Explotando jerarquías de memoria distribuida/compartida con Hitmap

Ana Moretón-Fernández[1] ,Arturo González-Escribano [2] y Diego R. Llanos Ferraris [3]

Resumen—
Actualmente los clústers de computadoras que se utilizan para computación de alto rendimiento se construyen interconectando máquinas de memoria compartida. Como modelo de programación común para este tipo de clústers se puede usar el paradigma del paso de mensajes, lanzando tantos procesos como núcleos disponibles tengamos. Sin embargo, esta forma de programación no es eficiente. Para conseguir explotar eficientemente estos sistemas jerárquicos es necesario una combinación de diferentes modelos de programación y herramientas, adecuadas para los diferentes niveles de la plataforma de ejecución.

Este trabajo presenta un método que facilita la programación para entornos que combinan memoria distribuida y compartida. En nuestro modelo el esfuerzo de desarrollo de la coordinación en el nivel de memoria distribuida se simplifica usando la biblioteca Hitmap. Mostraremos como integrar Hitmap con modelos de programación para memoria compartida y con herramientas automáticas que paralelizan y optimizan código secuencial. Esta nueva combinación permitirá explotar las técnicas más apropiadas para cada modelo además de facilitar la generación de programas paralelos multinivel que adaptan automáticamente su estructura de comunicaciones y sincronización a la máquina donde se ejecuta. Los resultados muestran como la propuesta del trabajo mejora los mejores resultados obtenidos con programas de referencia optimizados manualmente usando MPI u OpenMP.

Palabras clave— **Programación paralela, clusters híbridos, heterogeneidad**

I. Introducción

Actualmente los sistemas de cómputo que pueden manejar paralelismo son cada vez más heterogéneos, mezclando dispositivos con diferentes capacidades en el contexto de clústers híbridos o mezclando subsistemas de memoria compartida y distribuida jerárquicamente. Además, se tiende a encontrar soluciones más diversas y complejas para las aplicaciones paralelas, explotando varios niveles de paralelismo con diferentes estrategias de paralelización. Esto lleva a un aumento en el interés que suscitan las aplicaciones capaces de adaptar automáticamente su estructura a cualquier sistema dado.

Muchos modelos y herramientas de programación paralela se han ido proponiendo y asentando, generando diferentes escenarios de abstracción sobre los sistemas reales. Las bibliotecas que usan el estándar MPI se han convertido en la opción habitual para sistemas de memoria distribuida. Modelos como OpenMP, Cilk o TBBs se utilizan habitualmente para simplificar el manejo de los hilos de ejecución y el acceso a memoria compartida global. Los lenguajes de tipo PGAS (Partitioned Global Address Space) como Chapel, X10 o UPC presentan un punto intermedio de abstracción para la gestión de espacios de memoria local o global. Sin embargo, el programador todavía debe tomar decisiones no relacionadas con los algoritmos de programación, pero que son decisiones clave para obtener programas eficientes. Esto incluye por ejemplo decisiones sobre la partición y la localidad de datos, teniendo en cuenta cómo afectan a los costes de sincronización/comunicación, la selección del grano de paralelismo a usar, o detalles sobre la planificación, mapping o layout. Además, muchas de estas decisiones pueden cambiar dependiendo de las características de la máquina donde se este ejecutando o incluso cuando el tamaño de los datos del problema es modificado.

Nuestra aproximación se construye sobre la biblioteca Hitmap [1]. Hitmap realiza eficientemente la distribución de datos y las comunicaciones asociadas expresadas de una forma abstracta. Las decisiones de mapping son automáticamente guiadas por la topología virtual, la partición de datos y las políticas de distribución. Estas políticas están encapsuladas en módulos plug-in.

Aunque los modelos de programación para memoria distribuida son válidos también para memoria compartida, los algoritmos de programación, estrategias de paralelismo o técnicas de mapping de datos no serán las más eficientes posibles. En este trabajo veremos como integrar Hitmap con modelos de programación de memoria compartida para generar programas multinivel que exploten entornos híbridos. El modelo de comunicación de Hitmap será usado para coordinar el mapping y la sincronización en el nivel de memoria distribuida dejando así un proceso para un nodo o un subconjunto de nodos. Así, las estrategias para programación en memoria compartida, los modelos de programación o la utilización de herramientas automáticas para la generación o transformación de código podrán ser usadas sobre ese único proceso y explotar mejor la memoria compartida local.

Para mostrar las ventajas del uso de esta propuesta usaremos tres benchmarks que implican diferentes ventajas e inconvenientes a la hora de ser programados con modelos de memoria distribuida o compartida: Jacobi (programa basado en la sincronización entre vecinos), Gauss-Seidel (usando el paradigma de pipeline) y una combinación multinivel de dos algoritmos para la multiplicación de matrices. Nuestros resultados mostrarán que el uso de nuestra propuesta

[1]Dept. Informática, Universidad de Valladolid, e-mail: ana.moreton@alumnos.uva.es
[2]Dept. Informática, Universidad de Valladolid, e-mail: arturo@infor.uva.es
[3]Dept. Informática, Universidad de Valladolid, e-mail: diego@infor.uva.es

consigue los mejores resultados comparados con programas de referencia escritos en MPI o OpenMP y manualmente optimizados.

El resto del artículo sigue la siguiente estructura: La sección II describe trabajo relacionado. La sección III revisa las principales características de Hitmap. La sección IV describe las características de los tres casos de estudio. La sección V describe nuestra propuesta y cómo debe ser integrada en un programa Hitmap. La sección VI analiza los resultados de nuestra experimentación y por último la sección VII presenta las conclusiones y el trabajo futuro.

II. Trabajo previo

Existen muchas propuestas de lenguajes de programación que soportan paralelismo y operaciones paralelas con arrays que hacen más transparentes algunas decisiones al programador (e.g. HPF, ZPL, CAF, UPC, Chapel, X10, etc.). La mayoría de ellos están basados en sofisticadas transformaciones del código en tiempo de compilación. Estos lenguajes presentan un limitado juego de funcionalidades de mapping y además presentan problemas para generar códigos que adapten su estructura para plataformas híbridas de ejecución o cuando se intenta explotar composiciones jerárquicas de paralelismo con niveles de granularidad arbitrarios.

