Parte II

Compilación

Capítulo 4

Soporte HPF-2 para códigos dispersos

Los esfuerzos de investigación y desarrollo en paradigmas de paralelismo de datos que simplifiquen la tarea de construir eficientes algoritmos paralelos, se han incrementado en las últimas décadas. De entre los distintos lenguajes de paralelismo de datos disponibles se erige como estándar el HPF-2 (*High-Performance Fortran 2*) [40, 85, 86, 104].

En el modelo de paralelismo de datos, el usuario escribe el código usando un espacio de iteraciones global y ciertas directivas para indicar la distribución de los datos. Los compiladores de paralelismo de datos utilizan esta información, junto con la regla "computa el propietario"[1], para traducir las direcciones globales a locales e insertar las etapas de comunicación necesarias que permitan la ejecución del código en un multiprocesador acorde a un modelo SPMD.

Por ahora no es posible describir un código de factorización dispersa LU bajo un paradigma de paralelismo de datos con la tecnología actual de sus compiladores. Principalmente, son las computaciones implicadas en el llenado de la matriz y las permutaciones de filas y columnas, las que dificultan la representación de este código bajo el paradigma en cuestión.

En este capítulo abordamos el problema de la paralelización de algoritmos que presentan llenado y pivoteo, desde el punto de vista de los lenguajes de paralelismo de datos y de la compilación semi-automática. Como se ha comentado, es necesaria una estructura de datos que implica la necesidad de manejar indirecciones, referencias a datos a través de punteros y reserva dinámica de memoria. Estas características escapan a la actual tecnología de compilación de paralelismo de datos. Nuestra solución consiste en proponer un pequeño conjunto de nuevas extensiones a HPF-2 que permitan el manejo de este tipo de códigos. El compilador se apoyará parcialmente en una librería (DDLY) de rutinas necesarias para contemplar las nuevas directivas. Esta aproximación se evalúa posteriormente en el Cray T3E para el problema particular de la factorización

[1]Es decir, las asignaciones a la parte izquierda de una sentencia las realiza el procesador propietario de la variable correspondiente.

dispersa LU.

La organización del presente capítulo queda como sigue. En la introducción presentamos los lenguajes de paralelismo de datos y la motivación del trabajo presentado en este capítulo. En la segunda sección comprobamos la ineficiencia de los compiladores actuales de paralelismo de datos al manejar problemas irregulares. En esta línea de razonamiento, los dos siguientes capítulos presentan soluciones basadas en la extensión de las directivas HPF-2, para dos familias de códigos dispersos. En la sección 4.5 se aplican estas soluciones al algoritmo de factorización dispersa LU. Seguidamente se describe el compilador capaz de procesar las nuevas directivas introducidas. Las prestaciones de los algoritmos que hemos implementado siguiendo esta aproximación se presentan en la sección 4.7. Finalmente se enumeran otros problemas que pueden ser abordados con este paradigma de paralelismo de datos si contemplamos las extensiones propuestas.

4.1 Introducción

Aparte del mencionado HPF-2, existen otros muchos lenguajes de paralelismo de datos populares como son el CM-Fortran [142], Fortran D [67], Vienna-Fortran [160], y Craft [111].

Todos estos lenguajes se han orientado inicialmente hacia problemas regulares, esto es, códigos bien estructurados que pueden ser paralelizados eficientemente en tiempo de compilación, usando sencillas distribuciones de datos y computaciones. Sin embargo, la situación es radicalmente diferente en códigos irregulares, donde los patrones de acceso a datos y/o la cantidad de trabajo se conoce únicamente en tiempo de ejecución. En la literatura podemos reconocer, principalmente, dos aproximaciones que permiten manejar este tipo de códigos irregulares. La primera desarrolla técnicas en tiempo de compilación, extendiendo el lenguaje paralelo con nuevas construcciones capaces de expresar paralelismo no estructurado. Con esta información el compilador puede realizar en tiempo de compilación algunas optimizaciones, generalmente delegando el resto de ellas en librerías en tiempo de ejecución. La segunda aproximación contempla las técnicas en tiempo de ejecución, donde el paralelismo no estructurado es explotado y manejado completamente en tiempo de ejecución.

Respecto a la primera estrategia, los lenguajes Fortran D y Vienna-Fortran, por ejemplo, incluyen cierto soporte para distribuciones de datos irregulares. En Fortran D, el programador puede especificar una asignación de elementos de un array a procesadores, usando para ello otro array. Vienna-Fortran, por otro lado, permite al programador definir funciones para especificar la distribución irregular. Sin embargo, HPF no soporta directamente ninguna de estas construcciones. Este hecho ha sido reconocido por algunos investigadores, que han propuesto ciertas extensiones en HPF para intentar corregir este defecto [41, 107], así como por el propio Foro HPF que tomó la decisión de desarrollar la versión 2.0 de HPF [86]. Esta segunda versión de HPF ha sido mejorada y proporciona ahora una distribución por bloques generalizada (GEN-BLOCK), donde las particiones contiguas de arrays pueden ser de tamaños diferentes, y una distribución indirecta (INDIRECT), donde se define un array (llamado en inglés *mapping array*) para especificar cualquier asignación arbitraria de coeficientes del array a procesadores.

Sin embargo, otros investigadores prefieren la segunda aproximación y proponen técnicas en tiempo de ejecución que automáticamente manejan la distribución de datos definida por el usuario, reparten las iteraciones de los bucles y generan las comunicaciones oportunas. La mayoría de estas soluciones se basan en el paradigma *Inspector-Ejecutor* [115, 114, 36].

En cualquier caso, las construcciones incluidas actualmente en estos lenguajes y las librerías en tiempo de ejecución que las soportan están pobremente desarrolladas, resultando en bajas eficiencias cuando se aplican a un espectro amplio de códigos irregulares, como los que aparecen en la mayoría de las aplicaciones científicas y de ingeniería. Para contribuir a la solución de este problema fueron desarrollados y validados extensivamente un cierto número de esquemas de distribución de datos pseudo-regulares (BRS y MRD), diseñados como extensiones naturales de las distribuciones regulares [17, 23, 127, 147, 152]. La clave de estos esquemas de distribución es su simplicidad para ser incorporados en un lenguaje de paralelismo de datos y ser usados por el programador. Adicionalmente resultan de gran efectividad y obtienen altas eficiencias en la paralelización de códigos irregulares.

Sin embargo, las estructuras de datos contempladas bajo los esquemas de distribución mencionados resultan poco efectivas a la hora de manejar operaciones de llenado y pivoteo. Por lo tanto, presentamos en este capítulo las aportaciones necesarias para permitir la gestión del llenado y pivoteo en matrices dispersa en un entorno de paralelismo de datos. Como ejemplo, la factorización LU dispersa, que presenta este tipo de operaciones, se podrá describir bajo este paradigma. Claramente, tendremos que utilizar el lenguaje Fortran, en lugar de C, para describir los códigos correspondientes.

4.2 Una implementación inicial con Craft

Craft [111] es el nombre del compilador de paralelismo de datos disponible en las plataformas masivamente paralelas de Cray Research. Básicamente, consiste en el lenguaje Fortran 77 más algunas extensiones del Fortran 90, como la sintaxis de arrays y funciones intrínsecas para realizar operaciones globales con datos compartidos. También incluye un conjunto de directivas (que deben comenzar con la palabra clave `CDIR$`) para distribuir los datos. Aparte del modelo de paralelismo de datos, Craft también soporta otros paradigmas (pase de mensajes, datos compartidos y trabajo compartido) que se pueden combinar en un mismo programa. Más recientemente, Cray ha desarrollado un híbrido entre Craft y HPF que llaman HPF_Craft [46].

Como primera aproximación al estilo de programación de paralelismo de datos, intentamos implementar un algoritmo LU disperso simplificado (sin pivoteo de columnas ni reserva dinámica de memoria por estar escrito en Fortran77) mediante Craft para el Cray T3D. Básicamente, el algoritmo secuencial sigue una política *right-looking* y utiliza una estructrua de datos estática como la CCS (ver sección 1.2.2). Para gestionar el llenado es inevitable un movimiento de columnas en cada iteración que permita la inclusión de las nuevas entradas generadas durante la actualización. La estrategia descrita en [57, Cap. 2.4] se aplica para actualizar cada columna (`actcol()`). Aunque se ha mencionado en la sección 3.3.4, recordemos que consta de tres etapas: descompresión del vector empaquetado en uno denso (operación *scatter*); actualización del vector denso; y empaquetado de los elementos no nulos de éste último (operación *gather*). Las operaciones de *scatter* y *gahter* se muestran en el ejemplo 4.1 (a) y (b) respectivamente. Este algoritmo se detalla en el capítulo 5 y en [145].

Craft sólo proporciona la distribución clásica `BLOCK(x)` para matrices densas, donde `x` indica el tamaño del bloque contiguo del array que se asigna a cada procesador, normalmente conocida como distribución cíclica por bloques. Si utilizamos este esquema para distribuir los tres vectores, `Val`, `Fil` y `Colpt` que representan una matriz dispersa, `A`, con formato CCS, el primer resultados es una alta perdida de localidad. En efecto, la distribución cíclica por bloques no atiende a las coordenadas de las entradas y asigna trozos de columnas de tamaño variable a procesadores. En cuanto a la distribución del trabajo, encontramos un problema similar. Por ejemplo, en el código del ejemplo 4.1 (a), la directiva `DO SHARED` distribuye el trabajo del bucle, asignando la iteración `i` al procesador que tenga asignado la componente `Val(i)`. Esta sección de código transforma una columna comprimida de la matriz `A` en un vector denso, `Dens`. La sentencia de asignación con una indirección en la parte izquierda dará lugar a una pérdida de localidad inevitable, ya que aunque garanticemos que `Val(i)` esté alineado con `Fil(i)`, no podremos hacer lo mismo con `Dens(Fil(i))`.

Ejemplo 4.1 Extractos del código LU en Craft

```
CDIR$ DO SHARED(i) on Val(i)
DO i = Colpt(k), Colpt(k+1)-1
   Dens(Fil(i)) = Val(i)
ENDDO
```

(a)

```
induc = pos
DO i = 1, n
   IF (Dens(i) .NE. 0.0) THEN
      Val(induc) = Dens(i)
      Fil(induc) = i
      induc = induc + 1
   ENDIF
ENDDO
```

(b)

Otro problema de similar importancia consiste en la imposibilidad de paralelizar en Craft ciertos bucles que son paralelizables aplicando la distribución apropiada, por ejemplo BCS. El código del ejemplo 4.1 (b) realiza la operación opuesta a la comentada anteriormente: empaqueta los elementos no nulos de un vector denso en un vector comprimido. El código es paralelizable si los vectores `Val` y `Fil` se distribuyen cíclicamente en función de las coordenadas almacenadas en `Fil`, como ocurre en la distribución BCS. De esta forma un vector denso, `Dens` distribuido cíclicamente quedaría automáticamente alineado. Sin embargo Craft reporta resultados erróneos al intentar ejecutar ese bucle en paralelo con su distribución regular, principalmente debido a la variable de inducción `induc` que se incrementa dentro de una cláusula condicional. Por otro lado, como se indicó en el capítulo 3, no sólo la operación `actcol()` es paralela, sino que es posible actualizar todas las columnas de la submatriz reducida en paralelo. Con la distribución BCS eso es totalmente factible, sin embargo con Craft no podemos manejar adecuadamente la variable de inducción. Estos resultado son a grandes rasgos extrapolables al resto de las herramientas de compilación semi-automática disponibles, como Fortran-D, Vienna-Fortran, etc.

