Paralelizacion Automatica de Codigos Fortran Irregulares

R. Asenjo
E. Gutierrez
Y. Lin
D. Padua
B. Pottenger
E.L. Zapata

September 1997
Technical Report No: UMA-DAC-97/13

Published in:

VIII Jornadas de Paralelismo
Caceres, Spain, September 10-12, 1997, pp. 451-462

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Paralelización Automática de Códigos Fortran Irregulares

Rafael Asenjo, Eladio Gutierrez, Yuan Lin*, David Padua*, Bill Pottenger*, Emilio Zapata

Dept. Arquitectura de Computadores
Campus de Teatinos. Universidad de Málaga
Apdo. 4114 - 29080 MALAGA
+34-(9)5-213-27-91, fax: +34-(9)5-213-27-90
{asenjo,eladio,ezapata}@ac.uma.es

* Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801-2307
{potteng,yuanlin,padua}@csrd.uiuc.edu

Resumen

En este artículo se presenta una comparación del resultado obtenido por los paralelizadores automáticos PFA (Power Fortran Accelerator) y Polaris sobre un conjunto de códigos cuyas estructuras de datos son irregulares, proponiendo técnicas adicionales que podrían ser incluidas en estos paralelizadores para obtener un mejor rendimiento

Palabras claves: Paralelización automática, reducción, reducción de histograma, privatización, expansión, "copy-in", "copy-out".

1 Introducción

Los patrones de acceso irregulares han tenido tradicionalmente dificultades en su detección y paralelización automática y en muchos casos ni siquiera es posible. No obstante, esta clase de problemas son importantes ya que representan una fracción considerable de los que encontramos en aplicaciones, que por su naturaleza, son irregulares.

Aunque algún trabajo se ha realizado en este sentido, son pocos los estudios que tratan con códigos completos[18]. En este artículo presentamos diversas técnicas que son aplicables a una colección de códigos reales con patrones de acceso irregulares, realizando una comparación entre los resultados obtenidos por dos herramientas automáticas: el paralelizador comercial PFA de SGI[22], y el transformador de código Polaris[6].

2 Códigos evaluados

El conjunto de códigos sobre los que hemos experimentado está formado por una colección de aplicaciones irregulares y dispersas. Estos códigos pertenecen al conjunto de aplicaciones motivadoras del HPF-2 [10], excluyendo el producto disperso de matrices y el algoritmo de Lanczos. El código de dinámica de fluidos GCCG ha sido desarrollado en el Instituto de Tecnología del Software y Sistemas Paralelos de la Universidad de Viena, Austria. A continuación sumarizamos los códigos empleados resaltando las características más importantes.

CHOLESKY: La factorización de Cholesky de una matriz dispersa simétrica definida positiva A genera una matriz triangular inferior L de forma que $A = LL^T$. El código emplea un método directo para resolver el sistema de ecuaciones lineales.

La matriz usada fue la BCSSTK30 de la colección Harwell-Boeing[9]. Se muestra a continuación el tipo de aceso que encontramos en este código:

```
do s = 1,nsu
   do j = isu(s),isu(s+1)-1
      snhead(j) = isu(s)
      nafter(j) = isu(s+1) - 1 - j
   enddo
 enddo
```

DSMC3D: Este programa constituye una modificación de DSMC (Direct Simulation Monte Carlo) en 3 dimensiones. DSMC implementa la simulación del comportamiento de partículas de un gas en el espacio usando el método de Monte Carlo[5]. Un ejemplo del tipo de acceso que aparece en esta aplicación es el siguiente :

```
do i = 1, NM
   if (mcell(i)+1 .eq. ncell(i)) then
         cellx(mcell(i)) = cellx(mcell(i)) + 1
   endif
 enddo
```

EULER: Este código es una aplicación que resuelve las ecuaciones del mismo nombre en una malla irregular. La computación se basa en descripciones, referenciadas con indirecciones, de la malla de trabajo. Además encontramos dichas indirecciones a la derecha e izquierda de la asignación. El siguiente fragmento sirve de ejemplo de estos patrones de indirección:

```
do  ng=1,ndegrp
 do  i=ndevec(ng,1),ndevec(ng,2)
   n1        = nde(i,1)
   pw(n1,1)  = pw(n1,1) + qw(n2,1)*eps(i)
 enddo
enddo
```

