Paralelización de reducciones por afinidad de escritura

Eladio Gutiérrez, Sergio Romero, Oscar Plata, Emilio L. Zapata

Resumen— **Dada la importancia que las operaciones de reducción tienen en el núcleo de aplicaciones numéricas, se han propuesto numerosos métodos para su paralelización. Un grupo de estos métodos se basa en la distribución de los vectores de reducción entre los procesadores permitiendo, entre otras cosas, mejorar la escalabilidad en cuanto a gasto de memoria auxiliar de otras técnicas bien conocidas como las basadas en privatización. Además con ello se explota la localidad de los accesos a los vectores de reducción, algo fundamental para un buen rendimiento en multiprocesadores de memoria distribuida CC-NUMA. Este trabajo se centra en esta clase de métodos ofreciendo un marco general donde encuadrar las posibles soluciones existentes. Aunque su aplicabilidad práctica está limitada, dicho marco permite analizar las propiedades y características de estas técnicas, permitiendo su mejora o el desarrollo de otras posibles.**

Palabras clave— **Reducciones irregulares, afinidad de escritura, máquinas de memoria compartida NUMA.**

I. Introducción

EN el contexto de los códigos regulares, existen numerosas transformaciones orientadas a mejorar el aprovechamiento de la localidad de la jerarquía de memoria y/o beneficiar el paralelismo [1]. Para garantizar la validez de dichas transformaciones es fundamental el análisis de las dependencias.

Algunas de estas transformaciones se basan en la reorganización de los bucles, modificando el orden en que se acceden los vectores con respecto al orden inicial. En este grupo encontramos transformaciones como es el *loop interchange* o *loop tiling* [1]. Otras reestructuran el bucle de manera que las computaciones se realizan sin alterar su orden relativo original. Algunas de ellas, habitualmente implementadas en compiladores, son *loop unrolling*, y *software pipeline*.

En códigos irregulares, al efectuarse los accesos mediante arrays de indirección, se dificulta la posibilidad de aprovechar la localidad aunque tengamos la seguridad de que no existen dependencias inevitables entre las iteraciones del bucle a optimizar ó paralelizar. En este artículo nos centraremos en las operaciones de reducción, operaciones que aparecen frecuentemente ligadas a los accesos irregulares. Por definición una reducción es una operación conmutativa y asociativa que generalmente se aplica sobre los elementos de un vector.

Centrémonos en una reducción múltiple sencilla como la mostrada en la figura 1. Sobre el array de reducción A se realizan dos operaciones de reducción (representadas por $\oplus$) a través de los arrays de indirección $f1$, $f2$. Este hecho causa la aparición de dependencias ligadas al bucle, pero en ausencia de otras dependencias el bucle puede ser paralelizado gracias a la asociatividad y conmutatividad del operador. Este tipo de reducción efectuada sobre un array recibe el nombre de reducción de histograma.

```
do i = 1,N
   Computar ξ1, ξ2
   A(f1(i))=A(f1(i)) ⊕ξ1
   A(f2(i))=A(f2(i)) ⊕ξ2
enddo
```

Fig. 1. Reducción múltiple sencilla.

Nuestro punto de partida es la detección por parte del compilador del lazo de reducción, proporcionando además la información que definimos a continuación:

Definición 1 Sea A un array de reducción en un bucle iterado por i. Definimos el conjunto de expresiones de acceso a A (denotado $Exp_{(i)}(A)$) como aquellas expresiones, en función i, a través de las cuales se realiza una reducción en A. □

Definición 2 Sea A un array de reducción en un bucle iterado por i. Definimos el conjunto de acceso a A como la evaluación de las expresiones de acceso para un valor determinado del índice del lazo. Denotaremos con Acc el conjunto de accesos: $Acc_{(i=k)}(A) = Exp_{(i)}(A)\big|_{i=k}$. El conjunto de accesos es un conjunto formado por valores comprendidos en el rango de definición del array A. □

En caso de tratarse de lazos anidados, estas definiciones se expresarán en función del vector de iteración $\vec{\imath}$ y denotaremos $Exp_{\vec{\imath}}(A)$ y $Acc_{\vec{\imath}}(A)$.

