Capítulo 5

Paralelización automática de códigos dispersos

Esta claro que las herramientas de paralelización automática, son las que conducen a menores tiempos de desarrollo en la construcción de un algoritmo paralelo. Estas herramientas se materializan en compiladores fuente-fuente que a partir de un código secuencial, detectan las secciones paralelas para luego reestructurar y añadir las directivas, variables, sentencias, y comunicaciones necesarias que resultan, a la salida, en un código paralelo. Aunque la idea del compilador paralelo pueda parecer novedosa, en realidad no tiene otra misión que la que pueda tener cualquier compilador: ocultar al programador los detalles de la arquitectura para la que desarrolla sus códigos. Llevando esta idea al extremo, el lenguaje utilizado para expresar los programas debe ser también independiente de la arquitectura que presenta la plataforma, ya sea superescalar, vectorial, o, ¿por qué no?, paralela.

Bajo este paradigma, el programador sólo debe escribir y depurar la versión secuencial de su algoritmo, ya que delega en el compilador toda la responsabilidad de la paralelización. Sin embargo, tampoco es difícil imaginar que, actualmente, las prestaciones que presenta el algoritmo paralelo obtenido automáticamente son claramente superadas por una versión paralelizada a mano. Esta yuxtaposición entre facilidad de uso y prestaciones finales es comparable a la que existía entre la programación en ensamblador y con un lenguaje de alto nivel. Pero en este caso, es patente como la primera alternativa dejó de usarse[1], cuando los compiladores de lenguajes de alto nivel consiguieron, con el tiempo, rendimientos similares amén de una mucho mayor facilidad de programación. La fracción de la comunidad científica que trabaja en el ámbito del paralelismo sabe que, tarde o temprano, terminarán teniendo el mismo éxito los compiladores paralelos. La clave está en aumentar la potencia y el radio de acción de los compiladores paralelos para, también esta vez, romper el compromiso entre la comodidad y la eficiencia de la aproximación utilizada.

Es el objetivo de este capítulo el presentar las propuestas que aportamos en esta dirección. Embarcados, como estamos, en la resolución de problemas de índole irregu-

[1]Salvo para núcleos computacionales de poca longitud y de los que se requiere una alta optimización.

lar, más particularmente en algoritmos dispersos, cabe preguntarse cuanto paralelismo puede explotar un compilador paralelo automático de uno de nuestros códigos. En cierto modo, ésto ya ha supuesto ampliar el radio de acción de estos compiladores, que hasta ahora no habían encarado específicamente este tipo de problemas. Efectivamente, los compiladores automáticos suelen conseguir resultados competitivos en códigos regulares, debido a la mayor simplicidad en los bucles. Sin embargo, inicialmente, estos compiladores no son capaces de extraer paralelismo de los códigos dispersos con los que los alimentábamos. Ha sido necesario el desarrollo de una serie de técnicas adicionales y de algunas simplificaciones sencillas en los códigos para que las salidas paralelas fuesen competitivas.

Existe un determinado número de grupos desarrollando y mejorando sus herramientas de paralelización automática, a saber: el proyecto Polaris dirigido por David Padua [32]; SUIF por Mónica Lam [84]; Parafrase-2 por Polychronopoulos [113]; y PFA [137], herramienta comercial desarrollada por David Kuck.

Actualmente, estos compiladores no son capaces de extraer paralelismo cuando el código secuencial presenta estructuras de datos dinámicas o basadas en punteros. Por tanto, sólo para poder empezar a trabajar, ha sido necesario desarrollar un algoritmo disperso de factorización LU en Fortran77. Este algoritmo, presentado en la sección 5.1, es paralelizado a mano obteniéndose buenas eficiencias. Sin embargo, en un primer intento, Polaris, compilador en el que nos centramos en el resto del capítulo, no consigue explotar suficiente paralelismo por las razones que se comentan en la sección 5.2. La elección de Polaris como compilador paralelo viene avalada por los excelentes resultados que esta herramienta ha proporcionado con un gran número de códigos tanto del conjunto de *benchmarks*[2] SPEC [125] como Perfect Club [28], así como por su gran disponibilidad, lo que ha permitido a otros grupos investigar con el mencionado compilador. En la sección 5.3 se discuten un conjunto de técnicas orientadas a corregir las carencias y puntos débiles de Polaris al compilar códigos dispersos. Por último, en la sección 5.4, se presentan otros códigos que son paralelizados exitosamente mediante la aplicación de las mismas técnicas.

5.1 Factorización dispersa: Un caso de estudio

Actualmente, Polaris procesa código secuencial escrito en Fortran77 y, por otro lado, no está preparado para detectar paralelismo cuando los bucles recorren listas enlazadas. Por tanto, para chequear el compilador con un código disperso que presenta el problema del llenado, hemos escrito en Fortran77 el algoritmo de factorización dispersa LU.

5.1.1 Características generales

Este código, de tipo *right-looking*, emplea una estructura de datos CCS tanto para almacenar la matriz `A`, como los factores `L` y `U`. De esta forma evitamos la utilización de punteros, aunque seguimos utilizando una estructura comprimida. Al contrario que

[2]Término que identifica a los programas usados para evaluar las prestaciones de una arquitectura.

en la aproximación *left-looking*, donde para cada iteración `k` el llenado sólo tiene lugar en la columna `k`, en nuestro código *right-looking* el llenado aparece en toda la submatriz reducida. A pesar de ello, la organización *left-looking* no se presenta como una opción, ya que, como se comentó en el capítulo 3, es inherentemente secuencial.

El llenado en toda la submatriz reducida representada mediante la estructura de datos CCS puede ser manejado a costa de un inevitable movimiento de columnas. Para evitar un mayor tráfico de datos durante la fase de factorización, una etapa de análisis se encarga, en una primera fase, de determinar los vectores de permutación de filas y columnas. De esta forma se seleccionan los pivots que aseguran la estabilidad numérica y mantienen el coeficiente de dispersión. Durante esta fase de análisis también se decide la iteración a partir de la cual la submatriz restante puede ser tratada como densa y, por tanto, podemos conmutar a un código denso de factorización LU. Así pues, inicialmente se ejecuta un código de factorización dispersa, pero una vez alcanzada la iteración de conmutación, se permuta a un código denso basado en rutinas de nivel 2 (o 3 si la distribución es cíclica por bloques) BLAS. En este código denso se implementa pivoteo parcial de filas para asegurar la estabilidad numérica.

La distribución de datos en el código paralelo es BCS para la fase de factorización dispersa. Cuando esta fase termina, se realiza el cambio de estructura de datos en la submatriz reducida. Por tanto la conmutación a código denso consume un tiempo despreciable y la submatriz densa queda distribuida automáticamente según la distribución cíclica regular.

5.1.2 Estructura de datos

Para colaborar en la disminución del número de columnas a mover durante la factorización, en lugar de utilizar un algoritmo "in-place", ocuparemos una zona de memoria con la matriz de entrada y otra distinta para la matriz que contiene los factores `L` y `U`. La estructura de datos que organiza ambas zonas de memoria es CCS, es decir, un vector para almacenar coeficientes, otro para índices de fila y otro que contiene punteros al comienzo de cada columna. Además se utiliza un vector adicional con punteros, por lo que podemos llamar a la estructura completa CCS modificada. En la figura 5.1 presentamos gráficamente la estructura de datos mencionada.

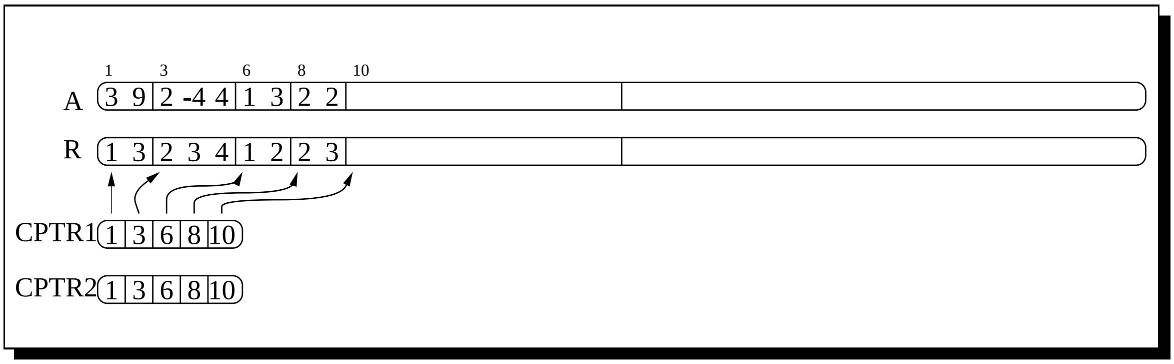


Figura 5.1: Estructura de datos CCS modificada para la matriz A.

Los arrays: `A` para los coeficientes; `F` para los índices; `CPTR1` para los punteros a comienzo de fila; y `CPTR2` usado para apuntar a la fila activa del pivot; contienen al comienzo del algoritmo la matriz inicial `A`. En la figura, estos tres vectores contienen los valores de la matriz dispersa $A^{(1)}$ del ejemplo 1.9, donde vemos que el vector `CPTR2` es una copia de `CPTR1` al comienzo del algoritmo. El espacio reservado en los arrays `A` y `F` se organiza en dos mitades, cuyo propósito se discute en la siguiente subsección. Inicialmente, $A^{(1)}$ debe caber en la mitad inferior.

La estructura de salida se muestra en 5.2, donde la matriz factorizada, $A^{(4)}$, del ejemplo 1.9, se organiza en el formato CCS modificado. El vector `FACT` contiene los coeficientes, `FR` los índices de fila y `FPTR1` los punteros al comienzo de cada columna. De nuevo, es necesario un vector adicional que apunte al pivot de cada columna. De esta forma marcamos donde acaba `U` y, por tanto empieza `L`, en cada columna. En la figura los pivots están marcados con fondo en gris. Esta estructura de datos es suficiente para diseñar el algoritmo disperso que se describe a continuación.