GCCG: Esta aplicación es un ejemplo extraído de la dinámica computacional de fluidos. El patrón de acceso es similar al que encontramos en los programas de elementos finitos, donde el valor de un elemento se determina a partir de las contribuciones de sus vecinos, empleando para ello indirecciones, que aparecen en el miembro derecho de la expresión que se computa:

```
   do  nc=nintci,nintcf
    direc2(nc)=bp(nc)*direc1(nc)       -bs(nc)*direc1(lcc(nc,1))
  *            -bw(nc)*direc1(lcc(nc,4))-bl(nc)*direc1(lcc(nc,5))
   enddo
```

LANCZOS: El algoritmo de Lanczos con reortogonalizacion completa, determina los autovalores de una matriz simétrica[11]. LANCZOS es una implementación de dicho algoritmo para matrices dispersas. La mayor carga computacional se centra en el cálculo de un producto matriz-vector y en la reortogonalización de una matriz densa. Los patrones de acceso incluyen subíndices con indirecciones en la parte derecha de la asignación como se muestra en el siguiente fragmento del código:

```
 do  j=1,a_nr
  do  k=ar(j),ar(j+1)-1
   r(j)=r(j)+ad(k)*q(ac(k),i)
  enddo
 enddo
```

MVPRODUCT: En este código hemos incluido un conjunto de operaciones básicas entre matrices dispersas [3, 11] tales como el producto matriz vector disperso y el producto y suma de 2 matrices dispersas. La representación de las matrices empleadas siguió los esquemas CRS (compressed row storage) y CCS (compressed column storage) [19]. El acceso típico de estas operaciones corresponde con el siguiente:

```
 do  i=1,a_nr
  do  k=1,b_nc
   do  ja=ar(i),ar(i+1)-1
    do  jb=bc(k),bc(k+1)-1
      if (ac(ja).eq.br(jb)) THEN
       c(i,k)=c(i,k)
&             + ad(ja)*bd(jb)
      endif
    enddo
   enddo
  enddo
 enddo
```

NBFC: El cálculo de fuerzas representa un elemento clave del cálculo en dinámica molecular[7]. NBFC determina interacciones, de tipo culombiano, entre partículas; aquí las fuerzas que actúan sobre una partícula se calcula a partir de una lista de partículas vecinas. Un ejemplo de patrones de acceso es el siguiente:

```
do k = 1, ntimestep
 do i = 1, natom
  do  j = inblo(i),inblo(i+1)-1
   dx(jnb(j)) = dx(jnb(j)) - (x(i) - x(jnb(j)))
   dx(i)      = dx(i) + (x(i) - x(jnb(j)))
  enddo
 enddo
enddo
```

SpLU: este código calcula la factorización LU de una matriz dispersa. La versión de SpLU sobre la que hemos experimentado es la incluida en la versión original de las aplicaciones motivadoreas del HPF-2[2]. El patrón de acceso incluye a límite de bucles en los que existen indirecciones:

```
do i=cptr1(j),cptr2(j)
   a(shift)=a(i)
   shift = shift + 1
enddo
```

3 Técnicas de Paralelización y transformación

Las técnicas empleadas en la paralelización de los códigos presentados se centran en los siguientes aspectos: Reducciones de Histograma, substitución de generadores de números pseudoaleatorios, comprobación de la monotonía en los vectores ínidices, comprobación de intervalos no solapados en variables de inducción, "Copy-in" y "Copy-out" y mejora en bucles con múltiples puntos de salida.

3.1 Reducciones de histograma

El siguiente código contiene una reducción en el array A cuyo subíndice contiene una función $f(i,j)$ que varía en cada iteración.

```
 do i=1,n
   do j = 1, m
      k = f(i,j)
      a(k) = a(k) + expression
   enddo
enddo
```

Debido a que el subíndice del array depende de las variables de iteración (i,j), pueden presentarse dependencias entre iteraciones en tiempo de ejecución. Este patrón se presenta comúnmente tanto en códigos densos como dispersos y se conoce con el nombre de *reducciones de histograma*[16, 13].

En el conjunto de códigos que estamos estudiando, dichas reducciones ocurren en los códigos: NBFC, CHOLESKY, DSMC3D y EULER. La paralelización de este tipo de reducciones se basa en técnicas en tiempo de ejecución que dependen del carácter asociativo de la operación realizada. El paralelizador automático Polaris es capaz de reconocer y transformar este tipo de reducciones[6].