Después de este intento, infructuoso pero definitivo, queda patente la imperiosa necesidad de enriquecer la potencia de los compiladores de paralelismo de datos de forma que permitan siquiera paralelizar los bucles que son inherentemente paralelos. De cara a conseguir una buena eficiencia bajo este paradigma, la localidad y la minimización de las comunicaciones son los factores que perseguiremos con más énfasis. Con todos esos propósitos encaramos las siguentes secciones.

4.3 Cuando el problema es estático

Los problemas mencionados en la sección anterior han sido presentados bajo el prisma de la factorización LU dispersa y en general de los problemas con llenado y pivoteo. En este tipo de problemas las entradas se mueven (pivoteo) y puede aparecer llenado por lo que puede ser considerado un problema dinámico. Una simplificación que permite una primera solución resulta de obviar las características dinámicas y abordar problemas donde no cambien las coordenadas de los coeficientes ni aparezcan o desaparezcan éstos durante el procesado. En estos casos, por ejemplo, en el producto matriz dispersa por vector, el problema es estático, aunque su naturaleza siga siendo irregular. A pesar de su mayor simplicidad, este conjunto de problemas tampoco era gestionado eficientemente por los compiladores de paralelismo de datos hasta muy recientemente.

Los esquemas de distribución CRS-MRD/CCS-MRD y BRS/BRD (ver sección 3.2) se adaptan perfectamente a muchos de estos problemas estáticos, pero no estaban cubiertos por las herramientas semi-automáticas. Ujaldón y col. (1997) [152] resuelven este punto al incorporar la directiva `SPARSE` en los compiladores Vienna Fortran y HPF. Aunque esos esquemas de distribución podrían ser especificados en HPF-2 mediante la directiva `INDIRECT`, el uso de los arrays de distribución (*maping arrays*) necesarios, da lugar a varios inconvenientes, entre ellos: el alto consumo de memoria; la necesidad de realizar todo el análisis de optimizaciones en tiempo de ejecución; y principalmente, el incremento de las comunicaciones que tiene lugar cuando los arrays de distribución están repartidos entre los procesadores. Sin embargo, el manejo de los esquemas de distribución pseudo-regulares son mucho más eficientes, ya que una simple operación de módulo o de comparación es suficiente para localizar cualquier elemento.

Con esta nueva directiva `SPARSE`, el esquema de distribución BCS puede especificarse como se indica en el ejemplo 4.2.

Ejemplo 4.2 Especificación HPF del esquema BCS

```
      REAL:: Val(alfa)
      INTEGER:: Fil(alfa), Colpt(n+1)
!HPF$ PROCESSORS, DIMENSION(P,Q):: malla
!HPF$ REAL, DYNAMIC, SPARSE(CCS(Val,Fil,Colpt)):: A(m,n)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO malla:: A
```

En este ejemplo, la directiva `SPARSE` informa al compilador que la matriz dispersa

`A`, de dimensión $m \times n$ y `alfa` entradas, esta representada en formato CCS mediante tre vectores, `Val`, `Fil` y `Colpt`. La palabra clave DYNAMIC, si acompaña a la directiva `SPARSE`, indica que la distribución se determina en tiempo de ejecución. De lo contrario, el compilador entiende que tanto la distribución como la representación local a cada procesador se puede construir en tiempo de compiliación. En la práctica, esto significa que el patrón de entradas está disponible antes de compilar el código, lo cual aunque raramente ocurre, debe contemplarse.

La directiva `DISTRIBUTE` informa de la distribución elegida para la matriz dispersa. Aplicando (CYCLYC,CYCLIC), consegimos un esquema de distribución BCS, ya que tenemos una distribución cíclica dispersa junto con una reprepresantación CCS de la matriz. De esta forma, obtenemos los beneficios de la distribución cíclica (balanceo de la carga y direccionamiento simple de los datos, lo que redunda en patrones de comunicación sencillos) aplicados a una matriz dispersa, independientemente de la estructura de datos comprimida que la representa. Otras alternativas, como (CYCLYC,:) o (:,CYCLYC) permiten distribuir las filas replicando las columnas y viceversa respectivamente. Aunque no es aplicable a nuestro problema de factorización, también está disponible la distribución (MRD,MRD) descrita en 3.2.

En el ejemplo 4.3 se muestra el resultado de aplicar esta directiva para una matriz dispersa con $n = 10$, $n = 8$, $P = Q = 2$ y $alfa = 14$. En este ejemplo la matriz dispersa (descomprimida), presentada en (a), se divide en un un conjunto de submatrices de tamaño 2×2, que son posteriormente proyectadas sobre la malla de procesadores. Por ejemplo, las entradas subrayadas de la matriz en (b), se asignan al procesador de coordenadas (0,0) de la malla. Finalmente, las submatrices locales se almacenan siguiendo el formato CRS, como se muestra en (c) para el procesador (0,0).

Ejemplo 4.3 Esquema BRS sobre una malla de 2×2 procesadores

$$
\begin{pmatrix}
0 & 53 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\
0 & 0 & 0 & 17 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 13 & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
0 & 23 & 69 & 0 & 37 & 0 & 0 & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{pmatrix}
$$

(a)

$$
\begin{pmatrix}
\underline{0} & 53 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\
\\
\underline{19} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\
\\
\underline{0} & 0 & \underline{0} & 17 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\
\\
\underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{13} & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
\\
\underline{0} & 23 & \underline{69} & 0 & \underline{37} & 0 & \underline{0} & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{pmatrix}
$$

(b)

DATA	COL	ROW
19	1	1
13	4	1
69	2	2
37	3	2
		3
		5

(c)

4.4 Y cuando el problema es dinámico

Hemos visto como la directiva SPARSE junto con las palabras clave CRS o CCS permiten contemplar los esquemas de distribución BRS y BCS respectivamente. Aplicado a problemas irregulares como el producto matriz dispersa por vector, esta aproximación reporta resultados hasta ahora no conseguidos por otras estrategias. Sin embargo, ya hemos comentado en la sección 1.2.2 los inconvenientes de las estructuras comprimidas por filas o columnas a la hora de manejar problemas con llenado o pivoteo. Por tanto, cuando en el problema encontramos estas operaciones dinámicas tiene sentido utilizar estructuras de datos también dinámicas, no contempladas tampoco por los compiladores de paralelismo de datos. Definir las estructuras de datos que necesitamos para una eficiente implementación de nuestro problema, y las modificiaciones necesarias para que sean contempladas por el compilador, conforman los objetivos de esta sección.

De entre los distintos compiladores de paralelismo de datos existentes, nos hemos centrado en HPF-2 para proponer sobre él las extensiones que necesitamos. Esta decisión se fundamenta en las siguientes razones:

- HPF-2 es un estándar diseñado por el *High Performance Fortran Forum* (HPFF). HPFF esta formado por grupos académicos y de la industria que trabajan en sugerir extensiones estándar en Fortran90. La intención es proporcionar un soporte para la programación paralela de altas prestaciones en un amplio espectro de plataformas, incluyendo sistemas masivamente paralelos SIMD y MIMD, asi como procesadores vectoriales. La última versión es el HPF 2.0 [86] que proporciona extensiones al actual estándar de Fortran, el Fortran95.

- La tendencia en paralelismo de datos es usar Fortran debido a su extensiva utilización en códigos científicos y por ser menos flexible que el C, lo cual simplifica el compilador. Esta excesiva simplicidad del Fortran77 nos impedía diseñar un código que realice eficientemente las operaciones de pivoteo y llenado. Sin embargo, el hecho de que HPF-2 se base en Fortran90, nos da flexibilidad para implementar estas operaciones, ya que este lenguaje permite el uso de estructuras dinámicas de memoria, punteros, reserva de memoria, etc, que son necesarios para soportar eficientemente el pivoteo y el llenado.

Pues bién, con todo lo dicho, nuestra intención es entonces extender HPF-2 de forma que contemple estructuras de datos apropiadas para manejar operaciones de llenado y pivoteo. En particular, contemplaremos las ya descritas en la sección 1.2.2: LLCS (*Linked List Column Storage*), LLRS (*Linked List Row Storage*), LLRCS (*Linked List Row-Column Storage*) y CVS (*Compressed Vector Storage*). Las tres primeras estructuras se utilizarán para representar matrices dispersas mientras que la última para almacenar vectores dispersos (arrays dispersos unidimensionales). El almacenamiento LLCS corresponde al presentado en la figura 1.4 de la sección 1.2.2. Es decir, la matriz se representa a través de un conjunto de listas enlazadas, una por cada columna. Aunque en la figura la lista es doblemente enlazada, también puede ser definida como simplemente enlazada. El esquema LLRS es análogo al LLCS pero enlazando por filas en lugar de columnas. Si enlazamos tanto por filas como por columnas tenemos la estructura LLRCS, cuya variante doblemente enlazada se representa en la figura 1.3. Finalmente la estructura CVS permite representar un vector disperso mediante dos arrays y un escalar: el array `Val` que contiene el valor de las entradas; el array `Ind` que almacena los índices asociados; y el escalar `Tam` que indica el número de entradas. La declaración de los tipos de datos en Fortran90 se encuentra en el ejemplo 1.5 de la sección 1.2.2.

Una vez han sido descritos los esquemas de almacenamiento, podemos combinarlos con la directiva `SPARSE`, descrita en la sección anterior. Pretendemos, de esta forma, informar al compilador que una matriz dispersa (o un vector disperso), por ejemplo `A`, se representa mediante una de estas estructuras de datos particulares. Esto es, `A` representa una plantilla (en inglés *place holder*) de la matriz dispersa, y la directiva `SPARSE` establece una conexión entre la entidad lógica `A` y su representación interna (la lista enlazada, por ejemplo). Las ventajas de esta aproximación son que podemos usar las directivas estándar de HPF-2, como `DISTRIBUTE` y `ALIGN` aplicadas a la matriz `A`, y al mismo tiempo almacenar ésta usando una estructura comprimida.

La directiva `SPARSE` descrita en la sección anterior puede ser extendida fácilmente para que incorpore las estructuras de datos mencionadas. En el ejemplo 4.4 se muestra la sintaxis simplificada BNF (Backus-Naur Form) [2] de nuestras propuestas para esta directiva, donde sólo vamos a contemplar las nuevas estructuras de datos.