Los modelos basados en PGAS (Partitioned Global Address Space) presentan una abstracción para trabajar con sistemas de memoria distribuida y compartida. Estos modelos no promueven las diferentes técnicas de paralelismo y optimización necesarias cuando los procesos y subprocesos se despliegan jerárquicamente en un entorno híbrido. En [1] se ha visto que Hitmap reduce la complejidad de programación cuando se le compara con UPC [2] mientras mantiene idéntica eficiencia. Otro ejemplo de PGAS es Chapel. Chapel propone una interfaz en la que se pueden añadir módulos de mapping para el sistema [3]. El sistema de run-time que Hitmap presenta se trata de una aproximación más generalizada con la posibilidad de explotar mappings jerárquicos para cada tipo de dominio usando el mismo interfaz. Además, Hitmap mejora las capacidades de otras bibliotecas jerárquicas tales como HTA [4].

Parray [5] es un modelo con una interfaz de programación flexible basada en tipos de array que pueden ser explotados en diferentes niveles jerárquicos de paralelismo sobre sistemas hetereogeneos. En su propuesta la gestión de dominios de datos densos y con stride no están unificados y las operaciones en el dominio de datos no son transparentes. Además, el programador todavía debe tomar decisiones acerca de la granularidad y sincronización en los diferentes niveles. Varios modificadores o tipos deben también ser usados para distribuir y mapear los datos en el nivel apropiado. Nuestra propuesta introduce una interfaz más genérica, portable y transparente.

El polyhedral model proporciona unas técnicas para realizar transformaciones automáticas a partir de un código fuente secuencial [6]. Algunas técnicas de paralelización y optimización de código pueden ser aplicadas a trozos de código secuencial para conseguir programas eficientes tanto en sistemas de memoria compartida como de distribuida. El polyhedral model es solo aplicable a códigos basados en bucles estáticos con expresiones afines. Además, esta técnica tampoco soporta composiciones recursivas de paralelismo. Sin embargo, es posible usar estas técnicas en el contexto de frameworks más genéricos y jerárquicos como Hitmap.

III. Hitmap: Descripción y estructura

Hitmap [1] es una biblioteca de funciones diseñada para el reparto y mapping jerárquico de estructuras de datos densas. También ha sido extendida para soportar estructuras de datos dispersas como matrices dispersas, o grafos [7] usando la misma metodología e interfaz. Hitmap está basado en el modelo de programación distribuido SPMD usando abstracciones para la declaración de estructuras de datos. Hitmap también automatiza la partición, repartición y comunicación de los diferentes niveles jerárquicos mientras mantiene un buen rendimiento.

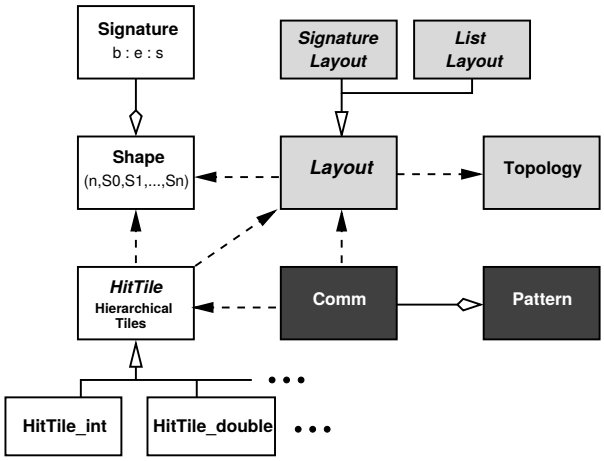

Fig. 1. Arquitectura de Hitmap. Las cajas blancas representan las clases que gestionan los dominios y las estructuras de datos. Las cajas gris claro representan las clases que aplican la partición y el mapping del dominio. Las cajas gris oscuro representan las clases que generan los patrones de comunicación adaptables y reutilizables.

A. Arquitectura de Hitmap

Hitmap esta diseñado como una biblioteca de funciones, con una aproximación de orientación a objetos, aunque esté programada en C. Las clases están implementadas como estructuras de C con funciones asociadas. La figura 1 muestra un diagrama de la arquitectura de la biblioteca. El diagrama muestra las clases que dan soporte a estructuras de datos con dominios densos o con stride. Además, Hitmap ha sido extendida para soportar grafos y matrices dispersas [7], usando la misma metodología e interfaz.

Un objeto de la clase *Shape* representa un subespacio de índices de un array definido como un paralelotopo rectangular de n-dimensiones. Sus límites serán determinados por objetos *Signature*. Cada *Signature*

es una tupla de tres números enteros $S = (b, e, s)$ (begin, end y stride) que representa los índices en cada uno de los ejes del dominio. Signatures con $s \neq 1$ definen un espacio regular de índices en un eje no contiguo. La cardinalidad de una signatura es $|S| = \lceil (e-b)/s \rceil$. Los elementos *begin* y *stride* de una signatura representan los coeficientes de una función lineal $f_S(x) = sx + b$. Aplicando la función inversa $f_S^{-1}(x)$ a los índices de la signatura obtendremos una dominio compacto contiguo que comienza en $\vec{0}$. Así, los índices del dominio representado por el Shape son equivalentes (aplicando la función inversa definida por las signaturas) a los índices del dominio de un array tradicional. Otras subclases de *Shape* se usan para dominios dispersos.

Un *Tile* mapea los datos actuales sobre el espacio definido por el shape. Internamente los tiles reservan la memoria necesaria en bloques de forma contigua. Además, es posible realizar subselecciones jerárquicas de un tile usando la información proporcionada por las signaturas para localizar y acceder a los datos eficientemente. Las subselecciones de tiles pueden reservar su propio espacio de memoria.

Las clases abstractas *Topology* y *Layout* son interfaces para dos diferentes sistemas de plug-ins. Estos plug-ins serán seleccionados por su nombre en la invocación del método constructor. Al estar implementado de esta forma el programador podrá desarrollar nuevos plug-ins para diferentes aplicaciones, aunque Hitmap ya tiene implementadas varias técnicas de particionamiento y balanceo de carga propias. Los plug-ins de topology implementan funcionalidades para organizar los procesadores físicos en una topología virtual y así construir las relaciones de vecindad entre procesadores sobre dicha topología virtual. Los plug-ins de Layout implementan métodos para distribuir un shape entre los diferentes procesadores de la topología virtual. El objeto Layout resultante contiene información acerca de la parte local del dominio y la relación de vecindad del proceso con otros. Este método de implementación a través de una topología virtual hace que los procesadores que queden inactivos sean transparentes para el programador y este no tenga que tenerlos en cuenta, ya que no estarán incluidos en la topología virtual.