La transformación necesaria en el bucle para hacerlo paralelo, puede realizarse de tres formas básicas: mediante *secciones críticas*, *privatización* o *expansión*. Ilustramos estas técnicas con varios ejemplos empleando la notacion del lenguaje IBM's Parallel Fortran [12].

- Sección crítica

La primera posibilidad implica la inserción de primitivas de sincronización rodeando la sentencia de reducción, convirtiendo la operación (en este caso la suma) en una operación atómica. Es una solución elegante para arquitecturas que proporcionan rápidas primitivas de sincronización.

- Privatización

En reducciones privatizadas, realizamos copias de la variables de reducción de forma que cada copia es creada y usada por cada procesador, de forma independiente, usando ésta en lugar de la variable original. Así realizaría dicha transformación en este ejemplo:

```
parallel loop i=1,n
  private a_p(sz)
  dofirst
    a_p(1:sz) = 0
  doevery
```

```
      do j = 1, m
         k = f(i,j)
         a_p(k) = a_p(k) + expression
      enddo
   dofinal lock
      a(1:sz) = a(1:sz) + a_p(1:sz)
enddo
```

Cada procesador ejecuta la sección `dofirst` del bucle paralelo al comienzo de su espacio local de iteraciones. La sección `doevery` es ejecutada en cada iteracion. Por último la sección `dofinal` es ejecutada una vez por cada procesador después de completar el número de iteraciones que le correspondía. El `dofinal` se ejecuta dentro de una sección critica. Este método es más rápido que el anterior y aplicable sólo a memoria distribuida.

- Expansión

Esta tercera técnica emplea una expansión para conseguir el mismo efecto que la privatización. En vez de poseer copias privadas de la variable de reducción, lo que se hace es expandir la variable, añadiendo un índice más, en un número igual al número de procesos (procesadores) que intervienen en la computación. Las variables de reducción se reemplazan por referencias a este nuevo array global que contiene una dimensión más que hace referencia al proceso (procesador) que esta ejecutando la iteración actual.

La inicialización y la suma (reducción) tienen lugar en bucles separados que se sitúan antes y después, respectivamente, del bucle original. En este método no son necesarios puntos de sincronización, y todos los bucles pueden ser ejecutados en paralelo. Un aspecto a tener en cuenta con este metodo es el fenómeno del "false sharing" en la cache, al haber aumentado la dimensión del array.

```
parallel loop j=1,threads
   do i=1,n
      a_e(i,j) = 0
   enddo
enddo
parallel loop i = 1, n
private int tid = thread\_id()
   do j = 1, m
      k = f(i,j)
      a_e(k,tid) = a_e(k,tid) + expression
   enddo
enddo
parallel loop i=1,n
   do j=1,threads
      a(i) = a(i) + a_e(i,j)
   enddo
enddo
```

3.2 Comprobación de la monotonía de los arrays índice

Una de las mayores dificultades en la paralelización automatica de códigos irregulares es el análisis de subíndices constituidos por una referencia a un array. Aunque este tipo de referencias no necesariamente causan dependencias de datos, sí llevan a decisiones

conservadoras a los test de dependencias. El fragmento de la figura 1(a) ilustra este tipo de patrones.

```
do i = 1, n
 k = ia(i)
 a(k) = ...
end do
```

(a)

```
do i = 1, n
 do j = ia(i), ia(i+1)-1
  a(j) = ...
 end do
end do
```

(b)

Figura 1: Patrones de acceso definidos por un array

Si ia es un vector de permutación, el subíndice $k = ia(i)$ contendrá diferentes valores para diferentes valores de i,y será posible ejecutar en paralelo el bucle. Otro patrón que ocurre con frecuencia es aquél en que los límites del lazo son referencias a arrays, como se muesra en el código de la figura 1(b).

En este último caso si queremos paralelizar el bucle más externo, los intervalos del espacio de iteraciones de j, esto es, $[ia(i), ia(i+1)-1]$ no deben de solaparse para ningún valor de i. Aunque esta condición no se puede garantizar en general, se observa que por naturaleza en los códigos que estamos analizando sí se verifica. Esto se debe al hecho de que las matrices dispersas a menudo se representan en formatos comprimidos (CRS, CCR) donde los valores de los elementos no nulos se almacena en un vector unidimensional y el intervalo anterior se refiere a punteros al primer y último elemento de cada fila (o columna).