Ejemplo 4.4 SINTAXIS BNF DE LA DIRECTIVA SPARSE

```
<sparse-directive>::= <datatype>, SPARSE (<sparse-content>) :: <array-objects>
<datatype>::= REAL | INTEGER
<sparse-content>::= LLRS (<ll-spec>)
                  | LLCS (<ll-spec>)
                  | LLRCS (<ll2-spec>)
                  | CVS (<cvs-spec>)
<ll-spec>::= <pointer-array-name>, <pointer-array-name>,
             <size-array-name>,
             <link-spec>
<ll2-spec>::= <pointer-array-name>, <pointer-array-name>,
              <pointer-array-name>, <pointer-array-name>,
              <size-array-name>, <size-array-name>,
              <link-spec>
<cvs-spec>::= <index-array-name>, <value-array-name>, <size-scalar-name>
<link-spec>::= SINGLY | DOUBLY
<array-objects>::= <sized-array>{,<sized-array>}
<sized-array>::= <array-name>(<subscript>[,<subscript>])
```

Las dos primeras estructuras de datos, LLRS y LLCS se definen mediante dos arrays de punteros (<*pointer-array-name*>), que apuntan al comienzo y al final, respectivamente, de cada fila (o columna), y un tercer array (<*size-array-name*>), que contiene el número de elementos por fila (para LLRS) o por columna (para LLCS). La opción <*link-spec*> especifica el tipo de enlace de la estructura enlazada: simple o doble. En cuanto a la estructura LLRCS, tenemos cuatro arrays de punteros que apuntan al comienzo y al final de cada fila y columna, y dos vectores adicionales para almacenar el número de entradas por fila y columna, respectivamente.

La siguiente sentencia es un ejemplo de uso de esta directiva:

```
!HPF$ REAL, DYNAMIC, SPARSE (CVS(Ind, Val, Tam)):: V(10)
```

Donde se declara una plantilla lógica, `V`, para un vector disperso, la cual no ocupa memoria. Lo que físicamente sí ocupa memoria son los coeficientes no nulos del array (`Val`), los índices correspondientes (`Ind`), y un entero para el número de entradas (`Tam`). La función de la plantilla es proporcionar un objeto abstracto que puede ser distribuido y con el que otros objetos pueden ser alineados. El atributo DYNAMIC lo usaremos siempre con la directiva `SPARSE` ya que como se indicó anteriormente la distribución de estas estructuras debe realizarse en tiempo de ejecución cuando se conocen los índices y el tamaño de los vectores. Además, en este contexto, el atributo DYNAMIC es indicativo de que el tamaño del array puede cambiar durante la computación.

Las directivas `DISTRIBUTE` y `ALIGN` de HPF-2, pueden ser aplicadas a las plantillas dispersas con la misma sintaxis que en el estándar. Como se comentó en la sección 4.3, al distribuir la plantilla dispersa ésta es considerada como densa. En el caso de la directiva `ALIGN`, sin embargo, la semántica puede ser ligeramente distinta. Consideremos el segmento de código del ejemplo 4.5.

Ejemplo 4.5 Especificación de un vector comprimido replicado

```
REAL, DIMENSION(10,10):: A
INTEGER, DIMENSION(10):: Ind
REAL, DIMENSION(10):: Val
INTEGER:: Tam
!HPF$ PROCESSORS, DIMENSION(2,2):: malla
!HPF$ REAL, DYNAMIC, SPARSE (CVS(Ind, Val, Tam)):: V(10)
!HPF$ ALIGN V(:) WITH A(*,:)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO malla:: A
```

El efecto de la directiva `ALIGN` en este fragmento de código es el siguiente: las entradas del vector `V` (es decir, `Val`), se alinean con las columnas de `A` dependiendo de los índices almacenados en el array `Ind`, y no de las posiciones del propio vector `Val` (como ocurriría bajo la semántica estándar para arrays densos). Ahora, la directiva `DISTRIBUTE`, replica el vector `V` en la primera dimensión del array de procesadores, `malla`, y lo distribuye en la segunda dimensión de la misma forma que fué distribuida la segunda dimensión de la matriz `A`. Así, cada fila completa de procesadores de la malla almacena una copia completa del vector `V`. Es importante subrayar que en esta operación de distribución, el vector `Ind` se utiliza como un array de índices de las entradas almacenadas en `Val`. Como podemos ver, para replicar un objeto (`V`) debe ser alineado con otro objeto que tenga una mayor dimensionalidad (matriz `A`) o una plantilla que sirva para este propósito. Aclaramos el proceso atendiendo a la figura 4.1, la cual muestra el efecto combinado de alineamiento/distribución para un caso particular. En la figura, el prefijo `loc` identifica a los arrays locales en cada uno de los cuatro procesadores de la malla.

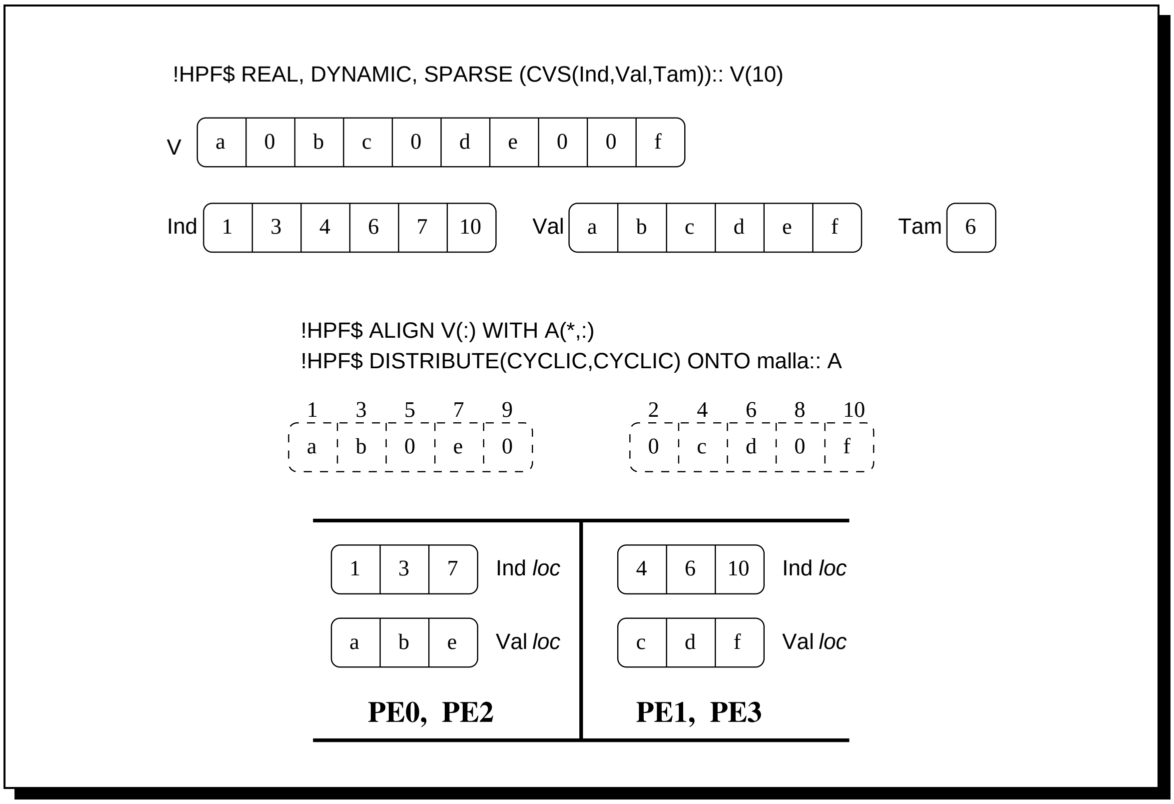

Figura 4.1: Alineamiento y distribución de una array disperso en una malla de 2×2.

4.5 Códigos en HPF-2 extendido

Mediante el uso de la directiva `SPARSE` hemos conseguido establecer un enlace entre un objeto disperso (matriz o vector) y su estructura de almacenamiento. En este punto, podríamos optar por ocultar la estructura de datos al programador, permitiéndole escribir código paralelo disperso usando notación de matrices densas. En esta propuesta inicial, discutida en [145], el compilador se hace responsable de traducir la notación densa a un código disperso y paralelo, teniendo en cuenta el esquema de almacenamiento utilizado. Sin embargo, esta aproximación supone un gran esfuerzo en la implementación del compilador, y para ser sinceros, no tenemos la medida objetiva de cómo de factible resulta tal implementación. Por otro lado, sería necesario combinar en el mismo código plantillas o *place holders* (es decir, entidades *lógicas*) con arrays *reales*, con una primera consecuencia: obtendremos resultados erróneos cuando compilemos el mismo programa para su ejecución secuencial. Esto contradice radicalmente la filosofía básica del paradigma del paralelismo de datos: la inclusión de directivas en el código sólo puede afectar al tiempo de ejecución de éste y nunca a sus resultados. Bik [29] y Pingali [97] proponen una aproximación similar, basada en la transformación de programas secuenciales densos, anotados con directivas dispersas, en los códigos secuenciales dispersos equivalentes. Sin embargo, incluso para códigos secuenciales, la implementación de tal compilador resulta tan compleja que no existe una versión disponible que maneje adecuadamente problemas reales.

Por tanto, siendo realistas, si los códigos dispersos son complejos, los programadores deben asumir parte de la complejidad al programar, dando pistas al compilador para que pueda realizar un buen trabajo. Mirando a través de ese prisma, la aproximación consiste en forzar al programador a usar explícitamente la estructura de datos que representa la información, y permitirle usar las plantillas únicamente con propósitos de alineamiento y distribución. Esta es la solución empleada en la versión de paralelismo de datos de la LU dispersa que presentamos a continuación. La misma aproximación es aplicada para la factorización QR dispersa en [15].

4.5.1 Código LU disperso

Presentamos los detalles de implementanción de las ideas expuestas apoyándonos en el algoritmo de factorización LU disperso. Más precisamente nos centraremos en la etapa de factorización con pivoteo parcial numérico, por lo que ejecutaremos la etapa de análisis en un paso anterior, la cual consume un tiempo despreciable.

Esta organización con pivoteo parcial elimina la necesidad de acceder tanto por filas como por columnas. De este modo, si optamos por un almacenamiento LLCS, donde enlazamos los elementos de una misma columna en una lista unidimensional, la permutación de columnas se puede llevar a cabo eficientemente. Con esta estructura de datos, como se discute en 1.2.2, ahorramos memoria, maneteniendo al nivel deseado la flexibilidad en el acceso y la versatilidad en la generación y movimiento de entradas.

Una vez elegida la estructura de datos, y para completar el esquema de distribución, paralelizaremos el código LU mediante una simple distribución cíclica de columnas sobre un array lineal de procesadores. En el ejemplo 4.6 se muestra la sección declarativa del código LU paralelo aplicando las extensiones propuestas en HPF-2.