Por último, Hitmap también posee dos clases relacionadas con la comunicación entre procesos. La clase *Communication* representa la información para la comunicación de tiles entre procesos. Esta clase nos da métodos constructores para construir diferentes esquemas en función de los dominios de los tiles, la información de los objetos layout, y las reglas de vecindad si fuesen necesarias. Esta clase encapsula comunicaciones punto a punto, intercambios entre vecinos, desplazamientos de datos a lo largo de la topología virtual, comunicaciones colectivas, etc. La biblioteca ha sido construida sobre el estándar MPI para que así pueda ser usada en diferentes arquitecturas. Internamente Hitmap explota varias técnicas de MPI que mejoran el rendimiento. Algunas de estas técnicas son el uso de tipos derivados de MPI y comunicaciones asíncronas. Los objetos Communication se pueden componer en Patterns reutilizables, es decir, se pueden realizar varias comunicaciones con una simple llamada.

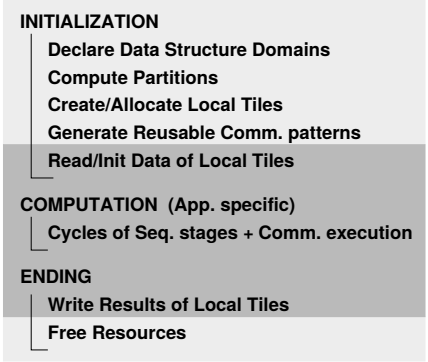

Fig. 2. Fases de una aplicación de cálculo numérico típica usando Hitmap. Los estados gris oscuro contienen código secuencial que puede ser jerárquicamente paralelizado.

B. Metodología de uso de Hitmap

En esta sección describiremos cómo un programa típico se implementa con Hitmap. Esta metodología propone unas guías claras y comunes para estructurar los códigos. La figura 2 muestra los fases de una aplicación científica típica programada con Hitmap.

El programador diseña el código paralelo usando las partes locales de la estructura de datos abstracta y diseñando el intercambio de información entre los nodos de la topología virtual. El primer paso es seleccionar el tipo de topología virtual apropiada para el algoritmo paralelo a desarrollar. Por ejemplo, podríamos escoger una topología rectangular donde los procesadores tienen dos índices (x,y).

El segundo paso en el diseño es la definición de los dominios. Todos los procesos lógicos declaran los shapes de la estructura de datos global. Seguidamente se usarán los objetos de layout para la partición y reparto del dominio de la estructura global entre los diferentes procesadores de la topología virtual. Los dominios locales obtenidos de los layout pueden ser expandidos haciendo que se solapen con los dominios de otros procesadores generando así *ghost zones*. Estas porciones de espacio serán compartidas pero no sincronizadas entre dos procesadores. Después de conocer el dominio de cada procesador, éste reservará el espacio de memoria necesario y el programador podrá empezar a usar los datos locales del subdominio tanto en coordenadas globales como en coordenadas locales. Esto ayuda a implementar la computación secuencial de los tiles.

Por último el programador deberá decidir las comunicaciones necesarias para la sincronización de los datos entre las fases de computación. Estas serán impuestas por el algoritmo paralelo que se desee desarrollar. La estructura de comunicación será creada a partir de los objetos de comunicación. Para construir los objetos de comunicación será necesario obtener el dominio donde recibir o enviar datos y la información

obtenida del objeto layout acerca de los vecinos y el tile local. Por ejemplo para las *ghost zones* el shape exacto donde se deben comunicar los datos puede ser obtenido automáticamente usando la funcionalidad de intersección que ofrece la biblioteca. Los objetos de comunicación tienen que ser creados al inicio del programa y podrán ser llamados cuándo y tantas veces como sea necesario.

Una vez finalizadas las fases de implementación obtendremos un código genérico que se adapta en tiempo de ejecución a: (a) los dominios globales declarados, (b) la información acerca de la topología física y (c) los plug-ins de partición o layout seleccionados. Por ejemplo el código de la figura 4 muestra la implementación en Hitmap de la multiplicación de matrices presentado en el caso de estudio 1.2 de la figura 3. En este código la función secuencial usada para el cálculo de la multiplicación en las matrices locales es una implementación del algoritmo presentado en el caso 1.1 de la figura 3.

IV. Casos de estudio para la programación sobre sistemas híbridos de memoria distribuida y compartida

Para mostrar las diferentes opciones o problemas que surgen en la integración de modelos de comunicación de memoria compartida con memoria distribuida se han propuesto tres diferentes clases de aplicaciones. En la figura 3 podemos ver los algoritmos de las aplicaciones escogidas.

A. Multiplicación de matrices: Algoritmos multinivel

En la figura 3 (izquierda) se muestran los dos algoritmos para la multiplicación de matrices usados en este artículo. El primero es una aproximación clásica secuencial basada en 3 bucles anidados. Los bucles exteriores de este algoritmo pueden ser paralelizados sin dependencias de escritura o condiciones de carrera. La paralelización de este método es apropiada para entornos de memoria compartida. Sin embargo, para entornos de memoria distribuida este algoritmo obliga a reservar grandes espacios de memoria y a realizar comunicaciones innecesarias.

Por otro lado, tenemos el algoritmo de Cannon, que trabaja con una partición de las matrices en k x k bloques, necesitando localmente en cada proceso solo uno de los bloques de cada matriz durante cada iteración. Utiliza un patrón de comunicaciones de desplazamiento circular (*circular shift*) para mover los datos entre los procesos (ver eg. [8]). Este algoritmo es apropiado en entornos de memoria distribuida pero es más complejo de programar e introducir pérdidas de rendimiento por el movimiento de datos en entornos de memoria compartida. Por ello, la mejor aproximación para sistemas jerárquicos híbridos será una combinación de ambos algoritmos.

B. Jacobi: Sincronización entre vecinos

El segundo caso de estudio se trata de la solución de una ecuación diferencial parcial (*PDE*) usando el método iterativo de Jacobi. El ejemplo escogido calcula la ecuación de transferencia de calor en un espacio discretizado de dos dimensiones. Será implementado como un Cellular Automata. En cada iteración cada posición de la matriz o célula será actualizada con los valores previos de los cuatro vecinos. Cuando usamos modelos de programación para memoria distribuida es necesario en cada iteración una comunicación de los nuevos valores de los bordes de la zona de la matriz asignada a cada proceso. Para guardar los valores previos mientras se realiza la computación de los datos será necesario una copia de la matriz original. La comunicación entre procesos se realiza con un patrón de comunicaciones *neighbor synchronization* en una topología virtual de dos dimensiones. Los datos que necesita cada proceso son los bordes de la matriz local de los procesos vecinos. La matriz local será expandida con una columna o fila más en cada dirección y sentido para poder almacenar esos elementos que provienen de los vecinos. Estas filas o columnas extras se denominan *ghost zones*. Si usásemos solo modelos de programación para memoria compartida el espacio de memoria extra no sería necesario. En cambio, el programa debe incluir un sistema de sincronización para asegurar que cada hilo no está utilizando datos que otro hilo está actualizando, o datos que no se encuentran actualizados en las zonas compartidas de los bordes.

