Trasgo: Sistema en tiempo de ejecución para código paralelo abstracto y portable

Arturo Gonzalez-Escribano, Ana Moretón Fernández, Diego Llanos Ferraris

Resumen— **En este trabajo presentamos la evolución de Trasgo, un sistema de generación de programas paralelos a partir de código abstracto y portable. Trasgo se apoya en una biblioteca de funciones que posterga importantes decisiones de mapeo a tiempo de ejecución. El lenguaje de entrada es un lenguaje de coordinación explícitamente paralelo dónde el programador no necesita tomar decisiones relacionadas con la granularidad, los recursos del sistema, o la estructura de comunicación. A partir de estos códigos abstractos Trasgo permite generar programas eficiente que se adaptan al sistema real, explotando diferentes tecnologías de paralelización en sistemas híbridos y/o heterogéneos.**

Palabras clave— **Lenguajes de programación paralela, generación de código, sistemas run-time**

I. Introducción

DEBIDO a la evolución de los sistemas de cómputo paralelo, el desarrollo de aplicaciones que explotan paralelismo ha ido cambiando paulatinamente su foco. Desde las clásicas aplicaciones numéricas en supercomputadores de alto-rendimiento, hacia las aplicaciones de streaming multimedia, y la computación de propósito general en cualquier ámbito.

Hoy por hoy interesa explotar el paralelismo tanto de algoritmos básicos como de complejos programas modulares. Las nuevas aplicaciones paralelas muestran diferentes niveles de paralelismo, con diferentes estrategias de paralelización. Muchas aplicaciones muestran comportamientos dinámicos, dependientes de datos, dónde las políticas de balanceo automático de carga son claves. A su vez, las aplicaciones deberían ser flexibles, para poder ejecutarse en entornos cada vez más diversos, mezclando clusters de memoria distribuida con multicores y dispositivos aceleradores cómo GPUs. La enorme diversidad y heterogeneidad de las plataformas, y las diferentes formas de componer dichos elementos presentan problemas para el desarrollo de aplicaciones portables que sean capaces de adaptarse de forma eficiente al entorno de ejecución.

Hasta ahora han aparecido diversos modelos de programación que han sido adoptados con éxito por la comunidad científica y de desarrolladores. En sistemas distribuidos las implementaciones eficientes del paradigma de paso de mensajes (p.e. las implementaciones de MPI). Para sistemas de memoria compartida las herramientas de control de threads de alto nivel (p.e. OpenMP). Para los dispositivos aceleradores, o sistemas híbridos que contengan aceleradores, modelos específicos (p.e. CUDA u OpenCL). En un intento de aunar diferentes tendencias y paradigmas los modelos PGAS (Partitioned Global Adress Space) (p.e. UPC, Chapel, X10).

Sin embargo, el programador que usa estos modelos o herramientas se enfrenta a diversas tareas que le obligan a razonar en términos de recursos físicos de una hipotética plataforma de ejecución, introduciendo código completamente ajeno a la lógica de la aplicación o algoritmo paralelo original, para preveer diversas situaciones. Estas tareas pueden incluir la maximización de la localidad de datos en los dispositivos, analizar costes de comunicación/sincronización, tomar decisiones sobre particiones de datos y granularidad de tareas, decidir las estrategias de paralelización para diferentes niveles de granularidad, escoger técnicas de mapeo, distribución de datos y planificación de tareas, o crear estructuras de comunicación y sincronización apropiadas.

En estos momentos es clave proveer a los desarrolladores con entornos y herramientas de programación productivas. Dónde estos detalles queden ocultos y sea posible programar, probar y depurar de forma fácil, e independientemente de los detalles de la máquina dónde se va a ejecutar la aplicación. En este trabajo presentamos la evolución de Trasgo [1], un sistema integral de programación paralela de éstas características. Trasgo es un sistema de transformación de código fuente a código fuente (ver figura 1). Su entrada es un lenguaje de programación paralela explícito, pero de alto nivel, abstracto e independiente de los detalles y decisiones relacionados con la plataforma de ejecución. El sistema de transformación automática genera código para múltiples plataformas y/o clusters heterogéneos, explotando eficientemente diversas herramientas, lenguajes y modelos de programación. El código generado se asienta sobre un sistema run-time que adapta las decisiones relacionadas con la partición de datos y tareas, y las estructuras de comunicación y sincronización a la plataforma de ejecución. Presentamos las nuevas características de Trasgo, los principios utilizados para incluir en el sistema de generación herramientas para sistemas heterogéneos, y casos de estudio para ejemplificar las