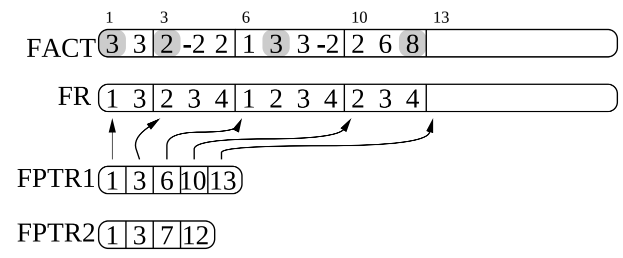


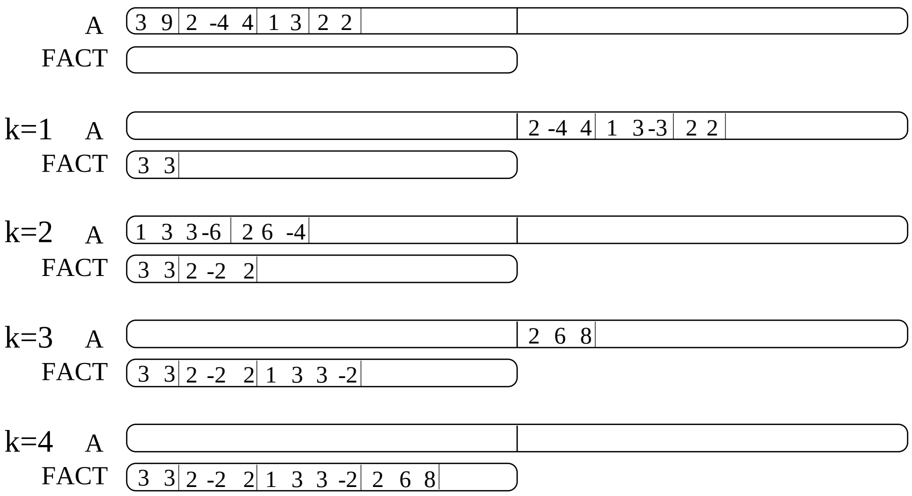
Figura 5.2: Estructura de datos CCS modificada para la matriz factorizada.

5.1.3 Organización del algoritmo

Como se ha repetido en otras ocasiones, la estructura de datos elegida es de una influencia determinante en la organización del posterior algoritmo disperso. En nuestro caso, el principal problema es el de encajar el llenado en la estructura comprimida. Es fácil entender que cuando aparece una nueva entrada en una columna debemos moverla a otra zona en la que quepa la columna completa. Así pues, durante la actualización de la submatriz reducida vamos actualizando y moviendo columnas para mantener éstas ordenadas tal y como impone la estructura CCS. A grandes rasgos, a cada iteración `k` del algoritmo, copiamos la columna del pivot (`k`) a la estructura de salida (`FACT, FR, FPTR`) una vez normalizada. El resto de las columnas de la matriz reducida se actualizan y mueven de una mitad de los arrays `A` y `R` a la otra, actualizándose convenientemente los punteros `CPTR1` y `CPTR2`. Cuando se alcanza la iteración de conmutación a código denso, la submatriz reducida restante se arregla en un array bidimensional, apropiado para la subsiguiente llamada a la rutina de factorización densa.

Antes de describir el código del algoritmo, nos detenemos un instante en el ejemplo 5.1 donde se muestra la evolución de los vectores de coeficientes `A` y `FACT` en las cuatro iteraciones necesarias para factorizar la matriz de la figura 5.1.

Ejemplo 5.1 Factorización de una matriz 4×4



Al comienzo, la matriz de entrada se encuentra completamente en el array `A`. Después de la primera iteración, la primera columna normalizada por el pivot pasa a formar parte de la estructura de salida (`FACT`), mientras, las restantes columnas se actualizan en la segunda mitad del array `A`. Es ahora esta segunda mitad la que se lee durante la iteración `k=2` para obtener la segunda columna de `FACT` y actualizar las restantes en la primera mitad de `A`. Al final del algoritmo, `FACT` contiene los coeficientes de la matriz factorizada.

Ejemplo 5.2 Factorización dispersa en F77 (1ª Parte)

```
      DO 20 I=1,N               !Inicialización
         CPTR2(I)=CPTR1(I)
 20   CONTINUE
      EYE=1
      DO 100 K=1,IterConm       ! Bucle externo
         FPTR1(K)=EYE           ! Comienza U
         DO 30 I=CPTR1(K),CPTR2(K)-1
            FACT(EYE)=A(I)
            FR(EYE)=R(I)
            EYE=EYE+1
 30      CONTINUE
         FPTR2(K)=EYE           ! Acaba U
C Copia el pivot
         AMUL=1/A(CPTR2(K))
         FACT(EYE)=AMUL
         FR(EYE)=R(CPTR2(K))
         EYE=EYE+1
         F1=EYE                 ! Comienzo del vector x en SAXPY (y=y-ax)
C Normaliza la columna del pivot
         DO 40 I=CPTR2(K)+1,CPTR1(K+1)-1
            FACT(EYE)=A(I)*AMUL
            FR(EYE)=R(I)
            EYE=EYE+1
 40      CONTINUE
         F2=EYE-1               ! Final del vector x
                                        •
                                        •
```

En el ejemplo 5.2, presentamos la primera parte del código. Al comienzo encontramos el bucle DO 20 encargado de inicializar el vector `CPTR2` para que apunte a la fila del pivot. A continuación tenemos el bucle DO 100 que recorre las `IterConm` primeras iteraciones de factorización dispersa. Las líneas de código mostradas en este ejemplo son las encargadas de copiar la columna `K` de `A` en `FACT`. El bucle DO 30 copia la fracción de esa columna que pertenece a `U`, a continuación se copia el pivot (apuntado por `CPTR2`) y finalmente se normaliza la columna al tiempo que se mueve a la estructura de salida en el bucle DO 40. Las variables F1 y F2 apuntan al comienzo y al final, respectivamente, de la columna del pivot activa. Esta subcolumna jugará el papel de vector x en las operaciones SAXPY dispersas ($y = y - \alpha x$) que tienen lugar en la subsiguiente actualización. En el ejemplo 5.3, presentamos la segunda parte del bucle: la actualización de la submatriz reducida.

Ejemplo 5.3 Factorización dispersa en F77 (2ª Parte)

⋮

```
C Actualiza la submatriz reducida
         SHIFT=MOD(K,2)*LFACT+1
         DO 50 J=K+1,N
            C1=SHIFT
C Copia la parte que no se modifica
            DO 60 I=CPTR1(J),CPTR2(J) ! Al menos se ejecuta una vez el bucle.
               A(SHIFT)=A(I)
               R(SHIFT)=R(I)
               SHIFT=SHIFT+1
 60         CONTINUE

            C2=SHIFT-1
C Copia el elemento de la fila K
            IF (R(C2).EQ.K) THEN
               AMUL=A(C2)
               C2=C2+1
C Scatter
               DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
                  W(R(I))=A(I)
 70            CONTINUE
C Actualiza
               DO 80 I=F1,F2
                  W(FR(I))=W(FR(I))-(AMUL*FACT(I))
 80            CONTINUE
C Gather
               DO 90 I=1,N
                  IF (W(I).EQ.0) GOTO 90
                  A(SHIFT)=W(I)
                  R(SHIFT)=I
                  SHIFT=SHIFT+1
                  W(I)=0
 90            CONTINUE
            ELSE
               DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
                  A(SHIFT)=A(I)
                  R(SHIFT)=R(I)
                  SHIFT=SHIFT+1
 95            CONTINUE
            END IF

            CPTR1(J)=C1
            CPTR2(J)=C2
 50      CONTINUE
         CPTR1(N+1)=SHIFT
         IF (SHIFT.GT.((MOD(K,2)+1)*LFACT)) STOP !Error por falta de espacio
 100  CONTINUE
```

La variable `SHIFT` es la encargada de apuntar al comienzo del vector `A` en las iteraciones impares o al comienzo de su segunda mitad en las iteraciones pares[3]. El bucle

[3] El array `A` es declarado de longitud 2×`LFACT`.

DO 50, cuya variable índice es `J`, es el encargado de recorrer las columnas a la derecha de la columna del pivot para su actualización. En el cuerpo de este bucle encontramos primero un bucle, DO 60, que copia la parte de la columna por encima de la matriz activa, ya que no va a ser modificada. La variable `SHIFT` evoluciona dentro de este bucle apuntando a las posiciones del vector `A` donde se copian las columnas. Cuando una variable se utiliza de esta forma (ver bucle DO 60), con incrementos de la variable a cada iteración, se la llama **variable de inducción** [159].

A continuación se chequea si existe una entrada en la fila `K` para esa columna. En caso de no existir se ejecuta el bucle DO 95 que sencillamente mueve el resto de la columna. Por el contrario, si existe tal valor, se almacena en la variable `AMUL` que representa el escalar α de la operación SAXPY dispersa. Esta operación se organiza en tres etapas, discutidas en detalle en [57, Cap. 2.4]. Primero desempaquetamos la columna comprimida `J` sobre un array denso, `W`, inicialmente a cero, en el bucle DO 70 (operación *scatter*). A continuación se realiza la operación SAXPY dispersa, donde el vector `W` toma el papel de la variable y, en el bucle DO 80. Por último, el bucle DO 90 empaqueta los elementos no nulos de `W` al tiempo que vuelve a dejar a cero los coeficientes usados de `W` (operación *gather*). Si `SHIFT` supera la mitad del array en el que escribe, el programa se aborta por falta de espacio para encajar el llenado. Con ésto se cierran los bucles DO 50 y DO 100.