C. Gauss-Seidel: Wave-front pipelining

El último caso de estudio computa la misma ecuación de calor, pero esta vez usando un método iterativo de Gauss-Seidel. En este método la convergencia se acelera utilizando valores ya computados durante la iteración actual. Este método no utiliza una copia para guardar los valores antiguos. Por tanto, cuando se va a actualizar una celda se usan los valores ya actualizados de los vecinos de arriba y de la izquierda de la matriz. Los valores de los vecinos de abajo y de la derecha serán los computados previamente en la iteración anterior. Por ello, el orden en el que se recorre la matriz en este caso es importante. La implementación secuencial se realiza con unos bucles internos que arrastran dependencias en cada iteración. Esto deriva en una aplicación paralela de tipo *wave-front*.

En modelos de programación de paso de mensajes la solución será similar al ejemplo de Jacobi pero usando dos patrones de comunicación. El primero recibirá los datos en las ghost zones de los vecinos de arriba y de la izquierda antes de la computación de los datos locales. El segundo se realizará después de la computación de lo datos locales. Recibirá los datos de los vecinos de abajo y de la derecha y realizará todos los envíos. Estos patrones generan un pipeline controlado por el flujo de datos.

Sin embargo, los modelos de programación para sistemas de memoria compartida no dan un buen soporte para aplicaciones de tipo pipeline. Hay dos aproximaciones que pueden ser usadas. La primera

```
** Case 1.1: MM CLASSICAL SEQ. ALGORITHM
1. For each i,j in C.domain
      For each k in A.columns
         C[i][j] += A[i][k] * B[k][j]


** Case 1.2: MM CANNON'S ALGORITHM
1. Split A,B,C in k x k blocks
   AA=Blocking(A), BB=Blocking(B), CC=Blocking(C)
2. Initial data alignment:
2.1. For each i in AA.rows
     Circular shift: Move AAi,j to AAi,j-i
2.2. For each j in BB.columns
     Circular shift: Move BBi,j to BBi-j,i
3. For s = 1 .. k
   3.1. CCi,j = CCi,j + AAi,j * BBi,j
   3.2. For each i in AA.rows
        Circular shift: Move AAi,j to AAi,j-1
   3.3. For each j in BB.columns
        Circular shift: Move BBi,j to BBi-1,j
```

```
** Case 2: JACOBI SOLVER
1. While not converge and iterations < limit
  1.1. For each i,j in Mat.domain
       Copy[i][j] = Mat[i][j]
  1.2. For each i,j in Mat.domain
       Mat[i][j] = ( Copy[i-1][j] + Copy[i+1][j] +
                     Copy[i][j-1] + Copy[i][j+1] ) / 4;


** Case 3: GAUSS SEIDEL SOLVER
1. While not converge and iterations < limit
   For i = 0 .. Mat.rows
     For j = 0 .. Mat.columns
       Mat[i][j] = ( Mat[i-1][j] + Mat[i+1][j] +
                     Mat[i][j-1] + Mat[i][j+1] ) / 4;
```

Fig. 3. Algoritmos de los tres casos de estudio

se basa en implementar directamente un sistema de sincronización que siga el orden impuesto por las dependencias de datos. Esto implica programar un patrón complejo con variables de bloqueo, y con decisiones sobre topología y partición de datos en función del número de threads. El segundo método consiste en realizar complejas transformaciones del espacio de iteraciones de los bucles secuenciales para generan un nuevo orden de ejecución en el que la aplicación pueda ser paralelizada en iteraciones sin dependencias de datos.

V. Integrando memoria compartida en Hitmap

A. Modelo de programación multinivel

Hitmap propone un modelo de programación multinivel que permite una coordinación eficiente y abstracta da tareas, integrando funciones de partición de datos, mapping y comunicaciones para memoria distribuida. Las estructuras de datos, sus tamaños y los límites de los subespacios locales se adaptan automáticamente en función de la información de los objetos de layout, que se han construído internamente teniendo en cuenta los detalles de la plataforma donde se ejecuta. En este modelo el código secuencial que ejecuta cada tarea local está claramente diferenciado y aislado. La ejecución de la parte secuencial ocurre entre las llamadas a funciones de Hitmap que ejecutan patrones de comunicación.

Así, el código dentro de cada tarea local puede ser paralelizado con el paradigma de memoria compartida de una manera sistemática. Esta paralelización puede ser realizada manualmente por un programador experimentado usando modelos de programación de memoria compartida. Sin embargo, también pueden ser aplicadas sobre las zonas secuenciales herramientas que generan automáticamente código paralelo en memoria compartida. Por ejemplo la figura 4 muestra una implementación del algoritmo de Cannon en Hitmap que llama a una función secuencial que realiza una multiplicación de matrices usando el algoritmo clásico. Paralelizando este trozo de código con el paradigma de memoria compartida obtendremos un programa paralelo con dos niveles de paralelismo. Este programa podrá ser ejecutado en plataformas híbridas donde haya una mezcla de memoria compartida y distribuida.

B. Metodología para usar OpenMP

En esta sección mostraremos, presentando finalmente ejemplos, la forma de utilizar OpenMP para explotar el paradigma de memoria compartida en un programa Hitmap. El primer paso será identificar la parte secuencial del código. Esto se puede hacer manualmente o automáticamente ya que siempre estas secciones se encuentran siempre entre llamadas a las funciones de Hitmap que ejecutan patrones de comunicación. Estas son *hit_comDo*, *hit_comStart*, *hit_comEnd*, *hit_patternDo*, *hit_patternStart*, *hit_patternEnd*.

Una primera aproximación para la introducción de directivas OpenMP en las partes secuenciales puede ser utilizar las primitivas sincronizadas *omp parallel for* para los bucles de alto coste computacional. Sin embargo, esta solución cuando se da en bucles interiores dentro de un anidamiento de bucles, puede ser ineficiente. En cada iteración del bucle exterior los threads deben ser creados, distribuidos y destruidos. Esta operación puede ser muy costosa dentro del contexto de una iteración.

