UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MASTER

MASTER UNIVERSITARIO EN INVESTIGACIÓN

EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

Explotando jerarquías de memoria distribuida/compartida con Hitmap

Autor:

Dña. Ana Moretón Fernández

Tutores:

Dr. Arturo González Escribano, Dr. Diego R. Llanos Ferraris

Valladolid, 3 de Julio de 2014

TÍTULO:	**Explotando jerarquías de memoria distribuida/compartida con Hitmap**
AUTOR:	**Dña. Ana Moretón Fernández**
TUTOR:	**Dr. Arturo González Escribano, Dr. Diego R. Llanos Ferraris**
DEPARTAMENTO:	**Informática**

Tribunal

PRESIDENTE:	**Pablo de la Fuente Redondo**
VOCAL:	**Anibal Bregón Bregon**
SECRETARIO:	**Miguel A. Martínez Prieto**
FECHA:	**3 de Julio de 2014**
CALIFICACIÓN:	

Resumen del TFM

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 entre todas las máquinas del clúster. 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, adecuada cada una de ellas 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. La coordinación en el nivel de memoria distribuida se facilita 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 nivel del sistema 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 experimentales muestran como la propuesta del trabajo mejora los mejores resultados obtenidos con programas de referencia optimizados manualmente usando MPI u OpenMP.

Palabras clave

Computación paralela, sistemas memoria jerárquicos, OpenMP , MPI

Abstract

Current multicomputers are typically built as interconnected clusters of shared-memory multicore computers. A common programming approach for these clusters is to simply use a message-passing paradigm, launching as many processes as cores available. Nevertheless, to better exploit the scalability of these clusters and highly-parallel multicore systems, it is needed to efficiently use their distributed- and shared-memory hierarchies. This implies to combine different programming paradigms and tools at different levels of the program design.

This work presents an approach to ease the programming for mixed distributed and shared memory parallel computers. The coordination at the distributed memory level is simplified using the library Hitmap. We show how this tool can be integrated with shared-memory programming models and automatic code generation tools to efficiently exploit the multicore environment of each multicomputer node. This approach allows to exploit the most appropriate techniques for each model, easily generating multilevel parallel programs that automatically adapt their communication and synchronization structures to the target machine. Our experimental results show how this approach mimics or even improves the best performance results obtained with manually optimized codes using pure MPI or OpenMP codes.

Keywords

Parallel computing, Hierarchical memory levels, OpenMP, MPI

Índice general

Índice de figuras

Capítulo 1

Introducción y Motivación

1.1. Motivación

En la actualidad las máquinas paralelas se están convirtiendo en dispositivos mas hetereogeneos con diferentes niveles jerárquicos de memoria tanto compartida como distribuida. Además, la programación de aplicaciones paralelas esta avanzando hacia soluciones más complejas explotando diferentes niveles de paralelismo con diferentes estrategias de paralelización. Esta situación justifica el gran existente interés actual por generar programas paralelos eficientes con la habilidad de adaptar automáticamente su estructura al sistema donde se va a ejecutar, ya sea un sistema con memoria distribuida, compartida o un sistema híbrido.

La programación en este tipo de entornos está cambiando, como muestra la cantidad de modelos de programación paralelos y herramientas que se han propuesto para entornos específicos. Sin embargo, el programador todavía debe lidiar con importantes decisiones ajenas a los algoritmos de programación paralelos pero que afectan seriamente al rendimiento del programa. 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* (despliegue, colocación o asociación de datos y procesos a los recursos computacionales). 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.

El sistema Trasgo [32] (posteriormente descrito en la sección 2.3) fue propuesto como marco de trabajo para desarrollar un sistema de traducción y compilación para programas paralelos a partir de un código de alto nivel, completamente independiente de decisiones relacionadas con los recursos de la máquina. Un prototipo de Trasgo ha sido descrito en [25], [28] y [29]. El prototipo genera códigos para memoria distribuida programados en C con llamadas a una librería de *run-time* propia denominada Hitmap [33] basada en MPI.

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. Estudios como [42] han demostrado la mejora en rendimiento que ofrecen códigos híbridos que usan MPI y OpenMP sobre códigos únicamente escritos en MPI.

En este trabajo veremos cómo integrar Hitmap con modelos de programación de memoria compartida para generar programas multinivel que exploten entornos híbridos. El modelo de comunicación de Hitmap será utilizado para coordinar el *mapping* y la sincronización en el nivel de memoria distribuida dejando un proceso para cada nodo o para cada subconjunto de nodos. Así, las estrategias para programación en memoria compartida, sus 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.

Con este trabajo se busca conseguir una metodología sistemática para la programación de códigos paralelos usando la librería Hitmap que aprovechen con la máxima eficiencia posible las capacidades de un sistema híbrido. Además, nos aportará una base sólida para poder modificar la implementación de Trasgo con el fin de generar códigos con mejor rendimiento.

1.2. Objetivos

Los objetivos de este trabajo fin de máster son:

- Estudio de los diferentes *frameworks* capaces de generar código paralelo para la explotación de memoria compartida y memoria distribuida simultáneamente.
- Diseño de una metodología capaz de conseguir códigos que exploten tanto memoria distribuida como memoria compartida sobre la librería Hitmap [33].
- Implementación de la metodología en aplicaciones programadas en Hitmap.
- Experimentación y verificación de la nueva metodología. Se presentarán casos de estudio para demostrar las ventajas de la programación con la nueva metodología con respecto a códigos programados sin ella.

El resultado final será la definición explícita de los pasos a seguir en la programación de códigos paralelos con la librería Hitmap. Estos códigos aprovecharán las mejores ventajas de la memoria distribuida y de la memoria compartida.

1.3. Estructura

La estructura de la memoria es la siguiente. En el capítulo 2 describiremos los diferentes modelos de programación usados para los diferentes sistemas de memoria en las plataformas de ejecución. Se expondrá la biblioteca usada en este trabajo, la forma de usarla y el sistema general de traducción que la utiliza. Además, se ha realizado un estudio de las diferentes herramientas encontradas en la literatura. En el capítulo 3 se desarrollará la propuesta de este Trabajo Fin de Máster. En el capítulo 4 se realiza un análisis de la propuesta por medio de una experimentación en una máquina de memoria compartida y en una de memoria distribuida para tres casos de estudio.

Por último se expondrán en el capítulo 5 las conclusiones y en el anexo A la publicación a la que este trabajo ha dado lugar.

Capítulo 2

Antecedentes/Contexto

2.1. Modelos de programación paralela

Actualmente existen diferentes modelos de programación para los diferentes tipos de sistemas paralelos. Muchos modelos y herramientas de programación paralela se han ido proponiendo y asentando, generando diferentes escenarios de abstracción sobre los sistemas reales. Las bibliotecas de paso de mensajes, por ejemplo las que usan el estándar MPI [1], se han convertido en la opción habitual para sistemas de memoria distribuida. Modelos como OpenMP [2], Cilk [3] o TBBs [4] se utilizan habitualmente para simplificar el manejo de los hilos de ejecución y el acceso a memoria compartida global.

2.1.1. Paso de mensajes: modelo para memoria distribuida

En el modelo de paso de mensajes cada nodo de computación realiza una serie de tareas computando sobre su memoria local. Al tener acceso únicamente a su memoria local es posible que la ejecución del programa necesite una sincronización o una comunicación de datos con otros nodos. La forma natural de comunicar y sincronizar procesos en los sistemas distribuidos es mediante paso de mensajes: los procesos intercambian mensajes mediante operaciones explicitas de envío (*send*) y de recibo (*receive*) que constituyen las primitivas básicas de comunicación de este tipo. Los sistemas de paso de mensajes pueden verse como una extensión de un mecanismo de sincronización como los semáforos, en la que añadimos la comunicación de datos entre los procesos sincronizados.

En toda comunicación, los emisores y receptores pueden no coincidir en el momento del envío y recepción del mensaje. Por ello, las comunicaciones pueden ser de dos tipos:

- **Síncronas:** En este tipo de operaciones el emisor queda bloqueado hasta que el receptor realice la función de recibir. Es decir, es necesario que el emisor y el receptor se sincronicen para que cualquiera de los dos procesos siga avanzando.

- **Asíncronas:** En este tipo de operaciones el emisor realiza el envío de los datos y continua avanzando sin percibir si el receptor ha recibido los datos.

2.1.2. Variables compartidas: modelo para memoria compartida

Los modelos de programación de memoria compartida asumen un espacio de direcciones global al que todos los procesos pueden acceder simultáneamente. Por tanto, no es necesario comunicar explícitamente datos. Un procesador escribe en ciertas direcciones y cualquier otro puede leerlos. En estos modelos el programador es responsable de introducir las primitivas de sincronización necesarias para organizar el orden de ejecución de las acciones paralelas que leen/escriben en la memoria global, para asegurar la corrección del resultado.

OpenMP [2] es una especificación para un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que pueden ser usadas para especificar paralelismo de alto nivel en programas escritos en Fortran, C o C++ para sistemas de memoria compartida.

Polyhedral model

El *polyhedral model* es una representación matemática para la optimización y paralelización de bucles con expresiones afines. Un conjunto de bucles anidados en los que se realizan operaciones que impliquen dependencias de datos no pueden ser paralelizados de forma directa con OpenMP ya que obtendríamos resultados erróneos.

Por ello existen varias propuestas [12] que usan el *polyhedral model* para que, realizando transformaciones en los códigos, se generen bucles que recorran el dominio de datos de una forma paralelizable con modelos de programación de memoria compartida.

Sin embargo, esta técnica posee algunas limitaciones. Principalmente:

- No soporta expresiones no afines.
- No soporta la manipulación de las iteraciones de tipo *while* o con condiciones genéricas.
- No soporta los programas recursivos.

Pluto [14] es un compilador en desarrollo que transforma código puramente secuencial siempre que cumpla las limitaciones impuestas por el *polyhedral model* en código paralelo usando OpenMP.

Un ejemplo del análisis y la transformación de código que realiza una herramienta poliédrica lo podemos ver en la figura 2.1. En esta figura vemos dos códigos con bucles que recorren las mismas iteraciones. Sin embargo el código de la derecha ha sido geneerado automáticamente y recorre las mismas iteraciones que el código de la izquierda pero en diferente orden. Además en este ejemplo vemos como para recorrer iteraciones en diferente orden es necesaria la programación de un código más complejo.

2.2. Hitmap

Hitmap [33] es una biblioteca de funciones diseñada para el reparto y *mapping* jerárquico de estructuras de datos originalmente densas. También ha sido extendida para soportar

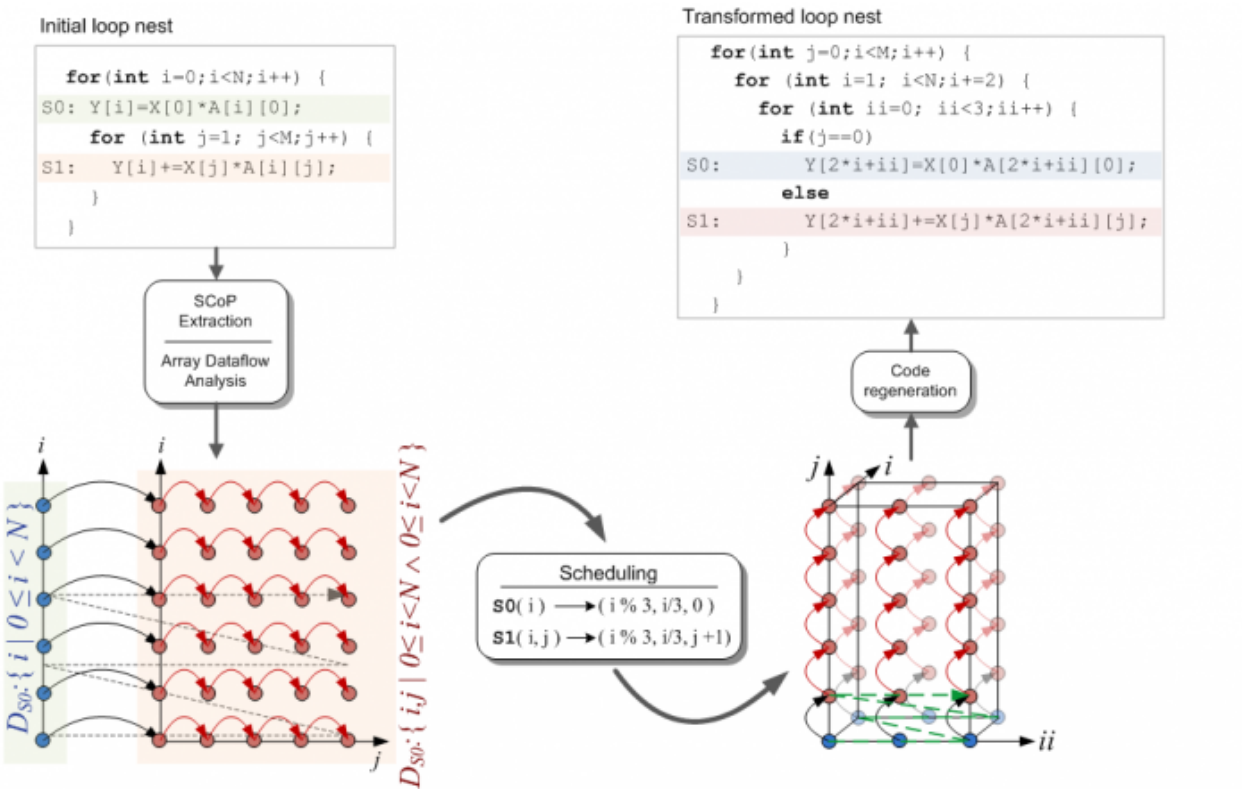

Figura 2.1: Ejemplo de análisis con *polyhedral model*

estructuras de datos dispersas como matrices dispersas o grafos [30] usando la misma metodología e interfaz. Hitmap está basado en el modelo de programación distribuido SPMD (Single Program, Multiple Data). Utiliza abstracciones denominadas *Tiles* para la declaración de estructuras o subestructuras de datos. Hitmap también automatiza la partición, repartición y comunicación de *Tiles* en los diferentes niveles jerárquicos de un programa paralelo, consiguiendo un buen rendimiento.

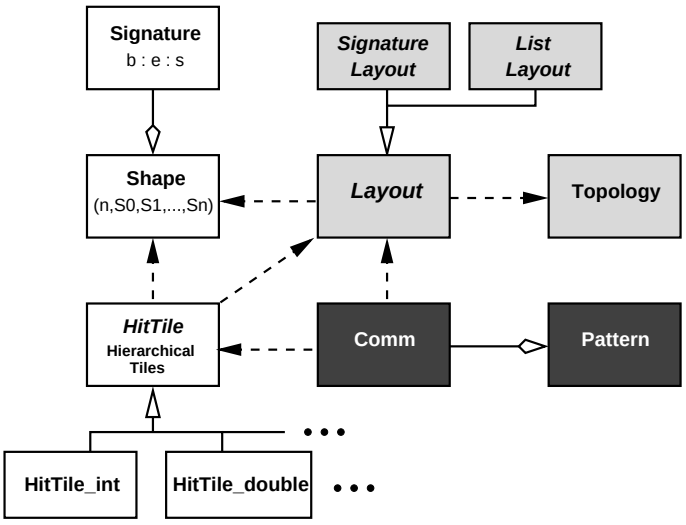

Figura 2.2: Arquitectura de Hitmap. Las cajas blancas representan las clases que gestionan los dominios y las estructuras de datos. Las cajas gris claro representan las clases que aplican la partición y el *mapping* del dominio. Las cajas gris oscuro representan las clases que generan patrones de comunicación adaptables y reutilizables.

2.2.1. 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 2.2 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 (salto regular). Detalles sobre como se extienden las clases para estructuras dispersas se pueden encontrar en [30].

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 reales sobre el espacio de índices definido por el *shape*. Internamente los *tiles* reservan la memoria necesaria en bloques de memoria 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 puedan ser declarados como inactivos por la política de generación de la topología 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 de comunicación en función de los dominios de los *tiles*, la información de los objetos *layout*, y las reglas de vecindad de la topología 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.

2.3. Uso de Hitmap en el sistema Trasgo

Trasgo [32] es un sistema de traducción y compilación capaz de generar código paralelo eficiente en C con llamadas a la librería Hitmap a partir de un código paralelo de alto nivel.

El modelo de arquitectura de Trasgo se muestra en la figura 2.3. La entrada a Trasgo es un código paralelo explícito, basado en un lenguaje tradicional añadiendo algunas extensiones para la programación en paralelo (ver fig. 2.3 (A)). En concreto, Trasgo utiliza por defecto un lenguaje basado en una extensión de C denominado cSPC.

Mostraremos un ejemplo de programación en cSPC. Se trata de una solución iterativa de Jacobi para el sistema de ecuaciones asociado a una discretización de las ecuación diferencial parcial de Poisson, que determina la dispersión del calor en una superficie. Se

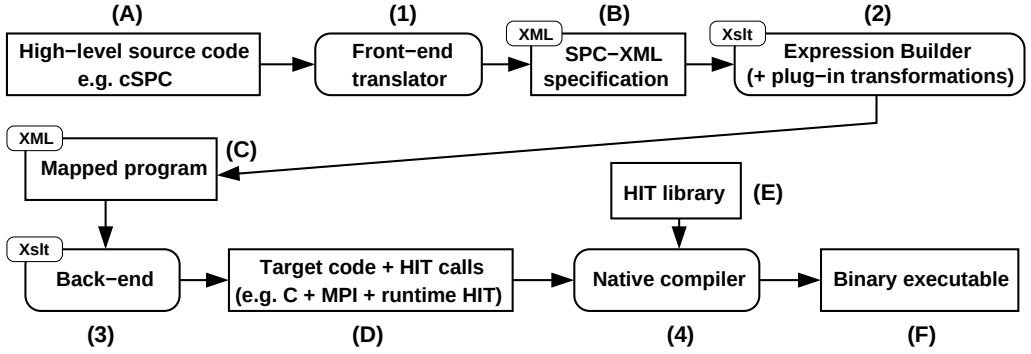

Figura 2.3: Arquitectura de Trasgo

implementa como una actualización iterativa de las celdas de una matriz, en función de los valores de sus cuatro vecinos, o celdas adyacentes. Este tipo de algoritmo se denomina también Cellular Automata.