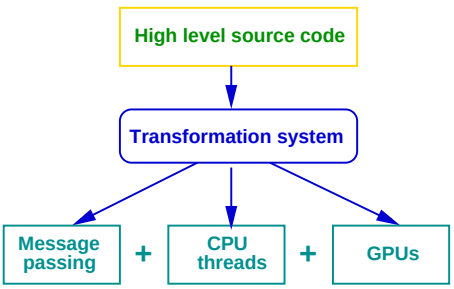

Fig. 1. Sistema de mapeo automático Trasgo

propiedades del sistema.

El resto del artículo se organiza de la siguiente forma. En la sección II se repasa el modelo de programación asociado al sistema Trasgo. En la sección III se discuten las generalidades del sistema de transformación de Trasgo y los cambios conceptuales en las nuevas versiones. En la sección IV se presentan las nuevas funcionalidades incluidas en el sistema de run-time y como utilizarlas en la transformación de código. En la sección V se comentan dos casos de estudio y resultados experimentales preliminares. La sección VI presenta las conclusiones y retos planteados en el estado de desarrollo actual de Trasgo.

II. El modelo Trasgo

A. Modelo de programación

El modelo de programación de Trasgo se basa en representaciones abstractas, estructuradas y de alto nivel de los algoritmos o programas paralelos. Una de las claves es que dicha representación está restringida al paradigma de *paralelismo anidado*. Una representación jerárquica y estructurada, de las construcciones de paralelismo, pero con un modelo de sincronización restrictivo. Es misión del sistema de transformación encontrar las formas no estructuradas de comunicación o sincronización más apropiadas para cada caso y reescribirlas en el código generado. Al mantener una representación jerárquica y estructurada el modelo se asienta en la composición de procesos SP (Serie-Paralelo) [2], con propiedades semánticas claras y bien definidas [3]. El resultado es un modelo sencillo, fácil de entender y programar, libre de condiciones de carrera, o comportamientos estocásticos y situaciones de *abrazo-mortal* difíciles de predecir. Conceptualmente los puntos de sincronización generan estados globales coherentes que favorecen las pruebas y la depuración del código. El modelo Trasgo extiende características del modelo poliédrico [4] y de álgebras de datos.

B. Lenguaje de programación

Se pueden desarrollar diferentes front-ends para traducir un lenguaje de entrada concreto a la representación interna que utiliza el sistema Trasgo. En su versión actual, el lenguaje de entrada utilizado por defecto es cSPC [5], para el que existe un front-end completo. cSPC es una extensión del lenguaje C estándar, con primitivas de coordinación. En la figura 2 se muestra el código cSPC de un PDE solver basado en el método de Jacobi para resolver la ecuación del calor en un espacio bidimensional.

El código secuencial se encapsula en funciones con parámetros formales en los que el programador expresa explícitamente el comportamiento de entrada/-salida de cada uno. El sistema usará esta información para determinar las dependencias de datos. Estas funciones secuenciales pueden ser del nivel de granularidad apropiado para el algoritmo, ignorando por completo decisiones relacionadas con la eficiencia. El sistema determinará si las tareas deben ser subdivididas o agrupadas. De hecho, las funciones secuenciales

```
void update( in double left,
             in double right,
             in double up,
             in double down,
             out double myself ) {
   myself = ( left + right + up + down ) / 4;
}

coordination void jacobiSolver(
             in int iterations
             inout double matrix[][] ) {

  double inside[][] = matrix[1:$-1][1:$-1];

  loop ( i, [1:iterations] ) {

    parallel( Map( inside.shape(),
                   blocks,
                   rectangular2D ) ) {
      map: update(
             inside[ paridx(0)-1 ][ paridx(1) ],
             inside[ paridx(0)+1 ][ paridx(1) ],
             inside[ paridx(0) ][ paridx(1)-1 ],
             inside[ paridx(0) ][ paridx(1)+1 ],
             inside[ paridx(0) ][ paridx(1) ]
           );
   }
}
```