Una segunda y mejor aproximación es lanzar los threads (usando la primitiva *omp parallel*) después de que los layouts sean construidos, se haya reservado la memoria necesaria y se hayan calculado los patrones de comunicación necesarios. Posteriormente, primitivas de control para memoria compartida como *omp for* o *omp sections* pueden ser introducidas para controlar los threads durante la inicialización de las estructuras de datos y el computo principal. El ejemplo visto en la figura 5 muestra la forma de programar una multiplicación de matrices usando Hitmap.

La ejecución de los patrones de comunicación que realiza el intercambio de datos entre procesos debe ser realizado solo por un thread. Por ello, la primitiva *omp single* debe ser introducida en cada llamada *hit_comDo*, *hit_comStart*, *hit_comEnd*, *hit_patternDo*,

```
 1:  hit_tileNewType( double );
 2:
 3:  void cannonsMM( int n, int m ) {
 4:     /* 1. DECLARE FULL MATRICES WITHOUT MEMORY */
 5:     HitTile_double A, B, C;
 6:     hit_tileDomainShape( &A, hit_shape( 2, hit_sig(0,n,1), hit_sig(0,m,1) );
 7:     hit_tileDomainShape( &B, hit_shape( 2, hit_sig(0,m,1), hit_sig(0,n,1) );
 8:     hit_tileDomainShape( &C, hit_shape( 2, hit_sig(0,n,1), hit_sig(0,n,1) );
 9:
10:     /* 2. CREATE VIRTUAL TOPOLOGY */
11:     HitTopology topo = hit_topology( plug_topSquare );
12:
13:     /* 3. COMPUTE PARTITIONS */
14:     HitLayout layA = hit_layoutWrap( plug_layBlocks, topo, hit_tileShape( A ) );
15:     HitLayout layB = hit_layoutWrap( plug_layBlocks, topo, hit_tileShape( B ) );
16:     HitLayout layC = hit_layoutWrap( plug_layBlocks, topo, hit_tileShape( C ) );
17:
18:     /* 4. CREATE AND ALLOCATE TILES */
19:     HitTile_double tileA, tileB, tileC;
20:     hit_tileSelectNoBoundary( &tileA, &A, hit_layMaxShape(layA));
21:     hit_tileSelectNoBoundary( &tileB, &B, hit_layMaxShape(layB) );
22:     hit_tileSelect( &tileC, &C, hit_layShape(layC) );
23:     hit_tileAlloc( &tileA ); hit_tileAlloc( &tileB ); hit_tileAlloc( &tileC );
24:
25:     /* 5. REUSABLE COMM PATTERNS */
26      HitComm alignRow = hit_comShiftDim( layA, 1, -hit_layRank(layA,0), &tileA, HIT_DOUBLE )
27      HitComm alignColumn = hit_comShiftDim( layB, 0, -hit_layRank(layB,1), &tileB, HIT_DOUBLE );
28:     HitPattern shift = hit_pattern( HIT_PAT_UNORDERED );
29:     hit_patternAdd( &shift, hit_comShiftDim( layA, 1, 1, &tileA, HIT_DOUBLE ) );
30:     hit_patternAdd( &shift, hit_comShiftDim( layB, 0, 1, &tileB, HIT_DOUBLE ) );
31:
32:     /* 6. INITIALIZE MATRICES */
33:     hit_tileFileRead( &tileA, "matrixA.dat" );
34:     hit_tileFileRead( &tileB, "matrixB.dat" );
35:     int aux=0; hit_tileFill( &tileC, &aux );
36:
37:     /* 7. INITIAL ALIGNMENT PHASE */
38:     hit_comDo( alignRow );  hit_comDo( alignColumn );
39:
40:     /* 8. DO COMPUTATION */
41:     int loopIndex;
42:     int loopLimit = max( layA.numActives[0], layB.numActives[1] );
43:     for (loopIndex = 0; loopIndex < loopLimit-1; loopIndex++) {
44:        matrixProduct( tileA, tileB, tileC );
45:        hit_patternDo( shift );
46:     }
47:     matrixProduct( tileA, tileB, tileC );
48:
49:     /* 9. WRITE RESULT */
50:     hit_tileFileWrite( &tileC, "matrixC.dat" );
51:
52:     /* 10. FREE RESOURCES */
53:     hit_layFree( layA ); hit_layFree( layB ); hit_layFree( layC );
54:     hit_comFree( alignRow ); hit_comFree( alignColumn );
55:     hit_patternFree( &shift );
56:     hit_topFree( topo );
57:  }
58:
59:  /* SEQ_1. CLASSICAL MATRIX PRODUCT */
60:  static inline void matrixProduct( HitTile_double A, HitTile_double B, HitTile_double C ) {
61:     int i,j,k;
62:     for( i=0; i<hit_tileDimCard( C, 0 ); i++ )
63:        for( j=0; j<hit_tileDimCard( C, 1 ); j++ )
64:           for( k=0; k<hit_tileDimCard( A, 1 ); k++ )
65:              hit_tileAt2( C, i, j ) += hit_tileAt2( A, i, k ) * hit_tileAt2( B, k, j );
66:  }
```

Fig. 4. Algoritmo de Cannon para la multiplicación de matrices en Hitmap.

```
/* MATRIX MULTIPLICATION: COMPUTATION STAGE INSIDE CANNON'S */
#pragma omp parallel private(loopIndex)
{
    hit_tileFileRead( &tileA, "matrixA.dat" );
    hit_tileFileRead( &tileB, "matrixB.dat" );
    #pragma omp single
    { hit_commDo( alignRow ); hit_commDo( alignColumn ); }

    for (loopIndex = 0; loopIndex < loopLimit-1; loopIndex++) {
       matrixProduct( tileA, tileB, tileC );
       #pragma omp single
       hit_patternDo( shift );
    }
    matrixProduct( tileA, tileB, tileC );
    writeResults( tileC );
}

static inline void matrixProduct( HitTile_double A, HitTile_double B, HitTile_double C ) {
   int i, j, k, ti, tj, tk;
   int numTiles0 = (int)ceil( hit_tileDimCard( C, 0 ) / tilesize );
   int numTiles1 = (int)ceil( hit_tileDimCard( C, 1 ) / tilesize );
   int numTiles2 = (int)ceil( hit_tileDimCard( A, 1 ) / tilesize );

   #pragma omp for collapse(2) private(i,j,k,ti,tj,tk)
   for (ti=0; ti<numTiles0; ti++) {
      for (tj=0; tj<numTiles1; tj++) {
         int begin0 = ti * tilesize;
         int end0 = min( hit_tileDimCard( C, 0 )-1, (begin0 + tilesize -1) );
         int begin1 = tj * tilesize;
         int end1 = min( hit_tileDimCard( C, 1 )-1, (begin1 + tilesize -1) );
         for (tk=0; tk<numTiles2; tk++) {
            int begin2 = tk * tilesize;
            int end2 = min( hit_tileDimCard( A, 1 )-1, (begin2 + tilesize -1) );
            /* MATRIX TILE BLOCK PRODUCT */
            for (i=begin0; i<=end0; i++)
               for (j=begin1; j<=end1; j++)
                  for (k=begin2; k<=end2; k++)
                     hit_tileAt2( C, i, j ) += hit_tileAt2( A, i, k ) * hit_tileAt2( B, k, j );
         }
      }
   }
```