Ejemplo 4.6 SECCIÓN DE DECLARACIÓN

```
INTEGER, PARAMETER :: n=1000
TYPE (ptr), DIMENSION(n):: first(:), last(:), vpiv(:)
INTEGER, DIMENSION(n):: vsize(:)
REAL, DIMENSION(n):: vcolv, vmaxval
INTEGER, DIMENSION(n):: vcoli, iq
INTEGER :: sizec,
REAL, PARAMETER:: u=0.1  !*** Parámetro de umbral

!HPF$ PROCESSORS, DIMENSION(P):: linear
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first,last,vsize,DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli,vcolv,size)):: VCOL(n)
!HPF$ ALIGN iq(:) WITH A(*,:)
!HPF$ ALIGN vpiv(:) WITH A(*,:)
!HPF$ ALIGN vmaxval(:) WITH A(*,:)
!HPF$ ALIGN VCOL(:) WITH A(:,*)
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
```

La estructura representada por la plantilla `A` contiene inicialmente la matriz de entrada, que será leída de un fichero, y al final almacena los factores `L` y `U` (el algoritmo es *in place*). La plantilla es físicamente accesible a través de los arrays de punteros `first` y `last`, que apunta a la primera y última entrada, respectivamente, de cada columna de `A`. También hemos definido un vector disperso `VCOL`, representado mediante una estructura de vector enpaquetado en formato CVS. Este array almacenará la columna del pivot después de normalizarla en la operación `divcol()` a cada iteración del bucle más externo.

La última sentencia de la sección de declaración distribuye cíclicamente las columnas de la matriz A sobre un array abstracto de `P` procesadores, declarado previamente con el nombre `linear` con la directiva `PROCESSORS`. Unas líneas antes, tres vectores densos, `iq`, `vpiv` y `vmaxval`, son alineados con las columnas de `A`. De esta forma, después de la distribución de `A`, estos tres vectores también quedan distribuidos cíclicamente sobre el array de procesadores. El vector `iq` representa el vector de permutaciones de columnas recomendado, por la etapa de análisis, para mantener el coeficiente de dispersión; `vpiv` es un vector de punteros a la fila del pivot y `vmaxval` se utiliza para calcular el máximo valor absoluto en la búsqueda de un pivot numéricamente estable.

Por último, el array disperso `VCOL` es alineado con las filas de `A`. Así pues, después de la distribución de `A`, `VCOL` queda replicado en todos los procesadores. La razón que justifica esta replicación es hacer visible la columna del pivot normalizada a todos los procesadores para completar en paralelo la actualización de la submatriz reducida. En efecto, como veremos más adelante, en cada iteración del bucle más externo, `k`, el procesador (`k`-1) mod P, propietario de la columna `k`, realiza la operación `divcol(k)` localmente. Por tanto, los coeficientes de la columna actualizados deben radiarse al resto de los procesadores, lo cual se realiza implícitamente si copiamos esa columna en `VCOL`, que está replicada en todos los procesadores. Queremos también subrayar que el uso, en esta sección de declaración, de un array de procesadores no es una limitación esencial. De hecho se puede extender de forma similar a una malla de procesadores, pero nuestra elección unidimensional se debe principalmente a los mejores resultados experimentales obtenidos en esta topología.

En la figura 4.2 aclaramos gráficamente la declaración del esquema de distribución propuesto para nuestro algoritmo. Podemos ver la partición final de la mayoría de los arrays implicados junto con la matriz `A`, para el caso de un array de $P = 2$ procesadores. También se supone en la figura que el valor actual de la variable índice del bucle externo `k` es cuatro. La leyenda en la esquina inferior izquierda contiene la correspondencia entre la notación empleada en la figura para nombrar los objetos y la notación usada en la sección de declaración del ejemplo 4.6.

El resto del código encargado de la factorización dispersa se muestra a continuación. Primero nos centraremos en la etapa de pivoteo, cuyo código aparece en el ejemplo 4.7. Las primeras líneas muestran la inicialización del vector `vpiv` para que apunte a la fila del pivot. El bucle que realiza esa inicialización es totalmente paralelo, lo cual se le indica al compilador mediante la directiva `INDEPENDENT`, que mantiene la funcionalidad estándar de HPF: las iteraciones se pueden ejecutar en cualquier orden (o concurrentemente) sin cambiar la semántica del bucle. Esta directiva junto con la regla "computa el propietario" da lugar a que cada procesador inicialice la porción de vector que tiene asignado, y dado que los dos vectores, `vpiv` y `first`, están alineados, dicha inicialización no requiere comunicaciones. En seguida encontramos la sentencia DO asociada al bucle más externo `k`. Considerando que la etapa de análisis ha determinado la iteración, `IterConm`, a partir de la cual compensa conmutar a un código denso, el espacio de iteraciones para el índice `k` es de uno a `IterConm`.

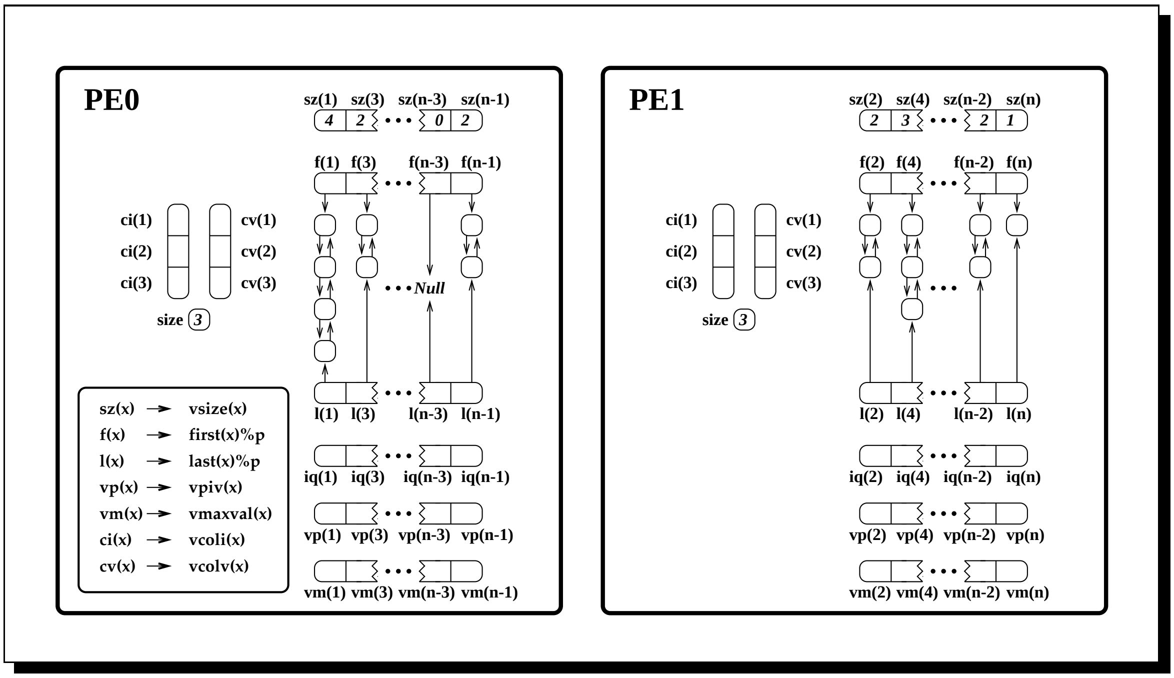

Figura 4.2: Esquema de distribución para la LU en HPF-2.

Ejemplo 4.7 ESQUEMA DEL CÓDIGO HPF-2. 1ª PARTE: PIVOTEO

```
! --> Inicialización
!HPF$ INDEPENDENT
      DO j = 1, n
        vpiv(j)%p => first(j)%p
      END DO

! --> Bucle principal de la LU
main: DO k = 1, IterConm

! --> Pivoteo
! --> Búsqueda del máximo valor absoluto en la fila activa k
!HPF$ INDEPENDENT
        DO j = k, n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          IF (.NOT.ASSOCIATED(vpiv(j)%p)) CYCLE
          IF (vpiv(j)%p%index /= k) CYCLE
          vmaxval(j) = ABS(vpiv(j)%p%value)
!HPF$ END ON
        END DO
        maxpiv = MAXVAL(vmaxval(k:n))
        maxpiv = maxpiv*u             ! Valor umbral

! --> Elección de un pivot numéricamente estable (operación de reducción)
        actpiv = vmaxval(k)
        pivcol = k
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
        DO j = k+1, n
          IF (vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j)) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
        IF(pivcol /= k) THEN
! ----> Intercambio de columnas
          CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF

                              •
                              •
                              •
```

La primera acción que tiene lugar dentro del bucle es la búsqueda de un pivot númericamente estable y, si puede ser, acorde con la recomendación de permutaciones en `iq` (proporcionada por la etapa de análisis) ya que ésta minimiza el llenado. Para asegurar la estabilidad numérica, el pivot debe superar un umbral, resultante de multiplicar el parámetro de entrada `u` por el máximo valor absoluto de la fila del pivot. El cálculo del máximo se realiza gracias a la función intrínseca `MAXVAL` del Fortran90, que tiene como entrada un vector, `vmaxval`, con el valor absoluto de todas las entradas en la fila del pivot activa. La actualización de `vmaxval` tiene lugar en el segundo bucle paralelo marcado con la directiva `INDEPENDENT`. Este bucle recorre la fila del pivot activa almacenando el valor absoluto de las entradas en `vmaxval`. La directiva `ON HOME(vpiv(j))` le indica al compilador que el procesador propietario de `vpiv(j)` será el encargado de la asignación de la iteración `j`, reafirmando la regla "computa el propietario". Además, esta directiva viene acompañada de la claúsula `RESIDENT(a(*,j))` que informa al compilador que toda la columna `j` es local al procesador propietario de `vpiv(j)`, y por tanto, no es necesaria ninguna etapa de comunicación.

Una vez tenemos el valor umbral, seleccionamos el pivot que lo supere y que, por otro lado, respete el coeficiente de dispersión de la matriz. Para ello, tendremos en cuenta que en la etapa de análisis se recomienda que la columna `r` preceda a la columna `s` si $iq(r) < iq(s)$. Seleccionar el pivot que cumpla esas condiciones es equivalente a una reducción, lo cual es indicado al compilador añadiendo a la directiva `INDEPENDENT`, la claúsula `REDUCTION()` con las dos variables que se reducen: `actpiv`, el valor absoluto del pivot; y `pivcol`, el índice de la columna a la que pertenece dicho pivot. Este tipo de reducción definida por el usuario no está considerada actualmente por el estándar HPF-2, pero su inclusión no añade ninguna complejidad significativa a la implementación del compilador. De hecho, existen investigaciones orientadas a detectar reducciones paralelas complejas [140], donde pueden aparecer sentencias `IF` dentro del bucle de reducción. Finalmente, si la columna `pivcol` no es la columna `k`, ambas se permutan, actualizando los vectores correspondientes, mediante la rutina `swap`.

Ejemplo 4.8 ESQUEMA DEL CÓDIGO HPF-2. 2ª PARTE: DIVCOL()

```
                              :
! --> La columna del pivot es actualizada y empaquetada en VCOL
!HPF ON HOME (vpiv(k)), RESIDENT (A(*,k)) BEGIN
        aux => vpiv(k)%p
        pivot = 1/(aux%value)
        aux%value = pivot
        aux => aux%next
        size = vsize(k)-1

        DO i = 1, size
          aux%value = aux%value*pivot
          vcolv(i) = aux%value
          vcoli(i) = aux%index
          aux => aux%next
        END DO
!HPF END ON
                              :
```

En la siguiente sección de código, presentada en el ejemplo 4.8, contempla la normalización de la columna del pivot por el valor de éste (operación `divcol`).

