducir conflictos o interferencias en el acceso a los recursos compartidos, se utilizan técnicas tales como (a) añadir memoria local v a los elementos de proceso, (b) incorporar memoria "cache", (c) aumentar el número de módulos de memoria independientes, (d) el uso de redes de interconexión con muy poca degradación (o sea, que permitan pasar el máximo posible de peticiones a los módulos de memoria), entre otras.

Algunos de los computadores de este tipo son, entre otros, el Sequent Balance [Sequ86] que permite hasta 30 elementos de proceso, el BBN Butterfly [BBN86] que permite hasta 256 elementos de proceso o el Denelcor HEP [Kowa85] que permite hasta 16 elementos de proceso.

De entre los multiprocesadores con elementos de proceso vectoriales cabe destacar la familia CRAY X-MP que permite un máximo de 4 elementos de proceso, Alliant FX8 [Alli85] o el ETA-10 [ETAS86] que permiten hasta un máximo de 8 elementos de proceso.

Sincronización

Un aspecto importante a considerar en las arquitecturas multiprocesadores con memoria compartida es la sincronización entre elementos de proceso, en cuanto a que determina en gran parte la eficiencia en ejecución del código paralelo.

La mayor parte de sistemas multiprocesadores incorporan mecanismos básicos de sincronización implementados directamente en "hardware" y que son la base de otros mecanismos "software" más complejos.

La implementación "hardware" de estos mecanismos básicos se puede basar en la capacidad por parte de los módulos de memoria de ejecutar primitivas asociadas a dichos mecanismos sobre variables compartidas. Estas

primitivas se ejecutan de forma indivisible (normalmente se denominan *primitivas de lectura-modificación-escritura)* y de entre ellas caben destacar las primitivas *test&set(a)* y *reset(a)* (implementadas por ejemplo en el Sequent Balance) y cuya semántica es

```
test&set(a)
{
tmp = a;
a = 1;
return (tmp)
}
```

```
reset(a)
{
a = 0;
}
```

La ejecución de estas primitivas puede causar que el procesador quede en un estadode espera actiya, ejecutando, dichas primitivas hasta que se cumpla una determinada condición, incrementando por lo tanto el tráfico a memoria. Para evitar este problema se utilizan mecanismos de interrupción, de manera que el procesador queda parado hasta que se genera la interrupción adecuada a la primitiva de sincronización que ha ejecutado. Por ejemplo en [AyLa88] se describe su implementación para las primitivas de sincronización *wait (sem)* y *signal (sem, val)* sobre *semáforos* y cuya semántica es

```
wait (sem)
  {
  tmp = sem;
  if((sem>0)
       sem = sem -1;
   return (tmp);
}
```

```
signal (sem, val)
      {
      sem = sem + val;
      }
```

Si N procesos intentan acceder a una misma variable de sincronización de forma simultánea, el módulo de memoria debe ejecutar N operaciones básicas sobre dicha variable de forma secuencial. Para evitar este problema, existen primitivas de sincronización cuya ejecución se distribuye a través de la red de interconexión, de manera que la variable es accedida una única vez. Este mecanismo se implementa en el NYU Ultracomputer [Gott83] ofreciendo la primitiva *fetch&add (a, b)* cuya semántica asociada es

```
fetch&add (a, b)
{
tmp = a;
a = a + b;
return (tmp);
}
```

En el HEP se marcan determinadas variables compartidas, utilizando un bit que indica si dicha variable ha sido accedida para escribir o leer *(full/empty bits)*. Una operación de lectura sobre una variable marcada es válida cuando previamente ha sido modificaday marcada como “full”. Después de la lectura, dicha variable es marcada como "empty". De la misma manera, una escritura es válida cuando dicha variable ha sido previamente leida y marcada como "empty".

Otros sistemas, como el Alliant FX8, utilizan un bus y hardware especializado para realizar la sincronización entre procesadores. De esta manera se reduele el tiempo de acceso a las variables de sincronización y el tráfico a través de los buses del sistema para acceso a datos e instrucciones.

Otro mecanismo de sincronización de más alto nivel es la *barrera (barrier synchronization)*. En este caso, todos los procesos a sincronizar deben llegar a la barrera antes de que cualquiera de ellos pueda continuar su ejecución. Este mecanismo se puede implementar en "software", como primitiva de alto nivel soportada por otros mecanismos "hardware", o directamente en "hardware" [BePo89],[GuEp89].

1.3.2 Lenguajes de programación

Para sistemas multiprocesadores con memoria compartida existen dos tendencias en desarrollo de lenguajes paralelos:

(a) extender lenguajes secuenciales ya existentes con construcciones y primitivas que permitan expresar la concurrencia de un algoritmo (por ejemplo el CEDAR FORTRAN, VPG [Guar88a],...).

(b) proponer nuevos lenguajes que soporten la programación concurrente (por ejemplo Ada).

Variables

En ambos casos, se ofrece al programador la posibilidad de indicar cuando una variable es compartida o privada a un proceso, con la finalidad de que el compilador decida su ubicación en memoria.

Creación de tareas

También se ofrecen rutinas o construcciones del lenguaje que permiten crear tareas paralelas y sincronizar su ejecución.