Observamos en el código 1 varias fuentes de localidad:

- Localidad asociada a los arrays de lectura y privatizables. En el código de la figura 1 estos arrays serán los arrays de indirección y aquéllos que intervienen en el cálculo de los valores ξ_1, ξ_2. Denominaremos localidad de lectura a la localidad derivada del acceso a estos vectores.
- Localidad asociada al array de reducción A. Éste es a la vez leído y escrito en cada sentencia de reducción determinando la localidad en dos sentidos:
  1. Localidad en cada iteración (intra-iteración). Ya que existe una secuencia de lecturas y escrituras en posiciones del array de reducción dentro de una misma iteración.
  2. Localidad entre iteraciones (inter-iteración). Ya que diferentes iteraciones acceden a posiciones (en principio desconocidas) del vector de reducción.

Nos referiremos con localidad de escritura a la localidad derivada del acceso al array de reducción. Estos accesos además son el origen de las dependencias a las que da lugar la reducción de histograma.

Si en máquinas con memoria distribuida la distribución de computaciones y datos está orientada a la disminución de las comunicaciones, en máquinas con memoria compartida-distribuida podemos englobar la minimización de las comunicaciones en la maximización de la localidad.

En ambos tipos de arquitecturas, el cómo se asignan los datos a las memorias de los procesadores es lo que se denomina función de distribución. Para un vector $A(1 : ADim)$ esta función de distribución vendrá expresada como una aplicación inyectiva $\Psi : \{1, 2, ... \ ADim\} \rightarrow P$ del conjunto de índices del array al conjunto $P = \{1, 2, ...nThreads\}$ de identificadores de los procesadores.

Podemos plantear la ejecución de un lazo en paralelo intentando aprovechar la localidad de las referencias. Para ello podemos partir de una distribución de los arrays que aparecen en el lazo. Las iteraciones se asignan a los procesadores siguiendo cierta expresión que relaciona la distribución de los datos y la iteración. Por ejemplo si tenemos un lazo iterado en i donde aparece la expresión $A(5 * i + 3)$, podemos asignar la iteración i al procesador propietario del elemento $A(5 * i + 3)$, de esta manera maximizamos el aprovechamiento de la localidad. Denominaremos *afinidad de datos* a esta manera de asignar las computaciones a los procesadores.

La paralelización de un bucle mediante afinidad de datos supone que:

- Se han resuelto los posibles conflictos a los que pudieran dar lugar las dependencias entre iteraciones, por ejemplo mediante primitivas de sincronización o comunicaciones, ...
- Se han distribuido los diferentes arrays acorde a la afinidad de datos elegida. Si se elige un array concreto para establecer la afinidad, la distribución del resto de arrays implicados ha de hacerse buscando la mayor localidad posible.

La paralelización del lazo de la figura 1 mediante afinidad de datos, con el objetivo de maximizar la localidad de los accesos, requiere elegir qué arrays han de ser distribuidos. Al efectuarse las únicas escrituras sobre los arrays de reducción una buena elección es la distribución de éstos. Sin embargo, el problema surge con las dependencias ya que una misma iteración podría escribir sobre posiciones que han sido asignadas a procesadores diferentes. Esta idea nos lleva a considerar cierta afinidad en los accesos a los elementos del array de reducción, afinidad que puede darse ó no en el interior de una misma iteración. En nuestro caso de estudio, es decir, operaciones de reducción, las siguientes definiciones recogen esta idea.

Definición 3 Sean $A(1 : ADim)$ un array de reducción, $P = \{1, 2, ...nThreads\}$ el conjunto de identificadores de los procesadores y $\Psi : \{1, 2, ...ADim\} \rightarrow P$ una función de distribución del array A entre los procesadores. Diremos que existe *afinidad de acceso de escritura* entre dos accesos $A(s)$, $A(t)$ si $\Psi(s) = \Psi(t)$. □

Definición 4 Sean A, P y Ψ los mismos que en la definición 3. Sean $\vec{\imath}$, $\vec{\jmath}$ los vectores de iteración correspondientes a dos iteraciones, y $\Psi(Acc_{\vec{\imath}}(A))$, $\Psi(Acc_{\vec{\jmath}}(A))$ el resultado de aplicar la función de distribución a los conjuntos de acceso. Diremos que existe *afinidad de escritura entre las iteraciones* $\vec{\imath}$, $\vec{\jmath}$ si $\Psi(Acc_{\vec{\imath}}(A)) = \Psi(Acc_{\vec{\jmath}}(A))$. □

Definición 5 Sean A, P y Ψ los mismos que en la definición 3. Sean $\vec{\imath}$, $\vec{\jmath}$ los vectores de iteración correspondientes a dos iteraciones, y $\Psi(Acc_{\vec{\imath}}(A))$, $\Psi(Acc_{\vec{\jmath}}(A))$ el resultado de aplicar la función de distribución a los conjuntos de acceso. Diremos que $\vec{\imath}$, $\vec{\jmath}$ son *incompatibles* desde el punto de vista de la afinidad de escritura si $\Psi(Acc_{\vec{\imath}}(A)) \cap \Psi(Acc_{\vec{\jmath}}(A)) = \emptyset$. □

Para una distribución de datos determinada por bloques contiguos, una iteración cuyos accesos al array de reducción sean afines tendrá mayor localidad que otra que lo sea. Así mismo dos iteraciones afines entre sí mostrarán características de localidad semejantes.