El análisis de tales patrones ha sido considerado dificultoso de acometer en tiempo de compilación. Sin embargo usando una combinacion de análisis en tiempo de compilación y tests simples en tiempo de ejecución es posible comprobar que en dicho intervalo los índices son no decrecientes. En el código CHOLESKY, por ejemplo, de forma estática se comprueba que el array índice isu (inicializado en SPARCHOL_GOTO290) es no decreciente. Esto es suficiente para poder ejecutar SPARCHOL_DO1017 en paralelo.

En casos donde el array índice se obtiene en una operación de lectura de datos, es sufiente con chequear si `ia(i+1).ge.ia(i)` para `i = 1,n` para probar que los intervalos no se solapan. Como se mencionó anteriormente, la representación de datos empleada en los códigos que hemos estudiado garantiza que $n \equiv m+1$, donde n es el tamaño del array índice y m es el número de columnas o filas. En la práctica $n << \alpha$, donde α es el número de elementos no cero de la matriz con lo que la sobrecarga introducida para el test será pequeña.

Si el posible, este test podría ser incluido en la parte inicial de la operación de entrada. Sin embargo, puesto que este bucle es esencialmente una reducción en ia, puede también ejecutarse en paralelo. Usando técnicas para manipular bucles con salidas condicionales, tal como las discutidas en la sección 3.5, el tiempo de ejecucion del bucle puede aún decrecer más.

Se comprueba que este patrón de acceso se encuentra en los códigos CHOLESKY y SpLU.

3.3 Comprobando el no solapamiento en rangos de variables de inducción

Gran parte del tiempo consumido en SpLU, implica accesos a arrays mediante variables de inducción que se incrementan de forma condicional. Encontramos patrones similares en DSMC3D. En estos casos un análisis estático del código proporciona las condiciones que deben chequearse en tiempo de ejecución para encontrar que los intervalos son independientes (no solapados). Consideremos el siguiente fragmento de SpLU (DPFAC_DO50):

```
      do 20 i=1,n
         ia_2(i) = ia_1(i)
20    continue
      do 100 k=1,n
         shift=mod(k,2)*lfact+1
         do 50 j=k+1,n
            c1=shift
            do 60 i=ia_1(j),ia_2(j)
               a(shift)=a(i)
               shift=shift+1
60          continue
            c2=shift-1
            if (fill-in) then
               c2=c2+1
               a(shift:shift+positive_inc)= ...
               shift=shift+positive_inc
            endif
            do 95 i=ia_2(j)+1,ia_1(j+1)-1
               a(shift)=a(i)
               shift=shift+1
95          continue
            ia_1(j)=c1
            ia_2(j)=c2
50       continue
         ia(n+1)=shift
100   continue
```

El bucle 100 es el bucle más externo, y se ejecuta para las n columnas de la matriz a. ia_1 e ia_2 son arrays índices. $shift$ es una variable que se usa también como índice de a. Dos hechos son suficientes para probar que el bucle do_50 puede ejecutarse en paralelo: el primero, que para cada j en do_50, el rango de $shift$ no se solapa en los intervalos $[ia\_1(j), ia\_1(j+1)-1]$ para las iteraciones del do_50 ejecutada en otros procesadores (no hay dependencias de flujo ni antidependencias), y segundo, que el rango de $shift$ no debe solaparse en el mismo rango de $shift$ para ejecuciones ejecutadas en otros procesadores (no hay dependencia de salida).

Para probar estos dos puntos debemos determinar que ia_1 e ia_2 son no decrecientes. Vemos que estos arrays índices son reasignados en cada iteración del do_50 complicando aún más el análisis del acceso. Sin embargo es posible determinar estáticamente en tiempo de compilación las condiciones bajo la cuales estos arrays contendrán valores no decrecientes. En primer lugar hay que señalar que la variable de inducción $shift$ es no decreciente. Si ia_1 e ia_2 son inicialmente no decrecientes, por indución esta condición invariante es suficiente para garantizar que permanecerán no decrecientes en la ejecución

completa del bucle más externo (do_100). Por tanto nuestra tarea de probar que estos arrays índices son no decrecientes se reduce a chequerar la condición $ia(i+1).ge.ia(i)$ para $i = 1, n$, en tiempo de ejecución.