Ejecución paralela de bucles

Estos lenguajes también ofrecen construcciones que permiten la ejecución paralela de iteraciones de un bucle. Las más utilizadas son DOALL y DOACROSS disponibles por ejemplo en el CEDAR FORTRAN. En un bucle DOALL todas las iteraciones del bucle pueden ser ejecutadas en paralelo y en cualquier orden. Sin embargo, en un bucle DOACROSS no puede iniciarse una determinada iteración del bucle hasta que todas las anteriores hayan sido iniciadas. Este último se utiliza cuando existen dependencias entre iteraciones del bucle.

Sincronización de tareas

En cuanto a posibilitar la sincronización entre tareas a ejecutar en distintos procesadores, estos lenguajes ofrecen primitivas del tipo *fetch_and_add, test_lock o wait* y *signal* sobre semáforos. También permiten la implementación de *secciones críticas* para garantizar el acceso único a variables compartidas.

1.3.3 Compiladores

La otra alternativa a la hora de programar estos sistemas es la utilización de compiladores capaces de extraer paralelismo de programas secuenciales.

El proceso de *paralelización* realizado por el compilador consiste en generar tareas que puedan ser ejecutadas sobre varios elementos de proceso, ya sea de forma independiente o apoyadas por mecanismos de sincronización.

Paralelización de bucles

Si se tiene en cuenta que el mayor potencial de paralelismo en programas numéricos se encuentra en bucles, el proceso de paralelización consiste en distribuir iteraciones del bucle entre elementos de proceso, añadiendo sincronización entre tareas generadas cuando sea necesario.

Así, por ejemplo, para el siguiente bucle secuencial

```
DO I = 1,100
    A(I) = B(I) * C(I)
ENDO
```

el compilador podría detectar que todas las iteraciones son independientes y que pueden ejecutarse en cualquier orden, generando el siguiente código paralelo

```
DOALLI = 1, 100
    A(I) = B(I) * C(I)
ENDOALL
```

Distribución de tareas

Otra tarea importante realizada por estos compiladores es el *"scheduling"* o *distribución de las tareas* generadas entre los elementos de proceso del sistema.

Esta distribución puede realizarse (a) en tiempo de compilación [P0KP86] asignando iteraciones del bucle a procesadores de forma estática o (b) en ejecución de forma dinámica, con la finalidad de equilibrar la carga asignada a los procesadores.

Algunas soluciones propuestas para la segunda alternativa son (a) Auto_distribución *(self_scheduling*[TaYe86]) consistente en que cada procesador, al finalizar el trabajo actual, entra en una sección crítica a fin de determinar la siguiente iteración o grupos de iteraciones a ejecutar o (b) auto_distribución controlada (guided *self_scheduling* [PoKu87]), consistente en reducir el tamaño de los grupos de iteraciones que los procesadores deben realizar a medida que quedan menos iteraciones del bucle a ejecutar, con la finalidad de evitar que determinados procesadores tengan que esperar mucho tiempo a que otros acaben el trabajo asignado.

1.4 PRESENTACIÓN DE LA TESIS Y APORTACIONES

Este trabajo se centra en el estudio y propuesta de técnicas de paralelización de bucles en programas secuenciales numéricos. Los principales problemas tratados son la existencia de recurrencias y sentencias condicionales. En este

trabajo se consideran únicamente bucles con un grado de anidación, o en caso contrario, únicamente el bucle más interno.

En el capítulo 2 se repasan las técnicas que permiten (a) realizar el análisis de las relaciones de dependencia entre sentencias de un bucle y (b) la reestructuración de los mismos a fin de obtener código paralelo que pueda ejecutarse de forma eficiente en sistemas multiprocesadores y computadores vectoriales segmentados.

La evaluación del paralelismo de un bucle es considerada en el capítulo 3 y se realiza a partir del grafo de dependencias entre sentencias obtenido en tiempo de compilación. El paralelismo evaluado es una buena medida de la eficiencia del proceso de reestructuración realizado.

Se propone un método, *Graph Traverse Scheduling* (GTS [L"aAy89], [ALTB89]) que incorporado en un compilador permite la extracción del máximo paralelismo del bucle. La distribución de operaciones realizada sé basa en recorridos a través de un ciclo del grafo de dependencias que cumple unas determinadas características.

La aplicación de GTS para multiprocesadores con memoria compartida se presenta en el capítulo 4 y permite la obtención de tareas independientes o sincronizadas según las dependencias existentes. Uno de los temas considerados es la reducción del número de sincronizaciones explícitas añadidas, así como el compromiso entre paralelismo obtenido y sincronización.

La aplicación de GTS para máquinas vectoriales se presenta en el capítulos y permite la obtención de instrucciones vectoriales de máxima longitud incluidas en bucles secuenciales.

GTS puede ser aplicado a otras arquitecturas como multiprocesadores con memoria distribuida y máquinas VLIW, aunque estos temas no han sido considerados en este trabajo.

En el capítulo 6 se evalúan los parámetros que indican la concurrencia de un bucle y se comparan los resultados obtenidos por GTS con los obtenidos aplicando otras técnicas de reestructuración ya existentes. Los resultados obtenidos se basan en datos obtenidos a partir de grafos de dependencia aleatoriamente generados.

El capítulo 7 concluye el trabajo y presenta las líneas abiertas de investigación.