Trasgo Frontend: hacia una generación automática de código paralelo eficiente

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

Resumen— **El frontend del sistema Trasgo (TF) es una herramienta que nos permite traducir un código paralelo de alto nivel a una representación intermedia basada en XML. Esta representación, a su vez se utiliza para generar código eficiente para sistemas de memoria compartida o distribuida. En este trabajo se enumeran las características de cSPCv.2, un lenguaje de descripción de problemas paralelos de alto nivel, que permite al frontend la extracción de la información necesaria para automatizar las comunicaciones entre procesos.**

Palabras clave— **Lenguajes paralelos, mapping automático, generación de código**

I. Introducción

HOY en día es un trabajo laborioso desarrollar aplicaciones informáticas que exploten la capacidad de cómputo simultaneo de las máquinas paralelas (clusters, multicores, many-cores, etc.). Aún no se ha impuesto un modelo de programación único para paralelismo. En la última década han surgido nuevos lenguajes y modelos de programación paralela que intentan facilitar al programador una visión más abstracta y portable de la arquitectura de la máquina. APIs portables, como los de paso de mensajes (p.e. MPI, PVM), o los mecanismos de manejo de threads (p.e. pThreads) facilitan la programación ocultando los detalles de la máquina, pero permitiendo obtener un buen rendimiento. Sin embargo, programar con modelos de coordinación no restringidos, puede ser extremadamente propenso a errores e ineficiente, ya que las dependencias de sincronización que puede generar un programa son complejas y difíciles de analizar por programadores o compiladores [1]. Por tanto las decisiones importantes en la trayectoria de implementación, en la planificación o en la partición de datos se convierten en decisiones extremadamente difíciles de optimizar.

El modelo de *paralelismo anidado* permite dar respuesta a estas cuestiones. Este modelo se basa en utilizar para expresar paralelismo una única estructura que permite especificar tareas de cualquier granularidad que se pueden ejecutar en paralelo, y que pueden anidarse libremente. Este modelo implica un sistema de sincronización muy sencillo y tiene grandes ventajas para el análisis y generación de código específico para una plataforma concreta. Los modelos basados en paralelismo anidado presentan un punto medio entre expresividad, complejidad y facilidad de programación. La restricción de anidamiento en sus estructuras de sincronización permite utilizar técnicas de compilación y planificación específicas que incrementan la portabilidad y eficiencia (p.e. work-stealing). Algunos ejemplos de modelos de programación basados en paralelismo anidado incluyen BSP [2], basados en esqueletos paralelos (p.e. SCL [3], Frame [4]), Cilk [5], or CUDA [6] (que restringen el paralelismo anidado a dos niveles).

Trasgo [7] es un sistema de programación que permite obtener código paralelo portable a partir de una representación de alto nivel del problema. Trasgo permite expresar con facilidad combinaciones complejas de paralelismo de datos y de tareas, utilizando una estructura común y ocultando los detalles del particionado de datos y tareas y la planificación.

En este artículo se propone cSPCv.2, una ampliación del lenguaje de alto nivel que sirve como entrada a Trasgo [8]. cSPC es una extensión del lenguaje C pensada para facilitar al programador habitual de C la especificación de aplicaciones paralelas en el modelo propuesto por Trasgo. Con esta ampliación seremos capaces de extraer toda la información necesaria para construir la partición de datos, planificación, balanceo de carga, etc, a partir de una única línea de código cSPCv.2.

El resto de este artículo se estructura como sigue. La sección II muestra la arquitectura de Trasgo.

La sección III enumera las características deseables para un lenguaje de programación ideal con paralelismo anidado, y las decisiones de diseño escogidas para la realización del código cSPCv.2. La sección IV definirá el lenguaje de programación cSPCv.2 creado. Seguidamente en la sección V describe el formato de la salida del frontend, para las distintas entradas en cSPCv.2. La sección VI describe un caso de estudio práctico implementado. Finalmente, la sección VII enumera las conclusiones.

II. Arquitectura de Trasgo

Trasgo es un sistema de traducción y compilación. El núcleo de Trasgo se compone de: un frontend que traduce el lenguaje de entrada cSPC a un lenguaje intermedio en XML, un sistema de reescritura de expresiones y un run-time con funciones de mapeo automático.