Las secciones de código de conmutación a código denso, factorización densa y resolución triangular son totalmente análogas a las descritas en capítulos anteriores. Por tanto pasamos a evaluar este nuevo algoritmo en sus versiones secuencial y paralela para comparar los resultados con los que inicialmente proporciona Polaris.

5.2 Un primer resultado con Polaris

El código descrito es fácil de paralelizar manualmente. Excepto el bucle exterior de índice `K`, el resto son paralelizables. El bucle DO 50 accede a las columnas que son independientes y los demás procesan estas columnas ya sea en la operación `divcol()` o en `actcol()`, inherentemente paralelas como se discute en el capítulo 1.

En resumen, el algoritmo paralelo necesita:

- Una distribución cíclica dispersa, que unida a la estructura de datos CCS conforman el esquema de distribución BCS. Tanto los índices de fila como los punteros a columnas son locales a la submatriz asignada a cada procesador.
- Dos etapas de comunicación: la primera radia por columnas de la malla la fila del pivot activa junto al propio pivot; y la segunda radia en las filas de la malla la columna del pivot activa.
- Modificación del espacio de iteraciones de los bucles para recorrer únicamente los coeficientes locales.

Con estas tres modificaciones realizadas sobre el algoritmo secuencial obtenemos el código paralelo que hemos testeado en el Cray T3D. El sólo hecho de distribuir desacopla las iteraciones que presentan dependencias generadas por bucle. Por ejemplo, la variable de inducción `SHIFT` es privada a cada procesador y recorre estructuras locales, por lo que se cancela la dependencia generada por bucle que presenta en el algoritmo secuencial. De igual forma, el bucle DO 50 es secuencial mientras no se asegure que las escrituras en las variables dentro del bucle se realizan en posiciones distintas de memoria a cada iteración. Por tanto la distribución, y la consiguiente privatización de las variables escalares así como el vector de trabajo `W`, consigue que cada procesador modifique su sección de datos local sin interaccionar con los demás. En cuanto a las etapas de comunicación, pueden ser interpretadas como una fase en la que los procesadores *leen* la información no local que necesitan de otros procesadores.

En la tabla 5.1 se cuantifican las prestaciones del algoritmo paralelo comentado. El interfaz de pase de mensajes utilizado es el que proporcionan las librerías SHMEM de Cray y como rutinas de test algunas del conjunto Harwell-Boeing.

	Tiempo (seg.)				Aceleración			Eficiencia(%)		
Matriz	sec.	2×2	4×4	8×8	2×2	4×4	8×8	2×2	4×4	8×8
STEAM2	2.33	0.77	0.34	0.21	3.02	6.85	11.09	75.64	42.83	17.33
JPWH 991	3.73	1.30	0.55	0.34	2.86	6.78	10.97	71.73	42.38	17.14
SHERMAN1	2.18	0.88	0.42	0.26	2.47	5.19	8.38	61.93	32.44	13.10
SHERMAN2	36.22	8.84	2.58	1.02	3.99	14.38	35.50	99.75	87.74	55.48
LNS 3937	157.48	44.20	12.17	4.24	3.56	12.94	37.14	89.07	80.87	58.03

Tabla 5.1: Tiempos, aceleración y eficiencia del algoritmo paralelo en F77 para distintos tamaños de la malla en el CRAY T3D.

Sin embargo, intentar obtener resultados similares mediante la paralización automática fueron inicialmente infructuosos. Una prueba es el resultado de ejecutar la salida paralela de Polaris en el SGI Power Challenge con 16 procesadores R10000 a 196MHz. En la tabla 5.2 se aprecia por un lado el coste añadido que introduce el compilador en el algoritmo paralelo[4], y por otro que los tiempos de ejecución en más procesadores son incluso peores.

Matriz	Secuencial	1	2	4	8	16
LNS 3937	77.14	179.41	350.66	425.49	492.75	710.01

Tabla 5.2: Tiempo (seg.) de ejecución del algoritmo paralelo generado por Polaris para distinto número de procesadores del SGI Power Challenge.

La razón de esto es clara. Para este código irregular, Polaris sólo detecta el paralelismo trivial de los bucles 20, 30 y 40, eliminando automáticamente la variable de inducción `EYE` cuando aparece. En el ejemplo 5.4 se muestra el código que genera Polaris para este bucle DO 40. En este caso, el bucle puede ser expresado en lo que conocemos como su **forma cerrada**, es decir substituyendo la variable de inducción por una función del índice, `I`, del bucle. Como vemos en el ejemplo, en la última iteración la variable de inducción `EYE` toma el valor adecuado, ya que es usada fuera

[4]Comparando el tiempo de ejecución secuencial con el paralelo ejecutado en un nodo.

del bucle. No es difícil imaginar que el coste introducido en la versión paralela no compensa el trabajo de procesar la columna en paralelo, ya que para matrices de, por ejemplo, $n = 1000$ una columna puede tener de media menos de 100 elementos a procesar. Además este bucle encargado de la operación `divcol()` sólo se ejecuta una vez por cada iteración del bucle externo. Por tanto es realmente el bucle DO 50 el que merece mayor atención, por ser el que completa las $n - k$ operaciones `actcol()` que actualizan la submatriz reducida, a cada iteración `k`.

Ejemplo 5.4 Código generado por Polaris para DO 40

```
                      eye1 = eye
                C$DOACROSS IF(((-1)+cptr1(1+k)+(-1)*cptr2(k).GT.5.OR.29.GT.300).AND.29+
                C$& 5*((-1)+cptr1(1+k)+(-1)*cptr2(k)).GT.150),LOCAL(EYE3,I),SHARE(EYE,
                C$& CPTR1,K,CPTR2,A,AMUL,EYE1,FACT,R,FR)
                CSRD$ LOOPLABEL 'DPFAC_do40'
B40-----------        DO i = 1, (-1)+cptr1(1+k)+(-cptr2(k)), 1
|               CSRD$ INDUCTION eye3, 'DPFAC_do40', 0, eye1, (-1)+cptr1(1+k)+eye1+(-1)*cptr2(k)
|                      IF ((-1)+cptr1(1+k)+(-cptr2(k)).NE.i) THEN
|                         fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|                         fr((-1)+eye1+i) = r(cptr2(k)+i)
|                         eye3 = eye1+i
|                  40     CONTINUE
|                      ELSE
|                         fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|                         fr((-1)+eye1+i) = r(cptr2(k)+i)
|                         eye = eye1+i
|                1002   CONTINUE
|                      ENDIF
B40-----------        ENDDO
```

Pues bien, el bucle DO 50 es considerado como serie por Polaris porque las variables `A`, `R`, `CPTR1`, `W` y `SHIFT`, pueden presentar dependencias generadas por bucle. Discutimos este punto más adelante. Sin embargo dentro de este bucle Polaris detecta una operación de **reducción de histograma** [119]: el bucle DO 80 que escribe en el vector W(I). Para este bucle, la salida de polaris se muestra en ejemplo 5.5.

Este bucle, que realiza la operación `W(FR(I))=W(FR(I))-AMUL*FACT(I)` para el rango de `I` en (`F1:F2`) no representa en realidad una operación de reducción. Ya ha sido mencionado que el rango `F1:F2` marca la sección de la columna del pivot activa que actua de vector x en la operación SAXPY dispersa. Por tanto es el rango de una sección de un vector comprimido donde los índices no se repiten en el vector `FR(I)`. Se dice que, en ese rango, `FR(I)` es un vector de permutación. Por tanto las escrituras en `W` se realizan en posiciones distintas para cuaquier `I` en el rango `F1:F2`. Dado que las escrituras son en posiciones distintas, el bucle es paralelo y no presenta el coste asociado a la implementación de la reducción paralela que vemos en el ejemplo anterior, y que comentamos a continuación.

La reducción presentada en el ejemplo 5.5 tiene tres etapas, cada una de ellas marcada por la directiva del compilador de fortran de SGI `DOACROSS`, que antecede a un bucle paralelo. La primera de ellas se encarga de la puesta a cero en paralelo de tantos arrays privados, `W0`, como procesadores estén colaborando en la reducción[5]. El segundo bucle realiza en paralelo la operación de reducción local. Finalmente el último `DOACROSS` recorre en paralelo el vector `W` donde para cada coordenada se van

[5]Las variables `cpuvar` y `noproc` indican el número de procesadores trabajando, mientras que `mp_my_threadnum()` devuelve el identificador de cada proceso.

sumando secuencialmente las aportaciones correspondientes de la copia privada, `W0`, a cada procesador.