II. Paralelización máxima por afinidad de escritura

Son muchas las técnicas de paralelización de reducciones que encontramos en la literatura [3], [5], [6], [8], [11], dada su importancia en los núcleos de aplicaciones numéricas. En esencia, dichos métodos abordan la paralelización desde tres posibilidades: inserción de secciones críticas, privatización del array de reducción ó partición del array de reducción. Este trabajo se centra en esta última solución ofreciendo un marco donde encuadrar las posibles soluciones existentes.

En esta sección planteamos la paralelización automática de bucles con reducciones de histogramas irregulares mediante afinidad de datos de escritura a través de una transformación del código original. La transformación parte de cierta distribución de los vectores de reducción y trata de distribuir las iteraciones entre los procesadores. Los objetivos deseables, a los que ha de ajustarse esta técnica de paralelización son los que se describen a continuación:

Maximizar el paralelismo. Por un lado se ha de intentar que el número máximo de procesadores que intervienen estén ejecutando código útil y por otro reducir el tiempo empleado en el *overhead* adicional de carga que supone la transformación.

Maximizar la localidad. Esto se consigue mediante una asignación conjunta de datos/computaciones que focalice los accesos de cada procesador en zonas de los arrays. La asignación de los datos vendrán determinadas por los vectores de reducción que son los modificados en la reducción.

Minimizar la replicación de computaciones.
Dado el carácter conmutativo y asociativo de la operación de reducción una transformación *naïve* que elimina la falta de afinidad entre accesos dentro de una iteración es *loop splitting* del lazo. Para la reducción de la figura 1 dicha transformación se muestra en la figura 2.

```
do i = 1,N
   Computar ξ1
   A(f1(i))=A(f1(i)) ⊕ξ1
enddo
do i = 1,N
   Computar ξ1
   A(f2(i))=A(f2(i)) ⊕ξ2
enddo
```

Fig. 2. *Loop splitting* aplicado a una reducción doble.

En el cálculo de ξ_1, ξ_2 de códigos reales participan gran número de expresiones comunes privatizables que acceden a los mismos arrays. Por tanto, dividir el lazo de esta manera implica multiplicar tanto el acceso a estos arrays de lectura o privatizables y multiplicar el tiempo de cálculo de estas expresiones comunes.

Minimizar el overhead de memoria. Las técnicas existentes basadas en la privatización del array de reducción, en mayor o menor medida, realizan copias privadas de ciertos vectores y variables, o bien elaboran estructuras de datos adicionales.

Minimizar el número de barreras. Se ha de buscar un compromiso entre la cantidad de memoria utilizada en las copias privadas y el número de puntos de sincronismo introducidos en el código para solventar los conflictos de escritura originados por las dependencias.

El código de la figura 3 muestra una reducción sobre un array A. Existen $nLoops$ lazos anidados, siendo el vector de iteración $\vec{\imath} = (i_1, i_2, \ldots\ i_{nLoops})$. El espacio de iteraciones, que viene dado por todos los valores que toma $\vec{\imath}$ en la ejecución, lo denotaremos por $\mathcal{S}$. El número de sentencias de reducción es $nInd$, caracterizadas por un conjunto de expresiones de acceso $Exp_{\vec{(i)}}(A) = \{$'$f_1(i_1, i_2, \ldots\ i_{nLoops})$', '$f_2(i_1, i_2, \ldots\ i_{nLoops})$',... '$f_3(i_1, i_2, \ldots\ i_{nLoops})$'$\}$. Supondremos que los arrays de indirección no se modifican en el interior del bucle de reducción.

La definición 4 nos establece una relación entre iteraciones que, en el conjunto de iteraciones, es de equivalencia (es reflexiva, simétrica y transitiva). Ello permite agrupar las iteraciones en clases de equivalencia.