Como efecto lateral de este analisis hemos probado que no hay dependencias de salida a lo largo de las iteraciones del do_50. Esto es una consecuencia de que $shift$ sea no decreciente en dicho lazo.

Dado que ia_1 e ia_2 son no decrecientes, el siguiente paso es mostrar que para cada j en el bucle do_50, el rango de $shift$ no se solapa en el intervalo $[ia\_1(j), ia\_1(j+1)-1]$ para iteraciones del do_50 que se ejecutan en procesadores diferentes (estamos ahora probando que no hay anti-dependencias o dependencias de flujo en do_50). Esto se puede conseguir usando un test que compare el límite superior de $shift$ con el límite inferior de ia_1 y el límite inferior de $shift$ con el límite superior de ia_1. La presencia de un incremento condicional de $shift$ complica el análisis. Sin embargo podemos emplear una estimacion del maximo valor de $shift$ determinando una cota superior del espacio de iteraciones del do_50. Basándonos en las condiciones iniciales y en el comportamiento no decreciente de $shift$, $ia\_2(j) \geq ia\_1(j)$. Con estas ideas podemos proponer el siguiente test, en tiempo de ejecución, para chequear si no existe solapamiento entre las lecturas y escrituras de a:

```
min_i=ia(k+1)
max_i=ia(n+1)-1
min_shift=shift
max_shift=shift+(max_i-min_i)+((n-k)*positive_inc)
if(min_shift.gt.max_i .or. max_shift.lt.min_i) then
       parallel=.true.
else
       parallel=.false.
end if
```

Este test se coloca fuera del lazo do_50, y como resultado introduce una pequeña sobrecarga. Cuando la condición se cumple el lazo se puede ejecutar en paralelo. Cuando no, de forma conservadora, se ejecuta en serie. Lo que el test realmente confirma es que las escrituras en a son independienetes de las lecturas en a durante la ejecución del lazo do_50.

3.4 Copy-in y Copy-out

Después de identificar la monotonía en los arrays índices y de probar que las variables de inducción no pertencen a intervalos solapados, a menudo hay que eliminar dependencias de salida y antidependencias *privatizando* variables escalares o vectores que se definen y utilizan en cada iteración[20]. En DSMC3D, COLLMR_DO100, por ejemplo, las variables del bloque /elast/ son privadas. Similarmente existen variables en DPFAC_DO50 de SpLU que requieren privatización. Sin embargo algunas variables contienen valores iniciales que tienen que ser cargados en cada copia privada de los procesadores antes de que comience la ejecución en paralelo. Esto requiere extender las técnicas que Polaris implementa.

A esta operación de privatización se la conoce con el término "copy-in". Asociado a ella encontramos la operación de "copy-out", en la que el valor de las variables privadas correspondiente a la última operación se copia en las variables originales para un uso posterior en la ejecución del programa.

3.5 Lazos con salidas condicionales

En varios casos encontramos lazos en los que existen salidas múltiples que se alcanzan de forma condicional. Tales bulces presentan dificultades en la paralelización debido a los efectos laterales de las iteraciones cuando se ejecutan en paralelo. Sin embargo ciertos tipos de operaciones como las reducciones, pueden ser paralelizadas a pesar de la pesencia de estos efectos laterales. Como se discute en la sección 3.1, se puede hacer esto para operaciones asociativas, tales como las reducciones de histograma si la variable de reducción se privatiza o se expande.

El problema que aparece cuando existen salidas condicionales es la necesidad de cada procesador de descartar el resultado del resto de iteraciones ejecutadas, una vez que la salida tiene lugar. En las máquinas SGI Challenge no hay mecanismos que proporcionen explícitamente tomar una salida anticipada de un bucle paralelo. Para hacer esto, crearemos un lazo nuevo externo que ejecuta una iteración por procesador. En su interior las iteraciones se intercalan de forma que los procesadores ejecutan pequeños trozos del espacio de iteraciones inicial. Una variable global y compartida se usa para almacenar el mínimo número de iteraciones en el que se alcanza la condición de salida. Cualquier iteración con índice mayor que ese mínimo sale anticipadamente del lazo externo. La transformación se ilustra en el siguiente fragmento:

```
        geti = n+1
        stagesize = blocksize * maxproc
        do 100 k = 1, maxproc
           do j = (k-1)*blocksize+1, n, stagesize
              do i = j, j+blocksize-1
                 if ( i > geti ) then goto 100
                 ...
                 if ( a(i) ) then
                    call lock
                       if ( i < geti ) then geti = i
                    call unlock
                 end if
                 ...
              end do
           end do
100     end do
```