Ejemplo 5.5 CÓDIGO GENERADO POR POLARIS PARA DO 80

```
                  ptr = malloc(8*n*noproc)
            C$DOACROSS IF((cpuvar.GT.5.OR.n.GT.300).AND.cpuvar*(6+n).GT.1000),
            C$& LOCAL(TPINIT,I),SHARE(CPUVAR,N,WO)
            CSRD$ LOOPLABEL 'DPFAC_do80_RINIT'
C-----------      DO i = 1, cpuvar, 1
|           CSRD$ LOOPLABEL 'DPFAC_do80_RINIT_0'
| D---------       DO tpinit = 1, n, 1
| |                 w0(tpinit, i) = 0.0
| D---------       ENDDO
C-----------      ENDDO
            C$DOACROSS IF(8+(eye+(-1)*f1).GT.75),LOCAL(I),SHARE(F1,EYE,WO,FR,FACT,
            C$& AMUL)
            CSRD$ LOOPLABEL 'DPFAC_do80'
C80---------      DO i = f1, (-1)+eye, 1
|                  w0(fr(i), mp_my_threadnum()+1) = w0(fr(i), mp_my_threadnum()+1
|              *)+(-fact(i))*amul
|            80    CONTINUE
C80---------      ENDDO
            C$DOACROSS IF((n.GT.5.OR.5*cpuvar.GT.300).AND.5*cpuvar+5*n.GT.150),
            C$& LOCAL(I,TPINIT),SHARE(N,CPUVAR,W,WO)
            CSRD$ LOOPLABEL 'DPFAC_do80_RSUM_0'
C-----------      DO tpinit = 1, n, 1
|           CSRD$ LOOPLABEL 'DPFAC_do80_RSUMI_0'
| D---------       DO i = 1, cpuvar, 1
| |                 w(tpinit) = w(tpinit)+w0(tpinit, i)
| D---------       ENDDO
C-----------      ENDDO
                  CALL free(ptr)
```

Aunque Polaris hubiese detectado de alguna forma que `FR(I)` es un vector de permutación en el rango `F1:F2` la paralelización del bucle DO 80 como tal, en lugar de su reducción, tampoco hubiera resuelto demasiado la falta de eficiencia. De hecho, en una modificacion manual de la salida de Polaris se paralelizan también los bucles 60, 70, 90 y 95 consiguiendo una sustancial mejora en el sentido de que el código paralelo al menos no es más lento que el secuencial, pero el resultado global sigue siendo muy bajo, ya que en general la eficiencia no superó el 2%. En efecto, los bucles 60, 70, 90 y 95 que Polaris, conservativamente, marca como secuenciales por existir riesgo de dependencia generada por bucle[6], son, al contrario, paralelos. Por ejemplo, es fácil demostrar que los bucles 60 y 95 no presentan dependencias generadas por bucle en las variables `A` y `R` si estos arrays han sido declarados con suficiente espacio. En particular, si una mitad del array es capaz de almacenar las columnas de la submatriz activa actualizada en cada iteración `k`, no existe solapamiento entre la zona del array `A` (`R`) que se lee y la que se escribe. Por otro lado, en caso contrario, incluso el programa secuencial se aborta por falta de espacio para encajar el llenado.

Al mismo tiempo, la dependencia generada por bucle en DO 70 debido a la sentencia `W(R(I))=A(I)`, se anula al comprobar que el rango de asignaciones corresponde al de una columna reducida, por lo que `R(I)` en ese rango es un vector de permutación.

El bucle 90 también se puede paralelizar mediante una privatización. La idea es que cada procesador ejecute ese bucle en paralelo leyendo cada uno de un bloque distinto de `W`. Las escrituras se deben realizar en arrays privados a cada procesador,

[6] Las variables que pueden provocar estas dependencias son `A` y `R` en los bucles 60, 90 y 95, `W` en el bucle 70, y `SHIFT` también en el bucle 90.

por ejemplo, `pA` y `pR`. Después, esos arrays privados se copian en una etapa paralela en la que cada procesador vuelca sus coeficientes en las variables de memoria compartida `A` y `R`, conociendo cada uno donde debe empezar a escribir. A esta última operación se la llama etapa de *copy-out*.

Sin embargo, como ya hemos dicho, aún con la ejecución en paralelo de todos estos bucles (30, 40, 60, 70, 80, 90 y 95) el resultado no es alentador. La razón es que estos bucles recorren columnas o secciones de columnas. El coste asociado a la ejecución paralela del procesado de una columna, no se compensa con la pequeña carga computacional presentada por las columnas dispersas.

Queda otra alternativa: es cierto que el trabajo dentro de una columna es independiente, pero también lo son las columnas entre sí. Retomemos la discusión sobre el paralelismo que presenta el bucle DO 50. Como decíamos, Polaris advierte que las variables `A`, `R`, `CPTR1`, `W` y `SHIFT`, pueden presentar dependencias, por tanto, conservativamente marca el bucle como secuencial. La situación es complicada ya que los arrays `A` y `R` se acceden mediante la varible de inducción `SHIFT`, que para empeorar las cosas, se incrementa condicionalmente (dependiendo de la cantidad de llenado en cada columna) en el bucle 90.

En efecto, las dependencias existen y el bucle no se puede ejecutar en paralelo en memoria compartida sin más. Pero tampoco deja de ser verdad que el procesado de una columna (`actcol()`) es independiente de las demás. Para desacoplar el trabajo entre columnas es necesario privatizar las columnas que van a ser procesadas en paralelo por cada procesador. Adicionalmente, el vector de trabajo `W` también se privatiza ya que se harán operaciones distintas sobre él en paralelo. Evidentemente, cada procesador tendrá una copia local de `SHIFT`. El problema es detectar automáticamente que esta paralelización es factible y realizar las privatizaciones correspondientes. Abordamos este aspecto en la siguiente sección.

5.3 Técnicas orientadas a códigos dispersos

Determinar de forma automática que las iteraciones del bucle DO 50 son independientes y paralelizarlo conlleva precisamente esas dos etapas. Por un lado hay que completar los test de dependencias para que sean capaces de inferir que el bucle es paralelo. En segundo lugar las dependencias generadas por bucle, que también van a ser detectadas, se pueden romper gracias a la privatización adecuada de las variables (escalares o arrays) que se definen y usan dentro de una única iteración [150]. Dedicamos una subsección a cada aspecto.

5.3.1 Detección de paralelismo: test de dependencias

Es evidente que dentro del bucle DO 50, las variables `A` y `R` se leen indexadas por el índice `I` que recorre una columna desde `CPTR1(I)` a `CPTR1(I+1)-1`, como bien puede inferir el analizador de acceso a regiones de Polaris [33]. Por otro lado las escrituras en `A` y `R` dentro del bucle siempre se indexan mediante la variable `SHIFT`. Con esto, y como discutimos en [16], dos hechos son suficientes para demostrar que el bucle que nos concierne se puede ejecutar en paralelo:

1. El rango de `SHIFT` no puede solapar el mismo rango de `SHIFT` para iteraciones del bucle `J` ejecutadas en otros procesadores. Estamos diciendo que no se puede escribir donde otro procesador va a escribir en paralelo. Más formalmente: decimos que no van a existir dependencias de salida.

2. El rango de `SHIFT` no puede solapar el rango `CPTR1(j), CPTR1(j+1)-1` recorrido en otros procesadores. Igual que antes, es intuitivo que no puedes leer de la zona en la que otro procesador va a escribir en paralelo. Formalmente, imponemos que no existan dependencias de flujo ni anti-dependencias.

Para probar el primer punto es claramente suficiente con realizar el siguiente análisis simbólico: siempre que se escribe en `A` y `R` la variable `SHIFT` se incrementa, por lo que dentro del bucle nunca se va a escribir dos veces en la misma posición de estos vectores. Es decir, no existen dependencias de salida.

Es más, el compilador podría hacer, aunque no es necesario para este primer punto, un análisis más completo demostrando que la variable `SHIFT` es estrictamente creciente. Esto puede ser inferido fácilmente por el compilador, haciendo un estudio simbólico de las sentencias que escriben en `SHIFT` dentro del bucle. Es fácil ver que la variable nunca se decrementa. Adicionalmente, la condición inicial impuesta en el bucle DO 20 garantiza que el bucle 60 ejecuta al menos una iteración, con lo que `SHIFT` se incrementa al menos en una unidad. Por tanto, uno de los invariantes de este bucle es que la variable de inducción `SHIFT` es estrictamente creciente. Por tanto, no existe dependencias de salida.

El segundo test debe demostrar que no existen dependencias de flujo ni anti-dependencias, es decir que el rango de `I (CPTR1(j):CPTR1(j+1)-1)` no va a solapar el rango de `SHIFT` en otros procesadores. En este caso necesitamos de un test en tiempo de ejecución ya que el contenido de `SHIFT` y `CPTR1` no es conocido en tiempo de compilación. Este test debe probar que:

- el índice más pequeño que accede al vector `A` para lectura, `min_i`, es mayor que el índice más grande que accede al vector `A` para escritura, `max_shift`, y,

- el índice más grande que accede al vector `A` para lectura, `max_i`, es menor que el índice más pequeño que accede al vector `A` para escritura, `min_shift`.

Si no tenemos ninguna información adicional sobre el vector `CPTR1`, la búsqueda del valor mínimo y máximo de `I` debe hacerse recorriendo las `n-k` componentes del vector. Sin embargo, el compilador puede eliminar complejidad al comprobar que `CPTR1` es no decreciente[7] antes de entrar en el bucle 50 y para cada iteración del bucle `K`. Efectivamente, un análsis simbólico determina que a este array se le asigna, en cada iteración del bucle, la variable `C1`. Ésta, a su vez toma su valor de `SHIFT`, que, como ya hemos demostrado, es estrictamente creciente. Por tanto en este caso, `CPTR1` es

[7] Es decir `CPTR1(J)`$\leq$`CPTR1(J+1)` para todo el rango de `J` (entre `K+1` y `N`).

estrictamente creciente después de la primera iteración del bucle K[8]. Si esto es así, para una iteración completa del bucle DO 50 con J barriendo entre K+1 y N, tenemos min_i=CPTR1(K+1) y max_i=CPTR1(N+1)-1.

En cuanto al valor mínimo y máximo de SHIFT, la situación es un poco más compleja, ya que SHIFT se incrementa condicionalmente en el bucle 90. Por un lado, tenemos trivialmente que min_shift=SHIFT, pero debemos estimar el valor max_shift suponiendo el caso peor. Considerando el máximo número de escrituras en A y R obtenemos max_shift=SHIFT+(N-K)*(F2-F1+1)+max_i-min_i.

Con todo, en el ejemplo 5.6 se muestra el test de dependencias en tiempo de ejecución. Este test comprueba que no hay solapamiento en el bucle 50, entre posiciones leídas de A y R y posiciones escritas. Vemos como la variable booleana parallel se inicializa a .false. si hay solapamiento. En ese caso el bucle DO 50, conservativamente, se ejecuta en secuencial gracias a la clausual IF(parallel) de la directiva DOACROSS que precede al bucle.