Definición 6 Se define $\mathcal{C}_Q$, la clase de equivalencia de iteraciones con conjunto de acceso $Q \subset P$, como el conjunto de todas las iteraciones cuyo conjunto de acceso sea Q, es decir, $\mathcal{C}_Q = \{\vec{\imath} \in \mathcal{S} \mid \Psi(Acc_{\vec{\imath}}(A)) = Q\}$. □

Definición 7 El conjunto de todas las clases de equivalencia en que podemos particionar el espacio de iteraciones $\mathcal{S}$ constituye el conjunto cociente definido por la relación de afinidad entre iteraciones, y lo representaremos con $\mathcal{S}/\text{aff}$. □

```
REAL    A(1:ADim)
INTEGER f1(1:N1, 1:N2 , ... , 1: NnLoops)
INTEGER f2(1:N1, 1:N2 , ... , 1: NnLoops)
   ...
INTEGER fnInd(1:N1, 1:N2 , ... , 1: NnLoops)

h:  do i1 = 1,N1
      do i2 = 1,N2
        ...
        do inLoops = 1,NnLoops
         Computar ξ1, ξ2, ... ξnInd
         !! i⃗ = (i1, i2, ... inLoops)
         A(f1(i⃗)) = A(f1(i⃗)) ⊕ ξ1
         A(f2(i⃗)) = A(f2(i⃗)) ⊕ ξ2
         ...
         A(fnInd(i⃗)) = A(fnInd(i⃗)) ⊕ ξnInd
        enddo
        ...
      enddo
    enddo
```

Fig. 3. Reducción de histograma con múltiples indirecciones.

En caso de que la distribución de datos Ψ tuviera características de localidad, como podría ser una distribución tipo BLOCK, las iteraciones de una misma clase $\mathcal{C}_Q$ explotarán más localidad de escritura mientras menor sera el número de elementos de Q, es decir, mientras más se concentren los accesos en una región reducida del array de reducción.

Definición 8 Diremos que dos clases $\mathcal{C}_Q$, $\mathcal{C}_R$ son incompatibles desde el punto de vista de la afinidad si dada dos iteraciones $\vec{\imath} \in \mathcal{C}_Q$, $\vec{\jmath} \in \mathcal{C}_R$, se tiene que $\vec{\imath}$, $\vec{\jmath}$ son incompatibles desde el punto de vista de la afinidad. □

Lema 1 Dos clases $\mathcal{C}_Q$, $\mathcal{C}_R$ son incompatibles desde el punto de vista de la afinidad si y sólo si $Q \cap R = \emptyset$.
DEMOSTRACIÓN: Si $\vec{\imath} \in \mathcal{C}_Q$, $\vec{\jmath} \in \mathcal{C}_R$ entonces $\Psi(Acc_{\vec{\imath}}(A)) = Q$, $\Psi(Acc_{\vec{\jmath}}(A)) = R$. Decir que estos conjuntos de accesos son disjuntos, es equivalente a decir que las iteraciones son incompatibles según la definición 5. □

Lema 2 El número máximo de clases no vacías que en podemos agrupar el espacio de iteraciones es a lo sumo:

$$\sum_{p=1}^{r} \binom{nThreads}{p}$$

siendo $r = card(Exp_{\vec{\imath}}(A))$ el número de expresiones de acceso en el intervalo a paralelizar. Se asume la definición $\binom{m}{n} = 0$ para $n > m$.
DEMOSTRACIÓN: Para una iteración concreta, el conjunto $\Psi(Exp_{\vec{\imath}}(A))$ puede contener desde 1 a r elementos, que pertenecen al conjunto de identificadores de los procesadores $\{1, 2, \ldots nThreads\}$. Existirán $\binom{nThreads}{p}$ conjuntos diferentes de p elementos, y como p puede tomar los valores entre 1 y r, el número máximo de conjuntos diferentes es la expresión indicada en el lema. □

Partiendo de la distribución del array de reducción entre el conjunto de procesadores P hemos particionado el espacio de iteraciones en clases de equivalencia por afinidad. Para un lazo como el mostrado en la figura 3, no existe dependencia de datos alguna entre iteraciones que son incompatibles, puesto que al ser la función Ψ inyectiva, dos iteraciones incompatibles están exentas de conflictos de escritura. El siguiente lema recoge este hecho.