El código secuencial lo podemos ver en la figura 2.4. Esta aplicación será presentada de forma completa en la sección 3.3. La programación de esta aplicación en el lenguaje de alto nivel de entrada a Trasgo, cSPC la observamos en la figura 2.5.

En la linea 13 de la figura 2.5 se encuentra declarada la función secuencial que realizará cada elemento de la matriz. La única diferencia con el lenguaje C será la necesidad de añadir un modificador para cada parámetro que indique su función en el interfaz: *Entrada* (sólo se usan los datos), *Salida* (no se usan los datos de entrada, pero se modifican), la combinación de ambas, *Creación* (se define o altera dentro de la función la cantidad de datos contenidos en el tile), o *Creación e introducción* de nuevos datos en el tile.

En la linea 24 de la figura 2.5 comienza la programación en paralelo. Lo primero es la declaración de función paralela añadiendo el modificador *coordination* al inicio. Después, como en el lenguaje C, se llevarán a cabo las declaraciones de las variables o arrays necesarios en el programa. El siguiente paso es la partición de los datos. Esta la realizaremos con la sentencia *Map* que especificará los datos que se han de mapear para la ejecución en paralelo. Contiene los siguientes parámetros. El parámetro *shape* donde se especifica el dominio de datos en el que se trabajará. *Layout*, con el cual elegiremos la política de particionar el *shape* para su tratamiento y el parámetro *topology* que escogerá la política para crear una topología virtual de los procesadores. El siguiente paso será la asignación del trozo asignado al proceso a un array. Para ello usaremos la sentencia *Array*. En este caso estaremos indicando que el array *m*, tenga el tamaño de array que el *Map* ha asignado a el proceso que esta ejecutando. Estos pasos son prácticamente comunes en todos los programas en cSPC, con la única diferencia de la creación de más o menos objetos *layout* o *Array*.

Seguidamente pasaremos a la implementación del algoritmo de la aplicación. En este caso será un bucle en el que se ejecuten para cada elemento una función secuencial previamente declarada. Para realizar este algoritmo en paralelo usaremos la primitiva *parallel*. La sentencia *parallel* es la sentencia propia para ejecutar aplicaciones en paralelo. A esta sentencia se le pasa un único parámetro, el parámetro *Map*. Después se encontrarán unos o varios *parblocks*, estos son los diferentes bloques de sentencias que se ejecutarán en paralelo.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>

#define ROWS    atoi( argv[1] )
#define COLS    atoi( argv[2] )
#define STAGES  atoi( argv[3] )

int main(int argc, char *argv[]) {

        double mat[ROWS][COLS];
        double copy[ROWS][COLS];
        int i,j;
        int tiempo = 0;

        // Init
        for ( i=0; i<ROWS; i++ )
                for ( j=0; j<COLS; j++ )
                        mat[i][j] = 0.0;

        for ( i=0; i<ROWS; i++ ) {
                mat[i][0] = 3;
                mat[i][COLS-1] = 4;
        }

        for ( j=0; j<COLS; j++ ) {
                mat[0][j] = 1;
                mat[ROWS-1][j] = 2;
        }

        // Compute
        int stage;

        for ( stage=0; stage<STAGES; stage++ ) {
                // Update copy
                for ( i=0; i<ROWS; i++ ){
                        for ( j=0; j<COLS; j++ )
                                copy[i][j] = mat[i][j];
                }

                // Compute iteration
                for ( i=1; i<ROWS-1; i++ )
                        for ( j=1; j<COLS-1; j++ )
                                mat[i][j] = ( copy[i-1][j] + copy[i+1][j]
                                        + copy[i][j-1] + copy[i][j+1] ) / 4;
        }

        return 0;
}
```

Figura 2.4: Código secuencial de un *Cellular Automata*

```
/**
* Example: Basic cellular automata for the heat equation
*/

#include <stdio.h>
#include <stdlib.h>
#include <hitmap.h>
#include <omp.h>

/**
* Sequential Function to update one cell element
*/
void updatecell(in double up, in double down,
in double left, in double right, out double result ) {

/* TYPICAL POISSON EQUATION */
        *result = (up + down + left + right ) / 4 ;
}


/**
* Parallel Cellular automata in 2D
*/
coordination void main() {

/* Declarations */
int nIter=10;
double matrix[10][10];
double m[][];
layout mapA;

/* Partition */
mapA=Map(matrix[1:$-1][1:$-1].shape, Blocks, Array2DComplete) ;
/* Assign each tile  */
Array(m,mapA);

/* Init each tile*/
initMatrix(m);

// ITERATIONS LOOP
loop( i, [1:nIter] ) {

// PARALLEL SECTION: AS MANY LOGICAL PROCESSES AS INNER ELEMENTS
parallel (mapA ) {
parblock: updatecell(m[ paridx(0)-1 ][ paridx(1) ],
                m[ paridx(0)+1 ][paridx(1)],
                m[ paridx(0) ][ paridx(1)-1 ],
                m[ paridx(0) ][ paridx(1)+1 ],
                m[ paridx(0) ][ paridx(1) ]
                );
        }
    }
}
```

Figura 2.5: Ejemplo de programación en cSPC

Figura 2.6: Fases de una aplicación de cálculo científico típica usando Hitmap. Los estados gris oscuro contienen código secuencial que puede ser jerárquicamente paralelizado.

La sentencia *parallel* lanzará tantos procesos lógicos como vengan definidos en el *Map* que se le pasó. Lanzando así la función secuencial por cada elemento de la matriz. Posteriormente en las siguientes etapas del framework se agruparán los procesos lógicos en procesos reales para conseguir un grano más eficiente de paralelismo.

El código escrito en este lenguaje de alto nivel será traducido, usando un frontend, a un código intermedio basado en XML [28]. Este lenguaje intermedio se denomina xSPC (XML Series-Parallel Coordination) (ver fig. 2.3 (B)). Se utiliza un código intermedio en XML ya que existen potentes herramientas de transformación y generación de código como Xslt y Xpath2.0 [35]. Con estas herramientas se simplifica el detectar patrones en las estructuras del programa y aplicar transformaciones. En Trasgo se propone un módulo de reconstrucción de expresiones, que puede ser ampliado con diversos *plug-ins*, guiado por los patrones detectados en el código intermedio. Estos módulos anotan y transforman el documento inicial en xSPC a un documento nuevo en el que se incluye toda la información necesaria para realizar la distribución de datos y las comunicaciones (ver fig. 2.3 (C)). Finalmente un back-end transforma el código anotado en el código objetivo escrito en C tradicional (ver fig. 2.3 (D)). Este código en C con llamadas a la librería Hitmap (ver fig. 2.3 (E)) será compilado con un compilador nativo para producir el código ejecutable (ver fig. 2.3 (F)).

El código resultante generado por el sistema debe ser lo más eficiente posible. Uno de los objetivos de este Trabajo Fin de Máster es determinar el formato que deben tener los códigos generados para posteriormente realizar los cambios necesarios en Trasgo y conseguir que este genere los códigos más eficientes posibles según la metodología propuesta.

2.4. Metodología de uso de Hitmap

En esta sección describiremos cómo se implementa con Hitmap un programa típico de cálculo científico. Esta metodología permitirá derivar en una guías claras y comunes

para estructurar los códigos. La figura 2.6 muestra los estados en los que se divide una aplicación 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. Los pasos para la programación de típicas aplicaciones paralelas de cálculo científico con Hitmap se pueden extrapolar a muchas otras aplicaciones. Son los siguientes:

1. 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).

2. El segundo paso en el diseño es la definición de los dominios. Todos los procesos 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 los procesadores.

3. Después de conocer el dominio de cada procesador, este 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. El doble sistema de coordenadas ayuda a implementar la computación secuencial de los *tiles*.

4. 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 usando los objetos de comunicación. Para construir los objetos de comunicación será necesario especificar el *tile* local de los datos a recibir o enviar y dejar que Hitmap utilize la información contenida en el objeto *layout* acerca de los vecinos. Por ejemplo para las *ghost zones* el *shape* exacto donde se deben comunicar los datos puede ser obtenido automáticamente usando la funcionalidad de intersección de dominios 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 las figuras 2.7, 2.8 y 2.9 muestra la implementación en Hitmap del Cellular Automata usado como ejemplo en las secciones anteriores.

2.5. Trabajos relacionados

Además de Hitmap existen otras bibliotecas como PARTI/CHAOS [13, 38] que facilitan el desarrollo manual de código paralelo basado en el paso de mensajes. Existen

```
#include <stdio.h>
#include <stdlib.h>
#include <hitmap.h>

hit_tileNewType( double );

/* A. SEQUENTIAL: CELL UPDATE */
static inline void updateCell( double *myself, double up, double down, double left, double right ) {
        *myself = ( up + down + left + right ) / 4;
}

/* B. MAIN: CELLULAR AUTOMATA */
int main(int argc, char *argv[]) {

        hit_comInit( &argc, &argv );
        int rows, columns, numIter;

        /* 1. CREATE VIRTUAL TOPOLOGY */
        HitTopology topo = hit_topology( plug_topArray2DComplete );

        /* 2. DECLARE FULL MATRIX WITHOUT MEMORY */
        HitTile_double matrix;
        hit_tileDomain( &matrix, double, 2, rows, columns );

        /* 3. COMPUTE PARTITION */
        HitShape parallelShape = hit_tileShape( matrix );
        parallelShape = hit_shapeExpand( parallelShape, 2, -1 );
        HitLayout matLayout = hit_layout( plug_layBlocks, topo, parallelShape );

        /* 4. ACTIVE PROCESSES */
        if ( hit_layImActive( matLayout ) ) {

                /* 4.1. CREATE AND ALLOCATE LOCAL TILES */
                /* 4.1.1. LOCAL TILE WITH SPACE FOR FOREIGN DATA */
                HitTile_double tileMat, tileCopy, tileInner;
                HitShape expandedShape = hit_shapeDimExpand( hit_layShape(matLayout), 0,
                                                             HIT_SHAPE_BEGIN, -1 );
                expandedShape = hit_shapeDimExpand( expandedShape, 0, HIT_SHAPE_END, 1 );
                expandedShape = hit_shapeDimExpand( expandedShape, 1, HIT_SHAPE_BEGIN, -1 );
                expandedShape = hit_shapeDimExpand( expandedShape, 1, HIT_SHAPE_END, 1 );
                hit_tileSelect( &tileMat, &matrix, expandedShape );
                hit_tileAlloc( &tileMat );

                /* 4.1.2. COPY OF THE TILE, TO AVOID SEQUENTIAL SEMANTICS ON UPDATES */
                hit_tileSelect( &tileCopy, &tileMat, HIT_SHAPE_WHOLE );
                hit_tileAlloc( &tileCopy );

                /* 4.1.3. SUBSELECT THE INNER PART TO UPDATE */
                hit_tileSelectArrayCoords( &tileInner, &tileMat, hit_layShape(matLayout) );

                /* ... */
        }

        /* 6. FREE OTHER RESOURCES */
        hit_layFree( matLayout );
        hit_topFree( topo );
        hit_comFinalize();
        return 0;
}
```

Figura 2.7: Algoritmo de *Cellular Automata* en Hitmap.

```
/* 4.2. COMMUNICATION PATTERN */
hit_comTagSet( TAG_UP, TAG_DOWN, TAG_LEFT, TAG_RIGHT );
HitPattern neighSync = hit_pattern( HIT_PAT_UNORDERED );
hit_patternAdd( &neighSync,
                hit_comSendRecvSelectTag( matLayout,
                                          hit_layNeighbor( matLayout, 0, -1 ) , &tileMat,
                                          hit_shape( 2, hit_sigIndex(1),
                                                  hit_sig( 1, hit_tileDimCard(tileMat,1)-2, 1) ),
                                          HIT_COM_TILECOORDS,
                                          hit_layNeighbor( matLayout, 0, +1 ) , &tileMat,
                                          hit_shape( 2,
                                                  hit_sigIndex( hit_tileDimCard(tileMat,0)-1 ),
                                                  hit_sig( 1, hit_tileDimCard(tileMat,1)-2, 1) ),
                                          HIT_COM_TILECOORDS,
                                          HIT_DOUBLE, TAG_UP )
                                          );
hit_patternAdd( &neighSync,
                hit_comSendRecvSelectTag( matLayout,
                        hit_layNeighbor( matLayout, 0, +1 ) , &tileMat,
                        hit_shape( 2, hit_sigIndex( hit_tileDimCard(tileMat,0)-2 ),
                                                  hit_sig( 1, hit_tileDimCard(tileMat,1)-2, 1) ),
                                          HIT_COM_TILECOORDS,
                                          hit_layNeighbor( matLayout, 0, -1 ) , &tileMat,
                                          hit_shape( 2, hit_sigIndex(0), hit_sig( 1,
                                                      hit_tileDimCard(tileMat,1)-2, 1) ),
                                          HIT_COM_TILECOORDS,
                                          HIT_DOUBLE, TAG_DOWN )
                                          );
hit_patternAdd( &neighSync,
                hit_comSendRecvSelectTag( matLayout,
                                          hit_layNeighbor( matLayout, 1, -1 ) , &tileMat,
                                          hit_shape( 2, hit_sig( 1,
                                                  hit_tileDimCard(tileMat,0)-2, 1), hit_sigIndex(1) ),
                                          HIT_COM_TILECOORDS,
                                          hit_layNeighbor( matLayout, 1, +1 ) , &tileMat,
                                          hit_shape( 2, hit_sig( 1, hit_tileDimCard(tileMat,0)-2, 1),
                                                  hit_sigIndex( hit_tileDimCard(tileMat,1)-1 ) ),
                                          HIT_COM_TILECOORDS,
                                          HIT_DOUBLE, TAG_LEFT )
                                          );
hit_patternAdd( &neighSync,
                hit_comSendRecvSelectTag( matLayout,
                                          hit_layNeighbor( matLayout, 1, +1 ) , &tileMat,
                                          hit_shape( 2, hit_sig( 1, hit_tileDimCard(tileMat,0)-2, 1),
                                                  hit_sigIndex( hit_tileDimCard(tileMat,1)-2 ) ),
                                          HIT_COM_TILECOORDS,
                                          hit_layNeighbor( matLayout, 1, -1 ) , &tileMat,
                                          hit_shape( 2, hit_sig( 1, hit_tileDimCard(tileMat,0)-2, 1),
                                                  hit_sigIndex(0) ),
                                          HIT_COM_TILECOORDS,
                                          HIT_DOUBLE, TAG_RIGHT )
                                          );
```

Figura 2.8: Patrones de comunicación de un *Cellular Automata* con Hitmap.

```
/* 4.3. INITIALIZE MATRIX (IN PARALLEL) */
int i,j;

initMatrix( tileMat );
/* 4.4. COMPUTATION LOOP */
int loopIndex;
for (loopIndex = 0; loopIndex < numIter; loopIndex++) {

        /* 4.4.1. UPDATE TILE COPY */
        hit_tileUpdateFromAncestor( &tileCopy );

        /* 4.4.2. SEQUENTIALIZED LOOP */
        for ( i=1; i<hit_tileDimCard( tileMat, 0 )-1; i++ )
                for ( j=1; j<hit_tileDimCard( tileMat, 1 )-1; j++ )
                        updateCell(
                                & hit_tileElemAt( tileMat, 2, i, j ),
                                hit_tileElemAt( tileCopy, 2, i-1, j ),
                                hit_tileElemAt( tileCopy, 2, i+1, j ),
                                hit_tileElemAt( tileCopy, 2, i, j-1 ),
                                hit_tileElemAt( tileCopy, 2, i, j+1 )
                                );

        /* 4.4.3. COMMUNICATE */
        hit_patternDo( neighSync );