Fig. 5. Computación local de una multiplicación de matrices en un programa Hitmap paralelizado con OpenMP después de aplicar tiling y una inversión de bucles.

hit_patternStart. *hit_patternEnd*. Además, para asegurar que los datos a comunicar son correctos y no comunicamos datos que aun siguen computándose, es necesaria la sincronización de los threads antes y después de la comunicación. En el ejemplo presentado en la figura 5 las primitivas *for* y *single* de OpenMP llevan implícita la sincronización de los threads. Es posible explotar solapamiento de computación y comunicaciones usando las funcionalidades propias de Hitmap para comunicaciones asíncronas, como en el caso IV-B (Jacobi).

C. Modelo de ejecución

El modelo de ejecución de un código Hitmap es simple. El programa se lanza con un proceso por nodo, o por un subconjunto de núcleos. Por ejemplo, usando el lanzador *hydra* es posible desplegar tantos procesos MPI como se desee en cada nodo físico. Las políticas internas para topología y layout adaptarán automáticamente la partición de datos y los patrones de comunicación en el nivel de memoria distribuida.

Cada tarea local lanzará tantos threads como número de CPUs se hayan asignado. Por ejemplo, usando OpenMP, inicializar el número de threads se realiza directamente usando la función *omp_set_threads_num()*. En la experimentación realizaremos diferentes pruebas variando las configuraciones de threads/procesos usando un valor pasado como argumento en la línea de comandos al lanzar los procesos MPI.

D. Paralelización y transformación de código

Se pueden utilizar transformaciones de código fuente para la optimización de códigos paralelos y secuenciales, o para simplificar la introducción del paralelismo. Algunas potentes optimizaciones como el *tiling* pueden ser implementadas en códigos con una alta tasa de reutilización de datos. Hitmap trabaja con un grano grueso para minimizar los costes de la comunicación entre datos. Por tanto, la técnica de tiling debe ser usada en las partes secuenciales antes de aplicar las técnicas de paralelización para memoria compartida. Esto generará particiones de un grano un poco más fino, optimizando así el uso de la memoria local y las cachés. Podemos ver un ejemplo en la figura 5.

También se pueden aplicar transformaciones de código para paralelizar cuando existen dependencias de datos. Por ejemplo, una aplicación de tipo *wave-front* puede ser fácilmente expresada con bucles secuenciales. Sin embargo, estas aplicaciones no pueden ser fácilmente paralelizadas con modelos como OpenMP, donde las primitivas no dan soporte para un control guiado por el flujo de datos.

Una solución puede ser desarrollar manualmente un programa con variables de bloqueo en cada thread. En esta aproximación, el programador está replicando manualmente la partición de datos y las sincronizacines introducidas en Hitmap, pero en el nivel de memoria compartida.

Una aproximación más fácil es la transformación completa de los bucles. La transformación generará un bucle exterior que recorra el dominio de iteraciones siguiendo un hiperplano en el que no hay dependencias, y dónde es posible una paralelización inmediata. Por ejemplo, una aplicación de wave-front puede ser transformada en una estructura de pipeline donde el bucle exterior avanza siguiendo la dirección de avance de las dependencias. Cada paso de avance es equivalente a una etapa del pipeline y dentro de esa etapa todos los elementos se calculan de forma paralela sin dependencias.

Por ejemplo, en el caso del benchmark Gauss-Seidel, es posible generar un bucle que recorra las diagonales secundarias de la matriz. Aunque la computación de los datos de una diagonal depende de datos de la diagonal anterior, las actualizaciones dentro de cada diagonal pueden ser realizadas en paralelo de forma trivial.

E. Integración de transformaciones de código del polyhedral model

Las transformaciones previamente descritas pueden ser automáticamente aplicadas usando técnicas propias del *polyhedral model* [6]. Por ejemplo, para la computación de las partes locales de los ejemplos presentados en la sección IV es posible aplicar estas técnicas en la zona secuencial.

Pluto [9] es un compilador en desarrollo que aplica transformaciones y optimizaciones a nivel del código fuente en códigos compatibles con las restricciones impuestas por el polyhedral model. Hemos adaptado el front-end de Pluto v0.9.0 para que genere trozos aislados de código, sin la función main, la declaración de las variables compartidas, etc. De esta forma podemos usarlo para sustituir en nuestro programa las zonas secuenciales por los códigos generados por Pluto. La integración de Pluto con Hitmap genera programas automáticamente optimizados. Además, se ha desarrollado una modificación adicional que altera las directivas introducidas automáticamente por Pluto, para adaptarlas a la propuesta de colocación de directivas de OpenMP que se ha presentado en la sección V-B. La versión original de Pluto genera directivas *omp parallel for* para los bucles transformados. En nuestra propuesta hemos modificado esas directivas por directivas *omp for* ya que los threads ya habrán sid crearemos una única vez al inicio de la sección secuencial.

Pluto puede puede ser usado para realizar optimizaciones automáticas de las partes secuenciales e integrarlas posteriormente en un programa Hitmap para facilitar el paralelismo en programas con dependencias. Mediante la metodología propuesta y las herramientas descritas es posible transformar un código Hitmap en un código paralelo multinivel que explota de una manera eficiente entornos de memoria distribuida-compartida, adaptando su estructura a la plataforma de ejecución.

VI. Experimentación

Hemos llevado a cabo un estudio para la valoración de las ventajas de nuestra propuesta y para verificar la eficiencia de los códigos resultantes.