Ejemplo 5.6 Test en tiempo de ejecución en DO 50

```
        if(maxi.ge.minsh .and. maxsh.ge.mini) then
           parallel = .true.
        else
           parallel = .false.
        end if

C$DOACROSS if(parallel), local(...), share(...)
        DO 50 J=K+1,N
           ....
   50   CONTINUE
```

5.3.2 Paralelización: privatización automática

Una vez sabemos que no hay dependencias, es decir, el bucle es paralelo, hay que abordar la generación del código paralelo para ese bucle. Es decir, generar el código que permita que los procesadores procesen conjuntos disjuntos de columnas en paralelo. Sin embargo, en el código secuencial existen variables escalares (C1, C2, AMUL, SHIFT) y vectoriales (el vector de trabajo W) que se utilizan para procesar una columna. Cada procesador debe privatizar estas variables para que el procesado de su columna no interfiera con el procesado de las demás que se está realizando en paralelo.

La privatización es una técnica que elimina las anti-dependencias entre iteraciones (generadas por bucle) [151]. Una variable se privatiza cuando cada procesador realiza una copia de la variable y sólo accede a su copia local. En cada iteración de un bucle, si toda lectura de una variable está dominada por alguna definición de esta variable en la misma iteración, podemos afirmar que la variable es privatizable. También son privatizables las variables que tienen al final de una iteración el mismo valor que tenían al comienzo.

[8]Si queremos probar la monotonía del vector CPTR1 para K=1 podemos probar que CPTR(I) $\leq$ CPTR(I+1) para I=1,n.

Evidentemente son privatizables las variables que no conducen a dependencias generadas por bucle, es decir aquellas que se definen y usan en una iteración sin provocar efectos laterales en ninguna otra iteración, por ejemplo `C1, C2, AMUL`. Queremos recordar que esta privatización, que bajo este paradigma es automática, se lleva a cabo en los compiladores semi-automáticos de paralelismo de datos utilizando la cláusula NEW junto a la directiva `INDEPENDENT`.

La variable `SHIFT` sí tiene efectos laterales en otras iteraciones, pero es una variable de inducción que tiene un tratamiento especial a comentar en un parrafo posterior. Por último, se debe demostrar que el array `W` no induce a dependencias generadas por bucle para poder privatizar todas las variables de trabajo usadas en el bucle. Para determinar si un array es privatizable, analizamos las secciones del array que son definidas y usadas en cada iteración. La sección usada de un array está dominada por la sección definida de un array si la sección definida cubre la sección usada [151]. En nuestro caso, el patrón de acceso a la variable W(I) se muestra de forma simplificada en el ejemplo 5.7.

Ejemplo 5.7 Patrón de acceso a `W` en DO 50

```
do i=1,n
     W(i)=0
end do

do j=...
    ...
    do i=...
        W(R(i))=...
    end do
    do i=...
        W(FR(i))=...
    end do

    do i=1,n
        if(W(i).ne.0) then
            ...=W(i)
            ...
            W(i)=0
        end if
    end do
end do
```

Esta situación es más complicada que las que se presentan en [151], debido a la escritura en `W` mediante indirecciones a través de los arrays de índices `R` y `FR`. Sin embargo la solución es fácilmente implementable en Polaris. Un análisis simbólico reporta que el array `W`, inicializado a cero al comienzo del programa, vuelve a estar en ese estado al acabar cada iteración `j`, si `R(i)` y `FR(i)` toman valores entre uno y `n` en el rango en el que están definidos estos vectores. Esa comprobación es factible mediante un rápido test en tiempo de ejecución que determine que esa condición es cierta, una vez, al comienzo del programa, para `R(i)`, unido al siguiente análisis simbólico de propagación de rangos [33]: `FR(EYE)` se copia de `R(i)` fuera del bucle 50, y `R(SHIFT)` se copia de `R(i)` dentro de este bucle, excepto en el bucle 90 donde tenemos `R(SHIFT)=I` con `I` en el rango `1:N`. Por tanto Polaris puede concluir que el rango de valores almacenados en `R` y `FR` es (`1:N`) y por tanto el array es privatizable. Existen otros patrones, también en el ámbito de la computación dispersa, que se pueden tratar de forma similar y que abordamos en la última sección de este capítulo.

En cuanto a los arrays `A` y `R`, su acceso en escritura está determinado por la variable de inducción `SHIFT`. En estas situaciones la paralelización presenta dos alternativas:

- Cuando el bucle que maneja la variable de inducción se puede expresar en su forma cerrada (del inglés *closed form*), es decir expresar la variable de inducción como una función del índice del bucle, no hay problema. Bajo este supuesto es fácil determinar en qué posición del array debe escribir cada procesador.
- Sin embargo, existen situaciones en las que esto no es posible. Por ejemplo, en nuestro caso, la variable `SHIFT` se incrementa condicionalmente en bucle 90 (–etapa *gahter*– por lo que el incremento de `SHIFT` depende del llenado que sufra cada columna en particular). Por tanto un procesador no sabe donde debe escribir hasta que no se haya completado la escritura en el procesador que gestiona la columna previa.

Si el bucle es paralelizable, la solución en esta segunda situación está de nuevo en el uso de técnicas de privatización. Es decir, se debe reservar un espacio local y temporal para copiar las columnas que corresponda actualizar a cada procesador. Más precisamente, en nuestro caso, la escritura se realizará sobre vectores privados, `pA` y `pR`, indexados mediante la también privatizada `pSHIFT`[9]. Tras el proceso de actualización local, debe existir una etapa en la que se calcule a partir de qué posición del array global, `A` y `R`, debe escribir cada procesador. En este punto, cada procesador vuelca en paralelo, sus vectores locales sobre los globales, en una etapa llamada de **copy out**.

Otro prisma desde el que mirar el mismo problema se presenta en [119] en el que la privatización de `A` y `R` es consecuencia de la detección automática de la operación *gather* del bucle 90. El autor de este trabajo, presenta la técnica de privatización como método de paralelización de operaciones de fusión de bucle (del inglés *coalescing loop operation*) siempre que éstas sean asociativas, y la operación *gather* lo es.

5.3.3 Evaluación del código generado

Algunas de las técnicas comentadas no están totalmente disponibles aún en Polaris[10]. Antes de incluirlas en el compilador es necesario validar que permitirán conseguir mejoras significativas en la eficiencia de los códigos paralelos generados. Por tanto, hemos simulado a mano la hipotética salida del futuro compilador con este y con otros códigos, como presentamos en la siguiente sección. Como muestra, presentamos la sección de código generada para el bucle DO 50 en el ejemplo 5.8.

Las variables escalares se privatizan mediante la cláusula `local()` en el `DOACROSS`. La excepción es `pSHIFT` que se privatiza dimensionandola en función del número de procesos –`mp_numthreads()`–. Esto es así ya que `pSHIFT` debe mantenter su valor a la salida del bucle paralelo. Para evitar la compartición falsa (del inglés *false sharing*) en la cache, el array `pSHIFT` se accede en posiciones que corresponden a líneas distintas de cache. Los vectores `pA, pW, pR, pCPTR1 y pCPTR2` son privatizados siguiendo la

[9] Utilizaremos el prefijo "p" minúscula para referirnos a la copia local asociada a los distintos vectores y escalares.

[10] Aunque ya ha comenzado su implementación.

misma técnica: se les añade una dimensión adicional que es indexada en función del número de proceso –`mp_my_threadnum()`–. Como se puede apreciar en el ejemplo, el código no cambia, salvo en que se escribe en las variables privadas en lugar de en las variables globales.

En [16] presentamos una aceleración de 3.84, conseguida con esta versión del algoritmo, para la matriz LNS3937 (ver tabla 1.2), en los 8 procesadores R4400 a 150MHz del multiprocesador de memoria compartida SGI Challenge. Esa aceleración se traduce en una eficiencia del 48%, la cual mejora muchos de los resultados obtenidos en memoria compartida, como los discutidos en 3.1, con la diferencia de que estos últimos están desarrollados y optimizados manualmente.

Ejemplo 5.8 SIMULACIÓN DE LA SALIDA DE POLARIS PARA DO 50

```
        do i = 1,mp_numthreads()
           psi = lshift(i-1,6)+33
           pSHIFT(psi) = 1
        enddo
C$DOACROSS if(parallel),local(pc1,pc2,psi,amul,j,tid,i,ii),share(pA,pW,pR,pCPTR1,pCPTR2)
        DO 50 J=K+1,N
           tid = mp_my_threadnum() + 1
           psi = pSHIFT(lshift(tid-1,6)+33)
           pC1=psi
           DO 60 I=CPTR1(J),CPTR2(J) ! Al menos se ejecuta una vez el bucle.
              pA(psi,tid)=A(I)
              pR(psi,tid)=R(I)
              psi=psi+1
 60        CONTINUE
           pC2=psi-1
C Copia el elemento de la fila K
           IF (pR(pC2,tid).EQ.K) THEN
              AMUL=pA(pC2,tid)
              pC2=pC2+1
C Scatter
              DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
                 pW(R(I),tid)=A(I)
 70           CONTINUE
C Actualiza
              DO 80 I=F1,F2
                 pW(FR(I),tid)=pW(FR(I),tid)-(AMUL*FACT(I))
 80           CONTINUE
C Gather
              DO 90 I=1,N
                 IF (pW(I,tid).EQ.0) GOTO 90
                 pA(psi,tid)=pW(I,tid)
                 pR(psi,tid)=I
                 psi=psi+1
                 pW(I,tid)=0
 90           CONTINUE
           ELSE
              DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
                 pA(psi,tid)=A(I)
                 pR(psi,tid)=R(I)
                 psi=psi+1
 95           CONTINUE
           END IF
           pCPTR1(J,tid)=pC1
           pCPTR2(J,tid)=pC2
           pSHIFT(lshift(tid-1,6)+33) = psi
 50     CONTINUE
```

Un estudio más detallado que el presentado en [16] se ha realizado sobre una plataforma más moderna: el SGI Power Challenge con 16 procesadores R10000. La tabla 5.3 resume los resultados para la misma matriz LNS3937.