Esta operación, `divcol()`, sólo la realiza el procesador propietario de la columna `k`, lo cual indicamos mediante la directiva `ON HOME`. De nuevo, para simplificar el análisis que debe realizar el compilador y pueda generar un código más optimizado, le adelantamos que la columna `k` es local al procesador en cuestión anotando la directiva `RESIDENT`. Dado que estamos escribiendo en un vector replicado, como es `VCOL`, cualquier modificación realizada en un procesador debe verse reflejada posteriormente en todos los demás. Depende de la complejidad que derrochemos en la implementación del compilador, las comunicaciones implicadas pueden realizarse: (i) instantáneamente a cada modificación de una componente de `VCOL`; (ii) agrupadas al acabar el bucle que actualiza el vector; (iii) o incluso cuando otro procesador lee una componente de `VCOL` que no ha sido adecuadamente refrescada; obteniéndose una mayor o menor eficiencia en la ejecución del bucle en función de la opción seleccionada.

Finalmente, la sección de código donde se actualiza la submatriz activa es mostrado en el ejemplo 4.9.

Ejemplo 4.9 ESQUEMA DEL CÓDIGO HPF-2. 3ª PARTE: ACTUALIZACIÓN

•
•

```
! --> Actualización de la submatriz reducida de A
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
loopj:  DO j = k+1, n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          aux => vpiv(j)%p
          IF (.NOT.ASSOCIATED(aux)) CYCLE
          IF (aux%index /= k) CYCLE
          amul = aux%value
          vsize(j) = vsize(j)-1
          vpiv(j)%p => aux%next
          aux => aux%next
loopi:    DO i = 1, size
            product = -amul*vcolv(i)
            DO
              IF (.NOT.ASSOCIATED(aux)) EXIT
              IF (aux%index >= vcoli(i)) EXIT
              aux => aux%next
            END DO
outer_if:   IF (ASSOCIATED(aux)) THEN
              IF (aux%index == vcoli(i)) THEN
                aux%value = aux%value + product
              ELSE
! ----> Inserción en la primera posición o una intermedia
                CALL insert(aux,vcoli(i),product,first(j)%p,vsize(j))
                IF (vpiv(j)%p%index >= aux%prev%index) vpiv(j)%p => aux%prev
              END IF
            ELSE outer_if
! ----> Inserción un una posición final
              CALL append(vcoli(i),product,first(j)%p,last(j)%p,vsize(j))
              IF (.NOT.ASSOCIATED(vpiv(j)%p)) vpiv(j)%p => last(j)%p
            END IF outer_if
          END DO loopi
!HPF$ END ON
        END DO loopj
      END DO main
```

•
•

El cuerpo de esta actualización de la submatriz definida está delimitado por un bucle paralelo de índice `j` que recorre las columnas de la matriz. El bucle está por tanto marcado con la directiva `INDEPENDENT`, pero además se le añade la cláusula `NEW` para indicar que las variables especificadas entre paréntesis deben ser consideradas privadas a cada iteración. Esto permite eliminar las dependencias generadas por bucle que detectaría el compilador si considera que el valor de estas variables, a la salida de una iteración, se utilizan en la iteración siguiente. Sin embargo, con la cláusula `NEW`, el compilador entiende que el valor de las variables especificadas está indefinido al comienzo del bucle y que en cada iteración se les asignará un valor nuevo e independiente, volviendo a quedar indefinidas al final del bucle. En este bucle, la directiva `ON HOME ..., RESIDENT ...` también se incluye para facilitar al compilador las tareas de análisis y generación optimizada de código.

Este fragmento de código contiene llamadas a las rutinas `append()` e `insert()` incluidas dentro de un módulo de Fortran90. Estas rutinas son necesarias para el manejo de listas y el insertado de nuevas entradas cuando hay llenado. En el ejemplo 4.10 mostramos el código de la rutina `append()`, que añade una entrada al final de una lista.

Por otro lado, la rutina `insert()`, incluye una nueva entrada al principio o en algún punto intermedio de la lista. Este módulo requiere de otro, `data_structure`, no mostrado en el ejemplo, donde se incluyen las declaraciones de los tipos de datos derivados (ya descritos) que soportan la estructura de datos enlazadas.

Ejemplo 4.10 Módulo Fortran90 para el manejo de listas

```
MODULE list_routines
  USE data_structure
  IMPLICIT NONE

CONTAINS

  SUBROUTINE append(ind, product, fptr, lptr, nel)
    INTEGER:: ind, nel
    REAL:: product
    TYPE (entry), POINTER:: fptr, lptr

    TYPE (entry), POINTER:: new
    ALLOCATE(new)
    new%index = ind
    new%value = product
    nel = nel + 1


    IF (.NOT.ASSOCIATED(lptr)) THEN
      NULLIFY(new%prev, new%next)  ! Lista vacía
      fptr => new
      lptr => new
    ELSE
      new%prev => lptr
      NULLIFY(new%next)
      lptr%next => new
      lptr => new
    END IF
  END SUBROUTINE append

       ...

END MODULE list_routines
```

Es importante resaltar que en caso de compilar este código mediante un compilador de Fortran90 (es decir, si las directivas HPF-2 son consideradas simples comentarios), el código se ejecuta correctamente en secuencial.

4.5.2 Conmutación a código denso

Cuando el índice `k` alcanza la iteración `IterConm`, el bucle principal de factorización dispersa concluye y la submatriz reducida restante se procesa mediante un algoritmo denso. Es clara la posibilidad de describir este tipo de código denso y regular en un entorno de paralelismo de datos. Sin embargo, queremos presentar en esta sección la estrategia para conmutar de estructura de datos y la sencillez con que se puede abordar la transición entre un algoritmo disperso y otro denso, coexisitiendo en un mismo código, bajo este paradigma extendido con las aportaciones presentadas.

La sección declarativa debe ser extendida con las siguientes líneas:

```
      INTEGER, PARAMETER:: nd = n-IterConm
      REAL, DIMENSION(nd):: ipd
      REAL, DIMENSION(nd,nd):: densa
!HPF$ ALIGN densa(:,:) WITH A(:,IterConm+1:n)
```

La variable `nd` especifica la dimensión de la submatriz reducida que se va a factorizar según el algoritmo denso. El vector denso, `ipd`, de `nd` posiciones es utilizado durante la factorización densa para anotar las permutaciones de filas que aseguren la estabilidad numérica. Por último la matriz, `densa`, de `nd`×`nd` posiciones encargada de almacenar la submatriz reducida cuando termina la factorización dispersa. No merece una explicación exhaustiva la sentencia en la que se alinean las columnas de la matriz `densa` con las `nd` últimas columnas de la matriz `A`. En el ejemplo 4.11 vemos la sección de código encargada del cambio de estructura de datos y la llamada a la rutina de factorización densa, `FactorDensa()`.

Ejemplo 4.11 ESQUEMA DEL CÓDIGO HPF-2. 4ª PARTE: DENSA

```
                         :
! Inicializa la matriz densa densa
      densa=0
!HPF$ INDEPENDENT, NEW (aux,i)
      DO j=IterConm+1,n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)), BEGIN
        aux => vpiv(j)%p
        DO
           IF (.NOT. ASSOCIATED(aux)) exit
           i=aux%index
           densa(i-IterConm,j-IterConm) = aux%value
           aux=>aux%next
        END DO
!HPF$ END ON
      END DO
! Realiza la factorización densa.
      CALL FactorDensa(nd,densa,ipd)
                         :
```

En primer lugar, la matriz `densa` se inicializa a cero, para luego ocupar sólo las posiciones que especifica el patrón de la submatriz reducida. Para ello, un bucle `INDEPENDET` recorre las columnas de la lista enlazada y escribe en paralelo las columnas correspondientes de la matriz `densa`. Una vez hemos conmutado a la nueva estructura de datos, podemos llamar a una rutina HPF de factorización LU densa que nosotros hemos llamado `FactorDensa`. Una descripción más detallada de esta función se encuentra, como ejemplo, en [104].

4.5.3 Resolución triangular

Una vez tenemos los factores `L` y `U`, resolvemos el sistema $LUx = b$ mediante los procedimientos de substitución hacia adelante, $Ly = b$, y de substitución hacia atrás, $Ux = y$. A su vez estos procedimientos tienen su sección de código dispersa y la correspondiente densa. En esta sección, presentamos el código de la etapa substitución hacia adelante dispersa, obviando la substitución hacia atrás por su similitud y las etapas densas por su trivialidad.

De nuevo necesitamos extender la sección declarativa con unas cuantas líneas que especifiquen las nuevas estructuras de datos necesarias.

```
      REAL, DIMENSION(n):: x,y,b
      TYPE (ptr), DIMENSION(n):: vaux
!HPF$ ALIGN x(:) WITH A(*,:)
!HPF$ ALIGN y(:) WITH A(*,:)
!HPF$ ALIGN b(:) WITH A(:,*)
!HPF$ ALIGN vaux(:) WITH A(*,:)
```

Como vemos, tanto el vector `x` como el `y` están distribuidos cíclicamente en el array de procesadores. Sin embargo, el vector de términos independientes, `b` está replicado en todos los procesadores. El vector `vaux` se utiliza para permitir el acceso por filas durante el proceso, por lo que sigue una distribución cíclica. En el ejemplo 4.12 se muestra el código paralelo para la etapa de substitución hacia adelante dispersa.

Ejemplo 4.12 ESQUEMA DEL CÓDIGO HPF-2. 5ª PARTE: RESOLUCIÓN

```
                              :
! El bucle i recorre las filas
      DO i=1,IterConm
        acum=0.0
        vaux(i)%p => vpiv(i)%p%next
!HPF$ INDEPENDENT, REDUCTION(acum)
        DO j=1,i-1
          IF (ASSOCIATED(vaux(j)%p)) THEN
            IF (vaux(j)%p%index == i) THEN
              acum = acum + vaux(j)%p%value * y(j)
              vaux(j)%p => vaux(j)%next
            END IF
          END IF
        END DO
        y(i) = b(i) - acum
      END DO
```

El bucle `i` que recorre las filas es inherentemente secuencial. Para cada iteración externa se resetea el valor del acumulador `acum` y se inicializa el puntero `vaux(i)` para que apunte al primer elemento de `L` en la columna `i`. Es importante recordar, que al terminar la factorización dispersa el vector `vpiv` apunta a los elementos de la diagonal, los cuales pertenecen a la matriz `U`.

Por otro lado, el bucle `j` presenta una reducción paralela. En efecto, la variable `acum` debe acumular las contribuciones de las entradas en la fila `i` que posteriormente restaremos a `b(i)` para obtener la correspondiente componente del vector `y`. Así como ocurría en la selección del pivot, la reducción, en este caso una suma, consta de una etapa local y paralela en la que se considera la porción de fila asignada a cada procesador, seguida de una suma global implementada por el compilador. De nuevo, esta operación no está aún contemplada por HPF-2, pero es evidente que cualquier sentencia del tipo $a = a$ `op` b, donde la operación `op` es asociativa, es susceptible de su paralelización directa y trivial.