Fig. 2. cSPC: Extracto de un ejemplo. Jacobi solver.

pueden implementar desde operaciones de grano muy fino, hasta wrappers que reciben grandes estructuras de datos y llaman a funciones de biblioteca especializadas para el cálculo secuencial, o para dispositivos aceleradores. En el ejemplo, la función *update* es una función secuencial que actualiza una celda (parámetro out) de la matriz que representa el espacio discretizado, con la media de los valores de otras cuatro celdas (parámetros in).

Las funciones que contienen información de coordinación van precedidas de un modificador que lo indica. En estas funciones sólo se pueden declarar y seleccionar partes de las estructuras de datos. Pero no manipular sus valores. Esto debe hacerse siempre llamando a funciones secuenciales. En el ejemplo, la función *jacobiSolver* es una función de coordinación. Recibe una matriz de dos dimensiones como parámetro de entrada/salida, y un número de iteraciones.

La selección de partes de un array se realiza con una notación similar a la de Fortran95, o a las extensiones para arrays del compilador de Intel. El símbolo $ indica el último elemento del espacio de índices en esa dimensión. En el ejemplo, *inside* representa la parte de la matriz que excluye los bordes de la misma. Es la región a actualizar por el solver.

El lenguaje de programación ofrece una visión global de las estructuras de datos. No hay detalles sobre la gestión threads, o la comunicación entre procesos. El programador trabaja en términos de procesos lógicos, no físicos. El paralelismo se expresa con combinaciones jerárquicas de una primitiva paralela unificada. La primitiva que permite expresar paralelismo recibe como parámetro el resultado de una función de mapeo que no se resolverá hasta el momento de la ejecución, cuando tanto los tamaños y propiedades de las estructuras de datos, como los características

de la plataforma de ejecución son conocidos.

La primitiva de paralelismo junto con la función Map trabajan con tres niveles de abstracción, que se corresponden con los tres parámetros de la función Map. Se expanden tantos procesos lógicos en paralelo como se indique en su primer parámetro, que es un dominio de índices lógicos. Este dominio puede extraerse de una estructura de datos (para generar paralelismo de datos) o explicitarse de forma directa (más apropiado para paralelismo de tareas).

El segundo parámetro es el nombre de un módulo de *Layout*. Dichos módulos implementan funciones de partición y asignación del dominio de índices sobre la topología virtual de procesadores. Actualmente el sistema Trasgo incluye las técnicas de partición más comunes. Nuevas técnicas se pueden añadir como plug-ins en el sistema de ejecución.

El tercer parámetro es el nombre de una función de *Topología virtual*. Estas funciones reorganizan los elementos y dispositivo de proceso reales en una topología virtual donde se crear relaciones de vecindad de acuerdo con las reglas específicas de cada función. De nuevo se incluyen las más comunes con un interfaz que permite desarrollar nuevos plug-ins.

En el momento de la ejecución estas funciones determinarán la estructura de tareas y el nivel más apropiado de granularidad siguiendo las políticas incluidas en cada función.

En el ejemplo de la figura 2, dentro de la función de coordinación se ejecuta una primitiva *loop* que implementa un bucle secuencial en el que la variable i recorre el espacio de índices dado como segundo parámetro. En cada iteración de este bucle se utiliza una primitiva paralela cuya función Map lanza tantos procesos lógicos en paralelo como celdas hay en la parte interna de la matriz, utilizando una función de layout que agrupará los procesos lógicos en bloques rectangulares, sobre una topología virtual en forma de rectángulo. Dentro de la primitiva de paralelismo aparecen una o más clausulas *map* que indican el código que se asociará a los procesos lógicos. En este caso sólo aparece una clausula map cuyo contenido se replica en todos los procesos lógicos. La clausula contiene una llamada a la función secuencial para actualizar el valor de la celda asociada al proceso lógico. La función *parindex(dim)* devuelve el índice asociado al proceso lógico para la dimensión deseada. Cada proceso lógicos tiene una copia virtual de las estructuras de datos, de forma que todas las actualizaciones se ejecutan en paralelo independientemente de las agrupaciones, o secuencializaciones generadas por la implementación internamente. La semántica de alto nivel de la primitiva paralela implica que los procesos se sincronizan creando un estado global al final de la misma, en cada iteración. El código generado no tiene porque cumplir con dicha restricción mientras la semántica se mantenga.