En esta tabla, la primera columna contiene el tiempo en segundos del algoritmo secuencial inicial sin ninguna modificación. El resto de las columnas dan una medida de las prestaciones del algoritmo paralelo para distinto número de procesadores. Como

	Secuencial	1	2	4	8	16
Tiempo (seg.)	77	138	67	31	18	15
Aceleración	–	0.56	1.15	2.48	4.28	5.13
Eficiencia	–	56%	57%	62%	53%	32%

Tabla 5.3: Prestaciones del algoritmo en el SGI Power Challenge.

vemos, la eficiencia se mantiene por encima del 50% para 2, 4, y 8 procesadores, aunque baja significativamente en 16 PE's. La principal razón de esta pérdida de escalabilidad no es tanto el coste introducido por el algoritmo paralelo sino el desbalanceo en la planificación del bucle. Esto es así ya que las iteraciones del bucle 50 se planifican según un esquema por bloques para simplificar la operación de copy-out. Sin embargo esto redunda en una mayor carga para el procesador que se queda con el último bloque, debido al llenado de la matriz.

También es apreciable cómo el algoritmo secuencial es un 56% más rápido que el paralelo ejecutado en un único procesador. Esto nos da una idea del coste que está introduciendo la paralelización del bucle 50, principalmente la etapa copy-out que implica un gran volumen de movimiento de datos.

Por tanto, aunque suficientes, a pesar de todo estos resultados son mejorables, pero por el momento no automáticamente. En efecto, la etapa de *copy out* es costosa porque implica un gran movimiento de datos. Esta etapa ha sido eliminada manualmente para estudiar la mejora. Las modificiaciones implicaban la utilización de otro vector de punteros para indicar el final de cada columna, permitiendo así que las columnas no sean consecutivas. De esta forma se puede simular un algoritmo en el que el bucle DO 50 se ejecuta siempre sobre arrays privados, lo cual, en cierto modo, no es más que una emulación de la versión del algoritmo para memoria distribuida (ver sección 5.2).

5.4 Aplicación a otros problemas dispersos

A pesar de los resultados obtenidos, la dura tarea de automatizar estas técnicas en Polaris no se justifica si no van a tener mayor aplicación que la de paralelizar automáticamente nuestro código de factorización dispersa. Sin embargo, el caso de estudio elegido, como ya se ha dicho en otras ocasiones, presenta un conjunto de características que lo convierten en un buen caso para extraer conclusiones aplicables a problemas similares. Las características a las que nos referimos son básicamente dos: el uso de una estructura de datos comprimida (CRS o CCS) y la existencia de llenado durante la computación. En general son dos aspectos muy frecuentes en computación de problemas dispersos.

En [16] se discuten los resultados de la aplicacion de estas técnicas y otras a un conjunto de códigos irregulares. En las siguientes subsecciones presentamos un resumen de ese trabajo, significativamente extendido con otros códigos dispersos o irregulares. En general, hemos encotrando que las técnicas de detección de paralelismo y privatización discutidas anteriormente son aplicables con éxito a una gran variedad de códigos reales.

5.4.1 Operaciones vectoriales y matriciales

Uno de los códigos de test (*benchmarks*) discutidos en [16] contempla un conjunto de rutinas dispersas, entre las que se incluyen el producto matriz dispersa por vector y el producto y suma de dos matrices. En ese programa las rutinas incluidas siempre escriben el resultado de la operación sobre un vector o una matriz densa. En estos casos, la computación no presenta dependencias ya que las estructuras dispersas únicamente se leen, y la escritura en matrices densas es regular y fácilmente paralelizable. Por ejemplo, el producto $y = Ax$, donde A es una matriz dispersa en formato CRS o CCS, y x e y son vectores densos, presenta el código mostrado en el ejemplo 5.9.

Ejemplo 5.9 Códigos para el producto matriz dispersa vector

CRS

```
      DO 100 I=1,n
         Y(I)=0.0
         DO 100 J=Filpt(I),Filpt(I+1)-1
            Y(I)=Y(I)+Val(J)*X(Col(J))
100   CONTINUE
```

CCS

```
      Y(1:n)=0.0
      DO 200 I=1,n
         DO 200 J=Colpt(I),Colpt(I+1)-1
            Y(Fil(J))=Y(Fil(J))+Val(J)*X(I)
200   CONTINUE
```

En este ejemplo, el bucle DO 100 marca el núcleo computacional del producto, cuando la matriz `A` está representada mediante una estructura CRS. Como vemos, el bucle más externo es paralelo ya que las escrituras se realizan en componentes distintas de `Y` en cada iteración. Si representamos `A` mediante una etructura CCS, el algoritmo cambia, como se muestra en la sección de código delimitada por el bucle DO 200. En este caso, Polaris determina que el bucle más externo representa una reducción, y por tanto aplica la técnica de paralelización de histograma descrita en [119]. Idénticas conclusiones se extraen del estudio de códigos de suma y producto de matrices dispersas, cuando el resultado se almacena sobre una matriz densa.

El problema es más interesante y presenta un mayor desafío, si el resultado de la suma o producto de dos matrices dispersas se almacena también de forma dispersa. En cuanto al algoritmo de suma, como ocurre en otros problemas dispersos, éste se suele dividir en dos etapas: suma simbólica, en la que se determina el patrón de la matriz resultado; y suma numérica que realiza las operaciones de suma apoyándose en la información obtenida de la etapa anterior [112].

En el ejemplo 5.10 tenemos la sección de código asociada a la suma numérica de dos matrices dispersas, `A` y `B` en formato CRS, resultando la matriz suma `C` también en formato CRS. Las tres matrices se representan mediante los vectores `VAL`, para los coeficientes, `IND`, para los índices de columna y `PTR`, que apunta a las filas, precedidos de las letras `A`, `B` y `C`, respectivamente. Como vemos, si la etapa de anáisis ha determinado inicialmente (en un tiempo despreciable) el patrón de `C`, la suma numérica tiene tres pasos: (i) desempaquetar una fila de `A` en un vector de trabajo `X`; (ii) sumar en `X` la fila de `B`; y (iii) empaquetar los elementos no nulos de `C` en la estructura CRS.

Es fácil ver que una vez conocido el patrón de la matriz resultado, el bucle 50 del ejemplo 5.10 es paralelo. Polaris analiza los vectores en los que se escribe, `CVAL` y `X`. En cuanto al primero, vemos que está indexado por la variable `IP` recorriendo

CPTR(I):CPTR(I+1)-1. Por tanto es aplicable la técnica de monotonía en el vector de punteros, CPTR, que va a consistir en una única prueba en tiempo de ejecución antes del bucle. En cuanto a X, de nuevo, se puede demostrar que es un vector de trabajo privatizable ya que el análisis de rangos junto con un test en tiempo de ejecución pueden comprobar que los coeficientes escritos en X en los bucles 20 y 30 son leídos y reinicializados a cero en el bucle 40.

Ejemplo 5.10 Suma numérica de dos matrices dispersas

```
      DO 10 I=1,N
         X(I)=0
10    CONTINUE
      DO 50 I=1,N
         DO 20 IP=APTR(I),APTR(I+1)-1
            X(AIND(IP))=AVAL(IP)
20       CONTINUE
         DO 30 IP=BPTR(I),BPTR(I+1)-1
            X(BIND(IP))=X(BIND(IP)+BVAL(IP)
30       CONTINUE
         DO 40 IP=CPTR(I),CPTR(I+1)-1
            CVAL(IP)=X(CIND(IP))
            X(CIND(IP))=0
40       CONTINUE
50    CONTINUE
```

Comentarios análogos merece el estudio del producto disperso, cuya sección numérica presentamos en el ejemplo 5.11. En este caso, multiplicamos A $\times$ B, ambas con almacenamiento CRS y escribimos el resultado en la matriz dispersa C también CRS. La etapa de análisis previa proporciona el patrón de la matriz producto y consume un tiempo de ejecución insignificante. Al igual que en el caso anterior, las escrituras en las distintas filas de C se pueden ejecutar en paralelo al demostrar la monotonía del vector CPTR, así como la privatización del vector de trabajo se puede inferir del análisis de rangos discutidos en el párrafo anterior. De esta forma concluimos que también para este código, Polaris podría paralelizar automáticamente el bucle 50.

Ejemplo 5.11 Producto numérico de dos matrices dispersas

```
      DO 10 JP=1,N
         X(JP)=0
10    CONTINUE

      DO 50 I=1,N
         DO 20 JP=APTR(I),APTR(I+1)-1
            DO 30 KP=BPTR(AIND(JP)),BPTR(AIND(JP)+1)-1
               X(BIND(KP))=X(BIND(KP)+AVAL(JP)*BVAL(KP)
30          CONTINUE
20       CONTINUE

         DO 40 J=CPTR(I),CPTR(I+1)-1
            CVAL(J)=X(CIND(J))
            X(CIND(J))=0
40       CONTINUE
50    CONTINUE
```

5.4.2 Transposición de matrices

Otra situación distinta se presenta en el código de transposición de matrices descrito en el ejemplo 5.12. La matriz de entrada `A` de `N`×`M`, se almacena en formato CRS mediante los vectores (`AVAL, AIND, APTR`). A la salida, obtenemos la matriz transpuesta `AT` en una estructura CRS identificada por los vectores (`ATVAL, ATIND, ATPTR`).