Otra dirección desde la que abordar la resolución triangular se basa en el cambio de estructura de datos. Dado que durante esta fase no aparecen operaciones de pivoteo ni de llenado, no es necesaria una estructura dinámica de datos. Podríamos por tanto, al tiempo que inicializamos la matriz `densa`, conmutar a una estructura CCS o CRS para la parte dispersa de los factores `L` y `U`. Esta aproximación, redunda en un ahorro de memoria, pero principalmente, consigue un mejor aprovechamiento de la cache y evita los accesos a entradas mediante punteros, salvando el coste temporal asociado.

4.6 Compilación y soporte en tiempo de ejecución

En la sección 4.4 hemos propuesto ciertas extensiones a HPF-2 que deben ser soportadas ahora por un módulo de compilación correctamente integrado en un compilador HPF-2. A grades rasgos este compilador realizará las fases de análisis del código fuente, paralelización y generación del código SPMD.

Para ayudarnos en la tarea de construcción de este compilador hemos utilizado la herramienta *Cocktail* [78]. Consiste en un conjunto de utilidades apropiadas para la consecución de los distintos pasos en la construcción del compilador: un analizador léxico (*scanner*) y sintáctico (*parser*) leen el código fuente, chequean la sintaxis y generan un árbol sintáctico; el análisis semántico se realiza sobre este árbol que posteriormente se traduce a una representación intermedia; finalmente el generador de código produce el código de salida [2].

4.6.1 Soporte para compilación

El análisis léxico del código HPF-2 extendido es la primera etapa a completar por el compilador. Los caracteres (letras, dígitos y símbolos) que definen la directiva `SPARSE` especificada en el ejemplo 4.4 deben ser identificados como palabras clave del lenguaje. Para ello la secuencia de caracteres que identifica una palabra clave se asocia a un componente léxico (*token*), por ejemplo "LLRCS". El generador de analizadores léxicos de *Cocktail* se llama *Rex* (*Regular Expression Tool*) [80]. El analizador léxico generado consiste en un autómata finito determinista dirigido por tablas que reconoce y clasifica los componentes léxicos del código.

A continuación el analizador sintáctico determina si la cadena de componentes léxicos proporcionados por el analizador léxico se ajusta a una cierta gramática. Para ello se construye un árbol sintáctico según la secuencia de *tokens* en el código fuente. De entre los tres generadores de analizadores sintácticos que ofrece *Cocktail* (*Lalr*, *Ell* y *Lark* [79, 81]) hemos usado el primero, el cual procesa gramáticas del tipo *LALR(1)* (*lookahead-LR*) [2].

En los ejemplos 4.13 y 4.14 se muestra el contenido del fichero que especifica la sintaxis asociada a la directiva `SPARSE`. En el primero encontramos una sección etiquetada con la clave `TOKEN` donde se definen los componentes léxicos y su codificación (coherente con la que devuelve el analizador léxico). La sección siguiente, `OPER`, especifica la precedencia y asociatividad de los operadores para resolver ambigüedades. Estos operadores son necesarios para contemplar las expresiones enteras que pueden acompañar al atributo `DIMENSION`.

Ejemplo 4.13 Fichero de sintáxis de la directiva SPARSE (1)

```
SCANNER shpf PARSER hpf

TOKEN
  Identifier     = 1
  '('            = 2
  ')'            = 3
  ','            = 4
  '::'           = 5
  Int_constant   = 6
  '!HPF$'        = 15
  'DYNAMIC'      = 16
  'DIMENSION'    = 17
  'SPARSE'       = 10
  'LLRS'         = 20
  'LLCS'         = 21
  'LLRCS'        = 22
  'CVS'          = 23
  'SINGLY'       = 30
  'DOUBLY'       = 31
  'REAL'         = 40
  'INTEGER'      = 41
  '+'            = 50
  '-'            = 51
  '*'            = 52
  '/'            = 53
  '**'           = 54

OPER
  LEFT '+' '-'
  LEFT '*' '/'
  RIGHT '**'
  LEFT unary_defined_operator
                             •
                             •
                             •
```

En el ejemplo 4.14 se muestra la última sección del fichero, `RULE`, que contiene las reglas de la gramática. Estas reglas están expresadas en notación *EBNF* (*Extended BNF*) pero son convertidas internamente a la notación BNF que acepta el generador *Lalr* [81].

Una vez se ha completado la etapa de análisis, comienza la etapa de paralelización. Ésta consiste en extraer la información almacenada en el árbol sintáctico para aplicar las estrategias de paralelización adecuadas. La información para la paralelización se obtiene principalmente de las directivas que proporciona el programador. De esta forma se identifica la estructura de datos dispersa, su distribución y alineamiento. Por otro lado, se determinan los bucles a paralelizar, traduciendo el espacio de iteraciones y anotando los requerimientos de información no local para la posterior gestión de las comunicaciones.

En el último paso se genera el código SPMD en Fortran90. Este código contiene llamadas a nuestra librería en tiempo de ejecución DDLY que discutimos en el siguiente epígrafe. Por tanto se llevarán a cabo las transformaciones necesarias para insertar el soporte en tiempo de ejecución y las modificaciones anotadas en la etapa de paralelización previa. El código generado puede ser convertido en el correspondiente ejecutable mediante un compilador estándar de Fortran90 para las plataformas paralelas que soportan los interfaces de pase de mensajes discutidos en la sección 2.2.1 (PVM, MPI y Cray SHMEM).

Ejemplo 4.14 Fichero de sintáxis de la directiva SPARSE (y 2)

$$\vdots$$

```
RULE
hpf_directive: hpf_header sparse_directive '::' array_objects
             .
hpf_header: '!HPF$' datatype
          .
datatype: 'REAL'
        | 'INTEGER'
        .
sparse_directive: sparse_stmt
                | sparse_stmt opt_dyn
                | sparse_stmt opt_dim
                | sparse_stmt opt_dyn opt_dim
                | sparse_stmt opt_dim opt_dyn
                | opt_dyn sparse_stmt
                | opt_dim sparse_stmt
                | opt_dyn opt_dim sparse_stmt
                | opt_dim opt_dyn sparse_stmt
                | opt_dyn sparse_stmt opt_dim
                | opt_dim sparse_stmt opt_dyn
                .
array_objects: (Identifier [array_content]) || ','
             .
array_content: '(' int_expr [',' int_expr] ')'
             .
opt_dyn: ',' 'DYNAMIC'
       .
opt_dim: ',' 'DIMENSION' array_content
       .
int_expr: '(' int_expr ')'
        | int_expr '+' int_expr
        | int_expr '-' int_expr
        | int_expr '*' int_expr
        | int_expr '/' int_expr
        | int_expr '**' int_expr
        | '-' int_expr  PREC unary_defined_operator
        | Int_constant
        | Identifier
        .
sparse_stmt: ',' 'SPARSE' '(' sparse_content ')'
           .
sparse_content: 'LLRS'  '(' ll_spec ')'
              | 'LLCS' '(' ll_spec ')'
              | 'LLRCS' '(' ll2_spec ')'
              | 'CVS' '(' cvs_spec ')'
              .
cvs_spec: Identifier ',' Identifier ',' Identifier
        .
ll_spec: cvs_spec ',' link_spec
       .
ll2_spec: cvs_spec ',' cvs_spec ',' link_spec
        .
link_spec: 'SINGLY'
         | 'DOUBLY'
         .
```

4.6.2 Soporte en tiempo de ejecución

Dado que la distribución y el patrón de acceso a los datos depende de los datos que se vayan a procesar, parte del análisis de los códigos propuestos en HPF-2 extendido, se debe realizar en tiempo de ejecución. Para soportar el incremento de funcionalidad del compilador, hemos extendido nuestra librería en tiempo de ejecución DDLY (del inglés, `Data Distribution Layer`) [147]. Más precisamente, hemos añadido un conjunto de rutinas para manejar listas enlazadas, que serán llamadas por el código generado por el compilador de HPF-2 extendido.

Las rutinas DDLY realizan un conjunto de operaciones de alto nivel orientadas a la distribución y alineamiento de arrays en multiprocesadores de memoria distribuida. Concretamente, la librería proporciona un interfaz que permite el manejo de arrays como objetos abstractos, ocultando al usuario la estructura interna usada para almacenar y distribuir datos. También proporciona el manejo necesario en el pase de mensajes[2], optimización de las comunicaciones y gestión de la entrada/salida. El interfaz de usuario en DDLY está basado en descriptores de objetos. Una operación básica es la declaración de un descriptor de un array o matriz y su asignación a un objeto particular. Estos descriptores se gestionan de forma similar a como UNIX maneja los descriptores de ficheros.

Para declarar un descriptor disponemos de la rutina `ddly_new`, mientras que para asignar un descriptor a un objeto llamamos a la rutina `ddly_init`. Una vez enlazados el objeto y su descriptor, cualquier acceso y manipulación del primero se lleva a cabo a través de su correspondiente descriptor. Por ejemplo, la librería incluye ciertas rutinas para distribuir y alinear objetos. En la llamada a estas rutinas, sólo es necesario especificar el descriptor del objeto y algún que otro argumento más que indique el tipo de distribución/alineamiento, o el descriptor del objeto con el que se va a alinear.

Con este interfaz, cada directiva de distribución/alineamiento de HPF-2 es traducida en su correspondiente llamada a una de las funciones de la librería DDLY. El paradigma que subyace en la librería es SPMD, es decir, las funciones son llamadas por todos los procesadores con los mismos argumentos. De esta forma, las funciones se ejecutan de forma coordinada en el multiprocesador.

En los siguientes ejemplos presentamos el código generado por el compilador de HPF-2 extendido. En la salida hipotética en Fortran 90 de este compilador, todas las directivas introducidas en el código secuencial han sido traducidas en llamadas a las rutinas de la librería DDLY.

4.6.2.1 Sección de declaración

En el ejemplo 4.15 encontramos la salida compilada de la sección de declaración descrita en el ejemplo 4.6. La primera rutina DDLY que encontramos en el ejemplo es el procedimiento `ddly_new_topology` el cual inicializa la topología deseada sobre la que se ejecutará el código SPMD. En este caso se especifica un vector lineal de `Q` procesadores. La información relativa a la topología se almacena en el descriptor `td`. El procedimiento

[2]Para ello se apoya en los interfaces estándar MPI, PVM y SHMEM.

ddly_HB_read simplifica la lectura de la matriz de un fichero con formato Harwell-Boeing. La matriz dispersa se almacena en memoria en formato CCS (ver sección 1.2.2), mediante los tres vectores (Val,Col,Filpt) y el vector b para los términos independientes.