}
```

Figura 2.9: Computación secuencial de un *Cellular Automata* con Hitmap.

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. Otro modelo interesante es Parray [18]. Este introduce 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.

Existen multitud de frameworks basados en el *polyhedral model* capaces de generar código paralelo para sistemas de memoria compartida como PoCC [5], PolyOpt [6], LLVM's [7], Polly [8] o IBM's XL compiler [9].

URUK/WRaP-IT [20, 31] y CHiLL [17] son otras herramientas poliédricas conocidas que usan una representación poliédrica para realizar optimizaciones en diferentes niveles. Sin embargo estas herramientas son semi-automáticas ya que requieren especificaciones manuales realizadas por un experto.

Además, algunas herramientas poliédricas están orientadas o capacitadas para generar código para sistemas de memoria distribuida. El problema principal es que quedan limitados a las restricciones del modelo. Ferner [27] describe un método para el cálculo automático de comunicaciones en sistemas de memoria distribuida. Esta aproximación se basa en la técnica de paralelización de Lim y Lam [36]. En [37] se propone un algoritmo para la elección de buenas particiones de datos con el objetivo de obtener una buena reutilización de datos y la preservación de paralelismo inherente en el código fuente. En [19] presentan un método para la generación eficiente de código que es ejecutable en clústers de memoria distribuida. En este trabajo presentan un esquema para comunicaciones donde usan *tiling* para agregar datos en los buffers que serán transmitidos. Además han implementado la vectorización de los mensajes. Faber en [23, 24] describe un método que se basa también en el modelo poliédrico. Genera código escrito en HPF [34] que incluye anotaciones de cómo los datos se distribuyen entre los procesadores. El compilador HPF genera las comunicaciones necesarias para sistemas de memoria distribuida. Sin embargo, la aplicación de este método está restringida a HPF y a los fallos o limitaciones de este lenguaje. Otra de las herramientas más citadas es Pluto [14]. Pluto posee una versión experimental para la generación de código para sistemas distribuidos a partir de código secuencial. Sin embargo, a diferencia de Trasgo, este realiza la partición en el dominio de iteraciones a realizar en lugar de en el dominio de datos.

Los trabajos presentados en [21, 22, 40, 41] proponen una aproximación basada en inspector-ejecutor(I/E) para la generación de código para aplicaciones de cálculo numérico con patrones de acceso irregulares ampliando así el rango de aplicaciones en el que usar el *polyhedral model*. SPF es un framework capaz de realizar estas transformaciones. En [39] se propone una aproximación basada en este modelo para la ejecución en paralelo en un sistema de memoria distribuido para una computación de un bucle irregular. Este

modelo usa una combinación de análisis en tiempo de compilación y en run-time. Con algoritmos que analizan el código secuencial y generan un inspector y un ejecutor. El inspector capturará el comportamiento de las dependencias de datos cuando se ejecutan en paralelo y el ejecutor realizará la computación en paralelo.

Otros métodos fuera del modelo poliédrico han sido propuestos. En [11] introducen un método para el cálculo de comunicaciones en un sistema distribuido. Para ello, utilizan un análisis del flujo de datos para determinar el lugar exacto del valor leído en cada acceso. No llegan a generar código desde un alto nivel y necesitan una descripción de cómo la computación debe ser particionada entre los procesadores de la máquina.

El *polyhedral model* es sólo 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.

Capítulo 3

Trabajo desarrollado

3.1. Contribuciones

En este trabajo se presentará y desarrollará un método de programación eficiente para sistemas de memoria compartida y distribuida con la biblioteca Hitmap. Se presentará una metodología con la que el usuario programador será capaz de generar sin gran esfuerzo códigos con un rendimiento similar a programas optimizados por un experto. Para que el programador consiga estos códigos de forma simple se han implementado herramientas automáticas capaces de transformar código secuencial en código optimizado y paralelizado para memoria compartida.

La coordinación de los procesos en el nivel de memoria distribuida se realiza con un menor esfuerzo de desarrollo mediante el uso de Hitmap. Hitmap es una biblioteca de funciones diseñada para el reparto y *mapping* jerárquico de estructuras de datos. La principal contribución de este trabajo se resume en la creación de una metodología para la integración de programas Hitmap con modelos de programación y herramientas automáticas de paralelización y optimización orientadas a memoria compartida. Con esta metodología podremos generar programas paralelos multinivel de una forma más simple y automática. Estos programas adaptarán su estructura de comunicación y sincronización a la máquina donde se ejecuten, explotando de la forma más eficiente las ventajas de cada modelo en un sistema de memoria jerárquico.

En concreto en este trabajo presentaremos unas directrices y ejemplos de como usar el paradigma de programación de 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 en cada proceso MPI. Además, también mostraremos como integrar una herramienta experimental basada en técnicas asociadas al *polyhedral model* para automatizar y optimizar las partes secuenciales. Esta herramienta es necesaria para paralelizar de manera automática aplicaciones donde existen dependencias de datos. Nuestros resultados experimentales mostrarán que los programas híbridos generados con Hitmap mejoran los mejores resultados obtenidos con los programas de referencia manualmente desarrollados y optimizados.

3.2. Programación multi-nivel

Hitmap propone un modelo de programación que proporciona un entorno para la coordinación eficiente de las particiones de datos, *mapping* y comunicaciones en un sistema de memoria distribuida. Los datos, tamaños de estos, y los límites de los subespacios locales serán uniformemente adaptados en función del contenido de los objetos de layout que ha sido generado intérnamente por Hitmap a la plataforma donde se ejecuta. Con 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.

El código dentro de cada tarea local puede ser paralelizado con el paradigma de memoria compartida de una manera sistemática. Esta paralelización podría ser realizada manualmente por un programador con experiencia usando modelos de programación de memoria compartida. Sin embargo, también pueden ser aplicadas herramientas que generan código paralelo automáticamente para la paralelización en memoria compartida en las zonas secuenciales. Por ejemplo, la figura 2.7 muestra una implementación del algoritmo de *Cellular Automata* en Hitmap que llama a una función secuencial que a su vez realiza una actualización de las celdas. 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 sistemas de memoria compartida y distribuida, pudiendo así explotar de forma más eficiente la plataforma de ejecución.

3.3. Casos de estudio

Para poder entender mejor la propuesta y las ventajas que nos puede ofrecer en los distintos escenarios, describiremos previamente las diferentes aplicaciones sobre las que hemos realizado la experimentación. Los algoritmos que implementaremos serán los de la figura 3.1. Estas aplicaciones abarcan aplicaciones de grano fino (el método de Jacobi a través de un *Cellular Automata*), aplicaciones de grano grueso (Multiplicación de matrices) y aplicaciones que arrastran dependencias entre iteraciones (Gauss Seidel).

3.3.1. Jacobi 2D: Sincronización entre vecinos

El primer caso de estudio que trataremos será una implementación del método de Jacobi en dos dimensiones. Este caso de estudio se trata de la solución de una PDE (Partial Differential Equation) usando el método iterativo de Jacobi para calcular 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 actualización y comunicación de datos. 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 la realizaremos con un patrón de comunicaciones *neighbor synchronization* (sincronización entre vecinos) en una topología virtual de dos dimensiones. Los datos que

```
** 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.
           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+Compararemos1][j] +
                         Mat[i][j-1] + Mat[i][j+1] ) / 4;
```

Figura 3.1: Algoritmos de los tres casos de estudio

necesita cada proceso son los bordes comunes de la matriz local de cada uno de los procesos vecinos. La matriz local será expandida con una columna o fila más en cada dirección y sentido para poder almacenar los elementos que provienen de los procesos vecinos. Estas filas o columnas extras son las que hemos denominado *ghost zones*. Si usásemos solo modelos de programación para memoria compartida el espacio de memoria extra no sería necesario. En cambio, sería necesaria una sincronización para asegurar que cada hilo no está actualizando datos que otro hilo esta aun utilizando, o datos que no se encuentran actualizados en las zonas compartidas de los bordes.

3.3.2. Gauss Seidel: Wave-front pipelining

Este caso de estudio computa la misma ecuación de calor que el *Cellular Automata*, pero esta vez usando un método iterativo de Gauss-Seidel. En este método la convergencia es acelerada ya que usa valores ya computados durante la iteración actual. Este método no utiliza una copia para guardar los valores antiguos. Así, cuando se va a actualizar una celda se usan los valores ya actualizados de los vecinos de arriba y de la izquierda de la celda en 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 será realizada con bucles que arrastran dependencias en cada iteración. Con esto se genera una aplicación de tipo *wave-front*.

En modelos de programación de memoria distribuida la solución será similar a la del 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 recibirá los datos de los vecinos de abajo y de la derecha y realizará todos los envíos. Esta última comunicación se realizará después de la computación de los datos locales. Estos patrones generan un *pipeline* controlado por el flujo de datos.

3.3.3. Multiplicación de matrices: Algoritmos multinivel

En la figura 3.1 se muestran los dos algoritmos para la multiplicación de matrices usados en este trabajo. El primero es una aproximación clásica secuencial basada en 3 bucles anidados. Este algoritmo puede ser paralelizado 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 o a realizar comunicaciones innecesarias.

Mientras, el algoritmo de Cannon trabaja con una partición de las matrices de tamaño *k x k* usando solo una pieza local por cada proceso y un patrón de comunicaciones *shift* circular para mover los datos entre los procesos [16]. Este algoritmo encaja en entornos de memoria distribuida pero es más complejo de programar e introduce movimientos de datos innecesarios en entornos de memoria compartida. Así, la mejor aproximación para sistemas jerárquicos híbridos será una combinación de ambos algoritmos.

3.4. Metodología para implementar el paradigma de memoria compartida en un código Hitmap

En esta sección mostraremos a través de ejemplos basados en el modelo de programación OpenMP, la forma de explotar el paradigma de memoria compartida en un programa Hitmap. El primer paso será identificar la parte secuencial del código. Esto podrá ser hecho manualmente o automáticamente ya que siempre se encontrará entre las funciones de Hitmap que ejecutan los patrones de comunicación. Estas son *hit_comDo*, *hit_comStart*, *hit_comEnd*, *hit_patternDo*, *hit_patternDoStart* y *hit_patternDoEnd*. El *framework* Trasgo

```
1  /* MATRIX MULTIPLICATION: COMPUTATION STAGE INSIDE CANNON'S */
2  #pragma omp parallel private(loopIndex)
3  {
4       initmatrix(tileA);
5       initmatrix(tileB);
6
7      #pragma omp single
8      { hit_commDo( alignRow ); hit_commDo( alignColumn ); }
9
10     for (loopIndex = 0; loopIndex < loopLimit-1; loopIndex++) {
11        matrixProduct( tileA, tileB, tileC );
12        #pragma omp single
13        hit_patternDo( shift );
14     }
15     matrixProduct( tileA, tileB, tileC );
16     writeResults( tileC );
17 }
18
19 static inline void matrixProduct( HitTile_double A, HitTile_double B, HitTile_double C ) {
20    int i, j, k, ti, tj, tk;
21    int numTiles0 = (int)ceil( hit_tileDimCard( C, 0 ) / tilesize );
22    int numTiles1 = (int)ceil( hit_tileDimCard( C, 1 ) / tilesize );
23    int numTiles2 = (int)ceil( hit_tileDimCard( A, 1 ) / tilesize );
24
25    #pragma omp for collapse(2) private(i,j,k,ti,tj,tk)
26    for (ti=0; ti<numTiles0; ti++) {
27       for (tj=0; tj<numTiles1; tj++) {
28          int begin0 = ti * tilesize;
29          int end0 = min( hit_tileDimCard( C, 0 )-1, (begin0 + tilesize -1) );
30          int begin1 = tj * tilesize;
31          int end1 = min( hit_tileDimCard( C, 1 )-1, (begin1 + tilesize -1) );
32          for (tk=0; tk<numTiles2; tk++) {
33             int begin2 = tk * tilesize;
34             int end2 = min( hit_tileDimCard( A, 1 )-1, (begin2 + tilesize -1) );
35             /* MATRIX TILE BLOCK PRODUCT */
36             for (i=begin0; i<=end0; i++)
37                for (j=begin1; j<=end1; j++)
38                   for (k=begin2; k<=end2; k++)
39                      hit_tileAt2( C, i, j ) += hit_tileAt2( A, i, k ) * hit_tileAt2( B, k, j );
40          }
41       }
42    }
```

Figura 3.2: Computación local de una multiplicación de matrices en un programa Hitmap paralelizado con OpenMP después de aplicar *tiling* y una inversión de bucles.

puede realizar esta identificación de forma automática, ya que el propio código de alto nivel marca el lugar donde se ejecutan las funciones secuenciales (los diferentes *parblocks* dentro de un *parallel*).

Una primera aproximación simple para la introducción de directivas OpenMP en las partes secuenciales puede ser usar las primitivas sincronizadas *omp parallel for* para los bucles de alto coste computacional. Sin embargo, esta solución puede ser ineficiente en bucles anidados dentro de otros. En cada iteración del bucle exterior nuevos threads deben ser creados, distribuidos y destruidos. Esta operación se hace muy costosa en contextos como el descrito.

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 introducirse para controlar los threads durante la inicialización de las estructuras de datos y en los bucles de alto coste computacional. El ejemplo visto en la figura 3.2 muestra la forma de programar una multiplicación de matrices usando Hitmap y OpenMP. La ejecución de los patrones de comunicación que realiza el intercambio de datos entre procesos debe ser realizado sólo por un thread. Por ello, la primitiva *omp single* debe ser introducida en cada llamada a *hit_comDo*, *hit_comStart*, *hit_comEnd*, *hit_patternDo hit_patternDoStart* y *hit_patternDoEnd*. Además, para asegurar que los datos a comunicar son correctos y no comunicamos datos que aun siguen computándose por otro thread es necesario la sincronización de los threads antes y después de la comunicación. En el ejemplo presentado en la figura 3.2, las primitivas *for* y *single* de OpenMP llevan implícita la sincronización de los threads. Es posible realizar el solapamiento de la computación y las comunicaciones usando las funcionalidades dadas por Hitmap para comunicaciones asíncronas como en el caso de estudio del *Cellular Automata* (sección 3.3).

3.5. Modelo de ejecución

El modelo de ejecución de un código Hitmap es simple. El programa debe ser lanzado con un proceso por nodo o por un subconjunto de núcleos. Por ejemplo, usando el lanzador de *hydra* (incluido en las distribuciones típicas de MPI [10]) es posible desplegar tantos procesos MPI como se desee en cada nodo físico. Las políticas de topología y *layout* adaptarán automáticamente la partición de datos y los patrones de comunicación para el nivel de memoria distribuida.

Cada tarea local se configura para lanzar tantos threads como número de núcleos de CPU se hayan asignado a ese proceso. Por ejemplo, usando OpenMP, iniciar el número de threads se realiza directamente usando la función *omp_set_threads_num()*. En la experimentación realizaremos diferentes pruebas variando las configuraciones de número de threads/procesos, usando el valor de un argumento *n* en la linea de comandos utilizada para lanzar los procesos MPI.

3.6. Paralelización y transformación de código

En las últimas decadas se han estudiado múltiples transformaciones de código para la optimización de códigos paralelos y secuenciales, o para simplificar la introducción del paralelismo. Por ejemplo, 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 de partición de datos para minimizar los costes de la comunicación entre procesos. La técnica de *tiling* puede 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 caches. Un ejemplo lo vemos en la figura 3.2.

También pueden ser aplicadas transformaciones de código cuando existen dependencias de datos. Por ejemplo, una aplicación de tipo *wave-front* puede ser fácilmente expresada con bucles secuenciales. Sin embargo, estos no pueden ser fácilmente paralelizados con modelos como OpenMP, donde las primitivas no dan soporte para el control del flujo de datos a través de los threads.

La aproximación más radical es la transformación completa de los bucles. Esta transformación generará un bucle exterior (o combinación de bucles exteriores) que recorra el dominio en una dirección diferente, orientada según las dependencias. Por ejemplo, una aplicación de tipo *wave-front* puede ser transformada en una estructura de *pipeline* donde el bucle exterior avanza a través de las fases del *pipeline* y cada fase puede ser paralelizada de forma trivial al no contener dependencias.

Otra aproximación puede ser el desarrollo manual de un programa con variables de bloqueo en función del número de threads. Esta forma replica la idea de partición de Hitmap y sus sincronizaciones pero en el nivel de memoria compartida.

En el caso de Gauss-Seidel (sección 3.3) es posible generar un bucle que recorra las diagonales de la matriz. Aunque la computación de los datos de una diagonal depende de los datos de la diagonal anterior, las actualizaciones de los datos de una diagonal pueden ser realizadas en paralelo.

3.7. Uso de herramientas del *polyhedral model* para la implementación de memoria compartida

Las transformaciones previamente descritas pueden ser automáticamente aplicadas usando técnicas del *polyhedral model* [12]. Por ejemplo, para la computación de las partes locales de los tres casos presentados en la sección 3.3 es posible la aplicación de estas técnicas en la zona secuencial.

Pluto [15] es un prototipo de compilador 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 función main, declaración de variables compartidas, etc y así poder usarlo en nuestro sistema. Esto nos permite sustituir en nuestro programa las zonas secuenciales de código por los códigos generados por Pluto. La integración de Pluto en códigos Hitmap nos da programas automáticamente optimizados. El sistema de transfor-

```
#pragma scop

for (i1 = 1; i1 <= N1; i1 = i1 + 1)
  {
    for (i2 = 1; i2 <= N2; i2 = i2 + 1)
      {

        m[i1][i2]
          =
          Inm[i1 - 1][i2] + Inm[i1 + 1][i2]
         + Inm[i1][i2 - 1] + Inm[i1][i2 + 1] + 0;
      }

  }

#pragma endscop
```

Figura 3.3: Código de entrada a Pluto

maciones de Trasgo posee un módulo capaz de generar el código de entrada a Pluto a partir del código de alto nivel de entrada a Trasgo. El código generado por Trasgo que será la entrada a Pluto será similar al mostrado en el ejemplo de la figura 3.3. En este código se incluyen variables relativas al dominio de iteraciones, como N1 y N2, que mantendrán sus valores constantes a lo largo de la ejecución de este trozo de código. Estas variables serán inicializadas por el código generado por otros medios de Trasgo, en general en función de las particiones (topología y *layout*) de Hitmap. El *back-end* ejecutará Pluto e incluirá la salida obtenida en la traducción final en el programa Hitmap. La salida de Pluto será similar al ejemplo de código de la figura 3.4 que muestran la traducción del ejemplo 3.3. Podemos ver como este código tiene aplicadas optimizaciones como el *tiling*. Además vemos como el cálculo de actualización que se realizaba en el código de entrada es sustituido por una llamada de una función S1. Será otro módulo de Trasgo el encargado de generar en el código de Hitmap la definición de S1. Está tendrá la siguiente forma para el ejemplo mostrado en 3.3.

