Capítulo 2:
TÉCNICAS DE PARALELIZACIÓN AUTOMÁTICA

En este capítulo se describen algunas de las técnicas más importantes utilizadas en el proceso de paralelización automática de bucles en programas secuenciales.

En la sección 1 se describe brevemente el proceso de paralelización y las fases involucradas en el mismo. En la sección 2 se estudian las relaciones de dependencia entre sentencias, cuyo análisis constituye una de las fases iniciales de todo el proceso de paralelización, y su representación en un grafo de dependencias en el cual se basan las técnicas de paralelización descritas en la sección 3. En la última sección de este capítulo se describen las consideraciones adicionales a realizar cuando aparecen sentencias condicionales en el bucle.

2.1 PROCESO DE PARALELIZACIÓN

El proceso de paralelización consiste en transformar un programa secuencial en una nueva versión concurrente semánticamente equivalente. Este proceso puede ser realizado:

(a) de forma automática por el compilador, intentando obtener tareas o operaciones que puedan realizarse de forma paralela;

(b) manualmente programando el algoritmo en un lenguaje de programación paralelo que permita especificar tareas u operaciones a ejecutar de forma concurrente.

En un programa secuencial el paralelismo puede detectarse a 3 niveles:

(a) a nivel de procedimientos: en este caso se ejecutan varias llamadas a procedimientos de forma simultánea.

(b) a nivel de bucles: en este caso se ejecutan varias iteraciones y/o sentencias del bucle en paralelo.

(c) a nivel de bloque básico: en este caso se ejecutan en paralelo varias operaciones de un bloque de sentencias de asignación. Se utiliza para arquitecturas VLIW.

En este trabajo se estudia la detección de paralelismo a nivel de bucles, ya que es en ellos donde los programas numéricos consumen la mayor parte del tiempo de ejecución. Se pueden distinguir dos tipos básicos demueles, que debido a sus características, obligan a técnicas de paralelización distintas:

(a) bucles fijos: cuyo rango de iteración (valores inicial y final e incremento de la variable de control del bucle) es conocido en tiempo de compilación.

(b) bucles no fijos: cuyo rango de iteración se determina durante la ejecución del bucle y que por lo tanto, es desconocido en tiempo de compilación.

Las técnicas de paralelización descritas y presentadas en este trabajo están orientadas al primer tipo de bucles, no entrando en la problemática que presentan los segundos [Wolf89].

El proceso de paralelización se realiza a nivel de programa fuente en alto nivel, generando una versión equivalente paralela también en alto nivel. El lenguaje de alto nivel utilizado es indiferente aunque los ejemplos que aparecen a lo largo del trabajó están codificados en FORTRAN.

A partir del programa original se realiza un detallado análisis de las dependencias entre sentencias del bucle. Estas dependencias determinan un orden parcial de ejecución de las operaciones del bucle que debe ser preservado en la nueva versión generada. De este análisis se obtiene una representación interna, denominada grafo de dependencias, a partir de la cual se realiza todo el proceso de paralelización.

Dentro de las técnicas de paralelización existen aquellas que son independientes de la arquitectura de la máquina sobre la cual va a ejecutarse el programa (técnicas orientadas a separar partes paralelizables de secuenciales, técnicas orientadas a aumentar el paralelismo del programa,...). Por otro lado existen técnicas específicas para un determinado tipo de arquitectura (máquinas vectoriales, multiprocesadores escalares, computadores VLIW, ...). Por último también existen técnicas específicas para una determinada máquina orientadas a utilizar al máximo las características que ésta ofrece. En esta tesis sólo son consideradas técnicas generales no específicas para ninguna máquina comercial o experimental en concreto.

2.2 RELACIONES DE DEPENDENCIA ENTRE SENTENCIAS

En esta sección se estudian las relaciones de dependencia que aparecen entre sentencias de un bucle que establecen un orden parcial para estas operaciones.

El análisis dé dependencias lo realiza y utiliza el compilador con la finalidad de aplicar determinadas técnicas de reestructuración de código y obtener una versión equivalente del programa que pueda ser ejecutada de forma eficiente sobre una determinada arquitectura. Esta equivalencia es a nivel semántico, es decir, ambas versiones deben generar los mismos resultados.

Es importante que el compilador no detecte relaciones de dependencia inexistentes, ya que éstas pueden perjudicar el proceso de paralelización de código. Sin embargo, en caso de no disponer de información suficiente en

tiempo de compilación, deberá apostar por su existencia, ya que, en caso contrario, el código generado podría no ser semánticamente equivalente. Por lo tanto se requieren métodos que realicen un análisis de dependencias lo más preciso posible, aunque en determinados casos conservativo.

Podemos clasificar las relaciones de dependencia entre sentencias en dos grupos:

(a) *dependencias debidas a datos:* el acceso a una determinada variable escalar o estructurada por parte de las sentencias fuerza un orden parcial de ejecución de las mismas.

(b) *dependencias de control:* la ejecución de una sentencia está condicionada por el resultado obtenido en la ejecución de otra previa.

2.2.1 Dependencias de datos

A continuación se analiza con más detalle cuando aparecen relaciones de dependencia debidas al acceso a datos.

La figura 2.1(a) muestra un bucle secuencial constituido por dos sentencias y la figura 2.1(b) parte de su ejecución secuencial desarrollada. En la ejecución secuencial desarrollada se puede observar que un determinado elemento del vector A modificado en la sentencia S_1 es posteriormente utilizado dentro de la misma iteración por la sentencia S_2. Dado que S_2 requiere del nuevo valor asignado a .dicho elemento* su orden de ejecución-no puede- ser modificado. Cuando esta situación ocurre se dice que aparece una relación de dependencia de datos entre las sentencias S_1 y S_2. Se denomina sentencia destino de la dependencia a la sentencia cuya ejecución debe ser retardada y sentencia fuente de la dependencia a la sentencia que permite su ejecución.

Tipos de dependencias

En la literatura [Kuck78] se distinguen diferentes tipos de relación de dependencia de datos en función del orden de las operaciones de lectura y/o escritura sobre una determinada variable que realizan las sentencias involucradas en dicha relación de dependencia. Dadas dos sentencias S_i y S_i incluidas en el cuerpo de un bucle, precediendo una instanciación de Si a otra de S_i en la ejecución secuencial del mismo, se distinguen los siguientes tipos de dependencias de datos:

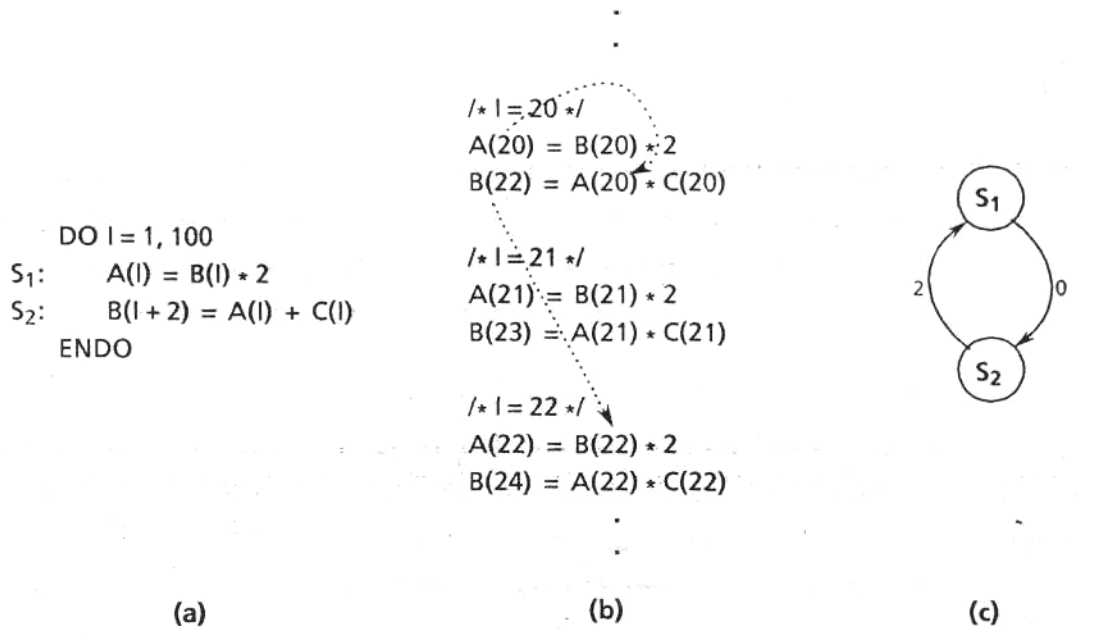


Figura 2.1: (a) Bucle secuencial, (b) ejecución desarrollada y (c) grafo de dependencias.

(a) *dependencia de flujo de datos (flow-dependence):* S_i modifica una variable que posteriormente puede ser consultada por S_j. Este tipo de dependencia se denota con $S_i \delta S_j$.

(b) *antidependencia (anti-dependence):* S_i consulta una variable que posteriormente puede ser modificada por S_j. Este tipo de dependencia se denota con $S_i \delta\text{-}S_j$.=

(c) *dependencia de salida (output-dependence):* ambas sentencias S_i y S_j pueden modificar el valor de una misma variable. Este tipo de dependencia se denota con $S_i \delta^o S_j$.

En realidad la única relación de dependencia inherente al algoritmo es la primera de ellas. Las otras dos se denominan dependencias añadidas y son debidas al tipo de lenguaje utilizado para especificar el algoritmo. Deben considerarse cuando el lenguaje de programación permite redefinir el valor de una variable. Así por ejemplo, en lenguajes de asignación única (SISAL [Live85],...) estas dependencias nunca aparecen. Estas dependencias añadidas son evitables y pueden ser eliminadas utilizando algunas de las técnicas que se comentan en la siguiente sección.

En la notación que se utiliza en este trabajo no se distingue el tipo de dependencia y se denota en cualquier caso con $S_i \delta S_j$.

Distancia

Una relación de dependencia puede aparecer entre instanciaciones de sentencias en iteraciones distintas. Así por ejemplo, la dependencia de la figura 2.1 entre S_2 y S_1 aparece debido a que un determinado elemento del vector B modificado en S_2 es consultado dos iteraciones más tarde por S_1.

Se denomina *distancia* [PaKL80] al número de iteraciones que cubre una relación de dependencia. Así por ejemplo en la figura 2.1 aparecen las relaciones de dependencia de flujo de datos entre S_1 y S_2 con distancia asociada 0 (dado que S_2 utiliza el valor en la misma iteración en que S_1 lo produce) y entre S_2 y S_1 con distancia asociada 2 (dado que S_1 consulta el valor dos iteraciones después de haberlo producido S_2).

2.2.2 Dependencias de control

La figura 2.2 muestra un bucle secuencial que incluye una sentencia estructurada condicional. En este caso, la ejecución de las sentencias S_3 y S_4 está condicionada al resultado obtenido en la ejecución de la sentencia S_2.