En el caso de repartir pocos elementos sobre un mayor número de procesadores virtuales, éstos se dividen en grupos, lo que permite construir particiones recursivas de forma natural. Algunas de las políticas de layout permiten hacer un balanceo de carga dinámico en función de pesos arbitrarios, lo que permite representar algoritmos con particiones de datos recursivas dependientes de datos. En la figura 3 se muestra el código de una función de coordinación que utiliza la primitiva paralela para mapear un espacio de sólo dos índices sobre grupos de procesadores. Esto genera una bipartición recursiva de los procesadores en grupos de diferente tamaño, ajustando las capacidades de cómputo a los tamaños de los subconjuntos de datos. La clausula *reduce* se utiliza al final de la primitiva de paralelismo para construir un estado global a partir de los datos modificados por cada proceso lógico. En el ejemplo se utiliza para llamar a una función que realiza un fusionado de las dos partes del conjunto de datos. Cuando sólo queda un procesador activo en un grupo, la recursión continúa de forma secuencial de forma natural, al tener que ejecutarse los dos procesos lógicos generados en cada etapa en el mismo dispositivo.

```
coordination recursiveDecomp( double data[] ) {

  if ( count(data) >= 2 ) {

    double parts[2][];
    splitElementsInTwoSets( data, parts );

    double weights[2] = { count(parts[0]),
                          count(parts[1]) };

    parallel( Map( [2],
                   groupBalance(weights),
                   linear ) ) {
       map:
          recursiveDecomp( parts[0] );
       map:
          recursiveDecomp( parts[1] );
       reduce:
          mergeElements( parts );
  }
}
```

Fig. 3. cSPC: Extracto de un ejemplo. Partición recursiva.

III. El sistema de transformación Trasgo

La figura 4 muestra un esquema de las capas del sistema de trasformación Trasgo. A la izquierda aparecen las diferentes representaciones que se van utilizando para el código y a la derecha las diferentes capas de transformación que se aplican.

El front-end traduce el lenguaje de entrada (p.e. cSPC) a una representación interna similar al código fuente pero en un formato XML. Esta representación en forma de etiquetas es apropiada para un lenguaje jerárquico y estructurado, y además disponemos de potentes herramientas estándar para identificar y localizar características del código (XPath) y ejecutar transformaciones (Xslt) en documentos XML.

Estas tecnologías se utilizan para escribir de forma compacta y reutilizable módulos de transformación de código. La capa central de transformaciones consta de un analizador y reconstructor de expresiones que identifican las dependencias a partir de las primitivas de coordinación y de los parámetros

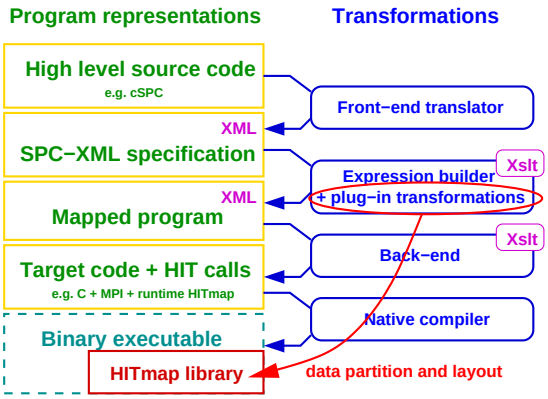

Fig. 4. Estructura del sistema de transformación.