```
#define S1( zt1, zt2,_TT_i1 , _TT_i2) updatecell(
                hit_tileElemAt(mCopy, 2 , _TT_i1-1 , _TT_i2 ) ,
                hit_tileElemAt(mCopy, 2 , _TT_i1+1 , _TT_i2 ) ,
                hit_tileElemAt(mCopy, 2 , _TT_i1 , _TT_i2-1 ) ,
                hit_tileElemAt(mCopy, 2 , _TT_i1 , _TT_i2+1 ) ,
                & hit_tileElemAt(m, 2 , _TT_i1 , _TT_i2 ) );
```

Además se ha desarrollado una modificación adicional que permite generar código en el que se encuentre integrada la propuesta de colocación de las directivas de OpenMP que se ha presentado en la sección 3.4. Pluto original genera directivas *omp parallel for* para los bucles transformados. En nuestra propuesta hemos modificado esas directivas, como se puede ver en el código 3.4, por simplemente directivas *omp for* ya que los threads les crearemos una sola vez al inicio de la sección secuencial.

Como hemos visto a lo largo de esta sección, Pluto puede ser usado para realizar optimizaciones automáticas de las partes secuenciales e integrarlas posteriormente en un

```
#include <omp.h>
#include <math.h>
#define ceild(n,d)  ceil(((double)(n))/((double)(d)))
#define floord(n,d) floor(((double)(n))/((double)(d)))
#define max(x,y)    ((x) > (y)? (x) : (y))
#define min(x,y)    ((x) < (y)? (x) : (y))




  int t1, t2, t3, t4;

 int lb, ub, lbp, ubp, lb2, ub2;
 register int lbv, ubv;