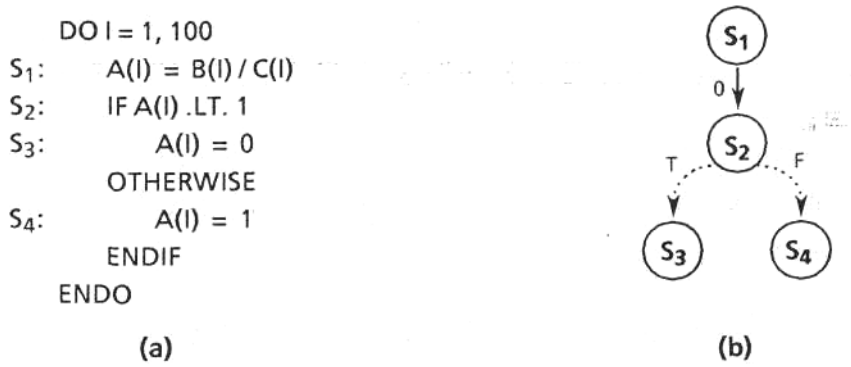


Figura 2.2: (a) Bucle con sentencia estructurada condicional y (b) grafo de dependencias asociado.

En general, se dice que existe una relación de dependencia de control entre la sentencia que evalúa la condición y todas las sentencias cuya ejecución dependen del resultado de dicha evaluación. Esta relación se denota con $S_i \delta^C S_j$.

Este tipo de relación de dependencia lleva siempre asociada una distancia 0, pues la ejecución de S_j está condicionada por el resultado de evaluar la condición necesaria en Si en la misma iteración.

En el caso de sentencias condicionales anidadas se considera que las dependencias de control son transitivas, es decir, si $S_i \delta^c S_j$ y $S_j \delta^c S_k$ también se cumple que $S_i \delta^c S_k$.

2.2.3 Grafo de dependencias

Las técnicas de reestructuración de código que se describen en esta tesis se basan en una representación abstracta de las sentencias del bucle y las relaciones de dependencia que existen entre ellas. Esta representación se denomina *grafo de dependencias* [Bane79]. El grafo de dependencias es un grafo direccional G (V, E) definido por el conjunto de nodos V, que representa a las sentencias del bucle

$$V = \{S_i \mid 1 \leq i \leq n\},$$

y por el conjunto de arcos E, que representa las relaciones de dependencia entre sentencias del bucle

$$E = \{d_{ji} \mid S_i, S_j \in V \wedge S_i \delta S_j\}.$$

Cada arco del grafo d_{ij} lleva asociada la distancia d_{ij} que define a la relación de dependencia que representa.

Con la finalidad de simplificar la notación, se suponerla existencia de un único arco entre dos nodos cualesquiera del grafo.

Tipos de arcos y etiquetado

En el grafo de dependencias se representan con arcos distintos los diferentes tipos de relación de dependencia que existen entre sentencias, tal como muestra la figura 2.3.

En el grafo de dependencias, los arcos asociados a dependencias de datos se etiquetan con la distancia asociada. Los arcos de control se etiquetan con el valor de la condición evaluada en la sentencia fuente que determina la ejecución de la sentencia destino.

Las figuras 2.1(c) y 2.2(b) muestran los grafos de dependencia asociados a los bucles respectivos.

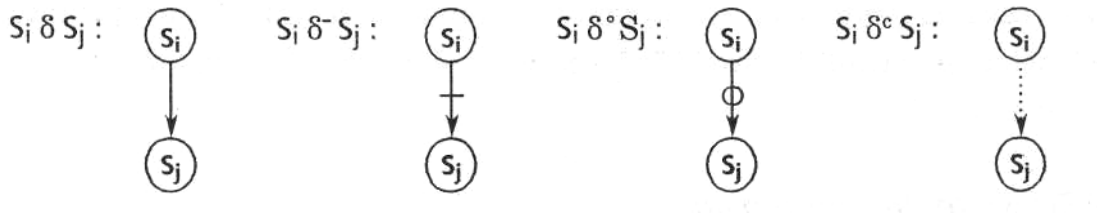


Figura 2.3: Representación de los tipos de dependencia en el grafo de dependencias.

Bucles anidados

Para el caso de bucles anidados, las relaciones de dependencia se etiquetan con un vector de distancias, representando cada elemento de este vector la distancia asociada a cada dimensión del bucle. La figura 2.4 muestra un bucle secuencial anidado y el grafo de dependencias asociado. En este caso todas las componentes del vector de distancias excepto la primera (correspondiente al bucle más externo) pueden tomar valores negativos, ya que el orden de ejecución secuencial supone que, para cada iteración de un bucle externo se ejecutan todas las iteraciones de los bucles internos.

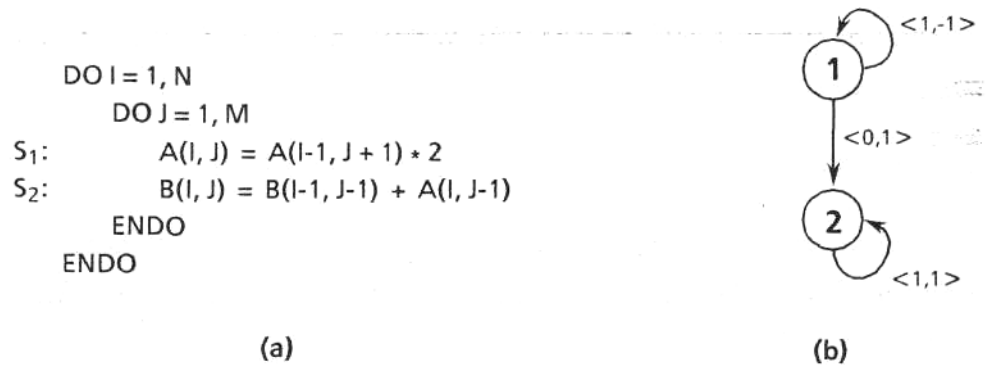


Figura 2.4: (a) Bucle secuencial anidado y (b) grafo de dependencias asociado.

Ramas de ejecución

En el caso de existir sentencias estructuradas condicionales se tendrán varias *ramas de ejecución,* entendidas como el subconjunto de sentencias del bucle que se ejecutan según las condiciones evaluadas. Para cada rama de ejecución

se define un *subgrafo de dependencias* G_s (V_s, E_s) en el que V_s representa el conjunto de sentencias incluidas en ella y E_s las relaciones de dependencia entre ellas. En la sección 3.4 se trata la obtención de este subgrafo para las posibles ramas de ejecución.

2.2.4 Detección de dependencias

La efectividad de cualquier técnica de reestructuración de código, basada en el estudio de las relaciones de dependencia, viene condicionada por la capacidad del compilador de realizar un buen análisis de las mismas.

Determinar la existencia de una relación de dependencia debida a variables escalares es inmediato, en cuanto a que se reduce a comprobar si dos sentencias del bucle utilizan esa misma variable de forma directa.

En el caso de variables estructuradas tipo vector, la detección de dependencias requiere un análisis de las expresiones de indexación del vector a fin de determinar si un mismo elemento de la estructura es referenciado por dos sentencias del bucle. Así, si $f_i(x)$ y $f_j(x)$ son las funciones de indexación que determinan el acceso a los elementos de una misma variable estructurada en sentencias S_i y S_j, existirá dependencia $S_i \, \delta \, S_j$ si

$$\exists x_a, x_b \mid (1 \leq x_a \leq x_b \leq N \wedge f_i(x_a) = f_j(x_b)) .$$

La distancia asociada a la relación de dependencia queda determinada por

$$D_{ij} = x_b - x_a$$

La condición anterior conlleva a una ecuación que relaciona x_a y x_b. Su resolución implica el encontrar una posible solución dentro del rango de iteración definido. Existen varios métodos descritos en la literatura [Bane79], [Wolf89] que consideran el análisis para una única dimensión de una estructura regular de datos, conociéndose además en tiempo de compilación el rango de valores que tomará la variable de control del bucle. Para estructuras multidimensionales, cada dimensión se analiza por separado. Estos métodos, para el caso de funciones de indexación complejas (que por ejemplo relacionan varias variables de control de los bucles) dan resultados muy conservativos, es decir, que encaso de duda apuestan por la existencia de dependencia.

En [LÍYZ89] se extienden los métodos anteriores a fin de subsanar los problemas que aparecen en el caso de estructuras de datos multidimensionales. Este método considera todas las dimensiones de forma

simultánea a fin de determinar la existencia de dependencia, obteniendo en general resultados más precisos.

Para el caso de funciones de indexación y rangos de iteración no conocidos en tiempo de compilación, se utilizan métodos que en tiempo de ejecución determinan si una iteración en particular depende o no de otras iteraciones [Poly88].

Otro problema importante en el análisis de dependencias es el acceso a variables de forma indirecta mediante punteros, lo que no suele ser frecuente en el caso de programas numéricos. En este caso, es difícil determinar cuando dos sentencias acceden a las mismas posiciones de memoria [Guar88b].

2.2.5 Notación y definiciones

Á continuación se definen algunos términos y notación que serán utilizados a lo largo de esta tesis.

Dado un grafo de dependencias G (V, E), se define *camino o cadena* C_{ij} como el conjunto de arcos distintos del grafo $d_{kl} \in E$ que generan una dependencia directa o indirecta entre las sentencias S_i y S_j del bucle. El *peso de un camino* w (C_{ij}) queda definido por la suma de las distancias asociadas a los arcos que lo componen, es decir

$$w(C_{ij}) = \sum_{d_{kl} \in C_{ij}} d_{kl}.$$

Se define el *camino transformado* C_{ij}* como el conjunto de sentencias S_m € V incluidas en un camino C_{ij}. La *longitud de un camino* 1 (Cij) queda definida por la cardinalidad del conjunto C_{ij}*.

Cuando se quiere considerar la influencia del tiempo de ejecución de las sentencias, se asocia a cada nodo del grafo $S_m \in V$ un peso directamente relacionado con el tiempo de ejecución de la sentencia que representa. En este caso se define la longitud de un camino C_{ij} como

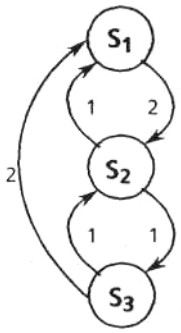
$$l(C_{ij}) = \sum_{S_m \in C^*_{ij}} S_m.$$

Se define *recurrencia* R como un ciclo o camino cerrado C_{ii} en el grafo, de manera que cada sentencia del conjunto C_{ii}* aparece una única vez. El *peso de una recurrencia* viene dado por

$$w(R) = \sum_{d_{kl} \in C_{ii}} d_{kl} .$$

Se denomina *recurrencia hamiltoniana* a una recurrencia R del grafo G (V, E) que cubre todos sus nodos.