El modelo de arquitectura de Trasgo se muestra en la figura 1. La entrada a Trasgo es un código paralelo explícito, basado en un lenguaje tradicional añadiendo algunas extensiones para la programación en paralelo (A). En concreto, Trasgo utiliza por defecto un lenguaje basado en una extensión de C denominado cSPC. El código escrito en este lenguaje de alto nivel será traducido, usando un frontend, a un código intermedio basado en XML. Este lenguaje intermedio se denomina xSPC(XML Series-Parallel Coordination)(B). Se utiliza un código intermedio en XML ya que existen potentes herramientas de transformación y generación de código como Xslt y Xpath2.0.

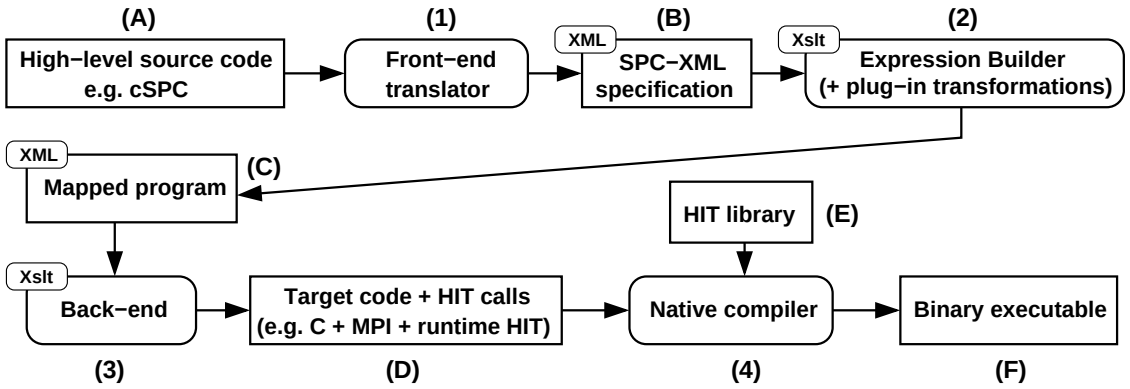

Fig. 1. Sistema de mapeo automático Trasgo

Con estas herramientas somos capaces de detectar patrones en las estructuras del programa y aplicar transformaciones(2). Trasgo dispone de 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(C). Finalmente un back-end transforma el código anotado en el código objetivo escrito en C tradicional(D). Este código resultante contiene llamadas a una librería de comunicaciones y run-time, denominada Hitmap(E). Este código en C con llamadas a la librería Hitmap será compilado con un compilador nativo para producir el código ejecutable (F) [7]. El presente trabajo se centra en un nuevo diseño del lenguaje de entrada cSPCv.2(A) y del frontend(1) que lo traduce a xSPC(B), para permitir extraer la información necesaria para un sistema de generación de expresiones más completo.

III. Principios de diseño del lenguaje cSPC

En esta sección se describen principios y elecciones de diseño utilizados para la nueva versión del lenguaje de entrada cSPCv.2, que permiten al frontend la extracción de la información necesaria en las siguientes fases de reconstrucción del código paralelo:

- La entrada será un lenguaje de coordinación [2]. El código secuencial se expresará en un lenguaje de programación tradicional. En nuestro caso C. Se utilizarán extensiones (nuevas primitivas, estructuras y modificadores) para expresar la coordinación entre las tareas secuenciales. Para simplificar la distinción entre código secuencial y de coordinación, el código secuencial estará siempre encapsulado en un tipo especial de función. En las expresiones de código de coordinación, sólo se podrá ejecutar código secuencial mediante llamadas a dichas funciones.
- Se utilizará una única primitiva o estructura para expresar paralelismo, que podrá anidarse tantas veces como sea necesario, incluso recursivamente a través de llamadas a la propia función. Dentro de esta estructura, se expresará la funcionalidad a ejecutar por cada proceso lógico. Para poder expresar paralelismo de datos de una forma sencilla, el programador podrá expresar una única funcionalidad que se replicará en cada proceso lógico.
- Las funcionalidades que se realizarán en paralelo deben poder especificarse sobre elementos de grano arbitrario. El programador no debe ser forzado a tomar decisiones sobre la granularidad en función de recursos de la máquina, sino en función de la aplicación o el algoritmo. Eso permitirá poder particionar de forma automática las estructuras de datos y agrupar los procesos lógicos en tareas planificables, sin restricciones artificiales.
- Los programas expresados en este lenguaje tienen que ser independientes de las políticas de reparto de datos que se aplicarán en tiempo de ejecución. A través de un parámetro de la primitiva de paralelismo el programador podrá sugerir al sistema una política de partición y distribución de datos adecuada al problema. Pero el resto del código será independiente de las decisiones tomadas por la misma, por lo que podrá cambiarse sin afectar al resto del código.
- Antes de una primitiva paralela, la computación tendrá un estado global, y un sólo hilo lógico de ejecución. Al lanzar tareas en paralelo, cada una tendrá una copia local del estado que podrá modificar independientemente. Cuando todas las tareas hayan llegado al punto de sincronización se volverá a consolidar un estado global único. La estructura de expresión de paralelismo debe proveer de una clausula que permita reducir adecuadamente variables que hayan sido modificadas de forma diferente en cada proceso lógico, para generar el estado global.
- Para relajar las sincronizaciones globales, transformándolas en patrones de comunicación eficientes, es necesario conocer las dependencias de datos entre procesos lógicos de estructuras paralelas que se ejecutan una detrás de otra (incluidas las forzadas por la ejecución de diferentes iteraciones de un bucle secuencial que contiene estructuras paralelas). Puesto que la reconstrucción de las dependencias es complicada a partir

```
/**
* Poisson equation
* Function to update one cell element
*/
void updatecell( in double up,
                 in double down,
                 in double left,
                 in double right,
                 out double result )
{
    *result = (up + down + left + right ) / 4 ;
}
```

Fig. 2. Ejemplo de declaración de función secuencial.

de un código tradicional, es deseable que el lenguaje obligue al programador a especificar las dependencias estrictamente necesarias. En nuestro caso, puesto que el código secuencial está encapsulado en funciones, se añade un modificador a los parámetros en la definición de las funciones que indica el carácter de lectura/escritura del mismo. Esta información ayudará a determinar las dependencias de datos entre funciones cuando se tomen las decisiones de granularidad y partición de datos.

IV. Formato y definición del lenguaje cSPCv.2

El lenguaje cSPCv.2 es una extensión del lenguaje C de la gramática de Kernighan and Ritchie [9] creado para expresar algoritmos paralelos de una forma simple, explícita e intuitiva para los programadores acostumbrados a lenguajes clásicos como C. La extensión cSPCv.2 consiste en una serie de primitivas que nos proporcionarán toda la información necesaria para el sistema de mapeo automático Trasgo.

A. Declaración de funciones

Las funciones clásicas de C contienen un trozo de código secuencial independiente de la ejecución paralela. La única diferencia es que se fuerza al programador a incluir para cada parámetro un modificador (*in*, *out*, *inout*) que indica si el parámetro es de entrada, entrada-salida o salida respectivamente. En la versión actual se fuerza a que el tipo de devolución de las funciones sea siempre *void*, utilizando parámetros *out* para devolver valores. Estos modificadores permitirán más adelante detectar las dependencias de datos entre las tareas paralelas. Como vemos en el ejemplo de la figura 2, la única diferencia apreciable con el lenguaje C se da en el paso de parámetros.

B. Declaración de funciones de coordinación

Esta sección explicará las estructuras de programación y las primitivas propuestas para paralelismo. Las funciones de coordinación son una extensión propuesta de cSPC. Se distinguen por el modificador *coordination* en la definición de la función delante del tipo de la misma. La utilidad y función principal de estas funciones consiste en componer el código a ejecutar en serie o paralelo. Un ejemplo de estas funciones se ve en la figura 3.