de las llamadas a funciones. En versiones previas de Trasgo en esta capa se procedía también a generar expresiones y código para aplicar *layouts* sobre topologías virtuales. En la versión actual de Trasgo esos módulos han sido movidos a la capa de run-time. Las estructuras de comunicación y/o sincronización también se construyen a partir de las dependencias de datos expresadas por la coordinación de tareas. Las expresiones relacionadas con el cálculo de comunicaciones llaman a las funciones de run-time para obtener la información necesaria, adaptándose en el momento de la ejecución a las partes de las estructuras de datos asignadas a cada procesador virtual por las políticas de layout. Este proceso es una generalización de las técnicas empleadas en el modelo poliédrico para generar código para sistemas de memoria distribuida [6].

El código transformado es reescrito por una capa de back-end en lenguaje C con llamadas a una biblioteca de funciones run-time denominada Hitmap, que incluye funcionalidades para la manipulación de estructuras de datos, módulos de mapeo (layout y topologías virtuales) y constructores de patrones de comunicación y sincronización. El código resultante se compila con un compilador nativo para generar el ejecutable.

IV. Biblioteca de funciones Hitmap

Hitmap [7] es una biblioteca de funciones para el mapeo automático y manipulación de arrays jerárquicos. Las versiones previas de Hitmap estaban orientadas al mapeo sobre un paradigma de paso de mensajes, más apropiado para sistemas de memoria distribuida. En esta sección se describen también las nuevas funcionalidades orientadas a mapear y manejar estructuras de datos entre los dispositivos dentro de un único nodo y cómo se integran con las anteriores.

A. Principales funcionalidades

En la figura 5 se muestra un esquema de los principales módulos de funcionalidad. Hitmap define objetos para declarar y operar con dominios de índices multidimensionales abstractos. Estos pueden usarse para declarar y seleccionar partes de arrays densos, u otras estructuras de datos dispersas [8]. Permite el manejo eficiente de tiles o particiones jerárquicas de dichas estructuras de datos. Incluye funciones para la declaración, reserva de memoria y liberación, copias entre zonas comunes seleccionadas, detección y construcción de *ghost zones* entre dominios de índices, etc.

Contiene también un sistema de plug-ins para la inclusión y utilización de módulos de construcción de topologías (para mapear los procesadores reales a una topología virtual) y de *layout* (para mapear dominios de índices sobre los procesadores de una topología virtual).

Finalmente, contiene también un conjunto de funciones para comunicar eficientemente tiles entre procesadores virtuales, y para construir patrones de comunicación reutilizables. Estas funciones utilizan internamente el estándar MPI para las comunicaciones. Su implementación explota técnicas eficientes de creación de tipos derivados y comunicaciones asíncronas. En Hitmap las comunicaciones se declaran en términos de procesadores virtuales y del resultado de un layout. Al cambiar la topología real de la plataforma de ejecución, o la política de layout aplicada, las estructuras de comunicación se calculan de forma diferente en tiempo de ejecución. De esta forma las estructuras de comunicación o sincronización se adaptan a la plataforma.

B. Módulos de mapeo

Los módulos relacionados con el mapeo se incorporan a la biblioteca a través de dos tipos de interfase. Uno para funciones de topología virtual y otro para funciones de layout. En el código se invocan por el nombre del plug-in.

En el caso de las topologías virtuales cada módulo puede definir los parámetros que el programador debe incluir en la llamada. En la mayor parte de los casos no es necesario ningún parámetro. El sistema incluye automáticamente un parámetro extra con una estructura que representa la información de la topología física. Esta información se obtiene en tiempo de ejecución y/o se completa con información de un fichero de definición de plataforma. El módulo devuelve una estructura que representa la asociación entre procesadores físicos y virtuales y las relaciones de vecindad entre procesadores. Actualmente Hitmap incluye módulos para construir topologías en forma de nube, cuadrados perfectos de procesadores, y varios tipos de paralelotopos multidimiensionales con diferentes restricciones.