A. Metodología

Para la evaluación de la propuesta se han diseñado varias configuraciones de ejecución teniendo en cuenta los procesos MPI lanzados vs. el número de hilos. Compararemos los resultados con códigos de referencia manualmente desarrollados y optimizados programados usando MPI para memoria distribuida y OpenMP para memoria compartida. Los códigos que usaremos son:

- **Ref-MPI** y **Ref-OMP**: Códigos paralelizados manualmente para memoria distribuida y compartida respectivamente.
- **Hitmap**: Código paralelizado con Hitmap donde las secciones secuenciales son una transcripción de los códigos de referencia.
- **Hitmap+PlutoOrig**: Códigos de la versión Hitmap sustituyendo la parte secuencial por el código secuencial generado por Pluto.
- **Hitmap+Pluto**: Códigos de la versión Hitmap utilizando las directivas OpenMP de la forma que describimos en nuestra propuesta, y usando cómo parte secuencial la versión optimizada por nuestra modificación de Pluto.
- **Hitmap-Locks**: En el caso de estudio de Gauss-Seidel la referencia con OpenMP es un pipeline construído manualmente, usando semáforos y realizando la partición de datos en función del número de threads.

Utilizamos dos plataformas de experimentación:

1. **Atlas**: Atlas es una máquina de memoria compartida perteneciente al grupo Trasgo de la Universidad de Valladolid. Se trata de un servidor Dell PowerEdge R815 con 4 AMD Opteron 6376 cuya velocidad de procesador es 2.3 GHz y con 16 núcleos cada procesador. Esto hace un total de 64 núcleos y 256 GB de RAM.
2. **Caléndula**: Es un clúster híbrido perteneciente a la Fundación C̈entro de Supercomputación de Castilla y León:̈ Los nodos están conectados mediante tecnología Infiniband. Cada nodo tiene dos CPUs Intel Xeon 545 de 4 núcleos. Usando 8 nodos del clúster podemos explotar 64 unidades computacionales.

En ambos casos compilaremos los códigos con el compilador GCC usando el flag *-fopenmp* y los flags de optimización *-O3* y *-funroll-loops*. En el nivel de memoria distribuida utilizamos *mpich2 v3.0.4* como implementación de MPI en Atlas y *OpenMPI v1.6.5* en Caléndula.

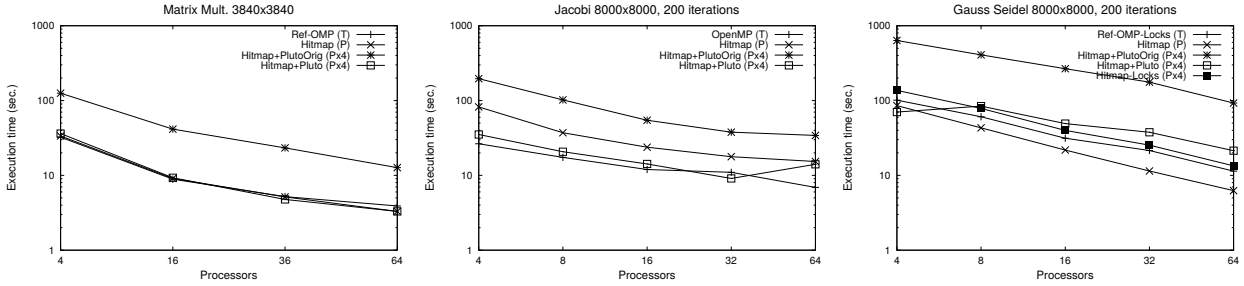

Fig. 6. Resultados de rendimiento en Atlas (plataforma de memoria compartida)

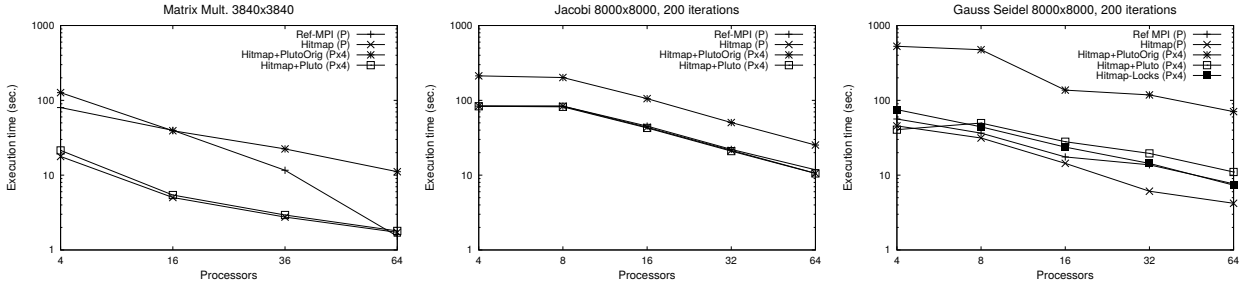

Fig. 7. Resultados de rendimiento en Caléndula (plataforma híbrida de memoria distribuida-compartida)

Para las estructuras de datos hemos seleccionado un tamaño suficientemente grande cómo para producir una carga computacional relevante con 64 unidades computacionales en memoria distribuida. Para los ejemplos de Jacobi y Gauss-Seidel presentaremos resultados de la computación de las primeras 200 iteraciones del bucle principal con matrices de 8000x8000 elementos. Para la realización de la experimentación hemos eliminado de los códigos la condición de convergencia por simplicidad y así poder centrarnos en los patrones de sincronización descritos en la sección IV. En el caso de estudio de la multiplicación de matrices realizaremos la computación de dos matrices de 3840x3840 elementos.

Los programas referencia de OpenMP han sido ejecutados en Atlas con T = 4, 8, 16, 32 y 64 threads. En Caléndula los programas de referencia de MPI han sido ejecutados con P = 4, 8, 16, 32 y 64 procesos. Los procesos han sido distribuidos por los nodos dependiendo del número de threads lanzados por proceso MPI. Los nodos serán completados con procesos o threads antes de lanzar más procesos en otro nodo diferente. Los programas **Hitmap** son ejecutados solo con procesos MPI (P) en ambas plataformas. El resto de modificaciones de Hitmap han sido ejecutadas en ambas plataformas con combinaciones de 1 a 16 procesos MPI con 4 threads en cada proceso (P x 4). También se han realizado experimentos con 8 threads por proceso en los cuáles hemos obtenido resultados similares a P x 4. Algunas de las configuraciones propuestas no serán válidas para la multiplicación de matrices debido a la restricción impuesta por el algoritmo de Cannon. Con el algoritmo Cannon es necesario que la topología de los procesos MPI sea un cuadrado perfecto.

Mostraremos los resultados en diferentes gráficas en las que enfrentaremos los tiempos de ejecución vs. el número de procesadores activos (núcleos con threads o procesos MPI asignados). Además, usaremos una escala logarítmica para observar de manera más fácil la escalabilidad resultante y potenciales perdidas de rendimiento constantes o proporcionales entre las diferentes versiones.