Por ejemplo, en el siguiente grafo de dependencias



tenemos, entre otros, el camino $C_{31} = \{d_{32}, d_{21}\}$ con peso asociado $w(C_{31}) = d_{32} + d_{21}=2$. El camino transformado $C_{31}^* = \{S_1, S_2, S_3\}$ y por lo tanto, la longitud del camino será $l(C_{31}) = 3$. En este grafo aparecen tres recurrencias, $R_1 = \{d_{12}, d_{21}\}$, $R_2 = \{d_{23}, d_{32}\}$, $R_3 = \{d_{12}, d_{23}, d_{31}\}$, cuyos pesos asociados son 3, 2 y 5 respectivamente. De ellas sólo R_3 es recurrencia hamiltoniana.

En [BCKT79] se define *ciclo máximal* como un ciclo del grafo que no queda incluido en otro ciclo del grafo. Esta definición no obliga a que cada nodo del grafo sea visitado una única vez, condición básica en la definición de recurrencia hamiltoniana. Así por ejemplo, en el grafo anterior, el ciclo formado por {día; d23, d32, d2l} es un ciclo máximal. También se define punto *aislado* del grafo como un nodo que no pertenece a ningún ciclo del grafo.

Sea B el conjunto de recurrencias R de un grafo G (V, E). Se dice que G es un *grafo direccional acíclico* (GDA) cuando $B = \{\varnothing\}$, es decir

$$\forall d_{ij} \in E \mid \exists C_{ij}$$

Por el contrario se dice que G es un *grafo direccional cíclico* (GDC) cuando posee al menos una recurrencia. Un GDC es hamiltoniano cuando existe al menos una recurrencia $R \in B$ hamiltoniana.

2.3. TÉCNICAS DE PARALELIZACIÓN

En esta sección se describen algunas de las técnicas orientadas a extraer o facilitar la extracción de paralelismo de bucles tanto para sistemas

multiprocesadóres como para máquinas vectoriales. Todas ellas se basan en el grafo de dependencias obtenido previamente por el compilador.

En los apartados 1.2.3 y 1.3.3 se han descrito las técnicas de vectorización total y paralelización DOALL respectivamente como técnicas triviales aplicables al caso de bucles con iteraciones independientes:

El primer grupo de técnicas descritas en esta sección son independientes de la arquitectura y están orientadas a facilitar la extracción de concurrencia:

- Distribución de bucles;
- Intercambio de bucles;
- Eliminación de dependencias.

El segundo grupo de técnicas están orientadas a la generación de código paralelo para sistemas multíprocesador y en general a resolver el problema de la existencia de recurrencias en bucles secuenciales:

- DOPIPE;
- DOACROSS;
- Particionado Parcial;

  Contracción de Ciclos;

Por último se consideran técnicas de reconocimiento de patrones o recurrencias específicas y generación de código vectorial o paralelo para ellas.

2.3.1 Distribución de bucles

La técnica de *distribución de bucles* [BCKT79] pretende distribuir sentencias de un bucle en bloques de sentencias sóbrelos cuales poder aplicar alguna técnica de paralelización. Los bloques generados pueden ser dependientes, en cuyo caso se ejecutarán de forma secuencial, o independientes, en cuyo caso se podran ejecutar en paralelo.

A partir del grafo de dependencias entre sentencias G (V , E) se obtienen un conjunto de π-*blocks,* constituidos bien por ciclos maximales del grafo (entendidos como los nodos del grafo que forman parte de un ciclo no incluido

en un ciclo mayor), o bien por puntos aislados (entendidos como nodos del grafo que no forman parte de ninguna recurrencia).

El conjunto de todos los π-blocks generados es una partición del conjunto V en subconjuntos disjuntos que se denotan ni. El conjunto de arcos E define las relaciones de orden parcial entre π-blocks. Dados π_i y π_j se tiene que

$$\pi_i \Delta \pi_j \text{ si } \exists (S_m \in \pi_i \wedge S_n \in \pi_j) | S_m \delta S_n$$

denotando Δ la relación de dependencia entre π-blocks. Si no se cumple la condición anterior, ambos π-blocks pueden ser ejecutados en paralelo o en cualquier orden.

Si se cumple la condición anterior se debe asegurar que π_j se ejecute secuencialmente después de π_i. Para ello se puede aplicar la técnica de *reordenación de sentencias* [Poly88]. La finalidad de aplicar la técnica de reordenación de sentencias es conseguir que todos los arcos del grafo de dependencias vayan en un determinado sentido, que es el que indica el orden en que van a ser ejecutadas las sentencias.

La figura 2.5 muestra un bucle secuencial y su grafo de dependencias asociado. En él se puede observar la existencia de dos π-blocks: $\pi_1 = \{S_1, S_2\}$ y $\pi_2 = \{S_3\}$ relacionados por $\pi_1 \Delta \pi_2$.

La técnica de distribución de bucles pretende separar partes del bucle que pueden ser vectorizadas o paralelizadas de otras partes que en principio deben ejecutarse de forma secuencial.

Vectorización

La técnica de *vectorización parcial* [Wolf 89] pretende vectorizar sólo aquellas sentencias incluidas en π-blocks acíclicos, generando para ellas sentencias vectoriales.

Las sentencias incluidas en π-blocks cíclicos se ejecutan de forma secuencial. Algunos compiladores utilizan técnicas adicionales, que se describen en esta misma sección, para tratar a los π-blocks cíclicos. Estas técnicas pretenden (a) eliminar ciclos de dependencia o (b) reconocer patrones para los cuales es capaz de generar código vectorial especial.

El resultado de vectorizar el bucle de la figura 2.5(a) aplicando distribución de bucles se muestra en la figura 2.5(c). Observar que el código generado para el ni cíclico es secuencial.

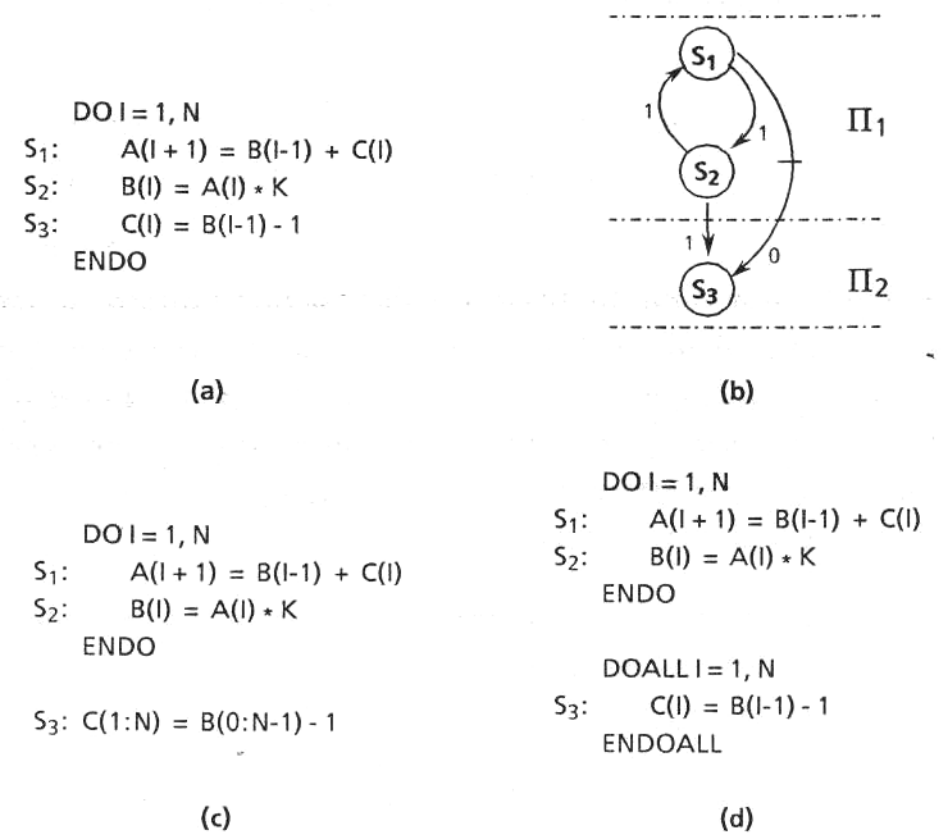


Figura 2.5: (a)Bucle secuencial y (b) n-blocks obtenidos tras aplicar distribución de bucles, (c) Código vectorial obtenido al aplicar vectorización parcial y (d) paralelización DOALL

La técnica de distribución de bucles puede también aplicarse para la vectorización de bucles anidados [Wolf89]. Existen varios métodos para ello que consisten en explorar cada uno de los bucles existentes. Una posibilidad es empezar por analizar el bucle más interno y acaba por el más externo. La figura 2.6(a) muestra un bucle secuencial constituido por dos bucles anidados. El grafo de dependencias para el bucle interno se muestra en la figura 2.6(b) y en él puede observarse la existencia de tres π-blocks acíclicos. Por lo tanto es posible aplicar vectorización obteniendo el código de la figura 2.6(c). A continuación se analiza el grafo de dependencias para el bucle externo (figura 2.6(d)) en que se observa un π-block cíclico constituido por las sentencias Si y

S_2 y otro acíclico constituido por S_3. Este último podrá ser vectorizado obteniendo el código vectorial de la figura 2.6(e).

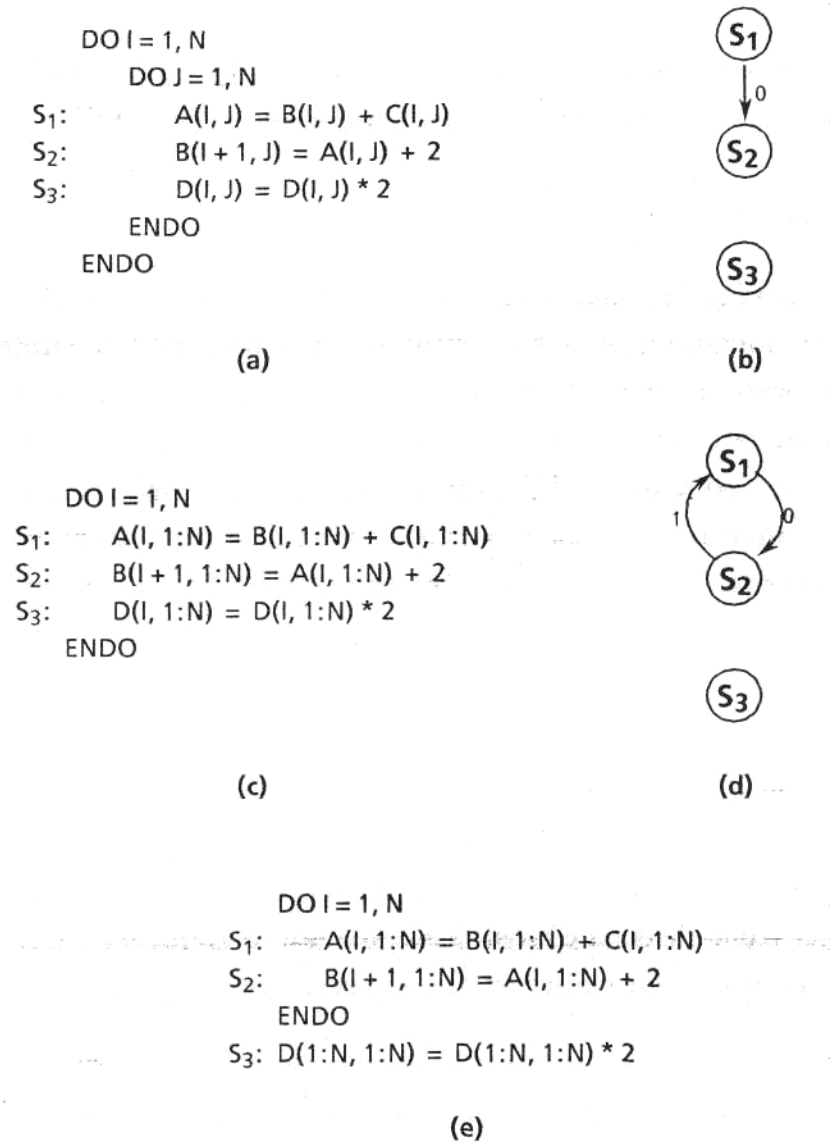


Figura 2.6: Vectorización de bucles anidados empezando por el bucle más interno.