En el caso de los módulos de layout el programa-

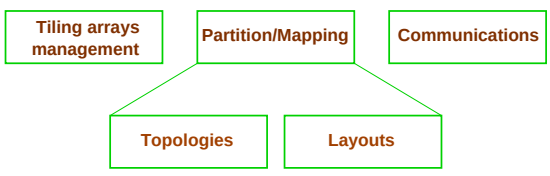

Fig. 5. Principales módulos de Hitmap.

dor debe suplir al menos dos parámetros obligatorios: Un dominio de índices y una topología virtual construida con alguna de las funciones anteriores. Cada módulo de layout puede definir otros parámetros necesarios. La función interna utilizará la información del tamaño de la topología virtual, sus relaciones de vecindad, y el dominio de índices para repartirlos. El resultado es un objeto con información sobre la zona de índices asignada al procesador local y métodos para obtener información de las zonas asignadas a otros procesadores o localizar índices. Actualmente Hitmap incluye módulos de layout para construir bloques multidimensionales, distribuciones cíclicas, balanceo de carga por pesos, particionado de grafos, etc.

C. Hitmap dentro de un nodo

Hitmap provee nuevas funcionalidades para manejar eficientemente los diferentes dispositivos dentro de un nodo en un clúster híbrido. Los diferentes procesos se asocian a dispositivos: Se asignan grupos de núcleos a un proceso de paso de menajes, y se asocia cada dispositivo acelerador a otro proceso de paso de mensajes. La comunicación entre dispositivos o grupos de dispositivos se ejecuta al mismo nivel que entre procesos de diferentes nodos. Internamente la comunicación entre los procesos de un mismo nodo se optimiza a través de la memoria compartida.

Se incluye en la nueva versión de Hitmap un nuevo tipo de módulos de mapeo: *Blocking/Tiling*. Estos módulos reciben un subdominio local y construyen la información necesaria para generar una rejilla de bloques o tiles de un tamaño/grano apropiado para computar de forma paralela en un conjunto de núcleos, o en el dispositivo acelerador asociado al proceso. El módulo utiliza la topología virtual para detectar el dispositivo asociado al proceso. En el caso de aceleradores como GPUs, el resultado indica los tamaños y formas de los bloques de threads. Cada módulo puede incluir diferentes políticas que el programador puede seleccionar en función del tipo de kernel, función secuencial o código a ejecutar, de los patrones de acceso, del uso de las memorias caché, etc.

Finalmente, en el caso de grupos de núcleos, se utiliza la información suministrada por el módulo para lanzar en paralelo tareas que subseleccionan los tiles correspondintes a cada bloque y ejecutan la función deseada utilizando alguna herramienta como OpenMP, TBBs, Cilk, o similar. En el caso de aceleradores, la comunicación de estructuras de datos (tiles) entre la memoria global y el dispositivo queda oculta en la llamada a las funciones. Se define un método común de lanzamiento de kernel o código, que internamente detecta el tipo de dispositivo asociado y ejecuta las acciones especiales necesarias para lanzar un kernel en el caso de una GPU.

D. Combinación de niveles

Las técnicas de partición se pueden combinar en diferentes niveles de forma natural. Dependiendo del tipo de aplicación puede resultar más natural una aproximación top-down o una bottom-up.

Por ejemplo, se puede usar primero una técnica de layout para distribuir un dominio entre procesos, y luego aplicar una técnica de blocking/tiling sobre el subdominio local del dispositivo o grupo de dispositivos. Por el contrario se puede utilizar una técnica de layout para distribuir bloques o tiles ya calculados previamente a lo largo de un conjunto de procesos.

También es posible anidar o construir estructuras recursivas que utilicen layouts en múltiples niveles para generar particiones recursivas a lo largo del conjunto de procesos.

V. Casos de estudio

Hemos seleccionado dos aplicaciones sencillas para probar la aplicación de estas combinaciones en clusters híbridos. La primera es el solver iterativo para PDEs basado en el método de Jacobi presentado en la figura 2. En este caso, en el código generado se utiliza primero un layout de bloques rectangulares para distribuir la estructura de datos matricial entre los procesos de paso de mensajes. Se utiliza posteriormente un módulo de Blocking básico para crear un tiling adecuado para los grupos de cores o GPUs. Los procesos asignados al mismo nodo se sincronizan en cada iteración para hacer las copias de las *ghost zones* o zonas fantasma entre diferentes procesos. Los procesos en diferentes nodos vecinos se comunican las zonas fantasma asíncronamente, de la forma habitual.