Como vemos, aparte del cálculo del histograma de `A` en el bucle 20, también presenta un paralelismo trivial el bucle 50. En éste, el índice `AIND(J)` selecciona a través de una indirección la posición `K` en la que se va a escribir. Para ejecutar este bucle en paralelo sólo hace falta demostrar que: (i) `AIND` es un vector de permutación en el rango `(APTR(I),APTR(I+1)-1)`; y que (ii) por otro lado, no se va a escribir dos veces en la misma posición de `ATIND` y `ATVAL`, ya que cada vez que escribimos en una posición `K=ATPTR(J)`, `ATPTR(J)` se incrementa en una unidad (monotonía).

Ejemplo 5.12 Transposición de una matriz dispersa

```
      DO 10 I=3,M+1
         ATPTR(I)=0
10    CONTINUE
      DO 20 I=1,APTR(N+1)-1
         J=AIND(I)+2
         ATPTR(J)=ATPTR(J)+1
20    CONTINUE
      ATPTR(1)=1
      ATPTR(2)=1
      DO 30 I=3,M+1
         ATPTR(I)=ATPTR(I)+ATPTR(I-1)
30    CONTINUE
      DO 60 I=1,N
         DO 50 JP=APTR(I),APTR(I+1)-1
            J=AIND(JP)+1
            K=ATPTR(J)
            ATIND(K)=I
            ATVAL(K)=AVAL(J)
            ATPTR(J)=K+1
50       CONTINUE
60    CONTINUE
```

5.4.3 Factorización QR y Cholesky multifrontal

Aparte de la factorización dispersa LU, existen otros métodos directos de aplicación en problemas más particulares. Por ejemplo, para factorizar una matriz simétrica, la factorización de Cholesky consigue menores tiempos de ejecución que el algoritmo general LU. Por otro lado, la factorización QR, aunque provoca un mayor llenado y consume un mayor tiempo en la factorización, es de utilidad en problemas de mínimos cuadrados o en la factorización de matrices muy mal condicionadas.

En cuanto a la factorización QR, las variantes Gram-Schmidt Modificado y Householder, presentan en su algoritmo una etapa de actualización análoga a la que encontramos en la factorización LU. En estas dos variantes, el procesado de las columnas es independiente y el bucle que actualiza las `n-k` columnas en la iteración `k` es también paralelo. Por tanto, la discusión acerca de las técnicas necesarias para paralelizar automáticamente estos códigos es idéntica a la expuesta en 5.3.

Un comentario análogo merece la factorización de Cholesky, si utilizamos una aproximación general (ver esquema de la página 8). Para encarar otro tipo de organizaciones, hemos estudiado la aplicabilidad de nuestras técnicas a un algoritmo Cholesky multifrontal, ya que ésta es la solución más eficiente para factorizar matrices simétricas. El código secuencial, discutido en [65], utiliza rutinas SPARSPACK [72] para las etapas de reordenación, usando el algoritmo de mínimo grado, y de factorización simbólica. A continuación construyen el árbol de eliminación para pasar a la etapa de factorización numérica. En este punto se conoce el patrón de la matriz factorizada. La etapa de factorización numérica está delimitada por un bucle de índice `k` que recorre las columnas. En su interior, se distinguen dos situaciones: la columna `k` es la última de un supernodo (ver figura 1.1), o no. En el primer caso un bucle, DO 1020, aplica una rutina de actualización densa a las restantes columnas del supernodo. En la seguna situación, el bucle DO 1021 actualiza las columnas afectadas mediante llamadas a una rutina de actualización dispersa. Estos dos bucles consumen el 82% del tiempo de ejecución del algorimto completo. Dado que la estructura de datos asociada es CCS, tiene sentido aplicar las técnicas de detección de paralelismo y privatización presentadas. En este caso sólo es necesario probar el no solapamiento de las secciones (columnas) escritas en las actualizaciones para detectar que los dos bucles son paralelos. De nuevo, la detección de arrays monótonos crecientes usados para acotar bucles, se muestra útil.

El código paralelo, entendido como la salida simulada de una futura versión mejorada de Polaris, reporta una eficiencia en el SGI Power Challenge, del 55% en 16 procesadores y 60% en 8, si atendemos sólo a las prestaciones de los bucles paralelos. Dado que un 18% del código queda sin paralelizar, las prestaciones del código completo son inferiores (20% y 35% respectivamente). El algoritmo puede ser optimizado a mano si paralelizamos la rutina de reordenación de mínimo grado que consume un 11% del tiempo total (la factorización simbólica y la construcción del árbol de eliminación son menos significativas). El nuevo código paralelo cubre el 93% del tiempo de ejecución y la eficiencia es de 30% en 16 procesadores y 47% en 8. Sin embargo, no está claro aún la posibilidad de desarrollar nuevas técnicas para la paralelización automática del algoritmo de mínimo grado, aspecto que mantenemos en evaluación.

5.4.4 Otros códigos irregulares

Adicionalmente, otros códigos irregulares son abordados en [16], y recogidos aquí en la tabla 5.4. Tres de ellos están incluidos en el compendio de aplicaciones que motivan la extensión de HPF-2 [66]. El código de Lanczos, desarroyado en nuestro grupo se discute detalladamente en [149], mientras GCCG proviene del Instituto de Tecnología Software y Sistemas Paralelos de la Univ. de Viena.

Aparte del tamaño en líneas de código de cada programa, presentamos en las últimas dos columnas de esta tabla, la aceleración alcanzada en los 8 procesadores del SGI Challenge, para la versión paralela generada por PFA [137] y por Polaris.

Si empezamos con los algoritmos que presentan mejores resultados, encontramos GCCG y LANCZOS. Ambos contienen indirecciones únicamente en la parte derecha de las asignaciones, por lo que no aparecen problemas de dependencias. Adicionalmente

Programa	Descripción	Origen	líneas	PFA	Polaris
GCCG	Dinámica de fluidos	Vienna	407	6.32	8.49
LANCZOS	Cálculo de autovalores	Malaga	269	7.36	7.17
NBFC	Dinámica molecular	HPF-2	206	0.99	5.00
EULER	Euler en una malla 3D	HPF-2	1990	1.02	1.97 (5.38)
DSMC3D	Monte Carlo	HPF-2	1794	1.02	1.22 (4.95)

Tabla 5.4: Aceleración para otros códigos.

las reducciones que aparecen son del tipo escalar o reducciónes de "dirección única"[11]. Por tanto, ni Polaris ni PFA encuentran demasiado problema al generar un buen código paralelo.

En cuanto al algoritmo NBFC, el problema es distinto. El núcleo de la computación está dominado por un triple bucle anidado que realiza una operación de reducción. En este caso la reducción es de histograma[12] ya que aparece una indirección en el lado izquierdo de la asignación asociada a la reducción. Esta situación está contemplada actualmente en Polaris [119], aunque no en PFA, por lo que apreciamos una diferencia en las prestaciones de una y otra versión.

En los dos últimos códigos, EULER y DSMC3D, Polaris consigue mejores resultados que PFA debido a que contempla la técnica de reducción de histograma. Sin embargo, como vemos la mejora no es suficiente debido por un lado al gasto que introduce la reducción de histograma (ver ejemplo 5.5), y por otro a que estas reducciones no tienen el suficiente peso en términos de porcentaje sobre el tiempo total de ejecución. En DSMC3D, hemos aplicado manualmente la técnica que prueba el no solapamiento de las variables de inducción junto con la detección de generadores de números aleatorios, consiguiendo un incremento de la aceleración de 1.22 a 4.95. En el código de EULER se ha mostrado útil la técnica que prueba la existencia de vectores de permutación. Explotando esta alternativa hemos conseguido subir la aceleración de 1.97 a 5.38 en 8 procesadores.

5.4.5 Resumen de técnicas

En general, el paralelismo entre columnas o filas de una matriz dispersa es factible en muchas ocasiones, sobre todo dentro del ámbito del cálculo numérico.

Evidentemente, el compilador no puede inferir de forma inmediata, y únicamente de la información que encuentra en el código, que éste procesa columnas o filas de una matriz dispersa. Es importante "enseñarle" cual es el patrón de acceso típico en esta clase de códigos. De otra forma: no siempre es útil la aplicación de las técnicas de detección y privatización de paralelismo discutidas en 5.3. Por tanto, debemos diseñar el compilador de forma que sepa cuándo debe aplicarlas. En una primera aproximación, el compilador aplicará estas técnicas cuando encuentre el patrón mostrado en el ejemplo

[11]Es decir, la variable que se reduce es un único elemento de una array.
[12]Varias componentes del array se reducen en cualquier orden. Ver [119].

5.13, y presente en todos los códigos discutidos en esta sección.

Ejemplo 5.13 Patrón del problema estudiado

```
      DO 10 I= ...
        ....
        DO 20 J=PTR(I),PTR(I+1)-1
          ....
          .... [ARRAY(IND(J))] .... [VAL(J)] ....
          ....
20      CONTINUE
10    CONTINUE
```

En este patrón, los arrays entre corchetes indican que pueden aparecer opcionalmente, y los puntos suspensivos, representan cualquier sentencia o función válida en F77. Este patrón es claramente indicativo de que, con un porcentaje elevado de probabilidades, el código esta haciendo uso de una estructura CRS o CCS. El compilador intentara demostrarlo probando las propiedades que caracterizan este tipo de estructuras de datos, a saber: el vector `PTR` es monótono no-decreciente; el vector `IND` es de permutación en el rango `PTR(I):PTR(I+1)-1`; y el array de índices contiene valores en un determinado rango.

Si se demuestran estas propiedades, es posible sacar provecho del paralelismo normalmente asociado a la independencia en el procesado de las filas o las columnas de la matriz. Como vemos, este patrón presentado puede resultar muy genérico.