Paralelización

La técnica de distribución de bucles también permite la generación de bucles paralelos DOALL para aquellas sentencias que quedan incluidas en n-blocks acíclicos. Si no es el caso, se aplican otros métodos para paralelizar recurrencias, algunos de los cuales se describen en esta misma sección.

En el caso de multiprocesadores vectoriales se pueden aprovechar los dos niveles de paralelismo que ofrecen (ejecución vectorial de los π-blocks acíclicos y ejecución paralela sobre distintos elementos de proceso de los π-blocks independientes).

El resultado de paralelizar el bucle de la figura 2.5(a) aplicando distribución de bucles se muestra en la figura 2.5(d) respectivamente.

2.3.2 Intercambio de bucles

La técnica de *intercambio de bucles* pretende, en el caso de bucles anidados, cambiar el orden en que los bucles son ejecutados con la finalidad de aumentar la eficiencia del proceso de paralelización. En [AIKe84] se describe la técnica formalmente y su implementación en el compilador PFC de la Rice University para bucles anidados perfectos. En [Wolf86] se describe su ampliación al caso de bucles anidados imperfectos. Esta descripción formal se basa en el grafo de dependencias y presenta las condiciones que permiten realizar el intercambio de forma correcta.

Vectorízación

Esta técnica puede ser utilizada para encontrar bucles vectorizables dentro de un conjunto de bucles anidados. En este caso interesa vectorizar el bucle más interno. Por ejemplo, el bucle interno de la figura 2.7(a) no es vectorizable. Si se intercambia el orden de.ejecuciónse tiene que ahora el bucle interno si lo es permitiendo la generación de código vectorial mostrada en la figura 2.7(c).

También puede aplicarse intercambio de bucles para obtener como bucle más interno aquel bucle que genere instrucciones vectoriales de máxima longitud. La figura 2.8(a) muestra un bucle secuencial al cual se le puede aplicar intercambio de bucles. Si el bucle más interno es el bucle I, las instrucciones vectoriales serán de longitud 10 mientras que si lo es el bucle J lo serán de longitud 100. Esta aplicación puede ser interesante en función de los parámetros que definen al procesador vectorial. La figura 2.8(b) muestra el código vectorial generado.

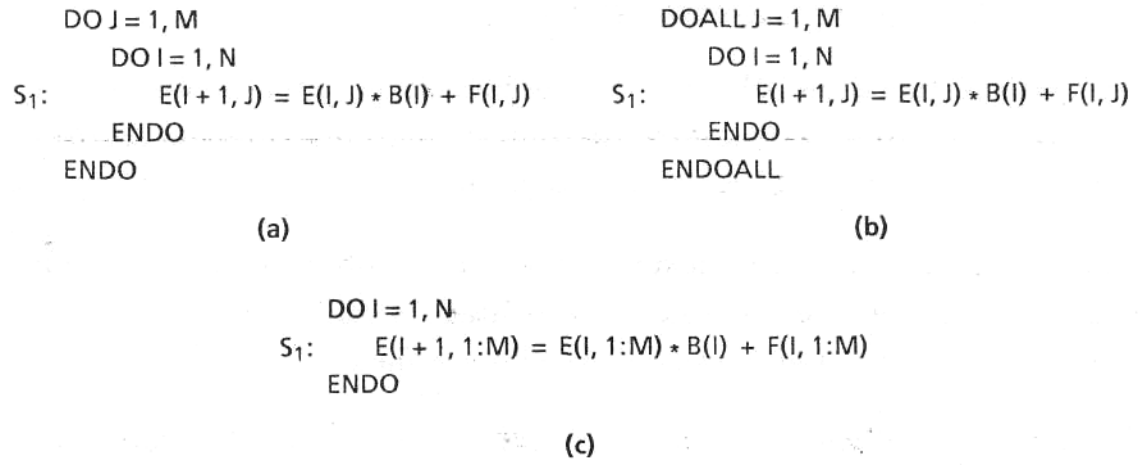


Figura 2.7: Intercambio de bucles para buscar bucles paralelizables (b) o vectorizables (c).

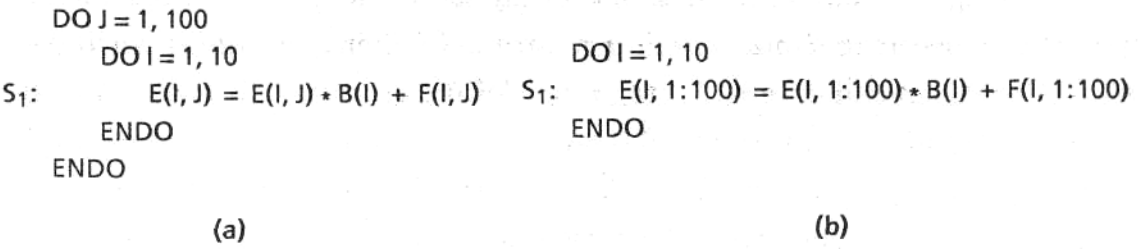


Figura 2.8: Intercambio de bucles para obtener instrucciones vectoriales de máxima longitud.

Paralelización

Por el contrario, en el caso de paralelizarinteresa tener como bucle más externo aquel que tenga un paralelismo mayor. Así por ejemplo, para el bucle de la figura 2.7(a) no se aplicaría intercambio de bucles y se generaría el código mostrado en la figura 2.7(b).

2.3.3 Eliminación de dependencias

En este apartado se describen algunas técnicas que permiten eliminar o modificar determinadas relaciones de dependencia del bucle. Las técnicas que van a describirse son las siguientes [KKPL81] [PaWo86]:

- *Cambio de nombre* (scalar renaming) y *expansión* escalar (scalar expansión).
- *Refinamiento de nodos* (node splitting).

2.3.3.1 Cambio de nombre y Expansión escalar

Estas técnicas permiten eliminar dependencias de datos debidas a la reutilización de variables escalares del bucle. Esta reutilización provoca antidependencias y dependencias de salida evitables.

La técnica de *cambio de nombre* asocia un nombre distinto en cada sentencia que define la variable escalar reutilizada, modificándolo también en todas aquellas sentencias que usan la definición asociada.

La técnica de *expansión escalar* convierte en vector aquellas variables escalares a las que se asigna un nuevo valor en cada iteración del bucle.

La figura 2.9 muestra un ejemplo en el que se pueden aplicar estas dos técnicas. La figura 2.9(a) es la versión original del bucle junto con su grafo de dependencias asociado. Notar que la primera definición y uso de la variable escalar U es independiente de la segunda definición y uso. Si aplicamos la técnica de cambio de nombre se obtiene el bucle y grafo de dependencias reducido de la figura 2.9(b). Las nuevas variables U1 y U2 pueden ser extendidas a vector, aplicando la técnica de extensión escalar, de manera que cada iteración utilice una variable distinta. El bucle resultante y grafo de dependencias se muestran en la figura 2.9(c) que sí puede ser vectorizado y paralelizado aplicando distribución de bucles.

2.3.3.2 Refinamiento de nodos

Algunos bucles presentan recurrencias que pueden ser eliminadas mediante copia temporal de alguna de las variables involucradas en antidependencias o dependencias de salida. La figura 2.10(a) muestra un bucle secuencial y grafo de dependencias asociado. La antidependencia que aparece puede ser eliminada insertando una nueva sentencia; de asignación a una variable temporal tal como se muestra en la figura 2.10(b). En el nuevo grafo de dependencias ha desaparecido la recurrencia que impedía la vectorización o paralelización del bucle.

La técnica de refinamiento de nodos puede también utilizarse para eliminar dependencias dé salida en recurrencias naciendo copias temporales intermedias de las variables que las ocasionan. La figura 2.11 muestra un ejemplo de aplicación.

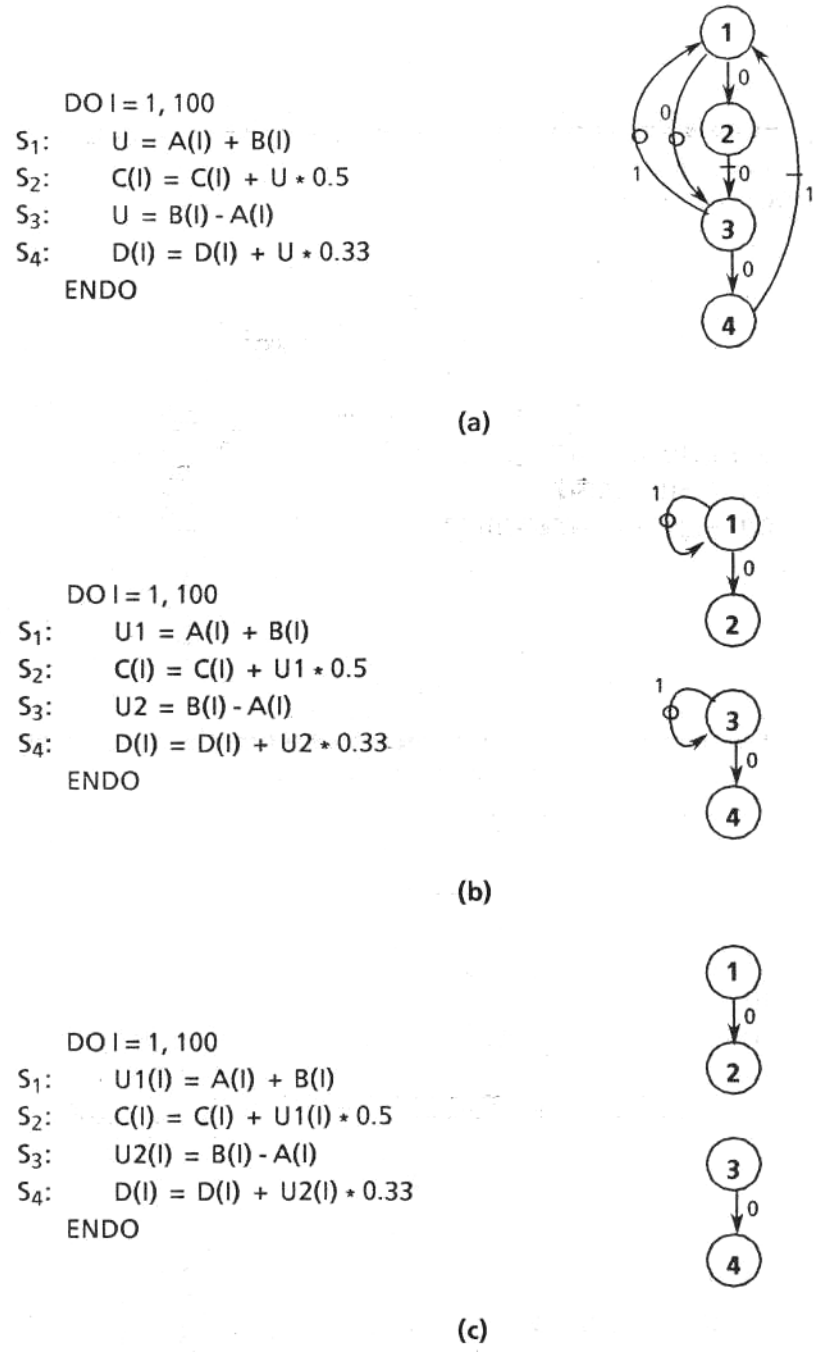


Figura 2.9: Ejemplo de aplicación de las técnicas de cambio de nombre (b) y expansión escalar (c) para eliminar antidependencias y dependencias de salida.