Ejemplo 4.15 SALIDA DEL COMPILADOR: SECCIÓN DE DECLARACIÓN

```
!HPF$ PROCESSORS, DIMENSION(Q):: linear
! --> td el el descriptor de la topología para una malla (1 x Q)
      CALL ddly_new_topology(td,1,Q)

!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first, last, vsize, DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(n)
! --> Lee la matriz Harwell-Boeing  (Val,Col,Filpt)
      CALL ddly_HB_read(n,alpha,Val,Col,Filpt,b)
! --> Crea el descriptor de la matriz A, md_a, en formato CCS
      CALL ddly_new(md_a,CCS,DDLY_MF_REAL)
! --> Inicializa md_a
      CALL ddly_init(md_a,n,alpha,Val,Col,Filpt)
! --> Crea el descriptor del array VCOL, vd_vcol, en formato CVS
      CALL ddly_new(vd_vcol,CVS,DDLY_VF_REAL)
! --> Inicializa vd_vcol
      CALL ddly_init(vd_vcol,vcoli,vcolv,size)

! --> Crea e inicializa los descriptores para otros arrays distribuidos ...
      ...
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
! --> Distribución BCS de A (especificada por md_a)
      CALL ddly_bcs(md_a, td)
! --> md_a es ahora el descriptor de la matriz distribuida
! --> Cambia la estructura de datos de CCS a LLCS
      CALL ddly_ccs_to_llcs(md_a,first,last,vsize,DOUBLY)

!HPF$ ALIGN iq(:) WITH A(*,:)
! --> iq es alienado con la segunda dimensión de A
      CALL ddly_alignv(vd_iq,md_a,SecondDim)

!HPF$ ALIGN vpiv(:) WITH A(*,:)
      CALL ddly_alignv(vd_vpiv,md_a,SecondDim)

!HPF$ ALIGN vmaxval(:) WITH A(*,:)
      CALL ddly_alignv(vd_vmaxval,md_a,SecondDim)

!HPF$ ALIGN vcol(:) WITH A(:,*)
! --> VCOL es alineado con la primera dimensión de A
      CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)
```

En la llamada ddly_new la matriz A se asocia al descriptor de matriz md_a, el cual indica que la matriz dispersa A está almacenada en formato comprimido por columnas (CCS) y que el tipo de datos de la matriz es real (DDLY_MF_REAL). Posteriormente ddly_init inicializa la matriz A según el contenido de los tres vectores (Val,Col,Filpt) leídos anteriormente.

En cuanto al vector comprimido VCOL el procedimiento es parecido al que hemos seguido con la matriz A: ddly_new asocia el vector VCOL (en formato CVS y con entradas reales) con el descriptor de vectores vd_vcol; posteriormente ddly_init inicializa el descriptor asociándole la variable size y los vectores vcoli y vcolv (inicialmente vacíos). El resto de los vectores que se distribuyen o alinean (iq, vpiv y vmaxval) también se asocian a descriptores de vectores para simplificar la gestión de la distribución/alineamiento.

Una vez tenemos inicializados los descriptores de vectores y matrices pasamos a la distribución y alineamiento de estas estructuras. El procedimiento ddly_bcs distribuye la matriz A (identificada por su descriptor `md_a`) sobre el array de procesadores (identificado por el descriptor de topología `td`), aplicando el esquema de distribución BCS (ver sección 3.2). Mediante la llamada `ddly_ccs_to_llcs` todos los procesadores realizan localmente y en paralelo un cambio de la estructura de datos CCS a LLCS.

Por último se procede a alinear algunos vectores con la matriz distribuida. Como vemos en la sección de declaración los vectores pueden ser alineados con la primera o la segunda dimensión de la matriz `A`. De esta forma las directivas `ALIGN` de HPF se convierten en llamadas `ddly_alignv` (para alinear vectores densos) y `ddly_aligncvs` (para alinear vectores comprimidos). Dado que el vector comprimido `VCOL` se alinea con la primera dimensión de `A` el vector queda replicado.

Como vemos, ha sido necesario ampliar la librería DDLY con un conjunto de nuevas rutinas de inicialización/distribución/alineamiento para contemplar la amplia variedad de situaciones que podemos encontrar usando la directiva `SPARSE`.

4.6.2.2 Sección de ejecución

La salida asociada a la etapa de factorización dispersa discutida en los ejemplos 4.7, 4.8 y 4.9 se muestra en el código del ejemplo 4.16.

Como vemos, todos los bucles anotados con la directiva `INDEPENDENT` son ejecutados en paralelo debido a la distribución y la localización del espacio de iteraciones para cada procesador. Para ello, las funciones `ddly_LowBound` y `ddly_UpBound` se aplican a las cotas inferior y superior, respectivamente, de los bucles paralelos. Evidentemente, estas funciones, presentadas en el ejemplo 4.17, realizan la transformación dependiendo de la distribución que se esté utilizando. Estas funciones también se aplican a las cotas de las operaciones que utilizan notación de subarrays[3], ya que son totalmente equivalentes a un bucle DO paralelo.

Los comentarios "`Cuerpo del bucle`" y "`Cuerpo del ON HOME`" indican que la sección de código HPF-2 correspondiente de los ejemplos 4.7, 4.8 y 4.9 se copia directamente (sin modificaciones) en el código de salida.

Las operaciones de reducción realizadas en la etapa de búsqueda del pivot merecen una atención especial. De forma similar a como opera el paralelizador Polaris (que comentaremos en la siguiente sección), el compilador de HPF-2 divide los bucles de reducción (anotados con la cláusula `REDUCTION`) en dos pasos. En el caso de la reducción para obtener el máximo valor absoluto en la fila activa `k`, tenemos:

- Primero se realiza una reducción local donde la variable local `maxpiv` toma el valor máximo de la sección de array `vmaxval` que le corresponde.

- Seguidamente tiene lugar la reducción global de los valores `maxpiv` correspondientes a cada procesador. Para ello, la rutina `ddly_ReduceScalarMax` realiza las comunicaciones necesarias para que todos los procesadores almacenen en `maxpiv` el máximo global.

[3]Expresiones del tipo A(1:n), soportadas en F90 [1].

Esta reducción simple está contemplada en las versiones actuales de HPF, sin embargo, la que encontramos a continuación necesita un soporte adicional. Nos referimos a la reducción que selecciona un pivot numéricamente estable. Durante la reducción local se identifica el pivot que cumple la condición especificada en la sentencia `IF`. En una segunda etapa, la reducción global se realiza gracias al procedimiento `ddly_ReduceLocMaxAbs`.

Ejemplo 4.16 Salida del compilador: Sección de Ejecución

```
!HPF$ INDEPENDENT
! --> Bucle paralelo
      DO j = ddly_LowBound(1), ddly_UpBound(n)
        vpiv(j)%p => first(j)%p
      END DO

! --> Bucle principal de la LU
main: DO k = 1, n

! --> Pivoteo
! --> Búsqueda del máximo valor absoluto en la fila activa k
!HPF$ INDEPENDENT
! --> Bucle paralelo
        DO j = ddly_LowBound(k), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
              ****** Cuerpo del bucle ****** (local a cada procesador)
!HPF$ END ON
        END DO
! --> Reducción paralela en dos etapas
! --> ... Primero, reducción local, ...
        maxpiv = MAXVAL(vmaxval(ddly_LowBound(k):ddly_UpBound(n)))
! --> ... Segundo, reducción global
        ddly_ReduceScalarMax(maxpiv)
! --> Elección de un pivot numéricamente estable (operación de reducción)
        actpiv=vmaxval(k)
        pivcol=k
        maxpiv = maxpiv*u
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
! --> Bucle paralelo
        DO j = ddly_LowBound(k), ddly_UpBound(n)
          IF ( vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j) ) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
! --> Reducción global
        pivcol = ddly_ReduceLocMaxAbs(actpiv,maxpiv,pivcol,iq)
        IF (pivcol /= k) THEN
! ----> Intercambio de columnas
           CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF

! --> La columna del pivot es actualizada y empaquetada en VCOL
!HPF ON HOME (vpiv(k)), RESIDENT (A(*,k)) BEGIN
! --> Este bucle aparece debido a la directiva ON HOME
        DO dum = ddly_LowBound(k),ddly_UpBound(k)
          ****** Cuerpo del ON HOME ******
        END DO
!HPF END ON

! --> Radiación de la columna del pivot (dado que VCOL está replicado)
        CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)

! --> Actualización de la submatriz reducida de A
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
! --> Bucle paralelo
        DO j = ddly_LowBound(k+1), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          ****** Cuerpo del ON HOME ******
!HPF$ END ON
        END DO
      END DO main
```

Se han implementado varias rutinas DDLY para soportar una gran variedad de reducciones. Sin embargo, tampoco es difícil construir estas rutinas en tiempo de compilación a partir de las sentencias del núcleo de la reducción. Esto se puede llevar a cabo siguiendo la estrategia de generación de código de reducción que incorpora Polaris, con la ventaja de que no es necesario detectar el bucle de reducción (el cual está ya marcado con la cláusula `REDUCTION`).

Ejemplo 4.17 Funciones para determinar el espacio de iteraciones

```
! Los bucles paralelos (DO j = globa, globb) se convierten en
! (DO j = ddly_LowBound(globa), ddly_UpBound(globb))
! my_pe es una variable global que contiene el identificador de proceso
! N$PES = número de PEs (variable global en el Cray T3E)

      INTEGER FUNCTION ddly_LowBound (i)
        INTEGER i
        ddly_LowBound = (i-1)/N$PES+1
        IF (my_pe < MOD(i-1,N$PES)) ddly_LowBound = ddly_LowBound+1
      END FUNCTION ddly_LowBound

      INTEGER FUNCTION ddly_UpBound (i)
        INTEGER i
        ddly_UpBound = i/N$PES
        IF (my_pe < MOD(i,N$PES)) ddly_UpBound = ddly_UpBound+1
      END FUNCTION ddly_UpBound
```

En la sección de código en la que la columna activa `k` se actualiza y copia en VCOL, el compilador incluye un bucle de índice `dum`. De esta forma, siguiendo la directiva ON HOME, el procesador propietario de `vpiv(k)` es el único que ejecuta esta sección de código. El compilador detecta que la estructura de datos (CVS) asociada a `VCOL` es modificada. Dado que este vector está replicado, cada componente modificada en un procesador se debe actualizar en el resto de los procesadores. No es difícil incluir una etapa de optimización que agrupa todas las escrituras al final del bucle de actualización, realizando la radiación de todo el vector mediante la llamada a `ddly_aligncvs`.

4.7 Evaluación de resultados

En esta sección nos centramos en la evaluación de las prestaciones del algoritmo paralelo LU discutido en la sección anterior. Los experimentos se han realizado en el multiprocesador Cray T3E, utilizando su compilador de Fortran90 y las librerías de comunicación SHMEM [25] incluidas en las llamadas DDLY.

Como se ha mencionado, las columnas de la matriz `A` se distribuyen cíclicamente sobre una array unidimensional de `Q` procesadores, y se almacenan mediante listas doblemente enlazadas. Queremos remarcar que el algoritmo paralelo, presentado en el ejemplo 4.15, es básicamente la versión secuencial con índices locales en lugar de globales y rutinas SHMEM para realizar las operaciones comunicación/sincronización. Estas dos diferencias entre el código paralelo y el secuencial se ocultan al programador gracias a llamadas a las rutinas DDLY comentadas en la sección anterior. Este código paralelo debe ser considerado como la salida del compilador HPF-2 extendido y no una optimización del código realizada manualmente.

En la figura 4.3 presentamos los tiempos de ejecución y la aceleración variando el número de procesadores. Las matrices de prueba han sido escogidas del conjunto Harwell-Boeing y se describen en la tabla 1.2. Esta elección es fruto de la accesibilidad y uso extensivo de estas matrices, lo cual permite comparar los resultados de otras implementaciones.