Otra aproximación intermedia entre el paralelismo de datos y la paralelización automática consiste en darle alguna información sobre la estructura de datos al compilador. Por ejemplo, asegurandole que los vectores `VAL`, `IND` y `PTR` representan una matriz dispersa en formato CCS, libera al compilador de tener que demostrar que PTR es no decreciente, que los valores contenidos en `IND` estan en el rango `1:n` donde `n` es la dimensión de la matriz, y que `IND` puede ser visto en la sección que representa cada columna como un vector de permutación. Es decir, le damos las propiedades implícitas en el formato de representación de los datos.

Por tanto, una primera solución puede consistir en dejar al compilador intentar encontrar el máximo paralelismo posible. Pero si en alguna de las fases de análisis simbólico el compilador no es capaz de demostrar lo que busca, puede formular la pregunta pertinente al usuario, para volver a compilar sobre un supuesto adicional.

En resumen, las cuatro técnicas útiles para paralelizar muchos de los códigos dispersos (y otros por extensión que presentan similares propiedades) se discuten a continuación. Adicionalmente, ya que el patrón del ejemplo 5.13 es indicativo pero muy general, adjuntamos a cada técnica el patrón particular que provoca la aplicación de dicha técnica.

Monotonía en los arrays que acotan bucles

El ejemplo 5.14 muestra el patrón típico que identifica cuando usar esta técnica. Es claro que las escrituras en VAL son independientes para cada iteración `I` si el vector

PTR es mónotono no decreciente. Más formalmente, se debe cumplir la expresión:

$$\bigcap_{i=1}^{n}[PTR(I) : PTR(I+1)-1] = \emptyset \tag{5.1}$$

Ejemplo 5.14 PATRÓN PARA APLICAR "MONOTONÍA"

```
      DO 10 I= ...
         DO 20 J=PTR(I),PTR(I+1)-1
            VAL(J)=....
20       CONTINUE
10    CONTINUE
```

En general para detectar si un vector, `PTR` es no decreciente en un rango `(1:n)` se estudian tres situaciones:

- Si el vector no se modifica durante la ejecución del programa, es suficiente un sencillo test en tiempo de ejecución, consistente en un bucle al comienzo del programa que chequea que `PTR(I)` $\leq$ `PTR(I+1)` para `I=1,n`.
- Si las componentes del vector `PTR` son asignadas durante la ejecución, se hace un estudio simbólico de las variables relacionadas con la escritura en `PTR`. Suponiendo que la condición es cierta al comienzo del programa (lo cual es probado posteriormente mediante el test del punto anterior), en muchas situaciones se puede inferir que se mantiene invariante en ciertos puntos del programa.
- En el caso en el que el punto anterior no pueda decidir sobre la monotonía del vector, el compilador puede formular la pregunta correspondiente al usuario para volver a compilar con esa información.

No solapamiento de variables de inducción

El caso en el que la escritura en un array se realiza dentro de un bucle, indexada por una variable de inducción en lugar de la variable del índice, se presenta en el ejemplo 5.15.

Ejemplo 5.15 PATRÓN PARA APLICAR "NO SOLAPAMIENTO"

```
      DO 10 I= ...                     DO 20 J=M, N
         ...                              DO 30 I=PTR(J),PTR(J+1)-1
         VAL(J)=VAL(I) ...                   VAL(K)=VAL(I) ...
         ....                                K=K + (valor_positivo)
         J=J + (valor_positivo)      30       CONTINUE
 10   CONTINUE                       20    CONTINUE
```

Para determinar si el bucle `I` del ejemplo anterior es paralelo se estudian dos situaciones:

- Si podemos expresar la variable `J` en función del índice `I` decimos que el bucle se puede expresar en su forma cerrada. En este caso se sustituye `J` por la función de `I` y se aplican los tests de dependencias estándar. Adicionalmente, si el último valor de `J` se utiliza fuera del bucle, aplicamos técnicas de "valor último" [119].
- En caso contrario, por ejemplo cuando no se conoce la cantidad `valor_positivo` en tiempo de compilación, una primera aproximación que demuestra la inexistencia de dependencias de flujo o anti-dependecias consiste en probar que el rango de `J` no solapa el rango de `I`.

Este segundo caso se puede presentar en una gran variedad de situaciones complicando significativamente el estudio general. Por tanto nuestra solución se centra en un caso particular del anterior, caracterizado por el bucle `J` del ejemplo 5.15 y siempre que se pueda estimar una cota máxima del `valor_positivo` en todas la iteraciones. En este supuesto, para poder ejecutar el bucle `J` en paralelo debemos:

1. Encontrar el valor máximo y mínimo de `I` para todo el rango de `J`. Esto se puede inferir de recorrer el vector `PTR` en tiempo de ejecución, o más económicamente, si se ha probado la monotonía de `PTR` usando la técnica anterior.
2. Determinar el valor mínimo de la variable de inducción `K` y estimar en tiempo de compilación la formula que especifica el valor máximo de `K` en el caso peor.
3. Aplicar el test en tiempo de ejecución que prueba que

$$(min_I > max_K) \wedge (max_I < min_K) \tag{5.2}$$

Como antes, en caso de que en esta situación no pueda ser alcanzada, se puede preguntar al usuario por la información necesaria para completar alguno de los puntos de la lista anterior.

Detección de vectores de permutación

En los casos en los que se ha encontrado un patrón similar al presentado en 5.13, y/o adicionalmente encontramos una sección de código con la estructura del ejemplo 5.16, podemos aplicar esta técnica si las anteriores no han dado resultados positivos.

Ejemplo 5.16 PATRÓN PARA APLICAR "VECTORES DE PERMUTACIÓN"

```
      DO 10 I= ...
         DO 20 J=PTR(I),PTR(I+1)-1
            VAL(IND(J))=....
20       CONTINUE
10    CONTINUE
```

Si el bucle `I` de este ejemplo, no puede ser paralelizado por las técnicas anteriores, es posible paralelizar el bucle `J` si demostramos que `IND` es un vector de permutación en el rango `PTR(I):PTR(I+1)-1`. Contemplamos dos situaciones:

- Si el vector `IND` sólo es leído dentro del bucle `I`, y sólo es escrito una vez durante la fase de inicialización del programa, es suficiente con un test en tiempo de ejecución que compruebe que la condición es cierta después de la mencionada inicialización.
- Si el vector `IND` se modifica en otros puntos del programa se puede realizar un estudio simbólico que determine si los valores escritos en `IND` son distintos en cada rango `PTR(I):PTR(I+1)-1`.

De nuevo, si estos tests fallan, se puede pedir información al usuario.

Extensión de las técnicas de privatización

Como deciamos en la subsección 5.3.2, la privatización de variables es definitiva a la hora de romper dependencias de salida o anti-dependencias generadas por bucle. Distinguimos dos situaciones diferentes: la privatización de variables o arrays de trabajo por un lado, y por otro la privatización de arrays accedidos por variables de inducción, ambos discutidos en 5.3.2. Cada uno de ellos presenta un problema particular no contemplado hasta ahora por los compiladores paralelos.

En cuanto a la privatización de un array de trabajo `W`, el problema se presenta cuando pretendemos probar que los coeficientes escritos de `W` coindicen con los coeficientes leídos en cada iteración. El uso del array `W` puede estar caracterizado por un código como el presentado en el ejemplo 5.7 en cuyo caso un análisis simbólico acompañado de un sencillo test en tiempo de ejecución soluciona el problema (ver sección 5.3.2). Sin embargo, existen situaciones más complicadas como las que presentan los algoritmos de suma y multiplicación de matrices dispersas en los ejemplos 5.10 y 5.11, respectivamente. En este caso, el test que garantiza que el array de trabajo es privatizable puede consistir en la ejecución simbólica del bucle que se quiere paralelizar, eliminando las operaciones aritméticas y chequeando si se cumplen las condiciones para privatizar `W` en la subsiguiente ejecución del bucle original.

Esta aproximación basada totalmente en test en tiempo de ejecución puede llegar a ser demasiado costosa. En esta línea, Rauchwerger [123] propone una ejecución especulativa en paralelo del bucle. Simultáneamente, se chequea la condición de privatización y en caso de que ésta resulte falsa, se desecha el trabajo realizado y el bucle se vuelve a ejecutar en secuencial. Obviamente, es necesario mantener una copia del estado del proceso antes de comenzar el bucle. En la aproximación que proponemos, la alternativa elegida, introduce un menor coste añadido en el tiempo final de ejecución, pero complica el compilador. El entorno básico a partir del cual construir la técnica completa, está especificado en la memoria de tesis de Peng Tu [151] dedicada a la privatización y al análisis del flujo de datos. Las técnicas propuestas en este trabajo, ya implemetadas en Polaris, cubren la privatización de arrays accedidos en un rango continuo de posiciones. Como hemos visto en los ejemplos, en nuestro caso, cuando el array de trabajo se accede mediante indirecciones, las técnicas de Tu no son suficientes.

Por tanto, el esquema final va a consistir en una complementación de las ideas de Tu con los trabajos de Blume, orientados a la propagación de rangos [33], y algunas de las técnicas en tiempo de ejecución discutidas en [123].

Por otro lado, la privatización de arrays accedidos por variables de inducción que no pueden ser expresadas en su forma cerrada es más directa, siempre que se haya probado que no existe otro tipo de dependencias que las que provoca la variable de inducción. Sin embargo, ahora la complicación aparece al ser necesaria la generación automática de la etapa de copy-out. Este aspecto no esta aún implementado por Polaris pero no presenta ninguna complejidad adicional. De hecho, la etapa de copy-out puede ser interpretada como un caso particular de operación de reducción, ya evaluada en [119].