/* Start of CLooG code */
if ((N1 >= 1) && (N2 >= 1)) {
  lbp=0;
  ubp=floord(N1,32);
#pragma omp for private(lbv,ubv)
  for (t1=lbp;t1<=ubp;t1++) {
    for (t2=0;t2<=floord(N2,32);t2++) {
      for (t3=max(1,32*t1);t3<=min(N1,32*t1+31);t3++) {
        lbv=max(1,32*t2);
        ubv=min(N2,32*t2+31);
#pragma ivdep
#pragma vector always
        for (t4=lbv;t4<=ubv;t4++) {
          S1(t2,t1,t3,t4);
        }
      }
    }
  }
}
/* End of CLooG code */
```

Figura 3.4: Código de de salida de Pluto

programa Hitmap para facilitar la introducción de 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 pueda explotar de una manera eficiente entornos de memoria distribuida-compartida que adapten su estructura a la plataforma de ejecución.

Capítulo 4

Experimentación y Análisis de resultados

Se han desarrollado una serie de experimentos para validar las ventajas de nuestra propuesta de metodología para la introducción de primitivas OpenMP en un programa Hitmap. Para ello compararemos los resultados con códigos de referencia manualmente desarrollados y optimizados programados usando MPI para memoria distribuida y OpenMP para memoria compartida con los códigos híbridos. Con estas comparaciones podremos observar las mejoras de rendimiento que ofrece nuestra propuesta. Además, también se realizarán experimentos con los códigos generados con Pluto para demostrar que además de ventajas de facilidad en programación la herramienta nos ofrece ventajas en el rendimiento.

4.1. Metodología de la experimentación

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. Los experimentos serán lanzados en dos plataformas:

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 dispone de 256 GB de RAM.

2. **Caléndula**: Caléndula es un clúster híbrido perteneciente a la *Fundación Centro de Supercomputación de Castilla y León*. Los nodos del clúster están conectados mediante la tecnología Infiniband. Además cada nodo tiene dos CPUs Intel Xeon 545 con 4 núcleos cada CPU. Usando 8 nodos del clúster entonceCompararemoss explotaremos 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*. Para el nivel de paralelismo asociado a memoria distribuida usaremos *mpich2 v3.0.4* como implementación de MPI en Atlas y *OpenMPI v1.6.5* en Caléndula.

Con respecto al tamaño de datos para las aplicaciones seleccionadas, hemos elegido un tamaño suficiente para producir una carga computacional relevante cuando usamos 64 unidades computacionales en memoria distribuida. Para los ejemplos de Jacobi y Gauss-Seidel presentaremos los resultados para ejecuciones en las que se realizarán las primeras 200 iteraciones del bucle principal en matrices de 8000x8000 elementos. Según se describe en cada ejemplo, 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 principales que son el objetivo de estudio en este trabajo. 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 un número de threads T = 4, 8, 16, 32 y 64. En Caléndula los programas de referencia de MPI han sido ejecutados con un número de procesos P = 4, 8, 16, 32 y 64. Los procesos han sido distribuidos por los nodos dependiendo del número de threads lanzados por proceso MPI. Los núcleos de cada nodo serán completados con procesos o threads antes de lanzar más procesos en otro nodo diferente. Los programas Hitmap se ejecutan solo con P procesos MPI 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) o con 8 threads por proceso (P x 8). Algunas de las configuraciones de procesos y threadas propuestas no serán válidas en el caso de estudio de la multiplicación de matrices debido a la restricción impuesta por el algoritmo de Cannon: es necesario que la topología de los procesos MPI sea un cuadrado perfecto. Para este algoritmo lanzaremos 4, 16, 36 y 64 procesos MPI.

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

4.2. Jacobi 2D: Sincronización entre vecinos

El primer caso de estudio que trataremos será una implementación del método de Jacobi en dos dimensiones. Para este caso analizaremos los siguientes códigos:

- **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**: Serán los códigos de **Hitmap** pero habiendo sustituido la parte secuencial por el código secuencial generado por Pluto sin modificar según nuestra propuesta completa.
- **Hitmap+Pluto**: Serán los códigos de **Hitmap** pero usando las directivas OpenMP en el lugar en el que describimos en nuestra propuesta y usando como parte secuencial la versión optimizada por nuestra modificación de Pluto.

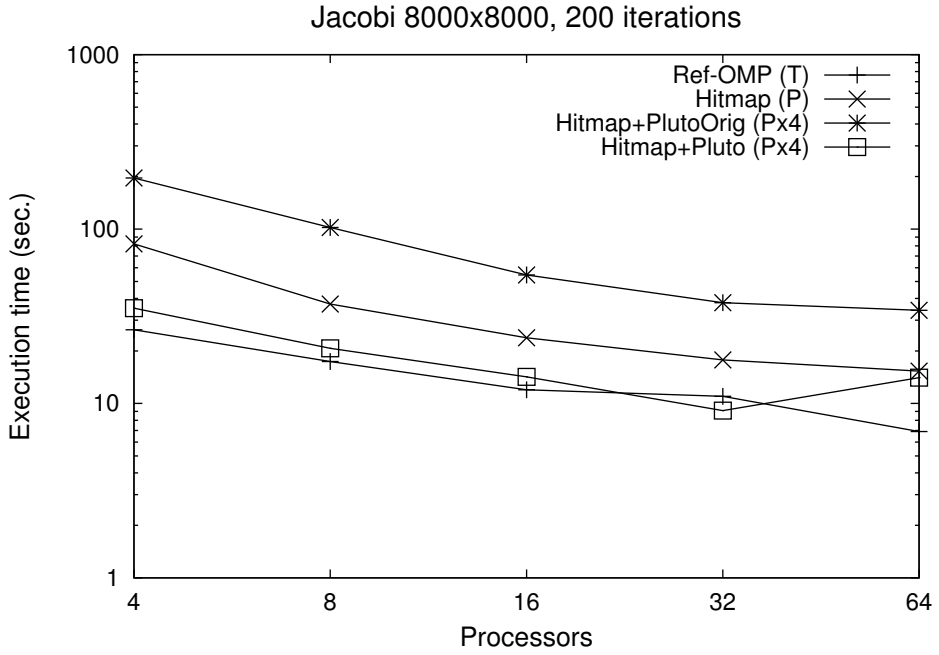

Figura 4.1: Resultados de rendimiento en Atlas para Jacobi 2D (plataforma de memoria compartida)

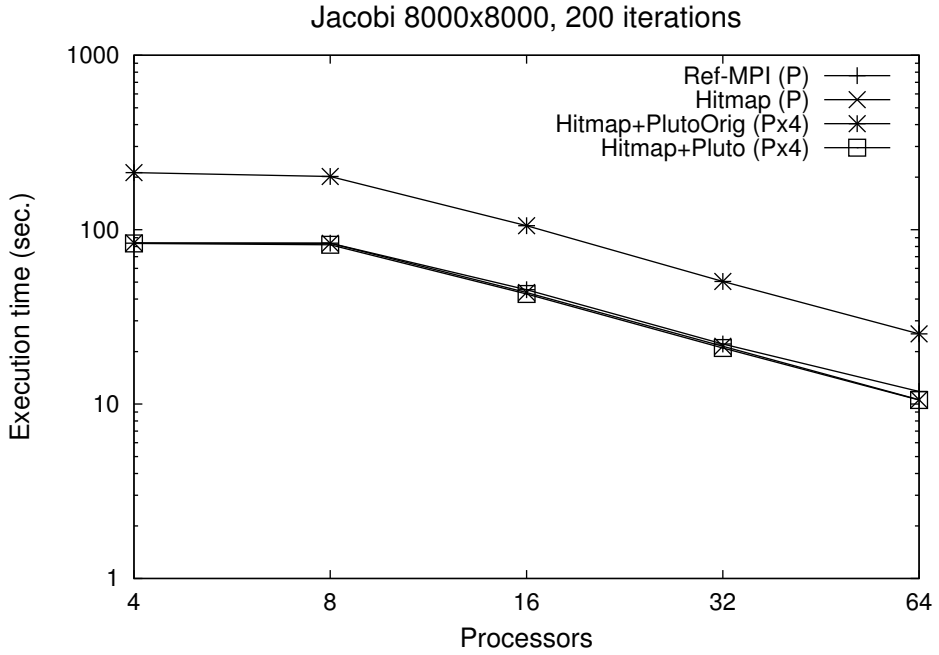

Figura 4.2: Resultados de rendimiento en Caléndula para Jacobi 2D (plataforma de híbrida)

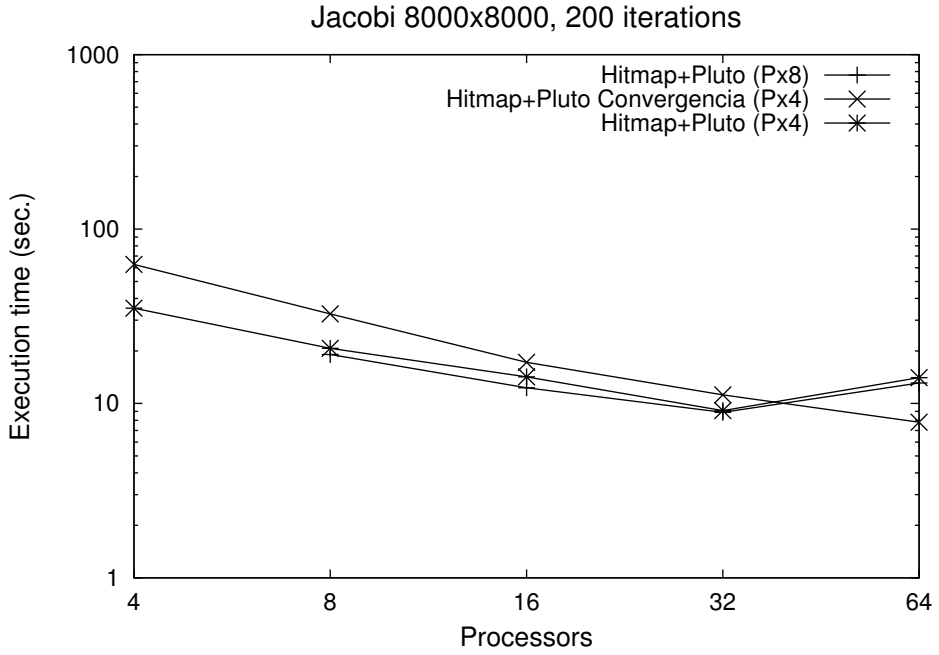

Figura 4.3: Resultados de rendimiento en Atlas para Jacobi 2D (plataforma de memoria compartida)

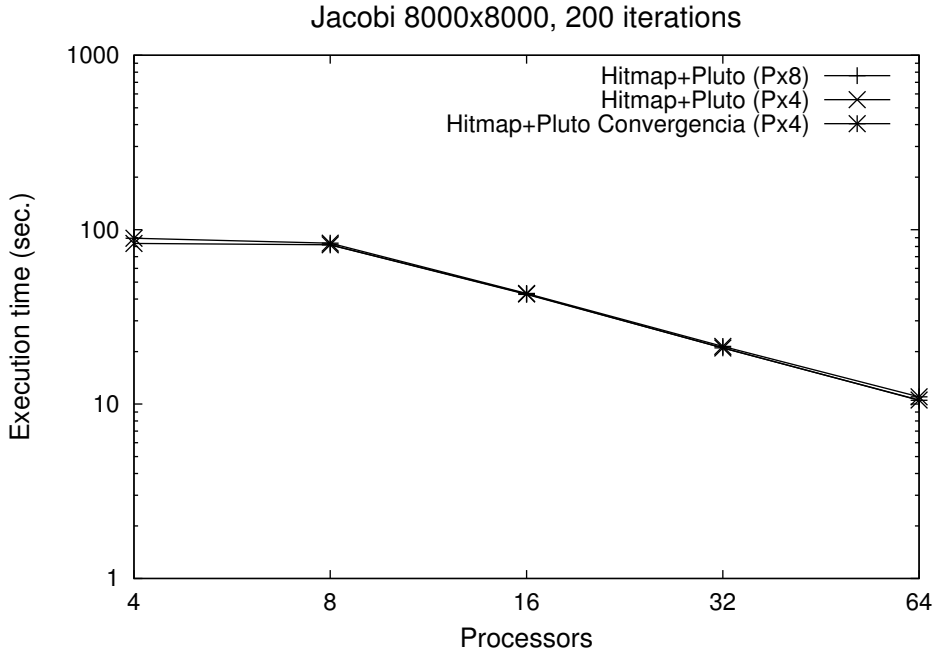

Figura 4.4: Resultados de rendimiento en Caléndula para Jacobi 2D (plataforma de híbrida)

- **Hitmap+Pluto Convergencia**: Serán los códigos de **Hitmap** pero usando las directivas OpenMP en el lugar en el que describimos en nuestra propuesta y usando como parte secuencial la versión optimizada por nuestra modificación de Pluto. Además se ha incluido la condición de convergencia para comprobar si la inclusión de la operación de reducción necesaria afecta a los resultados, aunque con 200 iteraciones no llegara a converger.

Como podemos ver en la figura 4.1, nuestra propuesta mejora los resultados del código en el que simplemente se lanzan procesos (**Hitmap**) en un entorno de memoria compartida. Vemos como el código **Hitmap+Pluto** es capaz de obtener similares resultados que la referencia de OpenMP.

En la figura 4.2 vemos como la nueva propuesta no ha introducido pérdidas ni grandes mejoras en el rendimiento de esta aplicación en un entorno distribuido.

Además se observa en ambas figuras (4.1 y 4.2) como el código de Pluto sin la aplicación de nuestra propuesta da peores resultados que el resto de códigos.

También se ha comprobado que aumentando el número de hilos por procesador se obtienen similares resultados como podemos ver en las figuras 4.3 y 4.4 donde se ha comparado experimentos lanzados con 4 y 8 hilos por proceso MPI.

En la figura 4.3 y 4.4 se han incluido además los resultados del código con la condición de convergencia. Podemos ver como este da resultados levemente peores en la plataforma de memoria compartida debido a que la operación de reducción está en cada iteración, pero en cambio se observa una escalabilidad más regular.

4.3. Gauss Seidel: Wave-front pipelining

Este caso de estudio computa la misma ecuación de calor que en la sección 4.2, pero esta vez usando un método iterativo de Gauss-Seidel. En este método la convergencia es acelerada ya que usa valores ya computados durante la iteración actual.

El caso de Gauss Seidel necesita un análisis más profundo debido a que este caso de estudio no se puede paralelizar en memoria compartida con OpenMP directamente al arrastrar dependencias (unos datos deben ser computados antes que otros para obtener los resultados correctos). Es posible realizar la implementación para memoria compartida de dos formas diferentes:

- Subdividiendo el dominio en trozos, controlando el flujo de datos mediante semáforos. Este método imita el algoritmo para memoria distribuida. Es más complejo de programar y no es fácil realizarlo de forma automática.

- Transformando los bucles que recorren el dominio para que lo recorran de una forma en la que los bucles exteriores no arrastren dependencias y que pueda ser paralelizada de forma trivial. Estas transformaciones pueden no obtener tanto rendimiento como el otro método. Sin embargo, cuenta con la ventaja de que puede ser implementado de forma automática usando herramientas basadas en el modelo poliédrico como Pluto.

Por ello los códigos que usaremos en este caso de estudio serán:

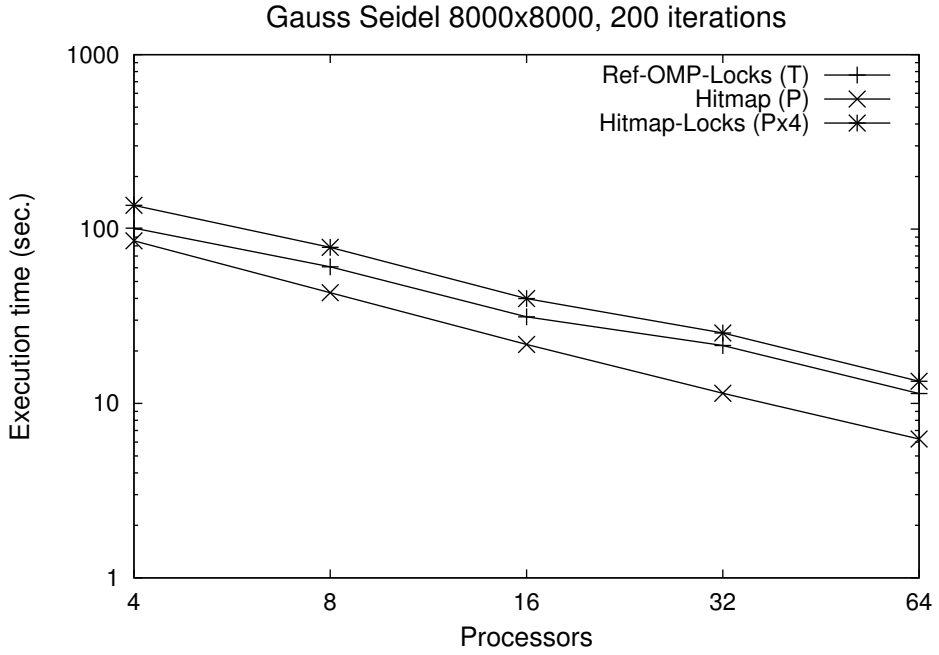

Figura 4.5: Resultados de rendimiento en Atlas para Gauss-Seidel 2D (plataforma de memoria compartida)

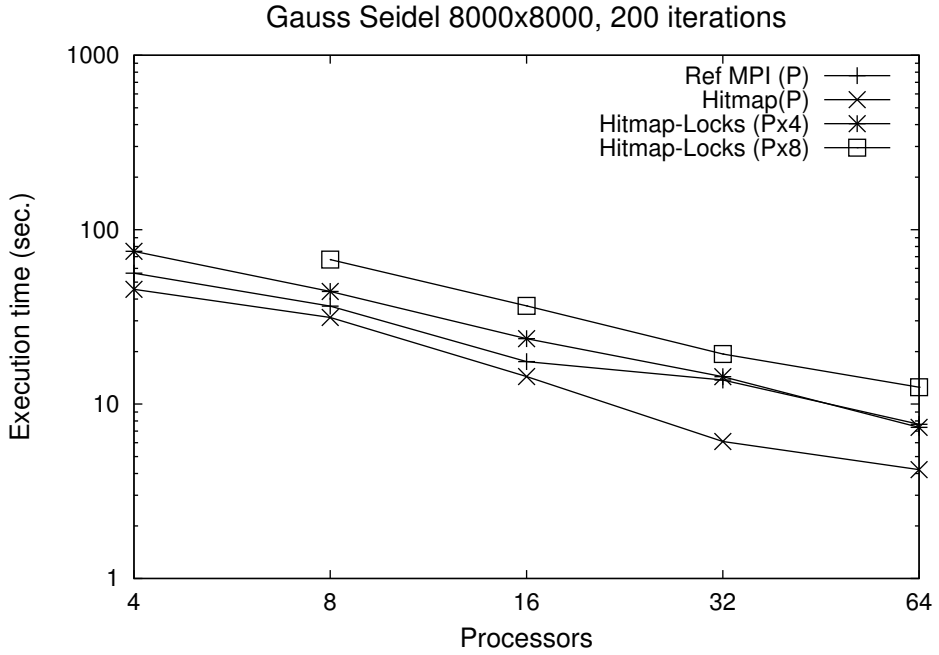

Figura 4.6: Resultados de rendimiento en Caléndula para Gauss-Seidel 2D (plataforma de híbrida)

- **Ref-MPI** y **Ref-OMP**: Códigos paralelizados manualmente para memoria distribuida y compartida respectivamente. En el caso de estudio del Gauss-Seidel la referencia con OpenMP es un *pipeline* construido manualmente usando semáforos y realizando una partición de datos en función del número de threads.

- **Hitmap**: Código paralelizado con Hitmap donde las secciones secuenciales son una transcripción de los códigos de referencia.

- **Hitmap+PlutoOrig**: Serán los códigos de **Hitmap** pero habiendo sustituido la parte secuencial por el código secuencial generado por Pluto.

- **Hitmap+Pluto**: Serán los códigos de **Hitmap** pero usando las directivas OpenMP en el lugar en el que describimos en nuestra propuesta y usando como parte secuencial la versión optimizada por nuestra modificación de Pluto.

- **Hitmap-Locks**: En el ejemplo de Gauss-Seidel también hemos desarrollado un código Hitmap en el que la parte secuencial es una versión adaptada del código de referencia en OpenMP, es decir, en la parte secuencial se realiza una partición de datos en función del número de threads y se controla el *pipeline* usando semáforos.

- **Hitmap+Pluto Convergencia**: Serán los códigos de **Hitmap** pero usando las directivas OpenMP en el lugar en el que describimos en nuestra propuesta y usando como parte secuencial la versión optimizada por nuestra modificación de Pluto. Además se ha incluido la condición de convergencia para comprobar el impacto de la operación de reducción necesaria en cada iteración (aunque con 200 iteraciones no llegara a converger).

En las figuras 4.5 y 4.6 comparamos el rendimiento obtenido por **Hitmap** solo cuando lanzamos procesos MPI, **Hitmap-Locks** lanzando diferentes números de hilos por cada proceso MPI y la referencia de OpenMP o MPI dependiendo de la plataforma donde se ejecute. Podemos ver como los mejores resultados se obtienen solo cuando usamos procesos MPI sin hilos. Esto es debido a que un programa *pipeline* es más apropiado para la comunicación entre procesos que para modelos de programación de memoria compartida.

En las figuras 4.8 y 4.9 vemos los resultados de los códigos una vez introducidos los códigos generados por nuestra versión de Pluto modificada. Podemos ver que al igual que en otros casos de estudio nuestra propuesta proporciona mejores resultados que usando la herramienta original de Pluto. La razón es porque en nuestra propuesta los threads son lanzados simplemente una vez y coordinados con primitivas de memoria compartida en la parte de memoria compartida del programa.

Además en este caso vemos como obtenemos mejores resultados con Pluto cuando lanzamos simplemente procesos MPI. Es decir, el código Hitmap transformado por Pluto mejora el rendimiento comparado con el código original Hitmap cuando sólo lanzamos procesos MPI en ambos.

La razón del mal comportamiento de la mezcla de hilos y procesos en esta aplicación cuando usamos transformaciones de bucles para la implementación de modelos de memoria compartida viene del propio concepto del modelo. Las transformaciones de Pluto hacen que recorramos el dominio por diagonales. Es decir, tendremos un bucle que nos haga saltar de diagonal y otro para recorrer esa diagonal como podemos ver en la figura 4.7.

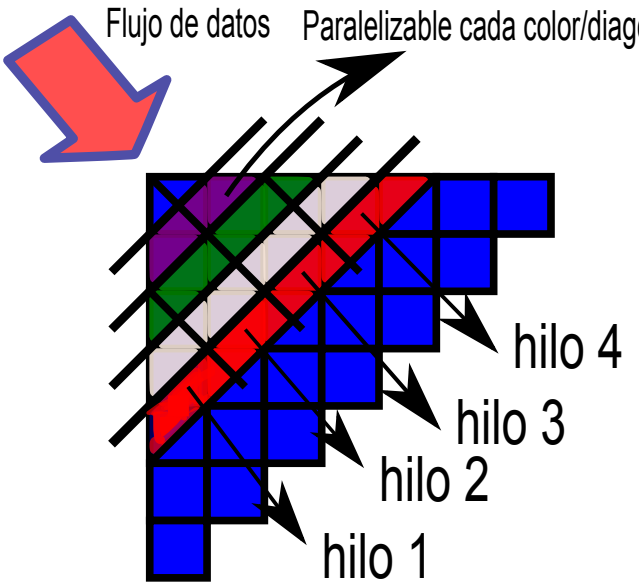

Figura 4.7: Recorrido de un dominio de datos con dependencias de modo que pueda paralelizarse

Así podemos paralelizar el recorrido de la diagonal ya que en esta no existe dependencias de datos. La diagonal mayor será la que más aproveche la paralelización. Sin embargo, el resto de diagonales puede que no tengan suficiente tamaño para explotar eficientemente el número de threads asignados. En las diagonales de pequeño tamaño como son las primeras y las últimas, cuando estas son divididas entre los diferentes threads, los threads son asignados a trozos pequeños. Por ello cuando se aumenta el número de procesos, los trozos asignados a cada proceso serán más pequeños. Esto lleva a tener diagonales más pequeñas haciendo que sea ineficiente la partición de las diagonales pequeñas entre los 4 threads lanzados en la experimentación. Este fenómeno produce también una reducción de la escalabilidad como vemos en los códigos **Hitmap+Pluto** cuando usamos hilos. Además, los datos utilizados en cada iteración no están alineados, por lo que el uso de las caches locales no es el óptimo.

Por último, en las figuras 4.10 y 4.11 vamos a comparar el uso de 4 u 8 hilos por proceso y el retardo que introduce la condición de convergencia. En esta aplicación podemos ver que el uso de 8 hilos por proceso da similares resultados que con 4 hilos.

Además, vemos como la condición de convergencia en este caso sí produce un empeoramiento en la escalabilidad de los resultados.

4.4. Multiplicación de matrices: Algoritmos multinivel

Para el análisis de la propuesta con el caso de estudio de la multiplicación de matrices usaremos los códigos:

- **Ref-MPI** y **Ref-OMP**: Códigos paralelizados manualmente para memoria distribuida y compartida respectivamente.

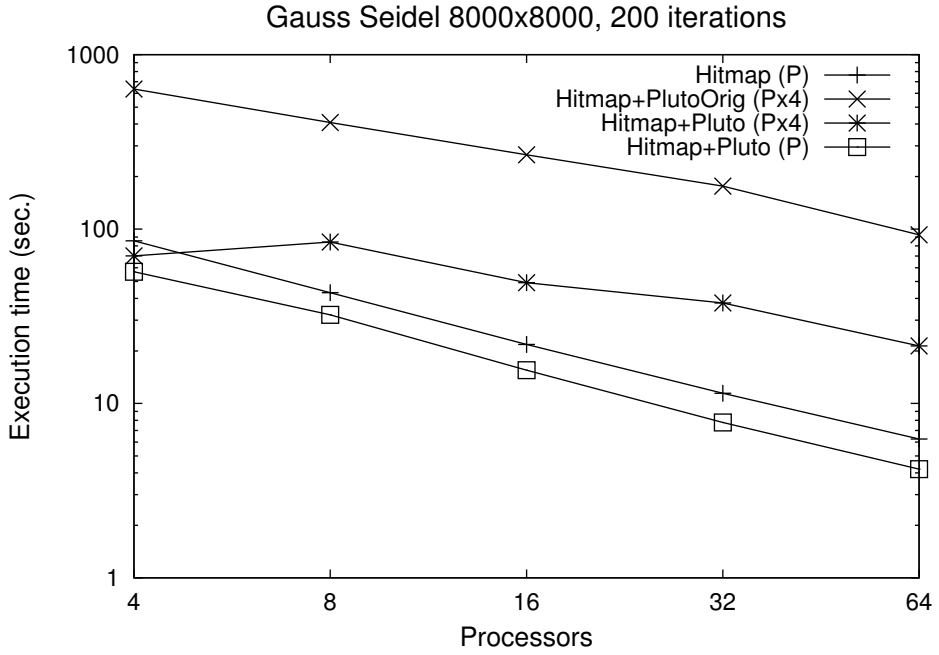

Figura 4.8: Resultados de rendimiento en Atlas para Gauss-Seidel 2D (plataforma de memoria compartida)

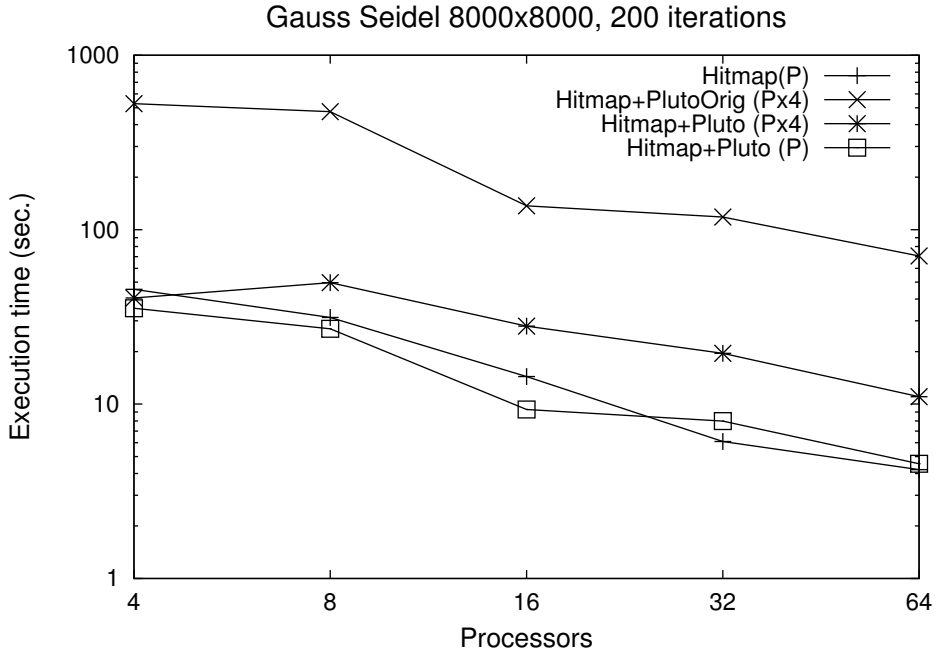

Figura 4.9: Resultados de rendimiento en Caléndula para Gauss-Seidel 2D (plataforma de híbrida)

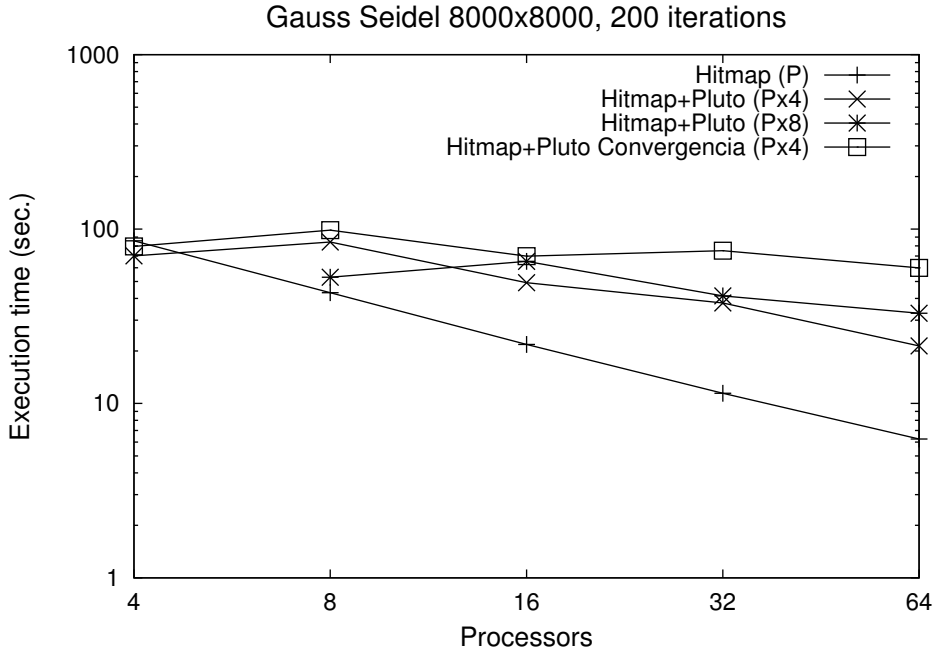

Figura 4.10: Resultados de rendimiento en Atlas para Gauss-Seidel 2D (plataforma de memoria compartida)

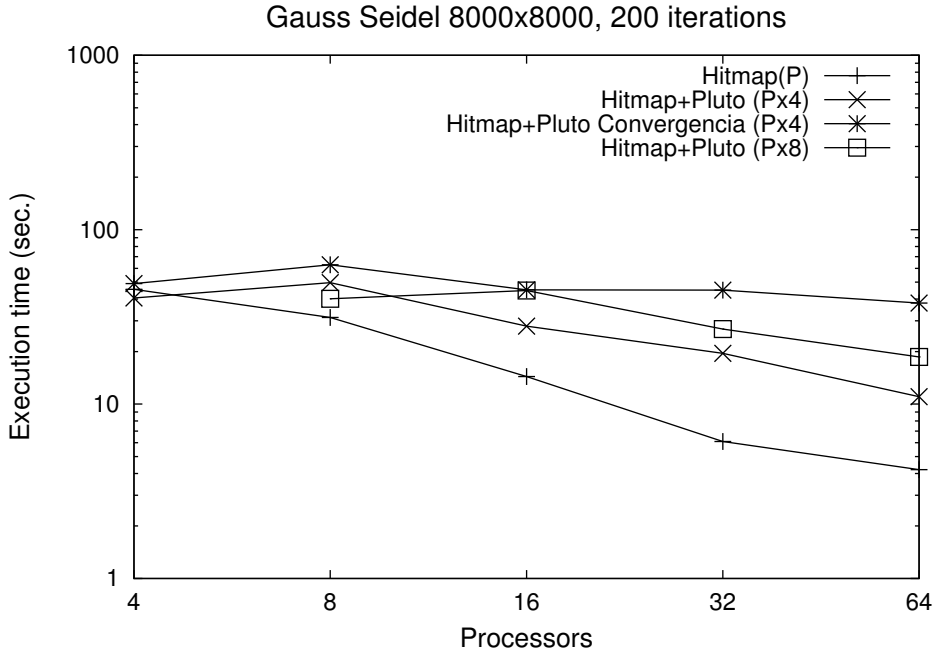

Figura 4.11: Resultados de rendimiento en Caléndula para Gauss-Seidel 2D (plataforma de híbrida)

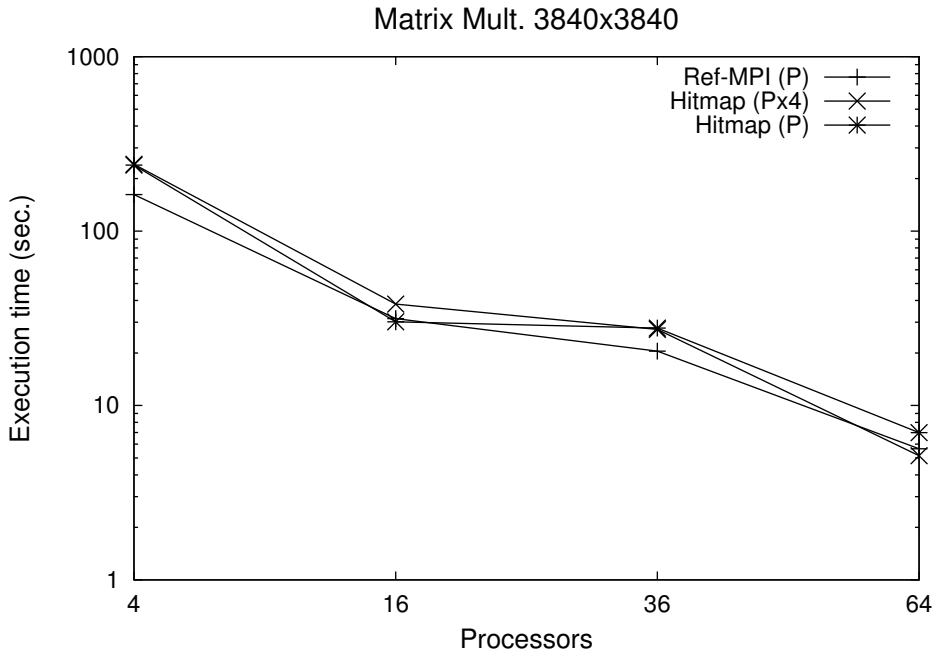

Figura 4.12: Resultados de rendimiento en Atlas para MM (plataforma de memoria compartida)

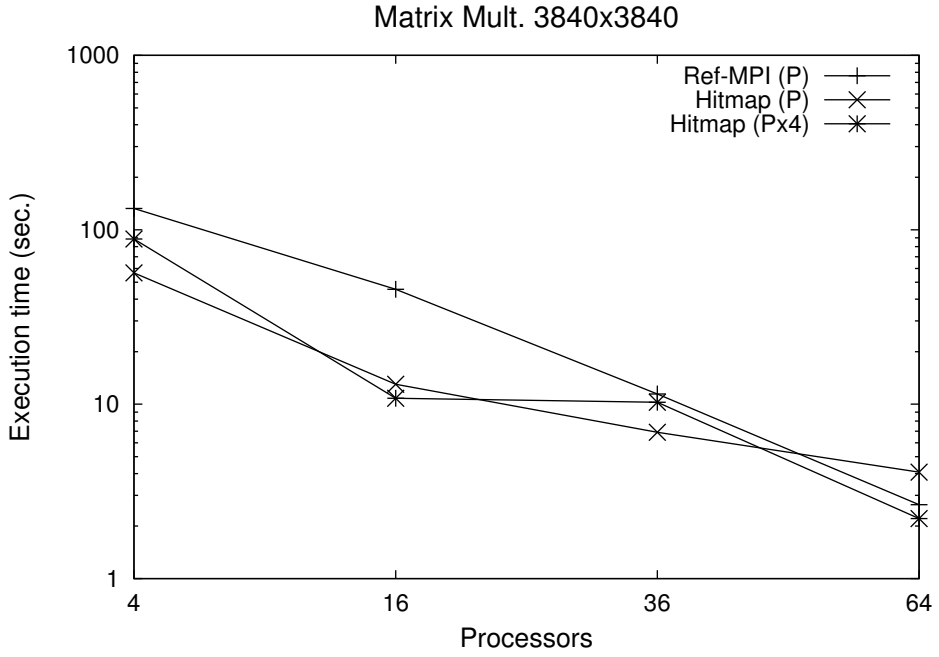

Figura 4.13: Resultados de rendimiento en Caléndula para MM (plataforma híbrida)

- **Hitmap**: Código paralelizado con Hitmap donde las secciones secuenciales son una transcripción de los códigos de referencia.
- **Hitmap+PlutoOrig**: Serán los códigos de **Hitmap** pero habiendo sustituido la parte secuencial por el código secuencial generado por Pluto.
- **Hitmap+Pluto**: Serán los códigos de **Hitmap** pero usando las directivas OpenMP en el lugar en el que describimos en nuestra propuesta y usando de parte secuencial la versión optimizada por nuestra modificación de Pluto.

En una primera aproximación, realizaremos una comparación del código **Ref-MPI** y **Hitmap** con un proceso MPI por procesador y **Hitmap** lanzando 4 hilos por cada proceso MPI. Como podemos ver en la figura 4.12 y 4.13, nuestra propuesta del uso de varios hilos por cada proceso MPI no produce grandes retardos e incluso es la mejor opción para algún número de procesadores. Estas mejoras se pueden observar más claramente en la figura 4.13 ya que el coste de las comunicaciones entre nodos en una plataforma de memoria distribuida es más costoso.

El siguiente estudio lo realizaremos incluyendo los códigos generados por Pluto en Hitmap. En este caso vemos como la introducción de la herramienta realiza un gran mejora en la eficiencia de los códigos. Esto es debido a que Pluto toma el código secuencial y realiza varias optimizaciones. En el caso de la multiplicación de matrices introduce *tiling* y una inversión de bucles que provoca una gran reutilización de datos. Cuando usamos el Pluto original observamos en las figuras 4.14 y 4.15 que obtenemos similares rendimientos a los códigos sin optimizaciones. Sin embargo cuando aplicamos nuestra propuesta a Pluto obtenemos grandes mejoras. Esto confirma la relevancia de nuestra propuesta.

Por últimos hemos realizado una pruebas realizando las mismas optimizaciones realizadas por Pluto pero de forma manual. En la figura 4.16 y 4.17 vemos como conseguimos similares resultados que los obtenidos con Pluto modificado. La única diferencia viene en la dificultad de programación. Mientras que la herramienta que incluimos en Hitmap lo realiza de forma automática, si queremos que los códigos optimizados manualmente obtengan similar rendimiento es necesario que el programador sea un experto con un gran conocimiento en el algoritmo y en técnicas de optimización.

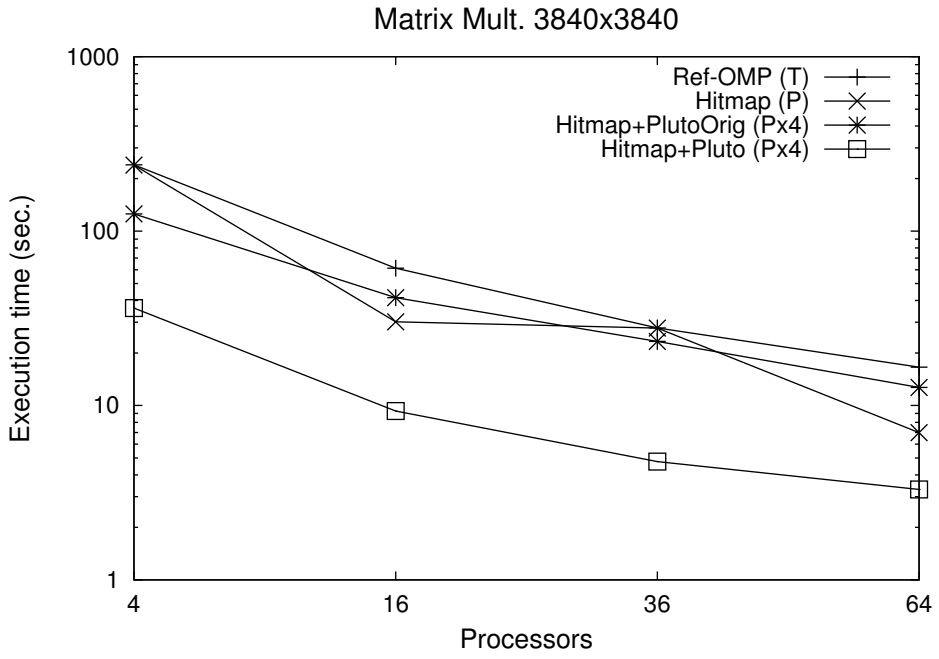

Figura 4.14: Resultados de rendimiento en Atlas para MM (plataforma de memoria compartida)

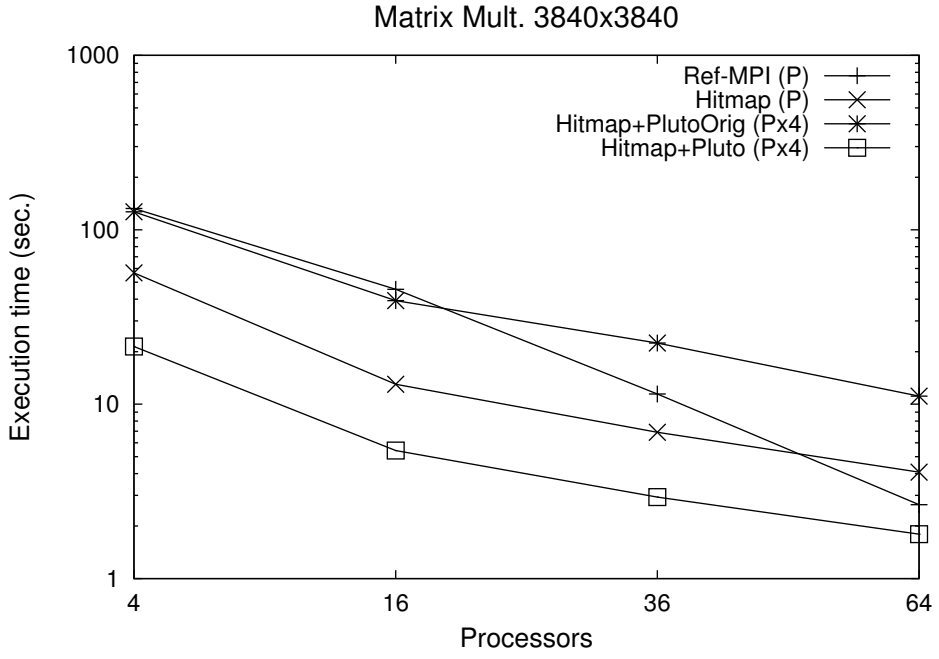

Figura 4.15: Resultados de rendimiento en Caléndula para MM (plataforma híbrida)

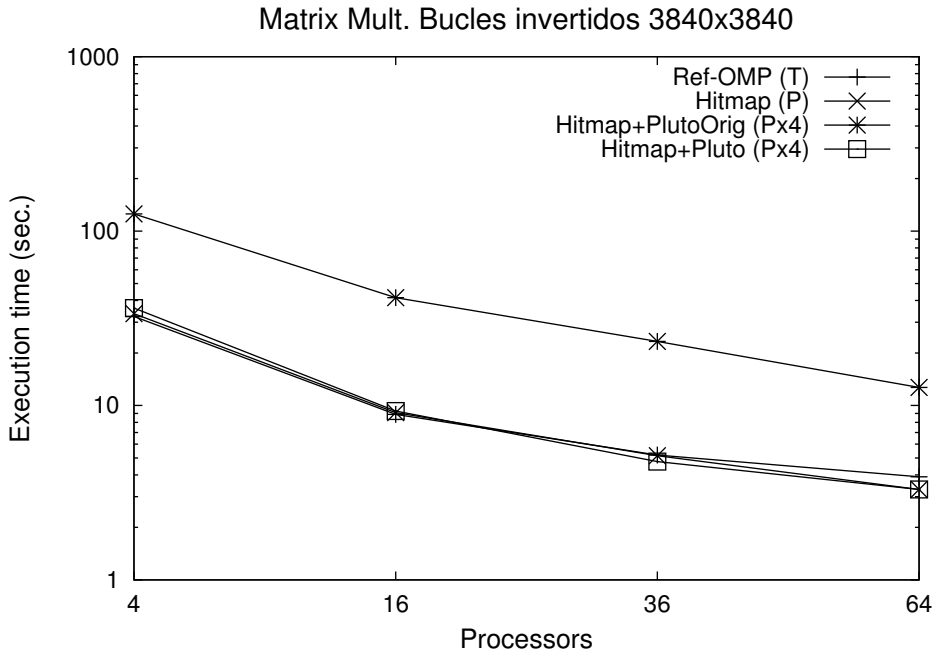

Figura 4.16: Resultados de rendimiento en Atlas para MM (plataforma de memoria compartida)

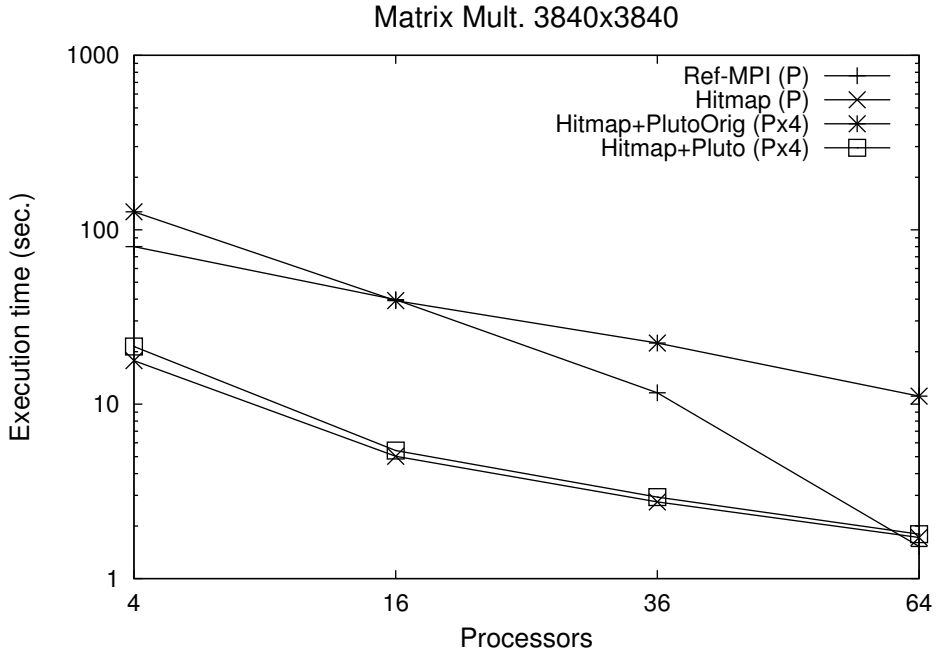

Figura 4.17: Resultados de rendimiento en Caléndula para MM (plataforma híbrida)

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo se ha presentado una nueva metodología para desarrollar programas portables y eficientes para sistemas híbridos de memoria compartida y distribuida (por ejemplo clústers de máquinas multicore). Esta metodología permite la creación de programas paralelos multinivel integrando programas escritos con Hitmap con modelos de programación y herramientas automáticas de paralelización y optimización específicamente orientados a memoria compartida. En concreto se han presentado directrices y ejemplos para integrar OpenMP en el contexto de programas Hitmap, y se ha estudiado la integración de una herramienta experimental basada en técnicas asociadas al polyhedral model para automatizar y optimizar la paralelización de las partes secuenciales del programa incluso en presencia de dependencias de datos. Se ha presentado trabajo experimental para validar la aplicabilidad de la metodología en diferentes casos de estudio, y la eficiencia de los códigos obtenidos

En este Trabajo Fin de Máster se han conseguido cumplir los siguientes objetivos:

- Se ha realizado un estudio de los diferentes *frameworks* capaces de generar código paralelo para sistemas de memoria distribuida y compartida. Se ha visto que la mayoría de los estudios de la literatura están basados en el *polyhedral model* y por consiguiente en sus restricciones.

- Se ha diseñado una metodología para explotar de forma eficiente programas Hitmap en entornos híbridos. Incluyendo la implementación de una herramienta para la paralelización automática con modelos de memoria compartida.