B. Resultados en memoria compartida

Los resultados de los tres casos de estudio en Atlas (plataforma de memoria compartida) se muestran en la figura 6. En todos los casos podemos ver que Hitmap no introduce importantes sobrecostes en el tiempo de ejecución. Los resultados muestran que el rendimiento obtenido en una arquitectura de memoria compartida por un código híbrido de Hitmap es similar a un código puro de OpenMP cuando explotan las mismas técnicas de optimización.

El código generado por Pluto funciona mejor cuando aplicamos a éste nuestra propuesta de metodología para la inclusión de directivas OpenMP en el contexto de una comunicación entre procesos. La razón es que en Hitmap los threads son lanzados simplemente una vez y coordinados con primitivas de memoria compartida durante el resto de esa fase de computación local. Después de integrar el código generado por Pluto en Hitmap, este produce buenos resultados, similares a nuestra mejor versión manual optimizada. Por ejemplo, las técnicas de optimización usadas en la versión manual de la multiplicación de matrices pueden ser automáticamente obtenidas con Pluto.

El caso de Gauss-Seidel requiere mayor atención. Ni las versiones modificadas de Pluto, ni la versión manualmente desarrollada y optimizada que usa variables de bloqueo consiguen mejores rendimientos que un programa exclusivamente basado en el paso de mensajes. Esto es debido a que el control

basado en flujo de datos del paradigma pipeline es más apropiado en el contexto de comunicación entre procesos que para los modelos de programación en memoria compartida como OpenMP.

C. Resultados en el clúster híbrido

Los mismos programas de Hitmap del experimento anterior pueden ser utilizados en la plataforma de memoria distribuida sin ningún cambio. Vemos los resultados de los tres casos de estudio ejecutados en Caléndula en la figura 7. Los resultados muestran que la comunicación entre diferentes nodos añade un retraso que minimiza el impacto de los tiempos de sincronización/comunicación dentro de los nodos. Así, cuando se usan threads en lugar de más procesos MPI para explotar los núcleos dentro de los nodos solo obtenemos una leve mejora en el rendimiento.

Los resultados para la multiplicación de matrices muestran que el código híbrido jerárquico de dos niveles de Hitmap con el código generado de Pluto modificado obtiene similares rendimientos que los códigos de Hitmap originales y mejores que los códigos de referencia MPI. Ambos códigos con la parte secuencial optimizada.

En el caso del ejemplo de sincronización entre vecinos (Jacobi) todas las versiones presentan el mismo rendimiento excepto la versión original de Pluto que es más lenta. En el caso de Gauss-Seidel se obtienen los mejores resultados con la versión original de paso de mensajes de Hitmap donde el código secuencial está mejor optimizado por el compilador que el código original de referencia. Estos resultados muestran que la principal ventaja del método de Hitmap es que usando una metodología simple y unas herramientas de optimización y paralelización automática es posible generar programas híbridos capaces de explotar las mejores ventajas de plataformas con memoria distribuida y compartida. Los programas adaptan sus comunicaciones y sincronizaciones a las características de run-time de la plataforma donde se ejecuta.

VII. Conclusiones

En este artículo hemos presentado un método para la programación en sistemas híbridos de memoria compartida y distribuida. Con esta metodología somos capaces de obtener códigos que nos dan resultados similares a programas cuidadosamente optimizados de forma manual, pero simplificando su programación con el uso de herramientas automáticas.

La coordinación en el nivel de memoria distribuida se simplifica utilizando Hitmap. Hemos mostrado una metodología para integrar Hitmap con modelos de programación para memoria compartida y con herramientas automáticas que explotan de la forma más eficiente las ventajas de cada modelo. Así, podemos generar fácilmente programas paralelos multinivel que adaptan su estructura de comunicación y sincronización a la máquina donde se ejecuta.

Concretamente, en este trabajo presentamos directrices y ejemplos de cómo usar el paradigma de programación en memoria compartida (usando OpenMP) para paralelizar las partes secuenciales de un programa en Hitmap para memoria distribuida. Todo con el fin de explotar paralelismo de threads dentro de cada proceso MPI. Además, mostramos como integrar una herramienta basada en técnicas del polyhedral model para automatizar y optimizar las partes secuenciales. Esta herramienta es básica para paralelizar de manera automática aplicaciones donde existen dependencias de datos. Nuestros resultados experimentales muestran que los programas híbridos de Hitmap igualan o superan los mejores resultados obtenidos con programas de referencia manualmente desarrollados y optimizados.

El actual framework (versión en desarrollo) y los ejemplos se encuentran disponibles bajo petición. El trabajo futuro incluirá el estudio de la aplicabilidad de estas técnicas para aplicaciones dinámicas y no poliédricas, la aplicación de la misma metodología en aplicaciones con matrices densas y dispersas y el aumento del número de niveles de paralelismo para soportar mayor hetereogenidad. Una versión extendida de este artículo ha sido remitida al congreso "High Performance Computing and Simulation (HPCS'2014)".

Agradecimientos

Esta investigación ha sido parcialmente financiada por el Ministerio de Economía y Competitividad (Spain) y el programa ERDF de la Unión Europea: Red CAPAP-H4 (TIN2011-15734-E), proyecto MOGECOPP (TIN2011-25639); y por la Junta de Castilla y León: proyecto ATLAS (VA172A12-2).

Referencias

[1] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D.R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE TPDS*, vol. (to appear), 2013, (doi:10.1109/TPDS.2013.83).

[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC : distributed shared-memory programming*, Wiley-Interscience, 2003.

[3] B.L. Chamberlain, S.J. Deitz, D. Iten, and S-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[4] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Mara J. Garzar¡n, David Padua, and Christoph von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. of the ACM SIGPLAN PPoPP*, New York, New York, USA, 2006, pp. 48–57, ACM.

[5] Y. Chen, X. Cui, and H. Mei, "PARRAY: A unifying array representation for heterogenous parallelism," in *Proc. PPoPP'12*. Feb 2012, ACM.

[6] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. PACT'04*. 2004, pp. 7–16, ACM Press.

[7] J. Fresno, A. Gonzalez-Escribano, and D.R. Llanos, "Extending a hierarchical tiling arrays library to support sparse data partitioning," *The Journal of Supercomputing*, vol. 64, no. 1, pp. 59–68, 2012.

[8] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta, *Introduction to Parallel Computing*, Addison-Wesley, 2 edition, 2003.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.