2.3.4. DOPIPE

DOPIPE [Davi81] es una técnica que no pretende paralelizar π-blocks cíclicos sino solapar al máximo su ejecución secuencial con la ejecución del resto de π-blocks, reduciendo por otro lado el número de procesadores necesarios para

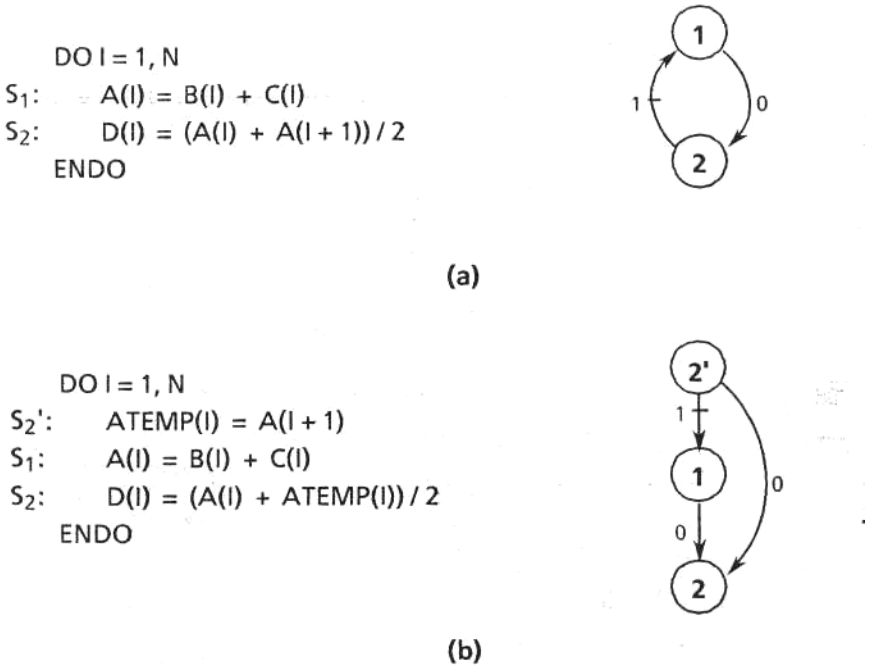


Figura 2.10: Ejemplo de aplicación de la técnica de refinamiento de nodos para eliminar antidependencias.

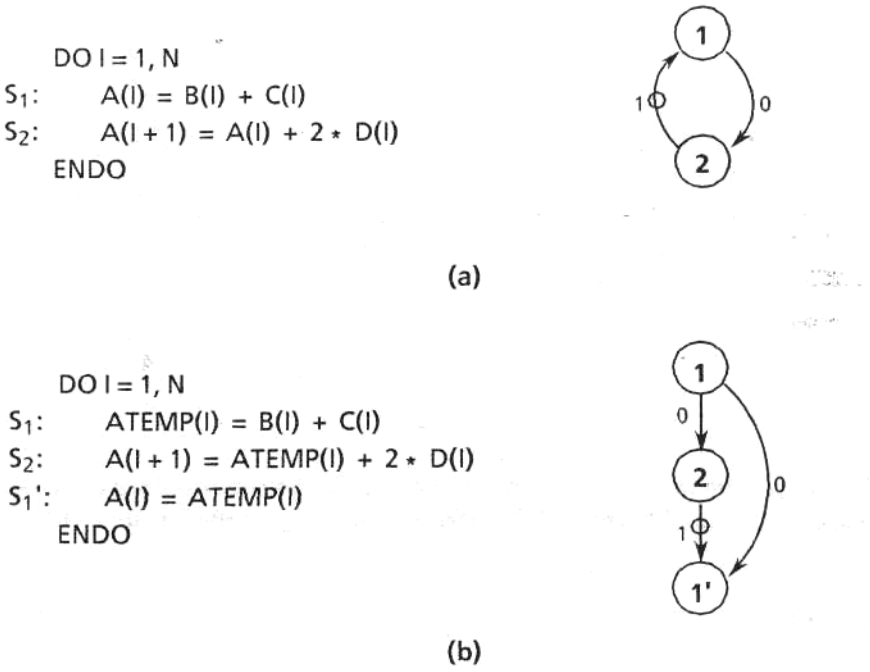


Figura 2.11: Ejemplo de aplicación de la técnica de refinamiento de nodos para eliminar dependencias de salida.

ello. DOPIPE necesita introducir sincronización a fin de asegurar el correcto solape en la ejecución de los π-blocks.

Este método distribuye las sentencias del bucle en *segmentos* que estarán constituidos bien por π-blocks cíclicos o bien por grupos de π-blocks acíclicos.

La figura 2.12(a) muestra un bucle secuencial y su grafo de dependencias asociado. En él puede observarse la existencia de 3 π-blocks, de los cuales π_1 y π_2 son acíclicos y π_3 constituye una recurrencia. Si π_3 se ejecuta de forma secuencial tardará 2*N unidades de tiempo. Tanto π_1 como π_2 pueden ejecutarse en paralelo como bucles DOALL tardando N/P unidades de tiempo si se utilizan P procesadores. La figura 2.12(b) muestra la ejecución del bucle tras aplicar distribución de bucles. En ella se representa con S_{ij} la ejecución de la iteración j de la sentencia S_i. Si queremos solapar la ejecución de los tres π-blocks, de nada sirve ejecutar π_1 y π_2 de forma separada como DOALLs ya que π_3 domina el tiempo de ejecución del bucle entero. DOPIPE agrupa entonces π_1 y π_2 en un mismo segmento que se ejecutaría de forma solapada y sincronizada con π_3. El código generado se muestra en la figura 2.12(d). La construcción SEGMENT permite definir las sentencias que constituyen cada segmento generado. Notar que la ejecución mostrada en 2.12(c) tarda el mismo tiempo que la mostrada en 2.12(b) pero utilizando únicamente dos procesadores. La sincronización entre segmentos se realiza en este caso utilizando semáforos.

La efectividad de esta técnica depende del algoritmo utilizado para determinarlas sentencias que constituyen cada segmento, intentando solapar al máximo la ejecución del segmento determinante del tiempo de ejecución del bucle con el resto de segmentos. Cada segmento se ejecuta secuencialmente en un procesador y puede ser necesario añadir sincronizaciones a fin de asegurar las relaciones de dependencia entre segmentos.

2.3.5 DOÁCROSS

La ejecución de un bucle utilizando DGALL no permite la existencia de recurrencias en el grafo de dependencias. Si existen, en [Cytr82] se propone su extensión a *DOACROSS* de manera qué cada iteración del bucle espera a que todos los datos que necesita para ser ejecutada estén libres de dependencias.

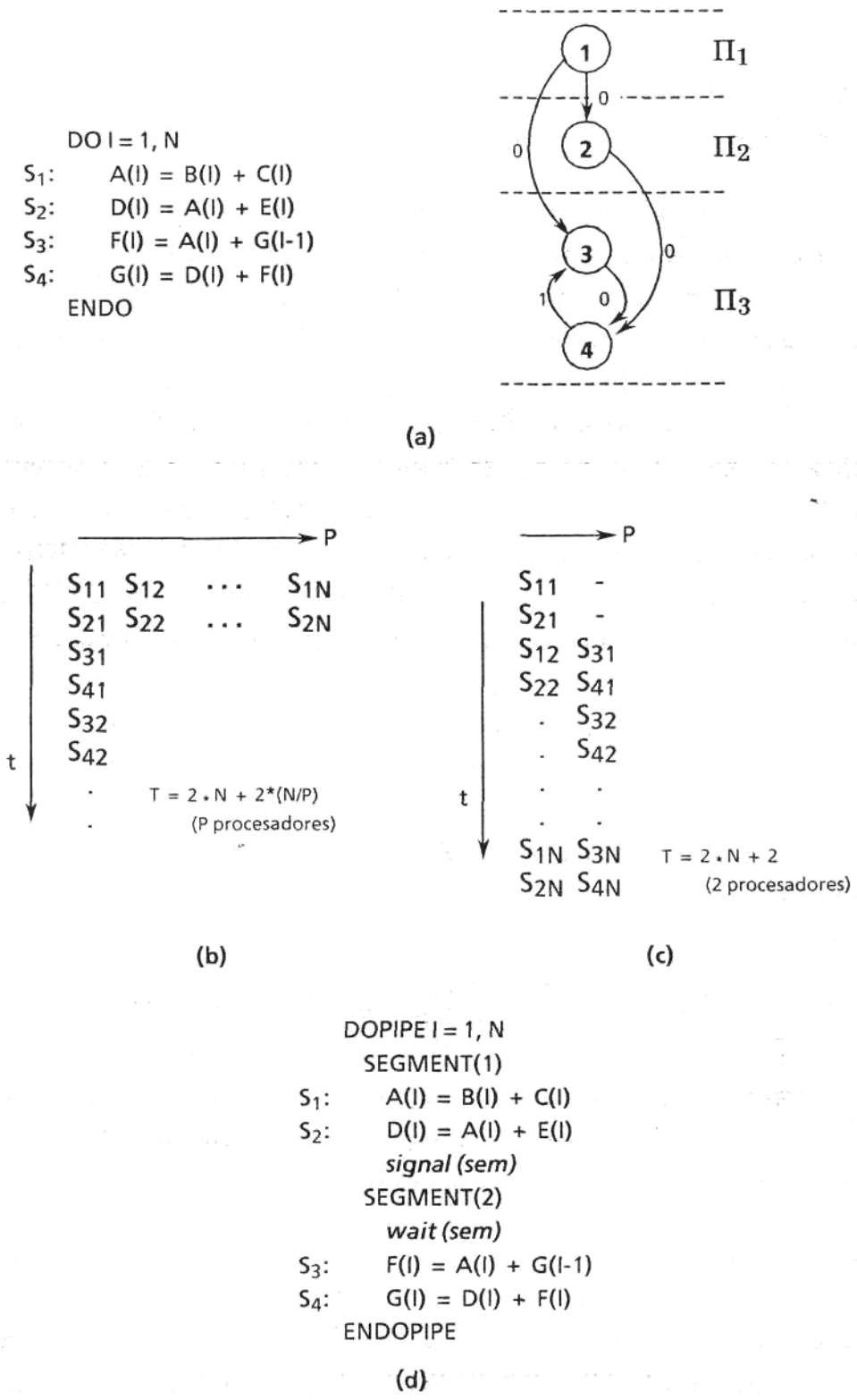


Figura 2.12: Ejemplo de aplicación de la técnica DOPIPE. (a) bucle secuencial y distribución en Π-blocks, (b) ejecución DOALL de Π_1 y Π_2, (c) ejecución DOPIPE agrupando Π_1 y Π_2 en un mismo segmento y (d) código paralelo generado.