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

  double matrix[10][10];
  int nIter=10;

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

    parallel( Map( matrix[1:$-1][1:$-1].shape,
                   blocks,
                   topology2D ) )
    {
      parblock: updatecell(
          matrix[ paridx(0)-1 ][ paridx(1) ],
          matrix[ paridx(0)+1 ][ paridx(1) ],
          matrix[ paridx(0) ][ paridx(1)-1 ],
          matrix[ paridx(0) ][ paridx(1)+1 ],
          matrix[ paridx(0) ][ paridx(1) ]
      );
    }
  }
}
```

Fig. 3. Ejemplo de declaración de función de coordinación.

La definición del dominio de índices de un array se diferencia de la definición en C clásico, porque admite una notación similar a Fortran95 (<inicio>:<final>:<salto>). Esto permite arrays que no tienen porque comenzar en la posición cero, o que pueden expresar un salto en los índices, almacenándose los datos de forma contigua en memoria.

Podemos encontrarnos con cinco tipos diferentes de sentencias de coordinación dentro del cuerpo de una función coordinada:

1. Sentencias de iteración o selección. Al igual que en C se pueden emplear las sentencias *if*, *if-else*, *do-while* y *while*.
2. Sentencia *loop*. Es una simplificación de una estructura *for* para estructuras iterativas controladas por un único índice. Su ventaja es que permite identificar más claramente el nombre del índice y los datos de comienzo, fin y paso del índice.
3. Sentencias de llamada a una función. Este será el recurso utilizado para la modificación de variables ya que no se permiten expresiones, ni asignaciones en las funciones coordinadas. Si queremos modificar alguna variable o realizar alguna expresión llamaremos a una función que lo haga. Puesto que en la definición de funciones los parámetros indican si son de entrada o salida se simplifica en gran medida el calculo de dependencia de datos. conociendo en cada momento que variable a podido ser modificada o no.
   Los valores de los parámetros pueden referirse a variables o a subdominios de arrays. Los subdominios se expresan en cada dimensión con una notación similar a Fortran95 (<inicio>:<final>:<salto>). En las expresiones de los índices se puede utilizar el símbolo $ para indicar el último índice del dominio en una dimensión (ver figura 3). Es posible añadir a los parámetros de entrada una alternativa para cier-

```
updateBorder( matriz[ -1:1 ][ 20:30:2 ]
              :
              matriz[ 0:2 ][ 20:30:2 ]
)
```

Fig. 4. Ejemplo de selección con valor alternativo.

tos casos en los que la selección de rangos de un array no se encuentre dentro del dominio original del array. Un ejemplo se muestra en la figura 4, en el que la sintaxis de los parámetros es (<valor>:<valor>). Si la primera selección de subdominio se sale fuera del dominio original, el parámetro tomará el segundo valor. Esto facilita la programación de, por ejemplo, códigos de simulación con condiciones de contorno.

4. La sentencia *Map* es una sentencia dedicada a la declaración de las políticas y detalles necesarios para distribuir un dominio de datos (p.e. el espacio de índices de un array). Su estructura es la siguiente: Map(<shape>, <layout>, <topology>). El parámetro *shape* especifica el dominio de datos que se distribuirá. El parámetro *Layout* indica el nombre y parámetros de una política de particionado. El parámetro *topology* indica el nombre de una política para crear una topología virtual de los procesadores. El resultado de esta función es una variable del tipo *map* que contiene la información del trozo de dominio asignado al proceso local y métodos para calcular los trozos asignados a otros procesos y la identificación de procesos con relaciones de vecindad.
5. La sentencia *parallel* declara una estructura para ejecutar tareas en paralelo. A esta sentencia se le pasa un único parámetro, el parámetro *Map*. Dentro de la estructura se encontrarán uno o varios trozos de código anotados por cláusulas *parblocks*. Estos son los diferentes bloques de sentencias que se ejecutarán en paralelo. La estructura *parallel* lanza tantos procesos lógicos como elementos haya en el dominio del parámetro *Map*. Si se han declarado varios *parblocks* cada uno se asocia a un proceso lógico. Si solo hay un *parblock* cada proceso lógico ejecuta una réplica del código. Estos procesos lógicos se agruparán en tareas siguiendo las políticas especificadas en los otros dos parámetros del *Map*. Dentro de una estructura parallel se pueden utilizar una función predefinida, *paridx(<dimension>)* (ver figura 3), que permite recuperar el índice asignado al proceso lógico en el que se esta ejecutando el código del *parblock*. También es posible introducir una cláusula *reduce* para especificar las reducciones necesarias para consolidar el estado global a partir de los valores locales de cada tarea.

V. Frontend del sistema Trasgo, cSPC-xSPC

El frontend de Trasgo es un traductor capaz de analizar el código cSPCv.2 (tanto funciones en C clásico como funciones de coordinación) y traducirlo a la representación interna en xSPC. xSPC es un lenguaje de etiquetas, este tipo de lenguajes conlleva que cada parte de la estructura del código viene delimitada por una etiqueta de inicio y otra de cierre. A partir de xSPC es posible sintetizar gran cantidad de información usando herramientas de transformación de XML(Xslt 2.0 y Xslt).