El segundo ejemplo es una factorización LU. En este caso, en el código generado se utiliza una técnica de Blocking/Tiling que calcula un tamaño de bloque genérico apropiado para comunicar entre los los diferentes dispositivos o grupos de dispositivos. Estos bloques se distribuyen a lo largo del conjunto de procesos utilizando una técnica de layout cíclica. El resultado es una distribución bloque-cíclica adaptada a las características de la plataforma de ejecución. Dentro de un nodo los procesos se sincronizan para copiar datos de las zonas compartidas. Entre nodos los procesos se comunican con operaciones colectivas mandando y recibiendo bloques o grupos de bloques según los patrones del algoritmo.

La biblioteca Hitmap ha demostrado previamente una buena eficiencia y escalabilidad en sistemas de memoria distribuida y compartida utilizando el paradigma de paso de mensajes [7]. Los experimentos preliminares con el nuevo sistema de capas incluyendo las abstracciones que se utilizan para los dispositivos dentro de un mismo nodo demuestran que la sobrecarga introducida por estas nuevas abstracciones es pequeña y fija, no afectando negativamente a la escalabilidad. En el caso de grandes cargas computacionales la sobrecarga es despreciable.

VI. Conclusión

En este trabajo se presentan innovaciones en el desarrollo de Trasgo, un sistema de programación capaz de generar códigos eficientes y adaptables a partir de

código abstracto y portable. Se muestra como ésto se consigue gracias a mover decisiones de mapeo a tiempo de ejecución dejando que las estructuras de comunicación y sincronización dependan de los resultados de las mismas. El sistema de generación de código se apoya en Hitmap, una biblioteca run-time que provee de las herramientas y utilidades necesarias para el mapeo en tiempo de ejecución. Se muestran nuevas técnicas orientadas a explotar múltiples niveles de paralelismo en clusters híbridos y/o heterogéneos.

El marco de trabajo planteado por el sistema Trasgo presenta aún varios retos importantes. Entre ellos se pueden destacar los siguientes. Incluir políticas automáticas de asignación de grupos de dispositivos a procesos de paso de mensajes para equilibrar la capacidad de computo. La integración de un sistema de generación automática del código de kernels para GPU u otros aceleradores a partir del lenguaje de entrada. Políticas de blocking/tiling que utilicen una mejor caracterización de los códigos para derivar tamaños de bloque apropiados. O la inclusión de técnicas de partición y layout irregulares que adapten la carga a las capacidades de diferentes dispositivos.

Agradecimientos

Este trabajo ha sido financiado parcialmente por el Ministerio de Industria (CENIT OCEANLIDER), Ministerio de Economía y programa FEDER de la Unión Europea (red CAPAP-H3 TIN2010-12011-E, red CAPAP-H4 TIN2011-15734-E, Proyecto MOGECOPP TIN2011-25639), y el proyecto HPC-EUROPA2 (project number: 228398), con el apoyo de la Comisión Europea (Capacities Area - Research Infrastructures Initiative).

Referencias

[1] A. Gonzalez-Escribano and D.R. Llanos, "Trasgo: A nested-parallel programming system," *The Journal of Supercomputing*, vol. 58, no. 2, pp. 226–234, 2011.

[2] A.J.C. van Gemund, "The importance of synchronization structure in parallel program optimization," in *Proc.* 11th *ACM ICS*, Vienna, Jul 1997, pp. 164–171.

[3] K. Lodaya and P. Weil, "Series-parallel posets: Algebra, automata, and languages," in *Proc. STACS'98*, Paris, France, 1998, vol. 1373 of *LNCS*, pp. 555–565, Springer-Verlag.

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

[5] 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.

[6] U. Bondhugula, "Automatic distributed memory code generation using the polyhedral framework," Tech. Rep. IISc-CSA-TR-2011-3, IISc, 2011.

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

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