La transformación consta de los siguientes pasos: (1) división del espacio de iteraciones en etapas, (2) cada etapa se divide en bloques cada uno de los cuales se asigna a un procesador, (3) cada procesador recorre todas las etapas y en cada una ejecuta el bloque que se le asigna, (4) cuando un procesador encuentra la condición de salida, carga en *geti* el valor de la iteración actual, (5) si un procesador se encuentra ejecutando una iteración mayor que *geti* va a la línea 100 descartando el resto de las iteraciones.

4 Resultados

La tabla 1 presenta una comparación entre los resultados obtenidos por los dos paralelizadores automáticos: PFA y Polaris y aquéllos obtenidos con mejoras introducidas en la sección 3. Los programas fueron ejecutados en 8 procesadores sobre una SGI Challenge con procesdores R4400 a 150MHz.

		Polaris	PFA	Nuevas téc.	Polaris	PFA	Nuevas téc.
Benchmark	T_{seq}	T_{par}	T_{par}	T_{par}	Speedup	Speedup	Speedup
CHOLESKY	4:25	6:42	4:20	3:40	0.66	1.02	1.20
DSMC3D	8:02	6:35	7:53	1:45	1.22	1.02	4.95
EULER	5:03	2:34	4:56		1.97	1.02	
GCCG	12:19	1:27	1:57		8.49	6.32	
LANCZOS	14:28	2:01	1:58		7.17	7.36	
MVPRODUCT	7:57	1:42	1:11		4.68	6.72	
NBFC	6:15	1:15	6:20		5.00	0.99	
SpLU	3:54	15:25	3:44	1:01	0.25	1.04	3.84

Tabla 1: Speedups: PFA, Polaris, y manual

La principal diferencia radica en las reducciones de histograma que son detectadas perfectamente por Polaris.

A continuación se explica código por código cuales son las claves que permiten su paralelización automática y que aspectos hay que modificar manualmente.

NBFC: Este código contiene un lazo principal que consume el 97% del tiempo de ejecución secuencial. Aparecen en él tanto reducciónes escalares como de histograma. En este código la paralelización de las reducciones es suficiente, proporcionando un buen speed-up. Observamos que PFA no es capaz de reconocer dichas reducciones.

CHOLESKY: Las técnicas aplicadas han sido: Reducciones de histograma, lazos con salida condicional, comprobación de la monotonía en arrays índice, y sustitución de generadores de números aleatorios.

Las reducciones de histograma se realizan en el lazo principal cuyo índice es la variable *kk* en UPDATE_DO_3. Este bucle consume un 20% del tiempo de ejecución serie. La transformación, sin embargo no aporta un speed-up significativo. Esto se debe a la sobrecarga adicional de inicializacion y reducción de la expansión que se realiza.

Las técnicas que se apuntaron en la sección 3.2 se han aplicado en los lazos SPARCHOL_DO1015 y SPARCHOL_DO1020. Juntos suponen un 24% del tiempo total serie. Se obtuvo un speed-up de 2.84 en 4 procesadores para estos bucles.

Aunque Polaris y PFA encuentran automáticamente un gran número de lazos paralelos en el código el pequeño speed-up obtenido se debe a la naturaleza del propio algoritmo (supernodo) empleado. En este ejemplo las modificaciones manuales se hacen necesarias.

DSMC3D: Las técnicas aplicadas a este código fueron: Reducciónes de histograma, sustitución de generadores de números aleatorios, comprobación de la monotonía en arrays índice, y paralelizacion de operaciones asociativas en manipulación de listas.

Las reducciones de histograma, discutidas en la sección 3.1, son importantes en algunos lazos: INDEXM_DO300, INDEXM_DO700, COLLMR_DO100, MOVE3_DO#3, y MOVE3_GOTO100. Estos 5 lazos suponen aproximadamente el 84% del tiempo secuencial.