Al igual que en DOALL, DOACROSS asigna iteraciones a procesadores y el inicio de cada una de ellas se retarda d unidades de tiempo respecto a la anterior. La figura 2.13 muestra la estructura general de un bucle DOAGROSS:Sfcseísupone queíeadansentencisrdelbucle tardaunaunidad de tiempo en ejecutarse, d se corresponde con el número de sentencias del bucle después de las cuales puede iniciarse la siguiente iteración. El retardo puede implementarse insertando primitivas de sincronización entre la sentencia S_d y S_1, dado que la propia ejecución de un bucle DOACROSS asegura que una iteración no se inicie hasta que todas las anteriores lo hayan sido también.

```
DOACROSS I =1,N
    delay(d*(i-1))
    S1
    S2
    .
    .
    Sn
ENDOAGROSS
```

Figura 2.13: Estructura general de un bucle DOACROSS, en el que se retarda la ejecución de la siguiente iteración en d unidades de tiempo.

En. [Gytr86] se presenta el cálculo de este retardo. Como puede observarse, si d=0 DOACROSS se comporta como un DOALL y si d coincide con el número de sentencias del bucle entonces se comporta como un bucle secuencial. El tiempo de ejecución del bucle viene dado por

$$\mathbf{T = (N - 1) \cdot \ d + T_{iter}}$$

siendo T_{iter} el tiempo necesario para ejecutar una iteración del bucle.

La figura 2.14(a) muestra un bucle secuencial y su grafo de dependencias asociado. La figura 2.14(b) muestra la ordenación y distribución de las operaciones obtenidas tras aplicar DOAGROSS. En este caso d = 2 de manera que la ejecución de S_1 en una determinada iteración no empieza hasta que finaliza la ejecución de la sentencia S_2 en la iteración anterior. A fin de mejorar la utilización de procesadores, es posible en este caso asignar grupos de iteraciones a procesadores tal como muestra la figura 2.14(c).En [Cytr84] se presenta el método general para definir este conjunto de iteraciones.

También es posible utilizar técnicas de reordenación de código con la finalidad de reducir el retardo introducido y disminuir por lo tanto el tiempo de ejecución del bucle paralelizado.

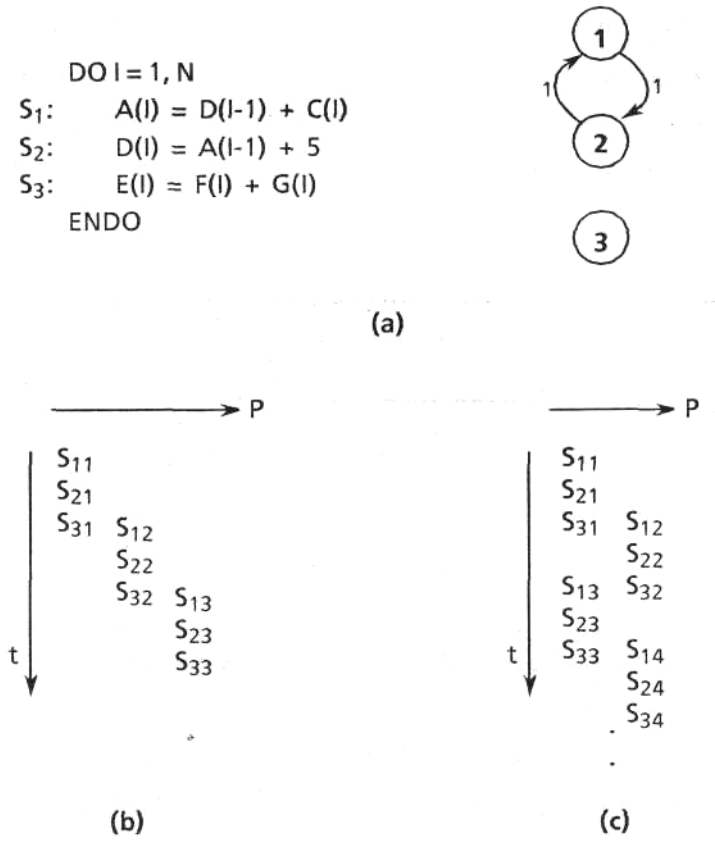


Figura 2.14: Ordenación y distribución de operaciones al aplicar DOACROSS a un bucle con GDC.

2.3.6 Particionado Parcial

Particionado Parcial [Padu79] es un método que permite la paralelización de recurrencias, asignando iteraciones del bucle a procesadores distintos de manera que no existan dependencias entre ellas. El número de tareas g viene determinado por el máximo común divisor de las distancias del grafo distintas de cero.

La figura 2.15 muestra-el código generado aplicando la técnica de particionado parcial para un bucle secuencial constituido por k sentencias incluidas en un ciclo de dependencia. El código generado está constituido por

dos bucles anidados: un bucle externo paralelo que determina las tareas independientes generadas y un bucle interno secuencial.

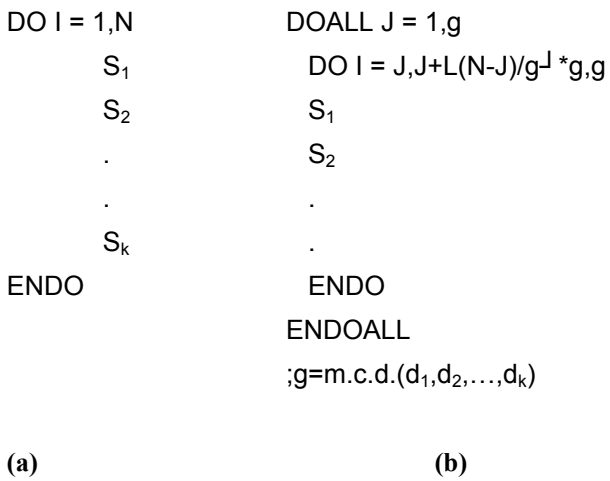


Figura 2.15: Código general obtenido al aplicar la técnica de particionado parcial.

Tal como muestra la figura 2.15(b), distintas iteraciones asignadas a un mismo procesador están separadas un múltiplo de g. Por lo tanto, si las distancias de los arcos son múltiplos de g, las dependencias quedarán aseguradas por la propia secuencialidad de la tarea asignada a cada procesador.

La figura 2.16(a) muestra un bucle secuencial y su grafo de dependencias asociado. Dado que el máximo común divisor de las distancias es 3, la técnica de particionado parcial genera 3 grupos e iteraciones que pueden ser ejecutadas en paralelo de forma independiente. La figura 2.16(b) muestra las tres tareas totalmente independientes generadas y la figura 2.16(c) el código paralelo generado;

La restricción más importante que presenta este método es que todas las dependencias de datos deben de tener asociada una distancia estrictamente mayor que la unidad a fin de obtener paralelismo de la recurrencia.

2.3.7 Contracción de Ciclos

La técnica de *contracción de ciclos* (Cycle Shrinking [Poly88]) es otro método que permite la paralelización de recurrencias. Se basa en ejecutar en paralelo

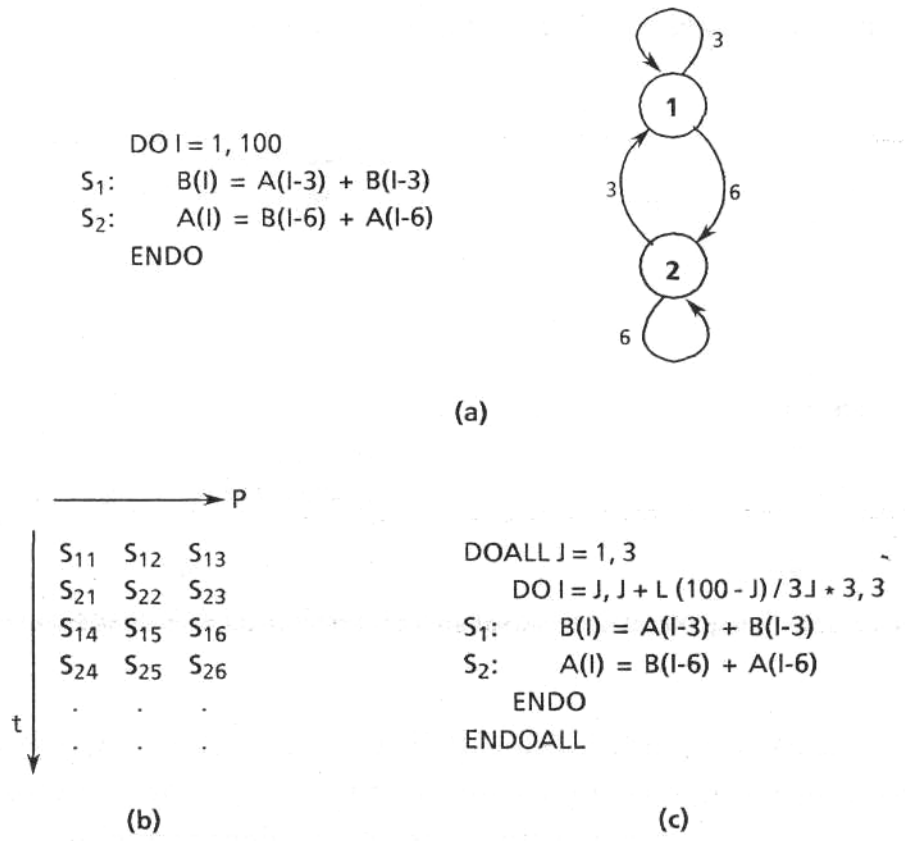


Figura 2.16: Ejemplo de aplicación de la técnica de particionado parcial.

todas aquellas iteraciones del bucle consecutivas que no dependen de ninguna iteración previa (inicialmente o porque ya han sido ejecutadas).