```
<call line="27" name="updatecell">

  <param>
    <select line="30" name="matrix">
      <dim>
        <par-index line="30">0</par-index>
      </dim>

      <dim>
        <op type="+">
          <par-index line="30">1</par-index>
          <value line="30">1</value>
        </op>
      </dim>
    </select>
  </param>

  <param>
    <select line="31" name="matrix">
      <dim>
        <par-index line="31">0</par-index>
      </dim>

      <dim>
        <par-index line="31">1</par-index>
      </dim>
    </select>
  </param>

  ...

</call>
```

Fig. 5. Extracto de una llamada a función en xSPC.

El código xSPC creado tendrá un formato similar al de la figura 5, en el que vemos un extracto de la representación en xSPC de la llamada a la función *updatecell* en la función coordinada de la figura 3.

Las funciones coordinadas propias de cSPCv.2 son completamente traducidas al lenguaje de etiquetas para así poder extraer la información necesaria para calcular después las particiones de datos y las comunicaciones. Sin embargo para las funciones de C clásico que contienen el código secuencial solo se transforma en etiquetas la cabecera de las funciones, manteniendo el código interno de la función en el mismo formato. Un ejemplo lo vemos en la figura 6.

El análisis del código xSPC permite detectar estructuras y patrones complejos de paralelismo. El sistema de generación de código utiliza módulos (plug-ins) asociados a los detalles estructurales del código o de la máquina objetivo, para optimizar la generación del código final. El resultado es un programa en lenguaje C, con llamadas a la librería Hitmap, en donde la partición de datos, las comunicaciones y la planificación han sido preparados por la transformación anterior. Internamente la versión actual de Hitmap utiliza MPI para realizar comunicaciones y sincronizaciones [10].

```
<function line="5" name="updatecell">

  <lang name="C"/>

  <description>
    /**
    * Poisson equation
    * Function to update one cell element
    */
  </description>

  <parameters>
    <in type="double" line="5" name="up">
      <shape>
      </shape>
    </in>

    ...

    <out type="double" line="11" name="result">
      <shape>
      </shape>
    </out>
  </parameters>