Lema 3 No existen dependencias de datos entre dos iteraciones $\vec{\imath}$, $\vec{\jmath}$ incompatibles desde el punto de vista de la afinidad, que pertenezcan a un bucle que efectúe una operación de reducción.
DEMOSTRACIÓN: En un bucle de reducción las dependencias de datos entre sus iteraciones se deben únicamente a las referencias a los vectores de reducción. Puesto que dos iteraciones $\vec{\imath}$, $\vec{\jmath}$ incompatibles acceden a conjuntos disjuntos de elementos del array de reducción, no existirán dependencias de datos entre ellas. □

El hecho expresado en este lema nos permite afirmar que dos iteraciones incompatibles pueden ser ejecutadas correctamente en paralelo, sin necesidad de privatizar el array de reducción, ni introducir puntos de sincronismo, ni replicar computaciones, en una máquina de memoria compartida. Por tanto, la ejecución paralela de iteraciones no–incompatibles, requiere introducir en el código factores que precisamente eran los que queríamos minimizar. Encontrar el máximo paralelismo nos llevará a buscar los conjuntos de clases incompatibles cuyas iteraciones son susceptibles de ser ejecutadas en paralelo, con el mayor número de clases posible.

Definición 9 Denominaremos grafo de no–incompatibilidad, que representaremos como $NIG(\mathcal{S}/\text{aff}) = (N_{NIG}, E_{NIG})$, al grafo no dirigido cuyos vértices son las clases de afinidad, existiendo un arco entre dos clases $\mathcal{C}_Q, \mathcal{C}_R \in \mathcal{S}/\text{aff}$ si las clases $\mathcal{C}_Q$,$\mathcal{C}_R$ no son incompatibles. Con N_{NIG} denotamos el conjunto de vértices de éste grafo, que es $\mathcal{S}/\text{aff}$ y con E_{NIG} el conjunto de arcos. □

El grafo de no–incompatibilidad nos informa de las dependencias en potencia que podrían existir entre las iteraciones para una cierta función de distribución Ψ definida. Encontrar el conjunto de clases incompatibles que nos proporcionará el máximo paralelismo por afinidad de datos de escrituras, se puede hacer con el siguiente procedimiento:

1. Definida la función de distribución $\Psi : \{1, 2, ...ADim\} \to P$ se construyen las clases de equivalencia por afinidad de escritura y $\mathcal{S}/\text{aff}$. Supondremos que las clases no vacías no se incluyen en $\mathcal{S}/\text{aff}$.
2. Se crea el grafo de no–incompatibilidad $NIG(\mathcal{S}/\text{aff})$ con arcos entre dos clases $\mathcal{C}_Q, \mathcal{C}_R$ si éstas no son incompatibles.
3. Aplicar un algoritmo de coloreado a los vértices del grafo de no–incompatibilidad. El resultado del coloreado serán los conjuntos de clases no incompatibles. Denotaremos esta operación $\mathcal{NI}(\mathcal{S}/\text{aff}) = Color(NIG(\mathcal{S}/\text{aff}))$ siendo $\mathcal{NI}$ el conjunto de conjuntos de clases con el mismo color.

Ejemplo 1 El código de la figura 1 corresponde a una reducción doble en un bucle de un sólo nivel. Se considera el conjunto de cuatro procesadores $P = \{1, 2, 3, 4\}$ y una función de distribución Ψ del array A sobre los procesadores. Consideremos que el conjunto de clases de equivalencia es el formado por:

$$\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{3,4\}}\}$$

El grafo de no–incompatibilidad se construye trazando un arco no dirigido entre cada par de clases que no sean incompatibles. Por tanto si $NIG(\mathcal{S}/\text{aff}) = (N_{NIG}, E_{NIG})$ siendo $N_{NIG} = \mathcal{S}/\text{aff}$ el conjunto de vértices y E_{NIG} el conjunto de arcos, tendremos que

$$E_{NIG} = \{(\mathcal{C}_{\{1\}}, \mathcal{C}_{\{1,2\}}), (\mathcal{C}_{\{1\}}, \mathcal{C}_{\{1,3\}}), (\mathcal{C}_{\{2\}}, \mathcal{C}_{\{1,2\}}), (\mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}), (\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{3,4\}})\}.$$

La aplicación de un algoritmo de coloreado de vértices al grafo de no–incompatibilidad asignará el mismo color a vértices que representan clases incompatibles. En el grafo anterior se obtendría:

$$\mathcal{NI}(\mathcal{S}/\text{aff}) = Color(NIG(\mathcal{S}/\text{aff})) = \{\{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3,4\}}\}, \{\mathcal{C}_{\{1,2\}}\}, \{\mathcal{C}_{\{1,3\}}\}\}$$