El número de iteraciones consecutivas A que se ejecutan en paralelo viene determinado por la mínima de todas las distancias del grafo de dependencias mayores que cero. Este bucle paralelo está incluido en un bucle secuencial que, en cada iteración, genera un nuevo conjunto de iteraciones libres de dependencia.

La figura 2.17 muestra la estructura general del código generado al aplicar la técnica de contracción de ciclos para un bucle general que incluye k sentencias involucradas en una ciclo de dependencia. En esta estructura general cabe resaltar la necesidad de sincronización tipo barrera al finalizar la ejecución del DOALL interno, dado que un conjunto de iteraciones no puede ser iniciado hasta que el anterior haya sido completamente ejecutado.

En la figura 2.18 se muestra un ejemplo de bucle secuencial con la versión paralela obtenida tras aplicar este método. En la ejecución desarrollada de la

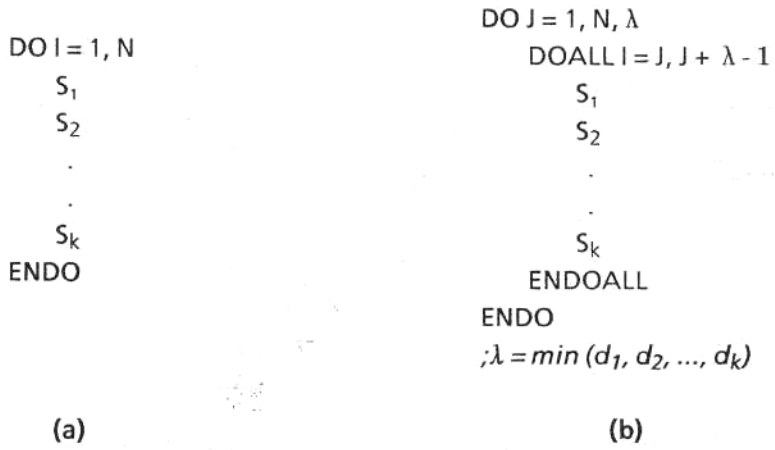


Figura 2.17: Código general obtenido al aplicar Contracción de ciclos.

misma se puede observar que es posible ejecutar en paralelo, después de cada barrera, todas aquellas iteraciones libres de dependencia.

En el ejemplo de la figura 2.18 también puede observarse que las sentencias ejecutadas en cada procesador son independientes, y por lo tanto, pueden ser ejecutadas también en paralelo.

En el grafo de la figura 2.19(a) y ejecución desarrollada de la figura 2.19(b) se observa que cada conjunto de iteraciones cubre dos iteraciones del bucle original. Debido a la relación de dependencia d_{23} con distancia asociada cero, no es posible ejecutar todas las sentencias del conjunto en paralelo. En este caso, es posible ejecutar S_1, S_2 y S_4 en paralelo y después de finalizar la ejecución de S_2 puede iniciarse la ejecución de S_3.

En general, el paralelismo del bucle viene dado por

$$\frac{\lambda \cdot n}{l(C)},$$

siendo A la distancia mínima del grafo, *n* el número de sentencias del bucle y C la cadena de dependencias más larga con peso w(C) = 0.

La técnica de contracción de ciclo presenta el mismo inconveniente que la técnica de particionado parcial, en cuanto a que requiere que las distancias del grafo sean estrictamente mayores que la unidad. Sin embargo, para el caso general, la técnica de contracción de ciclos consigue un mayor número de

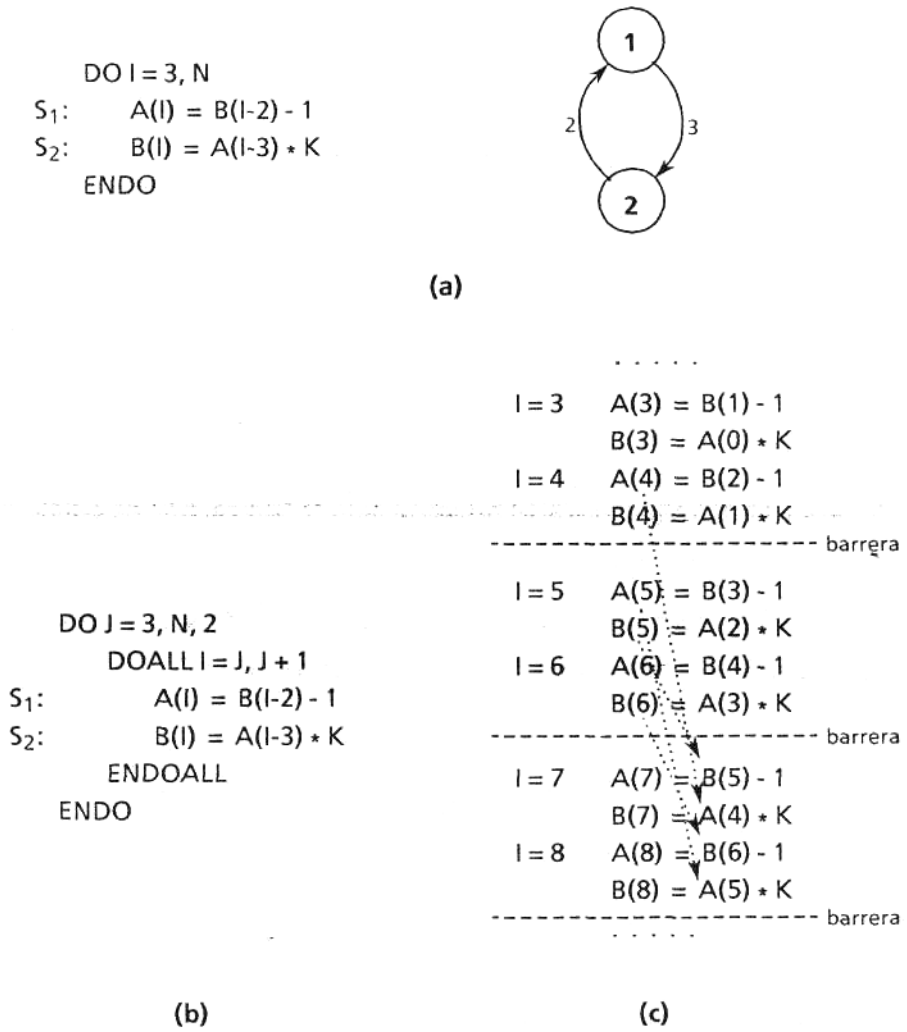


Figura 2.18: Aplicación de Contracción de ciclos a un bucle con GDC (a), código paralelo generado (b) y ejecución desarrollada del mismo (c).

operaciones ejecutables en paralelo que la técnica de particionado parcial. Esta comparación se realiza en el capítulo 6.

2.3.8 Reconocimiento de patrones

Algunos ciclos de dependencia que incluyen una única sentencia pueden ser detectados y reconocidos por el compilador. En este caso, el compilador puede sustituirlos por llamadas a rutinas de librería optimizadas para la máquina que las va a ejecutar (por disponer de "hardware" especializado) o por código equivalente que sí pueda ser paralelizado. Entre ellas suelen estar operaciones de reducción que obtienen un valor escalar a partir de un vector. Por ejemplo,

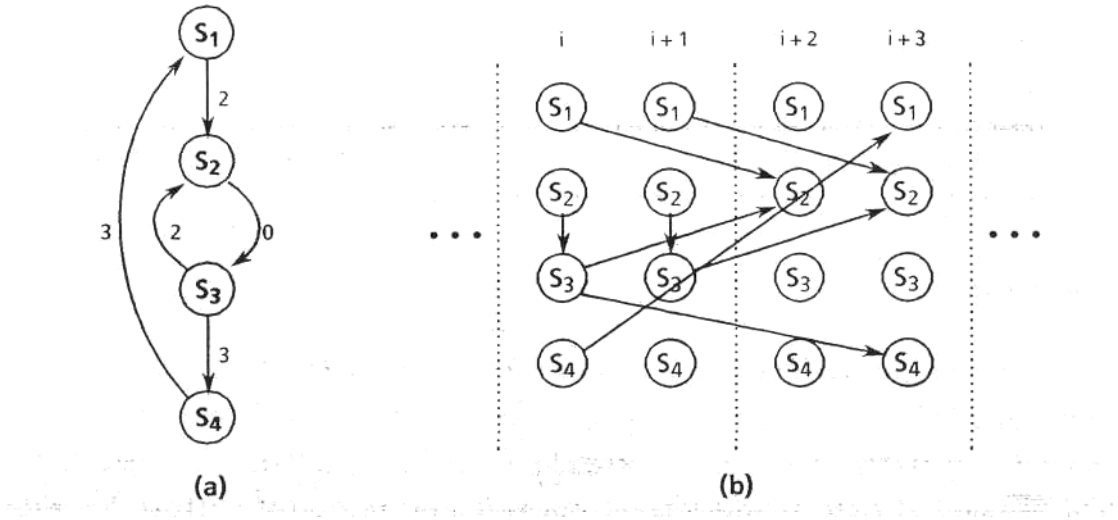


Figura 2.19: influencia de las dependencias con distancia cero en el paralelismo obtenido por la técnica de contracción de ciclos.

CEDAR FORTRAN posee implementadas el máximo, mínimo, suma y producto de los elementos de un vector entre otras.

Vectorización

La figura 2.20(a) muestra un bucle secuencial en el que aparecen dos recurrencias debidas a este tipo de operaciones, tal como muestra el grafo de la figura 2.20(b). El compilador puede analizar estas recurrencias y reconocer que son debidas a operaciones de reducción que conoce. En este caso podría generar el código vectorial mostrado en la figura 2.20(c).

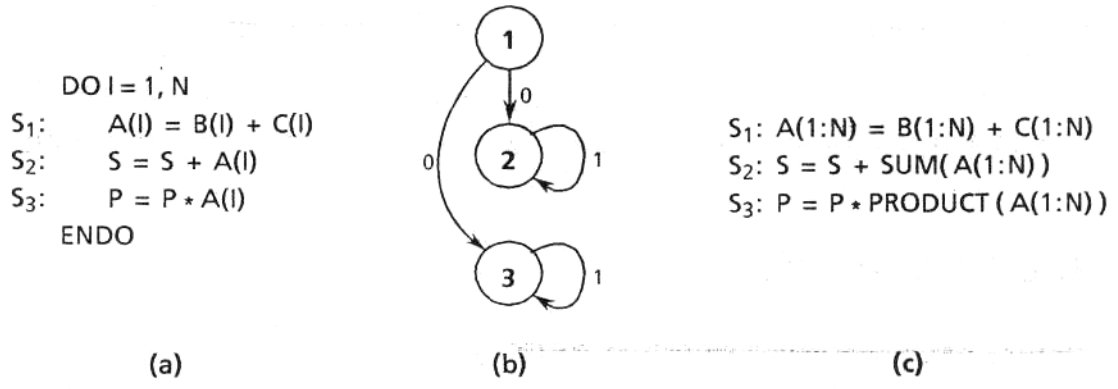


Figura 2.20: Vectorización de recurrencias reconocidas como operaciones de reducción.