Operaciones asociativas en manipulación de listas

DSMC3D contiene un lazo while en la rutina MOVE3 que supone el 35% de la ejecucion secuencial. Este lazo computa la fase de movimiento, la primera de las tres fases ejecutadas en cada iteración del bucle más externo (avance del tiempo). Las moléculas implicadas en esta fase están almacenadas en listas comprimidas en dos arrays globales. Cuando una mólecula abandona el flujo, se elimina de la lista y se sustituye por la última molécula en la lista. Esto genera dependencias ligadas al lazo. Sin embargo la eliminación de moléculas se puede aplazar hasta que la lista entera ha sido procesada. La molécula actual en se marcaría para ser eliminada más tarde. Después de salir del bucle paralelo, los elementos marcados se eliminan. La operacion de eliminación y sustitución de elementos en la lista es asociativa y por tanto puede ser paralelizada[21].

La combinación de estas técnicas en los lazos mencionados contribuyen a un speedup del 4.95 en la versión paralelizada a mano. Los speed-up proporcionados por Polaris incluyen la redución de histograma del lazo MOVE3_DO_3. PFA sin embargo no implementa ninguna de estas técnicas y por tanto alcanza un speed-up menor que Polaris aunque ambos son bajos.

EULER: Este programa contiene cinco lazos significativos: DFLUX_DO-100, DFLUX_DO-200, EFLUX_DO-100, EFLUX_DO-200, y PSMOO_DO-20 que suponen juntos un 70% de la ejecución secuencial. En estos cinco lazos, las reduciones de histograma suponen la única transformación que es necesaria para ejecutarlos en paralelo. Aunque el speed-up para este código es aproximadamente 2, a nivel de lazo el speed-up de los lazos transformados es bastante bueno (3 sobre 4 procesadores).

La pérdida de eficiencia en la ejecución global se debe a que las transformaciones implican la expansión en variables de reducción con la consiguiente sobrecarga en las fases de inicialización y reducción.

A pesar de estas dificultadas encontradas en la implementación de las reducciones en EULER el resultado de Polaris presenta cierto speed-up con respecto a PFA, ya que éste último ni detecta ni resuelve las reducciones mencionadas.

GCCG: El principal patrón de acceso en GCCG implica expresiones con indirecciones en el lado derecho de la sentencia de asignación. Éstas no plantean problemas especiales de dependencias debido al hecho de que las localizaciones de los arrays se leen pero no se escriben. Las reducciones que encontramos en GCCG son bien escalares o bien de dirección simple, en las cuales la variable de reducción es un elemento del array. Las técnicas acuales de paralelización son capaces de reconocer y transformar dichas expresiones para su ejecución en paralelo.

LANCZOS: LANCZOS presenta una situación similar a la que encontramos en GCCG en lo que respecta a los accesos en las asignaciones: patrones que implican indirecciones en las asignaciones en el miembro derecho, durante el producto matriz-vector. La reortogonalización de la matriz densa supone accesos a arrays mediante los índices del lazo. Como resultado no aparecen dependencias ligadas al bucle que impidan su paralelización. Este hecho se traduce en un buen speed-up para Polaris y PFA.

MVPRODUCT MVPRODUCT genera una matriz densa resultado del producto de dos matrices dispersas. Debido a este hecho, las indirecciones sólo aparecen el la parte derecha de asignacion. Este tipo de indirección no supone ningún problema a la hora de paralelizar, con lo que un buen speed-up se obtiene con PFA y Polaris.

SpLU En la paralelización de SpLU se emplearon las siguientes transformaciones: Comprobación de la monotonía en los arrays índices, comprobación de intervalos no solapados en variables de inducción, y "copy-in" y "copy-out".

El lazo DPFAC_DO_50 en SpLU consume casi el 100% del tiempo de ejecución serie. Como se discutió en la sección 3.2, en primer lugar es necesario probar que el array inidice contiene valore no decrecientes, y a partir de este hecho es posible desarrollar un test en tiempo de ejecución para comprobar la independencia de los intervalos de las variables de inducción. La transformación aplicada al lazo DPFAC_DO_50 implica las operaciones de "copy-in" y "copy-out" de los arrays privatizados: $a, rcptr1$ y $cptr2$.