- Se ha implementado la metodología en tres casos de estudio con características diferentes. Se ha evaluado el rendimiento de esos tres casos de estudio con respecto a códigos puramente basados en Hitmap y códigos de referencia manualmente desarrollados, con diferentes configuraciones en el número de procesos y de hilos lanzados.

El resultado final son unos pasos explícitos y simples a seguir a la hora de la programación de códigos paralelos con la librería Hitmap. Estos códigos aprovecharán las mejores ventajas de sistemas de memoria distribuida y de memoria compartida.

5.2. Trabajo futuro

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 aumentar el número de niveles hetereogeneos de paralelismo.

Además otra de las lineas de investigación de trabajo futuro será la ampliación del *framework* Trasgo para su aplicabilidad en algoritmos más complejos y la aplicación sistemática y generalizada de la metodología propuesta en este trabajo dentro del sistema general de traducción de Trasgo.

Apéndice A

Publicaciones asociadas a este Trabajo Fin de Máster

En este apéndice se cita una publicación científica derivada del trabajo realizado para este TFM. En concreto se trata de una presentación en el congreso HPCS 2014, un congreso CORE B internacional organizado por la asociación IEEE, con revisión por pares, que será presentado a finales de Julio de 2014. El artículo está aceptado y la versión definitiva que aparecerá en el libro de actas se incluye a continuación [26].

Exploiting distributed and shared memory hierarchies with Hitmap

Ana Moreton-Fernandez
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: ana.moreton@alumnos.uva.es

Arturo Gonzalez-Escribano
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: arturo@infor.uva.es

Diego R. Llanos
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: diego@infor.uva.es

***Abstract*—Current multicomputers are typically built as interconnected clusters of shared-memory multicore computers. A common programming approach for these clusters is to simply use a message-passing paradigm, launching as many processes as cores available. Nevertheless, to better exploit the scalability of these clusters and highly-parallel multicore systems, it is needed to efficiently use their distributed- and shared-memory hierarchies. This implies to combine different programming paradigms and tools at different levels of the program design.**

This paper presents an approach to ease the programming for mixed distributed and shared memory parallel computers. The coordination at the distributed memory level is simplified using Hitmap, a library for distributed computing using hierarchical tiling of data structures. We show how this tool can be integrated with shared-memory programming models and automatic code generation tools to efficiently exploit the multicore environment of each multicomputer node. This approach allows to exploit the most appropriate techniques for each model, easily generating multilevel parallel programs that automatically adapt their communication and synchronization structures to the target machine. Our experimental results show how this approach mimics or even improves the best performance results obtained with manually optimized codes using pure MPI or OpenMP models.

I. Introduction

Parallel machines are becoming more heterogeneous, mixing devices with different capabilities in the context of hybrid clusters, with hierarchical shared- and distributed-memory levels. Also, the focus on parallel applications is shifting to more diverse and complex solutions, exploiting several levels of parallelism, with different strategies of parallelization. It is increasingly interesting to generate application programs with the ability of automatically adapt their structure and load to any given target system.

Programming in this kind of environment is challenging. Many successful parallel programming models and tools have been proposed for specific environments. Message-passing paradigm (e.g. MPI libraries) have shown to be very efficient for distributed-memory systems. Global shared memory models, such as OpenMP, Intel TBBs, or Cilk, are commonly used in shared-memory environments to simplify thread and memory management. PGAS (Partitioned Global Address Space) languages, such as Chapel, X10, or UPC, present a middle point approach by explicitly managing local and global memory spaces. However, the application programmer still faces many important decisions not related with the parallel algorithms, but with implementation issues that are key for obtaining efficient programs. For example, decisions about partition and locality vs. synchronization/communication costs; grain selection and tiling; proper parallelization strategies for each grain level; or mapping, layout, and scheduling details. Moreover, many of these decisions may change for different target machine details or structure, or even when data sizes are modified.

Our approach is built upon the Hitmap library [1]. Hitmap reduces code development effort performing highly-efficient data distributions and aggregated communications, expressed in an abstract form. It can be used for indexed dense or sparse data structures, such as arrays, or graphs [2]. The mapping decisions are automatically guided by topology, partition, and distribution policies encapsulated in plug-in modules.

Although distributed-memory programming models can also be used for shared-memory environments, communication dominant models lead to the use of different base algorithms, parallelization strategies, or mapping techniques. In this work we show how to integrate Hitmap with shared-memory programming models, both manually and automatically, to generate a multilevel programming model that better exploits mixed distributed- and shared-memory environments. The Hitmap communication model is used to coordinate the mapping and synchronization on the distributed-memory level, exposing to the lower one only the local tasks mapped to one node. Thus, shared-memory programming strategies, programming models, and automatic code generation or transformation tools, can be transparently used to better exploit the local shared-memory environment, with reduced programming effort.

To show the application of this approach, we use three benchmarks that present different challenges to be programmed with distributed or shared memory programming models: A Jacobi solver (neighbor synchronization program), a wavefront Gauss-Seidel solver (using a pipeline paradigm), and a multilevel combination of matrix multiplication algorithms. Our experimental results for both shared- and distributed-memory environments show that the use of our approach obtains the best performance results when compared with pure MPI or OpenMP programs, manually optimized.

The rest of the paper is organized as follows: Section 2 discusses some related work. Section 3 reviews the main

features of Hitmap. Section 4 describes challenges presented by three cases of study. Section 5 describes our proposal to integrate shared-memory programming in Hitmap. Section 6 discusses an experimental evaluation of the proposal. Section 7 presents the conclusions and future work.

II. Related Work

There have been many proposals for programming languages to support parallelism and parallel array operations, making most mapping decisions transparent to the programmer (e.g. HPF, ZPL, CAF, UPC, Chapel, X10, etc.). Most of them are based on sophisticated compile-time transformations. They present a limited set of mapping functionalities, and they have troubles to generate programs that properly adapt their structures to hybrid execution platforms, or to exploit hierarchical compositions of parallelism with arbitrary granularity levels.

PGAS (Partitioned Global Address Space) models present an abstraction to work with mixed distributed- and shared-memory environments at the same level. These models do not promote the different parallelism and optimization techniques needed when processes and threads are hierarchically deployed in a hybrid environment. Comparing with UPC [3], it has been shown that Hitmap reduces the programming complexity and the memory footprint, while obtaining similar efficiency [1]. Another example of PGAS is Chapel, that proposes an interface to add mapping modules to the system [4]. The Hitmap run-time system presents a more general approach, with the possibility to exploit hierarchical mappings for any kind of domain using the same interface. Hitmap also improves the capabilities of other hierarchical tiling arrays libraries such as HTA [5].

Parray [6] is a model that introduces a flexible programming interface based on array types that can exploit hierarchical levels of parallelism for heterogeneous systems. In their approach the management of dense and strided domains is not unified and the domain operations are not transparent. Moreover, the programmer still faces decisions about granularity and synchronization details at different levels. Different modifiers and types should also be used to distribute and map arrays to the proper level. Our approach introduces a more generic, portable and transparent programming interface.

The polyhedral model provides a formal framework to develop automatic transformation techniques at the source code level [7]. Optimization and parallelization techniques applied to sequential code has been proven to be able to generate efficient programs for both, shared- and distributed-memory systems. The polyhedral model is only applicable to codes based on static loops with affine expressions. Moreover, it is not able to deal with generic hierarchical or recursive compositions of parallelism. Nevertheless, it is possible to use these tool in the context of more generic hierarchical parallel frameworks such as Hitmap.

III. Hitmap in a nutshell

Hitmap [1] is a library initially designed for hierarchical tiling and mapping of dense arrays. It has also been extended to support sparse data structures such as sparse matrices, or graphs [2], using the same methodology and interfaces. Hitmap is based on a distributed SPMD programming model, using abstractions to declare data structures with a global view. It automatizes the partition, mapping, and communication of hierarchies of tiles, while still delivering good performance.

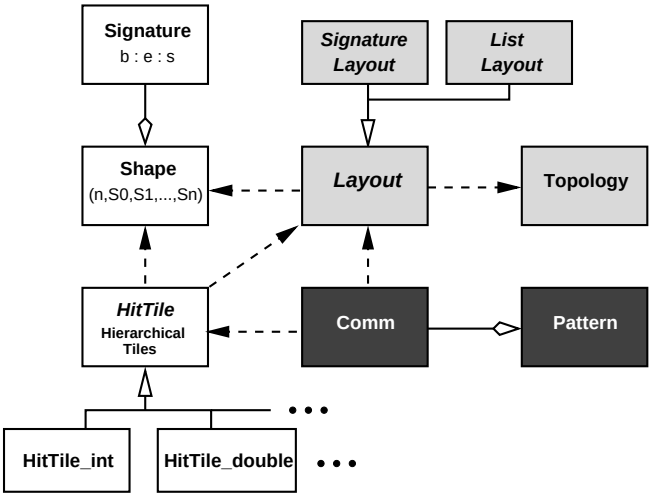

Fig. 1. Hitmap library architecture. White boxes represent classes to manage domain indexes and data structures. Light grey boxes represent classes to automatically apply domain partitions, and mappings. Dark grey boxes represent classes to generate adaptable and reutilizable communication patterns.

A. Hitmap architecture

Hitmap was designed with an object-oriented approach, although it is implemented in C language. The classes are implemented as C structures with associated functions. Fig. 1 shows a diagram of the library architecture. This simplified diagram shows classes to support data structures with dense or strided indexes domains.

An object of the *Shape* class represents a subspace of domain indexes defined as an n-dimensional rectangular parallelotope. Its limits are determined by n *Signature* objects. Each *Signature* is a tuple of three integer numbers $S = (b, e, s)$ (begin, end, and stride), representing the indexes in one of the axis of the domain. Signatures with $s \neq 1$ define non-contiguous yet regular spaced indexes on an axis. The index cardinality of a signature is $|S| = \lceil (e - b)/s \rceil$. Begin and stride members of a Signature represent the coefficients of a linear function $f_S(x) = sx + b$. Applying the inverse linear function $f_S^{-1}(x)$ to the indexes of a Signature domain we obtain a compact, contiguous domain starting at $\vec{0}$. Thus, the index domain represented by a Shape is equivalent (applying the inverse linear functions defined by its signatures) to the index domain of a traditional array. Other subclasses of *Shape* are used for sparse or non-array domains.

A *Tile* maps actual data elements to the index subspace defined by a shape. New allocated tiles internally use a contiguous block of memory to store data. Subsequent hierarchical subselections of a tile reference data of the ancestor tile, using the signature information to locate and access data efficiently. Tile subselections may be also allocated to have their own memory space.

The *Topology* and *Layout* abstract classes are interfaces for two different plug-in systems. These plug-ins are selected by

Fig. 2. Typical Hitmap stages for scientific computing applications. Dark grey stages contain sequential code that can be hierarchically parallelized.

name in the invocation of the constructor method. New plug-ins with different rules can be developed by the programmer and reused for other programs. Topology plug-ins implement simple functionalities to arrange physical processors in virtual topologies, with their own rules to build neighborhood relationships. Layout plug-ins implement methods to distribute a shape across the processors of a virtual topology. Hitmap has different partitioning and load-balancing techniques implemented as layout plug-ins. They encapsulate details which are usually hardwired in the code by the programmer, improving reusability. The resulting Layout object contains information about the local part of the domain, neighborhood relationships, and methods to locate the other subdomains. Topology and Layout plug-ins may flag some processors as inactive transparently to the programmer. For example, when there are more processors in the virtual topology than domain indexes to be distributed.

Finally, there are two classes related with interprocess communication. The *Communication* class represents information to synchronize or communicate tiles among processes. The class provides multiple constructor methods to build different communication schemes, in terms of tile domains, layout objects information, and neighbor rules if needed. This class encapsulates point-to-point communications, paired exchanges for neighbors, shifts along a virtual topology axis, classical collective communications, etc. The library is built on top of the MPI communication library, for portability across different architectures. Hitmap internally exploits several MPI techniques that increase performance, such as MPI derived data-types and asynchronous communications. Communication objects can be composed in reusable *Patterns* to perform several related communications with a single call.

B. Hitmap usage methodology

In this section, we discuss how a typical parallel program is developed using Hitmap. This methodology derives in clearly structured programs with common guidelines. Figure 2 shows the typical stages of a scientific computing application programmed with Hitmap.

The programmer designs the parallel code in terms of logical processes, using local parts of abstract data structures, and interchanging information across a virtual topology of unknown size. The first step is to select the virtual topology type appropriate for the particular parallel algorithm. For example, it could be a rectangular topology where processors have two indexes (x, y). Remind that topologies define neighborhood relationships.

The second design step is to define domains, starting with a global view approach. All logical processes declare the shapes of the whole data structures used by the global computation. Layout objects are instantiated for partitioning and mapping global domains across the virtual topology. The layout objects are queried to obtain the local subdomain shapes. Local domains may be expanded to overlap other processors subdomains, generating *ghost zones*, a portion of the subspace shared (but not synchronized) with another virtual processor. Once mapped, and after the corresponding memory allocation, the programmer can start to use data in the local subdomain, using local tile coordinates, or in terms of the original, global view coordinates. This helps to implement the sequential computations for the tiles.

The programmer finally decides which communication structures are needed to synchronize data between the computational phases. They are imposed by the parallel algorithm. Communication objects are built to create the communication structures designed. The programmer reasons in terms of tile domains and domains intersections. The objects are instantiated using a local tile (to locate the data in memory) and using information contained in a layout object about neighbors and domain partition. For example, for ghost zones, shape intersection functionalities automatically determine the exact chunks of data that should be synchronized across processors. The communication objects contain data-marshalling information. They can be created at program initialization, and invoked when they are needed, as many times as required.

The result of the implementation phase is a generic code that is adapted at run-time depending on: (a) the particular global domains declared; (b) the internal information about the physical topology; and (c) the selected partition/layout plug-ins. For example, the code of Fig. 4 shows the implementation in Hitmap of the Matrix Multiplication algorithm presented as case 1.2 in Fig. 3. In this code, the sequential function used to multiply the local matrices is an implementation of algorithm presented as case 1.1.

IV. Cases of study: Challenges for hybrid distributed/shared memory programming

Different classes of applications present different challenges for the integration of distributed and shared memory programming models. In this work we use three cases of study to show some of the challenges and possible approaches to solve them. Simple algorithm expressions for the three chosen scientific computing applications are shown in Fig. 3.

A. Matrix multiplication: Multilevel algorithms

Figure 3 (left) shows two algorithms for matrix multiplication. The first one is a classical approach that is typically implemented sequentially with three nested loops. The first two ones can be parallelized without write dependences or potential race conditions. This parallelization is appropriate for shared-memory environments. However, for distributed memory environments it may lead to bigger memory footprints