Otros tipos de operaciones reconocidas como patrones por algunos de los compiladores son:

```
DO I = 2, N
   X(I) = A(I) ± B(I) * X(I-1)
ENDO
```

```
DO I = 2, N
   X(I) = A(I) * ( B(I) ± X(I-1) )
ENDO
```

Paralelización

En el caso de paralelización también el compilador puede reconocer determinados patrones y generar código eficiente para ellos. Por ejemplo, para la operación de reducción suma mostrada en la figura 2.21(a), el compilador podría; generar el código equivalente mostrado en la figura 2.21(b). En este caso, cada uno de los NP procesadores realiza la suma parcial de un conjunto de elementos del vector. Al finalizar, debe acumularse esta suma parcial en la suma total, accediendo a la variable s dentro de una zona crítica.

```
    DO I = 1, N
S1:     S = S + A(I)
    ENDO
```

(a)

```
    DOALL P = 1, NP
        SP(P) = 0
        DO I = P, N, NP
S1':        SP(P) = SP(P) + A(I)
        ENDO
        <inicio sección crítica>
        S = S + SP(P)
        <fin sección crítica>
    ENDOALL
```

(b)

Figura 2.21: **Paralelización de recurrencias reconocidas como operaciones de reducción.**

Los principales problemas que esto presenta son posibles errores de redondeo debidos a que las operaciones se realizan en distinto orden [Wolf 89].

2.4 BUCLES CONDICIONALES

La existencia de sentencias estructuradas condicionales en el bucle es un tema a considerar en el diseño de un compilador para máquinas vectoriales y multiprocesadores escalares.

Tal como se ha descrito en la sección 2.2, las dependencias de control indican la ejecución condicionada de las sentencias incluidas en ramas de

ejecución bajo control de una sentencia tipo IF. En primer lugar, se consideran algunas de las técnicas existentes para la vectorización de este tipo de bucles, y en segundo lugar, se considera la paralelización de estos bucles para multiprocesadores escalares, analizando como tratan algunas de las técnicas descritas en la sección anterior a este tipo de bucles.

2.4.1 Vectorización de bucles condicionales

La vectorización de bucles condicionales se basa en la transformación de dependencias de control en dependencias de datos [AKPW83], convirtiendo las sentencias bajo control de un IF en sentencias cuya ejecución estará controlada por algún operando indicado en la misma instrucción.

La construcción WHERE descrita en el apartado 1.2.3 e incorporada en FORTRAN 8X y CEDAR FORTRAN permite la ejecución condicional de sentencias vectoriales.

La figura 2.22 muestra un bucle con instrucciones condicionales, su grafo de dependencias y una versión vectorial equivalente. En este caso, TMP(1:N) es evaluado en S_2 y constituye un nuevo operando que controla la ejecución vectorial de las sentencias S_3 y S_4.

El método es válido también en el caso de instrucciones condicionales anidadas, en cuyo caso el compilador debe generar e incluso simplificar las expresiones lógicas que controlan la ejecución de cada una de las sentencias incluidas en la estructura condicional.

Algunos ejemplos de implementación de este método se encuentran en [KKLW80] para el PARAFRASE, en [AIKe87l para el PFC y en [Cole87] para el vectorizador del UNISYS ISP.

Si la condición que decide la ejecución de la rama condicional se cumple muy pocas veces, se emplea gran parte del tiempo en ejecutar cálculos que no se van a almacenar posteriormente. Si el compilador es capaz de detectarlo, puede utilizar técnicas de compresión y expansión de vectores disponibles en las máquinas vectoriales. De esta manera, sólo se realizan aquellos cálculos que son realmente necesarios. La figura 2.23 muestra un ejemplo de un bucle secuencial con instrucción condicional y su vectorización utilizando técnicas de compresión y expansión. En este ejemplo, NI es el número de veces que se cumple la condición y equivale al número de unos que tendría el vector TMP.

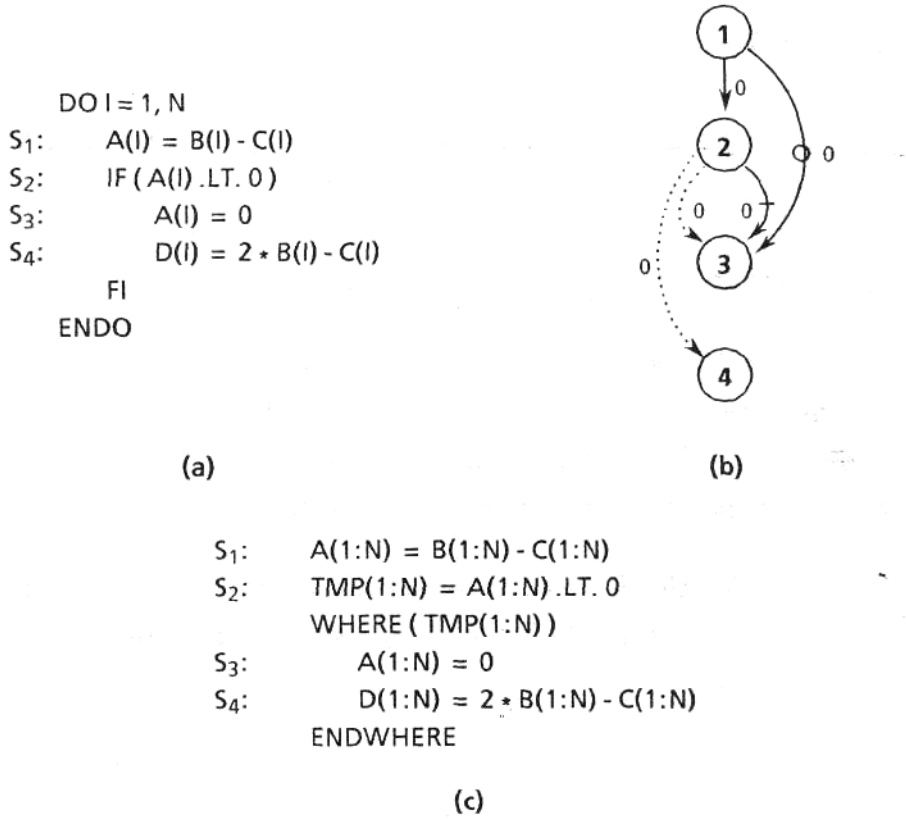


Figura 2.22: Vectorización de un bucle condicional.

NI es un valor obtenido mediante una función de reducción incluida en el lenguaje de programación (por ejemplo *count* del CEDAR FORTRAN que devuelve el número de elementos distintos de cero dentro de un vector).

```
    DOI = 1,N
ST:     IF(A(I).GT. 0)
S2:         B(I) = (A(I)*A(I) + B(I) * B(I)) * 0.5
        FI
    ENDO
```

(a)

```
TMP(1:N) = A(1:N).GT. 0
N1 = count (TMP(1:N))
TA(1:N1) = PACK(A(1:N),TMP(1:N))
TB(1:N1) = PACK(B(1:N),TMP(1:N))
TB(1:N1) = (TA(1:N1)*TA(1:N1) + TB(1:N1) * TB(1 :N1)) * 0.5
B(1:N) = UNPACK(TB(1:N1), TMP(1:N))
```

(b)

Figura 2.23: Utilización de las instrucciones de compresión y expansión para reducir el tiempo de ejecución.

La vectorización de patrones reconocidos en sentencias condicionales se considera en [Wolf89]. La figura 2.24(a) muestra un bucle secuencial que incluye una recurrencia dentro de una sentencia condicional. El compilador puede convertir la "recurrencia en incondicional utilizando variables temporales tal como se muestra en la figura 2.24(b).

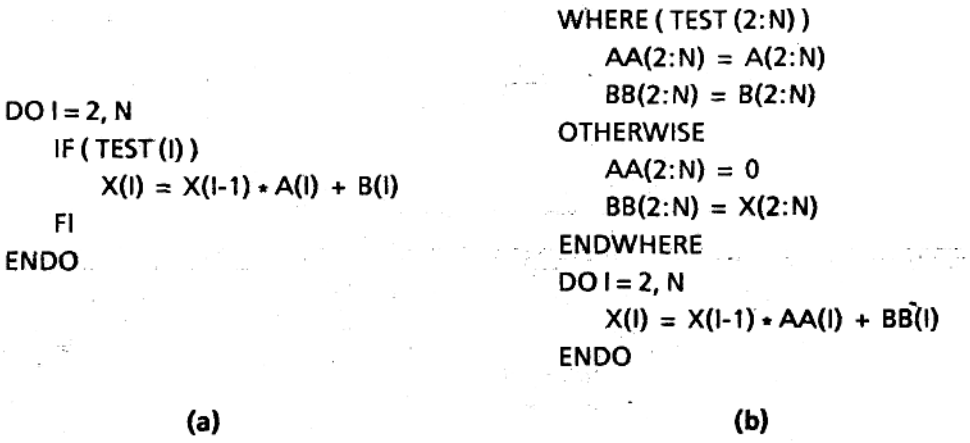


Figura 2.24: Recurrencias condicionales reconocidas como patrones.

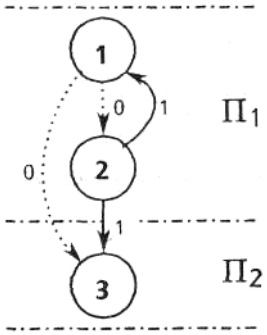
2.4.2 Paralelización de bucles condicionales

En este caso, no se aplican técnicas que-transforman dependencias de control en dependencias de datos como se ha descrito anteriormente para el caso de vectorización. Para el caso de distribución de bucles y DOPIPE, el compilador debe tener en cuenta los arcos de control entre π-blocks cuanda se ejecutan en procesadores distintos (ya que al aplicar DOPIPE varios π-blocks pueden ser englobados en un mismo segmento).

La idea básica [Padu79] es evaluar de forma distribuida la condición en todos los π-blocks o segmentos afectados por la dependencia de control. La figura 2.25(a) muestra un bucle condicional y su grafo de dependencias asociado. En este caso S_1 y S_2 forman un π-block y S_3 otro. La Figura 2.25(b) muestra el código generado al aplicar DOPIPE y la sincronización añadida para asegurar la dependencia de control utilizando semáforos.

```
     DO 1 = 1, N
St:      IF(A(I).LT. 10)
S2:          A(I+ 1) = B(I) + 10
S3:          D(I) = A(I) + C(I)*F(I)
         FI
     ENDO
```

(a)



(b)

```
DOPIPEI= 1,N
SEGMENT(1)
    TMP(I) = A(I).LT. 10
    IF(TMP(I))
        A(I + 1) = B(I) + 10
    FI
    signal(sem)
SEGMENT(2)
    wait(sem)
    IF(TMP(I))
        D(I) = A(I) + C(I)*F(I)
    FI
ENDOPIPE
```

(c)

Figura 2.25: Paralelización aplicando DOPIPE de un bucle con sentencia condicional.