Ni PFA ni Polaris implementan estas tres técnicas y los correspondientes speed-ups manifiestan este hecho. Polaris en particular, aplica transformaciones a las reducciones de histograma en bucles internos con pocas iteraciones con la consiguiente pérdida en la eficiencia.

5 Conclusiones

En nuestro estudio de los códigos irregulares y dispersos que hemos citado, se determina que las indirecciones en el miembro derecho de las asignaciones no suponen problemas para los paralelizadores automáticos. Hemos identificado así mismo, técnicas que hacen apuntar la posibilidad de paralelizar automáticamente códigos irregulares y dispersos que los paralelizadores automáticos no realizan actualmente.

Referencias

[1] Zahira Ammarguellat and Luddy Harrison *Automatic Recognition of Induction and Recurence Realtions by Abstract Interpretation.*. Proceedings of Sigplan 1990, Yorktown Heights, 25(6):283-295, June 1980

[2] R.Asenjo, M. Ujaldon and E. L. Zapata *Sparse LU Factorization. HPF2. Scope of Activities and Motivating Applications.* High Performance Fortran Forum, version 0.8, November 1994

[3] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM,Philadelphia, PA, 1994

[4] M. Berry, D.Chen, P. Koss, D. Kuck, L.Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Shwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin *The Perfect Club Benchmark: Effective Performance Evaluation of Supercomputers* . Int'l. Journal of Supercomputer Applications, Fall 1989, 3(3):5-40.

[5] G.A. Bird *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Oxford University Press, England, 1994.

[6] William Blume, Ramon Doallo, Rudolf Eigemann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. *Parallel Programming with Polaris.* IEEE Computer, 29(12):78-82, December 1996.

[7] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus. *CHARMM: A program for Macromolecular Energy, Minimization and Dynamics Calculations.* J. Comp. Chem., 4:187-217, 1983

[8] Luiz DeRose, Kyle Gallivan, Bret Marsolf, David Padua, and Stratis Gallopoulos. *FALCON: A MATLAB Interactive Restructuring Compiler* . Proceedings of the 8th Interntional WorkShop on Languages and Compilers for Parallel Computing, Columbus, pages 18.1-18.18, August 1995.

[9] Iain Duff, Nick Gould, John Reid, Jennifer Scott, and Linda Miles. *Harwell Subroutine Library. Technical Report http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html.* Council for the Central Laboratory of the Research Councils, Department for Computation and Information, Advanced Research Computing Division.

[10] Ian Foster, Rob Schreiber, and Paul Havlak. *HPF-2, Scope of Activities and Motivating Applications.* Technical Report CRPC-TR94492, Rice University, November 1994.

[11] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1993

[12] IBM. *IBM Parallel Fortran Language and Library Reference.* March 1988.

[13] Jee Myeong Ku. *The Design of an Efficient and Portable Interface between a Parallelizing Compiler and its Target Machine.* Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. and Dev., 1995.

[14] P.A.W. Lewis, A.S. Goodman, and J.M. Miller. *A Pseudo-Random Number Generator for the System/360.*. System Journal, 8(2):136-146, May 1969.

[15] G. Lueker. *Some Techniques for Solving Recurrences* . Computing Surveys, Vol. 12, No. 4, December 1980.

[16] Bill Pottenger and Rudolf Eigenmann. *Idiom Recognition in the Polaris Parallelizing Compiler.* Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain, pag. 444-448, July 1995.

[17] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M.L. Robinson. *Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator.* Proceedings of Supercomputing'97, Nov. 1994.

[18] Lawrence Rauchwerger and David Padua. *The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Redution Parallelization.* Proceedings of the SIGPLAN'95, Conference on Programming Language Design and Implementation, June 1995.

[19] L.F. Romero adn E.L. Zapata. *Distributions for Sparse Matrix Vector Multiplication.* J. Parallel Computing, 21(4):583-605, April 1995.

[20] Peng Tu and David Padua. *Automatic Array Privatization.*. Proc. 6th Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science, Vol. 768, pages 500-521, August 12-14, 1993.

[21] Guhan Viswanathan and James R. Larus *User-defined Reductions for Efficient Communication in Data-Parallel Languages.* Technical Report 1293, Univ. of Wisconsin-Madison, Computer Sciences Department, August 1996.

[22] Silicon Graphics, Inc. *IRIS Power Fortran Acelerator, User's Guide* . SGI, Inc. 1996