```
** 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. Algorithm expressions of three cases of study

and/or unnecessary communications. Cannon's algorithm [8] works with a partition of the matrices in $k \times k$ pieces, requiring no more than one local piece of the same matrix at the same time, and using a simple circular block shift pattern to move data across processes. This approach perfectly suits distributed-memory models, but is more complex to program, and it introduces no benefits for shared-memory environments. Thus, the best approach for a hybrid hierarchical memory system is to combine both.

B. Jacobi: Neighbor synchronization

The second case of study is a PDE solver using a Jacobi iterative method to compute the heat transfer equation in a discretized two-dimensional space. It is implemented as a cellular automata. On each iteration, each matrix position or cell is updated with the previous values of the four neighbors. With distributed memory programming models this leads to iterations of an update-communicate cycle. A second copy of the matrix is used to store the old values during the local update of the matrix. The communication across processes is done with a pattern known as *neighbor synchronization* in a virtual 2-dimensional processes topology. The data needed by each neighbor is only the nearest border of the local partition of the matrix. Thus, the local matrix is spawned with one more row/column on each direction to store the elements coming from neighbor processes. These extra rows/columns are known as *ghost zones*. Using a shared-memory programming model, extra memory space for the ghost zones is not needed. But synchronization is required to ensure that each thread is not using data that is updated or non-updated on time by other threads on the shared border zones.

C. Gauss-Seidel: Wave-front pipelining

The last case of study computes the same heat transfer equation, but using a Gauss-Seidel iterative method. In this method the convergence is accelerated using values already computed during the current time-step iteration. The method simply uses one matrix and no copy for the old values. Thus, when using the neighbor values of the up and left matrix positions, values already updated are used. The down and right values used during one matrix position update were computed on the previous time-step iteration. The order of traversing the domain indexes is important on this case. Thus, the implementation is done with loops that imply carried dependencies on each iteration. This leads to a wave-front kind of application on each convergence checking iteration.

With distributed-memory programming models, the solution is similar to the neighbor synchronization example; using ghost zones and two communication patterns. The first one receives data in the ghost zones from the up and left neighbors before local computation. The second one sends updated borders to down and right neighbors after local computation. This patterns naturally generates a pipelining computation controlled by the data-flow.

However, some shared memory programming models do not have a good support for data-flow or pipelining computations. There are two approaches that can be used. The first one is explicitly deal with the flow order imposed by the data dependencies to avoid potential race conditions. That means using sophisticated parallel constructs of lock variables, explicit topology and data partition, and reasoning in terms of the number of threads and the local thread rank. The second one is to apply complex loop transformations to generate a new order of execution that exposes parallelism.

V. Integrating shared-memory programming in Hitmap

A. Multilevel programming model

Hitmap approach proposes a multilevel programming model. Hitmap provides an environment to efficiently coordinate the data partition, mapping, and communication with a distributed memory approach. The sequential code that executes the local tasks is clearly identified and isolated. Sequential computation happens between calls to Hitmap functions that execute communication patterns. The data pieces, dimensional sizes, and limits of local indexes spaces are uniformly parametrized in terms of the results of the layout objects, internally adapted to the platform details at run-time. Thus, the code inside the local tasks can be parallelized with a shared memory paradigm in a systematic way.

```
 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(figurelayA));
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:     /* 6.1. INITIALIZE INPUT MATRICES */
33:     initMatrices( tileA, tileB );
34:     /* 6.2. CLEAN RESULT MATRIX */
35:     double 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:     writeResults( tileC );
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. Cannon's algorithm for matrix multiplication with no cardinality restrictions in Hitmap.