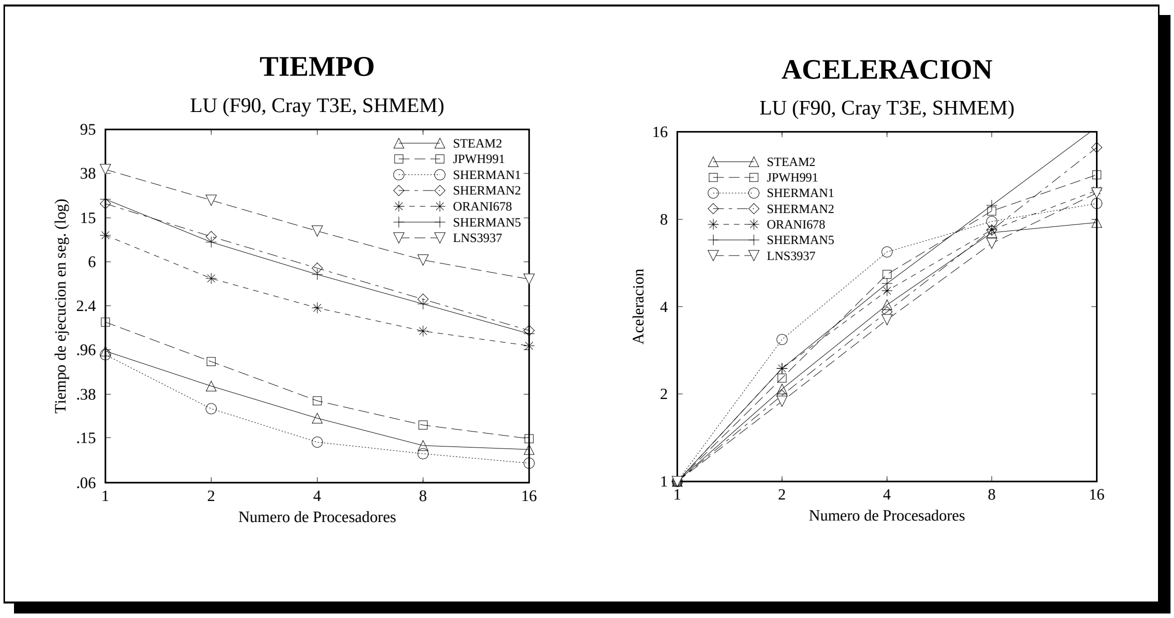

Figura 4.3: Tiempos de ejecución de la LU dispersa en paralelo y aceleración para diferentes tamaños del array de procesadores con comunicaciones SHMEM en un Cray T3E.

Obsérvese las altas eficiencias obtenidas cuando el tamaño de la matriz es suficientemente grande. De cualquier modo, matrices más grandes reportarían mejores resultados y nos permitirían explotar más paralelismo usando un mayor número de procesadores. También han sido evaluadas otro tipo de configuraciones de mallas en lugar de arrays de procesadores, pero los mejores tiempos paralelos se consiguen para topologías lineales, donde las matrices se distribuyen por columnas.

En nuestros experimentos, una etapa de análisis se ejecuta inicialmente para proporcionar los vectores de permutación adecuados. Esta etapa también proporciona la iteración `IterConm`, después de la cual se conmuta a código denso, lo cual tiene lugar en nuestros experimentos cuando la densidad de la submatriz reducida supera el 15%. El valor del parámetro `u` que afecta al umbral que han de superar los pivots tiene el valor 0.1 como se recomienda en el capítulo 3. Los tiempos de análisis y resolución triangular se muestran en la tabla 4.1, donde se aprecia que no suponen un tiempo significativo comparado con el tiempo de factorización.

También debemos justificar por qué el tiempo secuencial de nuestro algoritmo en F90 es muchas veces superior al de otras implementaciones codificadas en C, por ejemplo, la descrita en el capítulo 3. Después de realizar una serie de pruebas a los dos compiladores, en las que se evalúa el rendimiento de las operaciones con punteros, podemos afirmar que el compilador de F90 tiene un comportamiento pobre en relación al compilador de C al manejar este tipo de estructuras de datos (al menos en el Cray

	Tiempos		**Llenado**
Matriz	**Análisis**	**Resolución**	**n$^{\underline{o}}$ de entradas**
STEAM2	.059	.0078	11869+(314x314)
JPWH991	.054	.0074	8866+(305x305)
SHERMAN1	.028	.0037	6306+(192x192)
SHERMAN2	.43	.039	24281+(795x795)
ORANI678	.35	.029	32235+(638x638)
SHERMAN5	.35	.034	32650+(698x698)
LNS3937	.95	.067	78972+(961x961)

Tabla 4.1: Tiempos de la etapa de análisis y resolución triangular (en seg.) y llenado final en los factores LU.

T3E). Más precisamente, nuestros códigos de prueba realizan inserciones en listas (con llamadas a `ALLOCATE` en F90 y `alloc` en C), las recorre y borran elementos (con llamadas a `DEALLOCATE` en F90 y `free` en C). Después de un gran número de medidas el C ha resultado, en media, cuatro veces más rápido en la inserción, dos veces más rápido recorriendo listas, y cinco veces más rápido al borrar entradas.

Por otro lado, es importante validar las prestaciones de nuestro código secuencial comparándolo con otro considerado como estándar, por ejemplo, la rutina en Fortran77 MA48 de las Harwell Subroutine Library [60]. En la tabla 4.2 se presenta una comparativa de las características significativas de ambas implementaciones, como son el tiempo de ejecución secuencial y el error de factorización. Podemos apreciar que la rutina MA48 es más rápida que nuestro algoritmo, pero también es justo considerar la ineficiencia del compilador de F90 al manejar listas y estructuras dinámicas. Sin embargo, los errores de factorización son prácticamente idénticos en los dos algoritmos. De igual modo, los tiempos de análisis, resolución triangular y, por otro lado, el llenado de la matriz LU, son completamente semejantes. La principal ventaja de nuestra aproximación es que resulta fácilmente paralelizable, aspecto que hemos demostrado altamente cuestionable, en el capítulo 3, para el caso de la MA48, dadas sus características inherentemente secuenciales a causa de su organización *left-looking*.

	Tiempos		**Error**
Matriz	**F90 – MA48**	**ratio**	**F90 – MA48**
STEAM2	.9361 – .61	(1.53)	.15E-11 – .13E-11
JPWH991	1.707 – .88	(1.94)	.44E-13 – .82E-13
SHERMAN1	.8646 – .19	(4.55)	.14E-12 – .16E-12
SHERMAN2	20.21 – 16.7	(1.21)	.14E-5 – .15E-5
ORANI678	10.46 – 6.48	(1.61)	.70E-13 – .74E-13
SHERMAN5	22.24 – 11.0	(2.02)	.75E-12 – .59E-12
LNS3937	41.26 – 25.8	(1.60)	.15E-2 – .13E-2

Tabla 4.2: Comparación entre el algoritmo en F90 y la MA48 (tiempos en seg.)

También es interesante comparar nuestra solución paralela con otras aproximaciones a la paralelización semi-automáticas basadas en técnicas en tiempo de ejecución. Por ejemplo, la librería en tiempo de ejecución CHAOS [115, 132] se basa en el paradigma Inspector-Ejecutor para manejar accesos irregulares a datos. Mediante estas rutinas se pueden generar tablas distribuidas de traducción de índices globales a direcciones locales a cada procesador. Las rutinas CHAOS también generan la planificación de comunicaciones necesarias en los accesos a datos no locales. En [15] mostramos resultados en los que resulta manifiesta la superioridad de nuestra aproximación basada en distribuciones pseudo-regulares frente a la que puede ser implementada con CHAOS. En general, y para aplicaciones con llenado y pivoteo, como los algoritmos dispersos de factorización LU y QR, la implementación basada en el esquema de distribución LLCS-Scatter es casi un orden de magnitud más rápida que bajo el paradigma proporcionado por CHAOS. Resultados similares se presentan en [153] para problemas estáticos en los que no aparecen operaciones de llenado o pivoteo. Esto se justifica principalmente por la gran cantidad de comunicaciones y de posiciones de memoria, necesarias en CHAOS para acceder a grandes tablas de directorio distribuidas entre los procesadores. Por contra, el esquema LLCS-Scatter es adecuado para nuestros problemas dispersos ya que explota la localidad y minimiza las comunicaciones. Además no requiere de memoria adicional ni de comunicaciones para encontrar la dirección de datos no locales, ya que todos los procesadores conocen la posición de cualquier entrada global mediante una simple operación aritmética. Podemos concluir, por tanto, que nuestro esquema de distribución pseudo-regular consigue mejores prestaciones que el modelo general Inspector-Ejecutor proporcionado por CHAOS.

4.8 Otras aplicaciones

Aunque en este capítulo nos hemos centrado en el caso de la factorización dispersa LU en su versión *right-looking*, las directivas y aportaciones introducidas son válidas para un amplio espectro de aplicaciones dispersas en las que aparecen operaciones de llenado o pivoteo.

Por ejemplo, el código descrito en el capítulo 3 donde se explota el paralelismo asociado a la actualización de rango `m`, también podría ser expresado con las extensiones propuestas. En este caso, un esquema LLRCS-Scatter que permita acceso tanto por filas como por columnas y algunas etapas de reducción algo más complejas para determinar la compatibilidad de los pivots serían prácticamente suficientes. Sin embargo, también está claro que los tiempos de ejecución serán mayores, debido a que la herramienta de paralelización semi-automática no genera un código tan optimizado como el que obtenemos de paralelizar a mano.

En cuanto a las soluciones multifrontales, las implementaciones paralelas a nivel de tarea no se ajustan al paradigma implícito en HPF, donde los bucles constituyen la fuente esencial de paralelismo. Sin embargo, el paralelismo a nivel de bucle es también aplicable y puede ser representado en muchos casos mediante nuestras directivas. Por ejemplo, la factorización de Cholesky dispersa presentada en [65], en la que el problema del llenado y pivoteo está contemplado en una etapa de análisis anterior, un esquema BCS permite describir los dos bucles que consumen más de un 90% del tiempo de ejecución total. Lo dicho es aplicable hasta cierto punto en un algoritmo con filosofía supernodo [51], ya que puede ser necesaria una simplificación de la estructura de datos para hacer realizable la implementación en HPF-2. El paralelismo a nivel de tarea inherente en el árbol de supernodos o de eliminación puede explotarse gracias a la directiva `TASK_REGION` aprobada en HPF-2.

Otros algoritmos con llenado y pivoteo como las factorizaciones QR, ya sea mediante un algoritmo de Gram-Schmidt modificado (MGS) o por reflexiones Householder (HR) han sido totalmente expresados en HPF-2 extendido y validados experimentalmente en [15].

Aparte de los algoritmos de factorización, encontramos otro tipo de problemas dispersos que aunque no presentan operaciones de pivoteo, sí contienen operaciones de llenado. Nos referimos a códigos como la suma, el producto o la transposición de matrices dispersas. Estos algoritmos son totalmente abordables de forma sencilla en el entorno de paralelismo de datos con las nuevas directivas presentadas, y los consideramos un caso particular de los anteriores.