Ejemplo 2 Para 4 procesadores, $P = \{1, 2, 3, 4\}$, el conjunto de clases $\mathcal{S}/\text{aff}$ tendrá a lo sumo diez clases. En este caso el conjunto cociente es:

$$\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{2,3\}}, \mathcal{C}_{\{2,4\}}, \mathcal{C}_{\{3,4\}}\}$$

En la figura 4 se muestra el grafo de no incompatibilidad correspondiente a esta situación. Se observa que es posible colorear el grafo con 4 colores, proporcionando los siguientes conjuntos de clases incompatibles:

$$\mathcal{NI}(\mathcal{S}/\text{aff}) = Color(NIG(\mathcal{S}/\text{aff})) = \{\{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{4\}}\}, \{\mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{3,4\}}\}, \{\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,2\}}\}, \{\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,2\}}\}\}$$

Hemos introducido la operación de coloreado de los vértices del grafo de no–incompatibilidad como el método para obtener el máximo paralelismo por afinidad de escritura. Las iteraciones que pertenecen a clases incompatibles pueden ser ejecutadas en paralelo sin conflicto alguno. Los conflictos aparecen al lanzar iteraciones no incompatibles, por lo que es necesario situar un punto de sincronismo entre la ejecución paralela de clases a las que se ha asignado colores diferentes. En el pseudocódigo de la figura 5 se muestra cómo podemos ejecutar las diferentes iteraciones introduciendo sólo los puntos de sincronismo necesarios entre las clases que no son incompatibles.

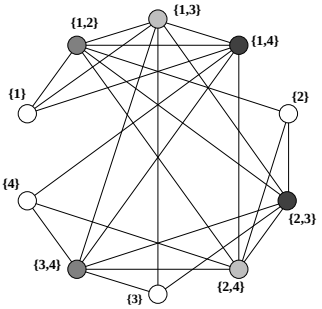

Fig. 4. Coloreado de los vértices del grafo de no incompatibilidad para todas las clases no vacías posibles con 4 procesadores.

Observamos que el número de barreras en el código de la figura 5 dependerá del número de colores en que se haya descompuesto el grafo $NIG(\mathcal{S}/\text{aff})$. El coloreado de los vértices de un grafo no es único, y por tanto existirán numerosas formas de obtener el conjunto $\mathcal{NI}(\mathcal{S}/\text{aff})$. Lo que se persigue es encontrar un conjunto $\mathcal{NI}(\mathcal{S}/\text{aff})$ que reduzca el número de barreras.

```
for color ∈ NI(S/aff)
    forall C ∈ color
        Ejecutar C
    end
    C$barrier
end
```

Fig. 5. Paralelización por afinidad de escritura.

El número de colores mínimo para colorear un grafo dado G se conoce como número cromático del grafo, y lo representaremos con $Crom(G)$. Es esta propiedad del grafo de no–incompatibilidad la que determina el número de puntos de sincronismo necesarios. El número cromático está en relación con el número de arcos de los que es parte un nodo del grafo. Se denomina grado de un vértice al número de arcos que posee dicho vértice y se habla de grado del grafo $Deg(G)$ al máximo de los grados del vértices del grafo. Para un grafo G se tiene que $Crom(G) \leq Deg(G) + 1$. Si el grafo es conexo, $Deg(G) \geq 3$ y existe al menos un par de vértices entre los que no existe arista, esta cota se reduce verificándose que $Crom(G) \leq Deg(G)$.

Si $NIG(\mathcal{S}/\text{aff})$ contiene el máximo número de nodos posibles para un número de procesadores dados $nThreads$, tendremos que $Crom(NIG(\mathcal{S}/\text{aff})) \geq nThreads$ ya que para un conjunto de elementos del array de reducción asignados a un procesador p, los nodos que representen a todas las clases $\mathcal{C}_P$ con $p \in P$ forman un subgrafo completo de al menos $nThreads$ nodos.

El grafo de la figura 4 ha sido coloreado con 4 colores, que es para este grafo su número cromático. El grado del grafo viene dado por el máximo grado de los vértices que es 6, ya que a lo sumo los vértices de este grafo están conectados a 6 aristas.