```
/* MATRIX MULTIPLICATION: COMPUTATION STAGE INSIDE CANNON'S */
#pragma omp parallel private(loopIndex)
{
    initMatrices( tileA, tileB );
    double aux=0; hit_tileFill( &tileC, &aux );
    #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 (k=begin2; k<=end2; k++)
                  for (j=begin1; j<=end1; j++)
                     hit_tileAt2( C, i, j ) += hit_tileAt2( A, i, k ) * hit_tileAt2( B, k, j );
         }
      }
   }
```

Fig. 5. Local computation stage of the matrix multiplication Hitmap program, parallelized with OpenMP after applying tiling optimization and loop reordering.

The parallelization of local tasks can be done manually by a programmer with experience using shared memory programming models. Nevertheless, tools to automatically generate parallel code can also be applied to the sections of code that perform local computations. For example, Fig. 4 shows a Hitmap implementation of the Cannon's algorithm that calls a sequential function implementing the classical algorithm to apply the local matrix multiplication. Parallelizing with a shared-memory programming model the sequential function leads to a two-level parallel program suitable for hybrid distributed- and shared-memory platforms.

B. Methodology to use OpenMP

In this section we show guidelines and examples to use OpenMP to exploit the shared-memory paradigm in the context of a Hitmap program. The first step is to identify the sequential sections of the code. This can be done manually or automatically, as they are enclosed by Hitmap functions that execute communications: *hit_comDo*, *hit_comStart*, *hit_comEnd*, *hit_patternDo*, *hit_patternStart*, *hit_patternEnd*.

The first approach to introduce OpenMP directives to parallelize the sequential parts of the Hitmap programs is to use synchronized *omp parallel for* primitives for computational intensive loops. Nevertheless, this solution can be inefficient when the loops to be executed in parallel are inner loops in the context of other loops that execute repetitions of the typical computation/communication cycles of distributed-memory programs. On each iteration of the outer loop, new threads should be spawned and destroyed for the worksharing loop. These operations are costfull.

A better approach is to spawn threads (using a *omp parallel* primitive) as soon as the layouts objects have been built, the local data structures has been allocated, and communication patterns have been created. Then, worksharing control primitives (such as *omp for*, *omp sections*, etc.) can be introduced to control the threads during the initialization of data structures, and the computation stages. The example in Fig. 5 shows the application of this guidelines to the Hitmap matrix multiplication code. The execution of communication patterns to move data from/to other remote processes should be done by only one thread inside the communicating process. Thus, an *omp single* primitive should be introduced for each group of consecutive communication calls: *hit_comDo*, *hit_comStart*, etc. To ensure correctness, the threads should be synchronized before and after the execution of the communication patterns that move from/to other processes pieces of data that are involved in the current workshared computation. In the example

presented in Fig. 5, the *for* and *single* OpenMP primitives imply the needed synchronization. Overlapping of computation and communication can be exploited, such as in the Jacobi example, using the Hitmap functionalities for asynchronous communication.

C. Execution model

The execution model of the hybrid distributed- and shared-memory Hitmap programs is simple. The program should be launched with one process per node, or per chosen subset of CPU cores. The current version of Hitmap is built on top of MPI. For example, using the common *hydra* launcher included in several MPI distributions, it is direct to deploy as many MPI processes as desired on each physical node. The topology and layout policies automatically adapt the data-partition and communication patterns for this distributed memory level.

Each local task spawns as many threads as the number of assigned computational elements (CPU cores). For example, using OpenMP, it is again direct to set the number of threads to be the number of assigned cores in the topology or layout objects, using the standard function: *omp_set_threads_num( )*. In the current experimental prototype we allow to test different thread/process configurations using the value of a program command line argument to set the number of threads on each particular MPI process.

D. Parallelization and code transformations

Code transformation techniques are typically used to: (1) Optimize the parallel and sequential codes; and (2) simplify the parallelization. For example, critical optimizations for codes with high data reutilization include the use of tiling. The Hitmap coordination code works with a coarse grain approach that minimizes communication costs. Tiling should be used in the sequential parts of the code before applying shared-memory parallelization. It generates a middle-grain partition, optimizing the use of local memory and cache hierarchies, and aligning the threads' work. See an example in Fig. 5.

More aggressive transformations to expose parallelism in the presence of data dependencies can also be applied. For example, wave-front computations are easily expressed with sequential loops with forward dependencies across loop iterations. This codes are not easily parallelized with models such as OpenMP where the work sharing primitives do not directly provide any kind of flow control across threads.

The tough approach is to manually develop parallel structures with lock variables, reasoning in terms of the number of threads and the local thread rank. This replicates the upper-level Hitmap partition and synchronization structure at the shared-memory level. An easier approach is to completely transform the loop codes to generate an outer loop that traverses the index domain space in a way that parallelism without dependences is exposed. For example, wave-front computations can be transformed in pipeline structures, where an outer loop advances through pipeline stages, and the stages are executed in parallel without dependencies. In the case of the Gauss-Seidel program, it is possible to generate a loop that sequentially traverses the secondary diagonals of the matrix. Although the updates on each element of a secondary diagonal depends on data generated on the previous diagonal, updates on the same diagonal can be done in parallel.

E. Integration of polyhedral model code transformations

The previously discussed transformations can be automatically applied to an important class of sequential codes using *polyhedral model* techniques [7]. For example, the local part computations in the three cases of study presented in section IV are suitable for using such techniques. The full sequential Matrix Multiplication algorithm is a good example. In the Jacobi and Gauss-Seidel algorithms, the codes inside the convergence checking loops are also suitable.

Pluto [9] is a state-of-the-art compiler that applies source-level optimization transformations and parallelization to polyhedral model compatible codes. We have adapted the front-end of Pluto v0.9.0 to generate isolated pieces of code (without a C main function, shared variable declarations, etc.). Thus, we can supply pieces of code from the Hitmap program to the Pluto transformation system, and automatically substitute them by the output generated. Besides this interface modification, we have developed an additional, tailored version of Pluto to allow the generated code to be integrated with the directives we introduced in Sect. V-B. The original Pluto generates *omp parallel for* directives to independently parallelize selected transformed loops. We have modified the directive generation to produce *omp for* directives instead.

This new fully automatic tool can be used to optimize the sequential parts of code in Hitmap programs, and to expose parallelism in the presence of loop-carried dependencies. With the proposed methodology and tools, it is possible to easily transform Hitmap codes in multilevel parallel programs that can efficiently exploit distributed/shared memory hybrid platforms, adapting the synchronization structure to the platform at run-time.

VI. Experimental evaluation

We have conducted an experimental study to validate the advantages of our proposal, and to verify the efficiency of the resulting codes.

A. Methodology

We evaluate several run-time configurations (MPI processes vs. number of threads), and compare the results with reference program versions manually developed and optimized using MPI for distributed memory, and OpenMP for shared-memory. We work with four types of parallel versions of the original codes:

- **Ref-MPI** and **Ref-OMP**: Manually parallelized and optimized versions for distributed and shared-memory. In the Gauss-Seidel case the OpenMP reference is a pipelined code manually built using data-partitions in terms of thread indexes and lock-variables.
- **Hitmap**: Versions parallelized with Hitmap. The sequential parts are transcribed from the reference codes.
- **Hitmap+PlutoOrig**: The same Hitmap codes, substituting the sequential parts with Pluto outcomes, and avoiding the use of other OpenMP directives proposed by our approach.
- **Hitmap+Pluto:** Hitmap implementations, using the OpenMP directives described in our approach, and

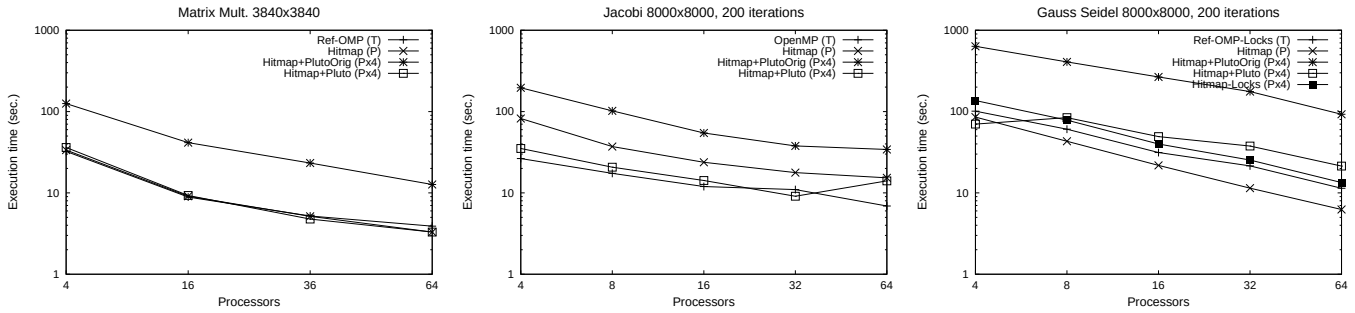

Fig. 6. Performance results for experiments in Atlas (shared memory platform)

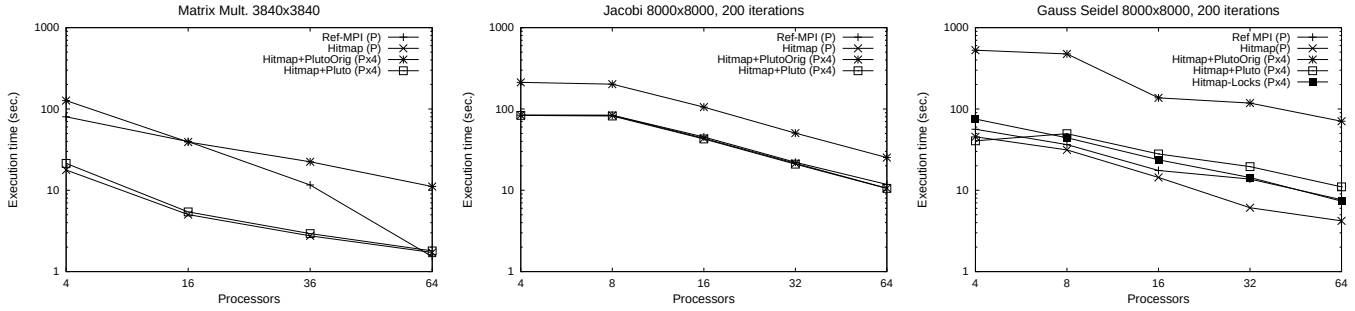

Fig. 7. Performance results for experiments in Calendula (hybrid distributed-shared memory platform)

using the sequential parts automatically optimized and parallelized by our modified version of Pluto.

- **Hitmap-Locks:** In the Gauss-Seidel case, we have also develop a Hitmap code in which the sequential part is substituted by an adapted version of the code used as OpenMP reference, using lock-variables.

The experiments were executed in two platforms. The first one is a pure shared memory machine (Atlas) of the Trasgo group, at the University of Valladolid (Spain). It is a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, making a total of 64 cores, and 256 GB of RAM. The second platform (Calendula) is a hybrid cluster that belongs to "Fundación Centro de Supercomputación de Castilla y León" (Spain). The cluster nodes are connected by Infiniband technology, and they have two Intel Xeon 5450 CPUs with 4 cores each. Using 8 nodes of the cluster, we exploit up to 64 computational units.

In both cases we compile the codes with the GCC compiler, using the *-fopenmp* flag, and the optimization flags *-O3* and *-funroll-loops*. For the upper level of parallelism, in Atlas we use *mpich2* v3.0.4 as MPI implementation. In Calendula we use the *OpenMPI* v.1.6.5 implementation.

We have selected input data sizes big enough to produce a computational load that is relevant when distributed across 64 computational units. For the Jacobi and Gauss-Seidel problems we present results for the execution of the first 200 iterations of the main loop in a 8000×8000 matrix. We have eliminated from the codes the convergence test reduction for simplicity and to focus on the main synchronization patterns described in section IV. The matrix multiplications programs compute the multiplication of two matrices of 3840×3840 elements.

The pure OpenMP programs used as reference, have been executed in Atlas with $T = 4$, 8, 16, 32, and 64 threads. In Calendula, the reference MPI programs have been executed with $P = 4$, 8, 16, 32, and 64 processes. The processes were distributed across the nodes depending on the number of threads spawned by each process. Nodes are filled up with processes/threads, one per core, before launching more processes in another node. The original Hitmap programs are executed with normal MPI processes (P). The multilevel Hitmap versions have been executed in both platforms with combinations of 1 to 16 MPI processes with 4 threads per process ($P \times 4$). Combinations with 8 threads per process have been also executed, obtaining similar results. Thus, they are omitted in the plots and discussion. Note that due to the perfect square topology of MPI processes imposed by the Cannon's algorithm, some configurations are not possible for the matrix multiplication programs.

We present the results in plots showing execution time vs. number of active processing elements (cores with assigned threads, or MPI processes). We use logarithmic axis to easily observe both scalability and constant or proportional performance degradations, when comparing different versions.

B. Results in shared memory

Figure 6 show the results of the three cases of study on Atlas, the shared memory platform. We can see that, for all the examples, Hitmap abstractions do not introduce significant overheads at run-time. The results show that inside a pure shared-memory architecture the performance obtained by the hybrid Hitmap programs is similar to a pure OpenMP program exploiting the same optimization techniques.

The code generated by Pluto works much better using

the proposed Hitmap methodology to introduce the OpenMP parallelization directives, integrating it in the context of a communicating process. The reason is that in Hitmap the threads are spawned only once, and coordinated with the work sharing primitives through their lives across the shared-memory part of the program. After integrating it in Hitmap, it produces very good performance results, similar to our best manual optimizations. For example, the optimization techniques used in the manual version of matrix multiplication (mainly tiling and loop reordering), can be automatically obtained with Pluto.

The Gauss-Seidel case behavior deserves further discussion. Even the Pluto modified versions do not achieve the same performance as a pure message-passing program, such as the original Hitmap version. Even a manually developed and optimized version using a sophisticated approach based on lock-variables (*OpenMP-Locks*) is outperformed by the message-passing Hitmap. The data-flow nature of this pipelined program is more appropriate for communicating processes than for shared-memory programming models.

C. Results in a hybrid cluster

The same Hitmap programs used in the previous experiments can be used in the distributed-memory platform without any changes. Figure 7 shows the experimental results of the three cases of study on Calendula. The results show that the communication time across different nodes impose a delay that minimizes the impact of the synchronization/communication times inside the nodes. Thus, using threads instead of more MPI processes to exploit the cores inside the nodes only delivers a slight performance improvement.

The results for the Matrix Multiplication case show that the hybrid two-level Hitmap program, with the modified-Pluto codes integrated, obtains the same performance as the original Hitmap, and better performance than the MPI reference code, both with carefully optimized sequential code. In this case, the native-compiler optimization modules works better with the sequential code in the context of the Hitmap abstractions. In the neighbor synchronization (Jacobi) example, all versions present the same performance, except the original Pluto version that is slower. In the Gauss-Seidel case, we again obtain the best results with the original message-passing Hitmap version, in which the sequential code is even better optimized by the compiler than for the original reference code.

These results show that the main advantage of the Hitmap approach is that using a simple methodology and automatic optimization and parallelization tools, it is possible to generate hybrid programs capable of exploiting the best advantages of distributed- or shared-memory platforms. The programs adapt their communication and synchronization structures to the platform features at run-time.

VII. Conclusions

This paper presents an approach to simplify the programming for mixed distributed and shared memory parallel computers. The development effort of the coordination at the distributed memory level is reduced using Hitmap, a library for distributed computing using hierarchical tiling of data structures. We present a methodology and tools to integrate Hitmap codes with shared-memory programming models, and automatic code-generation tools, to efficiently exploit the multicore environment of each multicomputer node. This approach allows to exploit the most appropriate techniques for each model, easily generating multilevel parallel programs that automatically adapt their communication and synchronization structures to the target machine.

We present guidelines and examples of how to use a shared-memory programming paradigm (such as OpenMP) to parallelize the sequential parts of the distributed memory Hitmap programs, in order to exploit thread parallelism inside the Hitmap communicating processes. We also show how to integrate the use of a state-of-the-art tool based on polyhedral model transformation techniques, to automatically parallelize the sequential parts in the presence of loop-carried data dependencies. Our experimental results show that the hybrid Hitmap codes mimics or even improves the best performance results obtained with manually developed and optimized programs using one of both, pure distributed, or pure shared memory programming models.

The current development framework and examples are available under request. Future work includes the study of the applicability of these techniques for more dynamic and non-polyhedral application classes, sparse data structures, and more levels of heterogeneous parallelism.

Acknowledgment

This research has been partially supported by Ministerio de Economía y Competitividad (Spain) and ERDF program of the European Union: CAPAP-H4 network (TIN2011-15734-E), and MOGECOPP project (TIN2011-25639); Junta de Castilla y León (Spain): ATLAS project (VA172A12-2), and "Fundación Centro de Supercomputación de Castilla y León".

References

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

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

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

[4] B. Chamberlain, S. 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.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarn, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. of the ACM SIGPLAN PPoPP*. New York, New York, USA: ACM, 2006, pp. 48–57.

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

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

[8] L. Cannon, "A cellular computer to implement the kalman filter algorithm," Doctoral dissertation, Montana State University Bozeman, 1969.

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

Bibliografía

[1] http://www.mpi-forum.org/ Ultimo acceso: 24 de junio de 2014.

[2] http://openmp.org/wp/ Ultimo acceso: 27 de mayo de 2014.

[3] http://supertech.csail.mit.edu/cilk/ Ultimo acceso: 24 de junio de 2014.

[4] https://www.threadingbuildingblocks.org/ Ultimo acceso: 24 de junio de 2014.

[5] **Pocc: Polyhedral compiler collection.** Avalaible at http://www.cse.ohio-state.edu/ pouchet/software/pocc/ .

[6] **Polyopt: a polyhedral optimizer for the rose compiler.** Avalaible at http://www.cse.ohio-state.edu/ pouchet/software/polyopt/ .

[7] **Polly: Polyhedral optimizations for llvm.** Avalaible at http://polly.llvm.org/.

[8] **Polly framework** Avalaible at http://polly.llvm.org/.

[9] **Ibm xl compiler.** Avalaible at http://ibm.com/software/awdtools/xlcpp/.

[10] http://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager Último acceso: 17 de junio de 2014.

[11] Saman P Amarasinghe and Monica S Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN Notices*, volume 28, pages 126–138. ACM, 1993.

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

[13] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, 1991.

[14] Uday Bondhugula. Automatic distributed-memory parallelization and code generation using the polyhedral framework. *rap. tech. ISc-CSA-TR-2011-3, Indian Institute of Science*, 2011.

[15] 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.

[16] Lynn E Cannon. A cellular computer to implement the kalman filter algorithm. Technical report, DTIC Document, 1969.

[17] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

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

[19] Michael Claßen and Martin Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE, 2006.

[20] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160. ACM, 2005.

[21] Raja Das, Paul Havlak, Joel Saltz, and Ken Kennedy. Index array flattening through program transformation. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 70. ACM, 1995.

[22] Raja Das, Joel Saltz, and Reinhard von Hanxleden. *Slicing analysis and indirect accesses to distributed arrays*. Springer, 1994.

[23] Peter Faber. *Transformation von shared-memory-programmen zu distributed-memory-programmen*. PhD thesis, 1997.

[24] Peter Faber, Martin Griebl, and Christian Lengauer. Issues of the automatic generation of hpf loop programs. In *Languages and Compilers for Parallel Computing*, pages 359–362. Springer, 2001.

[25] A. Moreton Fernandez, A. Gonzalez-Escribano, and D.R. Llanos. Trasgo frontend: Hacia una generación automática de código paralelo portable. In *Proc. Jornadas Paralelismo'12*, 2012.

[26] A. Moreton Fernandez, A. Gonzalez-Escribano, and D.R. Llanos. Exploiting distributed and shared memory hierarchies with hitmap. In *The 2014 International Conference on High Performance Computing Simulation (to appear)*, 2014.

[27] Clayton Ferner. Revisiting communication code generation algorithms for message-passing systems. *The International Journal of Parallel, Emergent and Distributed Systems*, 21(5):323–344, 2006.

[28] Ana Moretón Fernández. Generación de código xspc a partir de cspc. Bachelor Thesis. Universidad de Valladolid, 2011.

[29] Ana Moretón Fernández. Trasformación y generación de código paralelo. Master's thesis, Universidad de Valladolid, 2013.

[30] J. Fresno, A. Gonzalez-Escribano, and D.R. Llanos. Extending a hierarchical tiling arrays library to support sparse data partitioning. *The Journal of Supercomputing*, 64(1):59–68, 2012.

[31] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[32] A. Gonzalez-Escribano and D.R. Llanos. Trasgo: A nested parallel programming system. *Journal of Supercomputing*, doi:10.1007/s11227-009-0367-5, 2009.

[33] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D.R. Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE TPDS*, 2013.

[34] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[35] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*.

[36] Amy W Lim and Monica S Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214. ACM, 1997.

[37] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. Revisiting loop fusion in the polyhedral framework. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 233–246. ACM, 2014.

[38] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 361–370. ACM, 1993.

[39] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 72. IEEE Computer Society Press, 2012.

[40] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[41] Joel Saltz, Kathleen Crowley, Ravi Michandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, 1990.

[42] Xingfu Wu and Valerie Taylor. Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):56–62, 2011.