  <code line="8">
    *result = right + left + up + down / 4 ;
  </code>

</function>
```

Fig. 6. Extracto de declaración de función secuencial.

VI. Caso de estudio

Con todo lo explicado podemos ver que a partir del lenguaje cSPCv.2 podemos crear un programa en lenguaje C con llamadas a la librería Hitmap. Con cSPCv.2 reducimos el número de líneas de código relacionadas con los detalles de paralelismo. Como ejemplo mostramos el código de un *Cellular-Automata en 2D*, una aplicación orientada a paralelismo de datos [11]. El programa calcula la distribución de calor en un espacio bidimensional utilizando el método de Jacobi para resolver el sistema de ecuaciones diferenciales parciales(PDE) derivado de la ecuación de Poisson.

La parte de coordinación del código se muestra en la figura 3. La forma de actualizar cada punto de la matriz en cada iteración se especifica en la función *updatecell* de la figura 2, la cual realiza la media entre los valores de los elementos adyacentes en la matriz. Se realizará un bucle determinado por la sentencia *loop* durante un número de iteraciones determinado. Ese bucle lanzará en cada iteración una estructura paralela especificada en la sentencia *parallel*. Como se observa en el parámetro *Map* la partición de datos elegida es por bloques en cada dimension y la topología de los procesadores elegida es una matriz de 2D (*topology2D*). El parblock que realiza la función *updatecell* se realiza para cada elemento de la matriz en paralelo. Una vez terminadas las diferentes iteraciones se tendrá el resultado deseado.

El sistema Trasgo generará un código donde un subdominio del array se asigna a cada proceso MPI. La función de actualización de cada celda local se ejecuta en secuencial dentro del proceso. Para mantener la semántica paralela debe construirse automáticamente una copia del trozo local de la matriz para actualizar los elementos con los valores de la iteración anterior. Al final de cada iteración la sincronización implícita del final de la estructura *parallel* debe sustituirse por una comunicación entre vecinos de los bordes de la matriz, deducida a partir de las dependencias de datos expresadas en la llamada a la función *updatecell*. La implementación actual de Trasgo contiene un sistema de generación de expresiones básico capaz de generar comunicaciones optimizadas para casos sencillos como el del caso de estudio.

Se ha desarrollado un código optimizado manualmente en C con MPI para este problema. Ya que las estructuras de comunicación que genera Trasgo son idénticas, y la librería Hitmap es muy eficiente en el acceso secuencial a los elementos, no se producen perdidas de rendimiento apreciables. Se observa también que el número de líneas dedicadas a los cálculos y comprobaciones asociadas a la partición de datos, recepción de estos y creación de tipos compuestos en la versión MPI manual, se reduce dramáticamente comparándolo con la composición paralela en cSPCv.2. Ésta utiliza una única primitiva *Map*, que se escribe en una sola línea de código, para toda la partición de datos y tareas.

VII. Conclusiones

En este trabajo se ha presentado un nuevo frontend para el sistema de compilación Trasgo. Este nuevo frontend recibe como entrada códigos en cSPCv.2, un lenguaje de programación sencillo, simple e intuitivo para un programador de C, que facilita la programación en paralelo. La salida del frontend es xSPC, un lenguaje de etiquetas XML que representa el mismo código. xSPC se puede procesar con herramientas estándar de identificación de propiedades y reescritura del documento, para extraer información y generar un código que se adapta al entorno de ejecución.

Se presentan los criterios de diseño del nuevo lenguaje cSPCv.2 que permiten extraer la información necesaria para reconstruir una estructura de comunicaciones eficiente. Se presenta un caso de estudio que muestra que el uso de cSPCv.2 y Trasgo permiten obtener códigos eficientes a partir de especificaciones paralelas de alto nivel. cSPCv.2 permite al programador programar en paralelo sin entrar en los detalles de la comunicación y la partición de datos. Estos detalles se realizarán automáticamente utilizando la librería de run-time Hitmap.

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, TIN2011-25639, red CAPAP-H4 network TIN2011-15734-E), y el proyecto HPC-EUROPA2 (project number: 228398), con el apoyo de la Comisión Europea (Capacities Area - Research Infrastructures Initiative).

Referencias

[1] S. Gorlatch, "Send-Recv considered harmful? Myths and truths about parallel programming," in *PaCT'2001*, V. Malyshkin, Ed. 2001, vol. 2127 of *LNCS*, pp. 243–257, Springer-Verlag.

[2] L.G. Valiant, "A bridging model for parallel computation," *Comm.ACM*, vol. 33, no. 8, pp. 103–111, Aug 1990.

[3] J. Darlington, Y. Guo, H.W. To, and J. Yang, "Functional skeletons for parallel coordination," in *Europar'95*, 1995, LNCS, pp. 55–69.

[4] M. Cole, "Frame: An imperative coordination language for parallel programming," Tech. Rep. EDI-INF-RR-0026, Div. Informatics, Univ. of Edinburgh, Sep 2000.

[5] R.D. Blumofe and C.E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proc. Annual Symp. on FoCS*, Nov 1994, pp. 356–368.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar 2008.

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

[8] Javier Sáez Villagrá, "Front-end para un sistema de mapeo automático de paralelismo anidado," 2008, Proyecto fin de carrera.

[9] Nestor Gomez Muoz Brian W.Kernighan, Dennis M. Ritchie, *El lenguaje de programación C*, Prentice-Hall.

[10] Javier Fresno, Arturo Gonzalez-Escribano, and Diego R. Llanos, "Automatic Data Partitioning Applied to Multigrid PDE Solvers," in *Euromicro PDP 2011*, February 9-11, Ayia Napa, Cyprus, Feb. 2011, pp. 239–246, IEEE.

[11] Diego R. Llanos Javier Sáez-Villagrá, Arturo González Escribano, "Front-end para un sistema de mapeo automático de paralelismo anidado," in *XIX Jornadas de paralelismo*, 2008.