En el ejemplo 2 se determinó, para 2 indirecciones en las sentencias de reducción y 4 procesadores, el número óptimo de colores, que es 4. Esto permite ejecutar en paralelo las diferentes clases con sólo 4 barreras siguiendo el esquema de ejecución de la figura 5. Sin embargo observamos que de los conjuntos que surgen del coloreado del grafo, sólo uno de ellos contiene tantas clases como procesadores. El resto contiene menos clases que el número de procesadores, por lo que existirán procesadores ociosos en el lazo paralelo que recorre las clases del mismo color. Aunque hemos reducido uno de los factores a minimizar como es el número de puntos de sincronismo, hemos reducido con él uno de los factores a maximizar que es el paralelismo. El objetivo es pues obtener el máximo número de clases en cada color, con el menor número de colores.

III. Aplicabilidad

Como hemos visto la técnica anterior plantea un grave problema de aplicabilidad y es la necesidad de un inspector que elabore las clases incompatibles por afinidad. El coloreado de grafos es un procedimiento de elevada carga algorítmica. Encontrar el número cromático constituye un problema NP-completo, y en la práctica el coloreado se realiza con algoritmos que no proporcionan el coloreado óptimo. Además al determinar las clases no incompatibles el algoritmo debería de tener en cuenta no sólo encontrar el mínimo número de colores, sino también maximizar el número de clases por cada color y de forma balanceada.

La figura 6 muestra los resultados de colorear el grafo de no–incompatibilidad para diferente número de procesadores considerando un lazo con dos indirecciones. El algoritmo de coloreado aplicado es el de búsqueda secuencial (o *greedy*) [2], testeado sobre diferentes ordenaciones iniciales de los vértices del grafo. Un hecho interesante es que el algoritmo empleado es óptimo cuando el número de procesadores es potencia de dos. En este caso como el número de colores es igual al de procesadores y por tanto el óptimo. Como vemos el tiempo requerido para colorear el grafo crece con $\mathcal{O}(N^4)$ siendo N el número de procesadores.

En la práctica para que el método sea aplicable

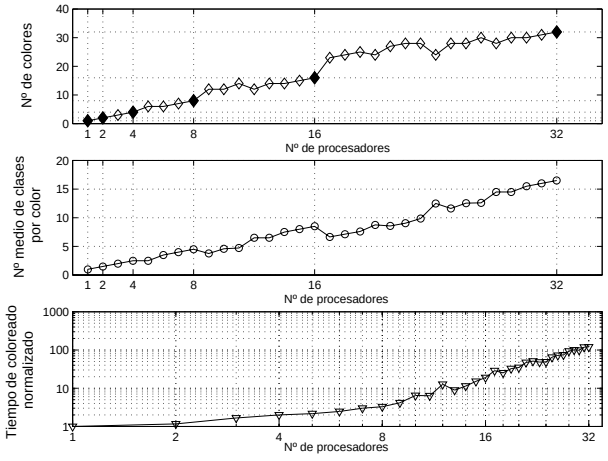

Fig. 6. Cálculo del grafo de no–incompatibilidad para diferente número de procesadores.

debe simplificarse la elaboración de las clases, para conseguir una etapa de inspección computacionalmente mucho menos costosa. Podemos mencionar dos opciones que encontramos en la literatura:

DWA–LIP: Esta técnica (acrónimo de *Data Write Affinity with Loop Index Prefetching*) realiza una restricción de la relación de afinidad con la cual es posible realizar un rápido recorrido de las clases incompatibles [3], [4], [5]. Por contra dicha simplificación se realiza a costa de cierta pérdida de paralelismo.

LocalWrite: Clasifica las clases $\mathcal{C}_Q$ en dos grupos: aquéllas con $Cardinal(Q) = 1$ (iteraciones locales) y aquéllas con $Cardinal(Q) \geq 2$ (fronteras). Las locales se ejecutan paralelamente, ya que les corresponderían colores diferentes. Las fronteras, tras un *loop splitting*, quedan convertidas en iteraciones con una única indirección, aplicándosele el mismo método que a las locales pero independientemente a cada lazo que surge del *spplitting* [6], [7]. Consecuentemente, éstas últimas generarán una replicación de computaciones.

IV. Resultados Experimentales

En la figura 7 se muestran las gráficas de aceleración del método analizado y las implementaciones DWA-LIP y LocalWrite, aplicados a un código de simulación de tejidos. También se incluyen resultados de un método basado en la privatización de los arrays de reducción (Array Expansion [11]). El código utiliza una discretización irregular del tejido en elementos finitos triangulares numéricamente modelados por una ecuación diferencial ordinaria [9], [10]. Para avanzar en el tiempo se emplea un método de integración implícito. Utilizando una expansión de primer orden de Taylor sobre dichas ecuaciones se obtiene un sistema lineal de ecuaciones algebraicas.

Dentro del lazo de tiempo se puede encontrar una reducción irregular sobre 3 indirecciones que recorre los triángulos. Las reducciones se efectúan sobre un vector tridimensional que representa la magnitud que se computa (fuerzas) y sobre una matriz dispersa, cuadrada y simétrica derivada del método implícito. El número de elementos del vector de reducción y el orden de la matriz es igual al número de nodos. El grado de dispersión de la matriz es 0.015%. Las simulaciones se han realizado sobre una malla irregular de 218272 nodos, 653100 aristas y 434829 triángulos.

Los resultados se han obtenido en un SGI Origin 2000 con procesadores MIPS R12000 a 400Mhz y 8Mbytes de memoria caché secundaria. Como se puede observar, las curvas de aceleración de DWA-LIP y LocalWrite son bastantes similares a la obtenida por coloreado del grafo de no–incompatibilidad, lo que muestra que estas dos aproximaciones en la práctica proporcionan resultados aceptables.

En cuanto a los métodos basados en privatización, cabría mencionar que dada la falta de escalabilidad en el uso de memoria y el gran tamaño de los arrays de reducción, uno de los cuales es una matriz, se hace imposible la ejecución del código testeado con más de 4 procesadores. Se pone de manifiesto así mismo que este último tipo de método no explota localidad, de ahí que se obtenga menor aceleración.

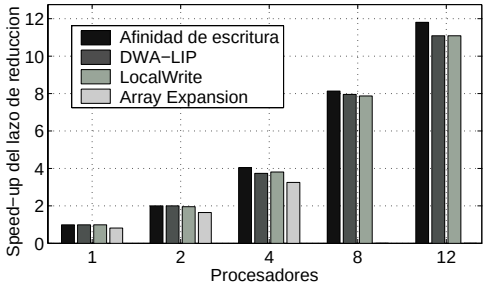

Fig. 7. Speed-up del lazo de reducción paralelo en el código de simulación de tejidos.

V. Conclusiones

En este artículo se ha presentado un marco general donde encuadrar un grupo importante de métodos de paralelización de reducciones: aquéllos basados en la distribución de los arrays de reducción entre los procesadores. Dicho marco se basa en la clasificación de las iteraciones según su afinidad de escritura, lo que permite explotar la localidad, algo primordial en la paralelización de reducciones irregulares sobre multiprocesadores de memoria distribuida NUMA.

Referencias

[1] D. F. Bacon, S. L. Grahan, and O. J. Sharp. Compiler transformation for high-performance computing. *ACM Compiler Surveys*, December 1994.

[2] A. Gibbons. Algorithmic graph theory. Cambridge University Press, 1999.

[3] E. Gutiérrez, O. Plata, and E.L. Zapata. A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors. *14th ACM Int'l. Conf. on Supercomputing (ICS'00)*, pp. 78–87, Santa Fe, NM, USA,May 2000.

[4] Eladio Gutiérrez Carrasco. *Tesis Doctoral: Paralelización Automática de Reducciones.* Universidad de Málaga, 2001

[5] E. Gutiérrez, O. Plata and E.L. Zapata. Improving Parallel Irregular Reductions Using Partial Array Expansion. *The IEEE/ACM Int'l. Conf. for High Performance Computing and Communications (SC2001)*, Denver, CO, USA, November 10-16, 2001.

[6] H. Han and C.-W. Tseng. Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes. *11th Workshop on Languages and Compilers for Parallel Computing*, pp. 181–196, Chapel Hill, NC, August 1998.

[7] H. Han and C.-W. Tseng. A Comparison of Parallelization Techniques for Irregular Reductions. *15th IEEE Int'l. Parallel and Distributed Processing Symp.* , San Francisco, CA, USA, April 2001.

[8] Y. Lin and D. Padua. On the Automatic Parallelization of Sparse and Irregular Fortran Programs. *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, Pittsburgh, PA, May 1998.

[9] S. Romero, L.F. Romero and E.L. Zapata. Fast Cloth Simulation with Parallel Computers. *6th Int'l. Euro-Par Conference (Euro-Par'2000).* Munich, Germany, August 29 - September 1, 2000, pp. 491-499.

[10] S. Romero, L.F. Romero and E.L. Zapata. Approaching Real-Time Cloth Simulation Using Parallelism, *16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation.* Lausanne, Switzerland, August 21-25, 2000.

[11] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. *14th ACM Int'l. Conf. on Supercomputing*, pp. 66–77, Santa Fe, NM, USA, May 2000.