Front-End para un sistema de mapeo automático de paralelismo anidado

Javier Sáez-Villagrá, Arturo González-Escribano, Diego R. Llanos[1]

Resumen— **El paralelismo anidado, como modelo de programación, resulta atractivo por su sencillez de comprensión y uso. Para este modelo, diversas técnicas de mapeo permiten generar automáticamente ejecutables eficientes, adaptados a la máquina para la que se compila el programa. El framework SPC-XML es una herramienta que permite incluir de forma incremental dichas técnicas de mapeo. Genera programas en lenguaje C con llamadas a librerías MPI para gestionar las comunicaciones y sincronizaciones. Sin embargo, su interfaz de programación utiliza una representación intermedia en XML. En este trabajo presentamos un front-end para el framework SPC-XML. Este permite escribir algoritmos de paralelismo anidado con un mínimo esfuerzo de aprendizaje para programadores habituados al lenguaje secuencial C, obteniendo ejecutables eficientes sin necesidad de especificar detalles sobre particiones de datos o comunicaciones.**

Palabras clave— **Paralelismo-anidado, mapeo, lenguajes de programación.**

I. Introducción

UNA práctica común para implementar aplicaciones de alto rendimiento es utilizar el modelo de programación de bajo nivel asociado a la máquina de destino. Los APIs portables, como los interfaces de paso de mensajes (p.e. MPI,PVM), proveen de una capa de abstracción sobre la arquitectura de la máquina, permitiendo obtener todavía un buen rendimiento. Sin embargo, estos modelos permiten una gran flexibilidad en las formas de sincronización y comunicación. Implican modelos no-restringidos de coordinación y la necesidad de que el programador implemente manualmente todas las decisiones sobre mapeo, distribución de datos, balance de carga, comunicaciones y sincronizaciones. Esto lleva a programas poco portables, difíciles de optimizar y depurar, suponiendo un alto coste de desarrollo y mantenimiento. Las dependencias que se generan son difíciles de analizar manual o automáticamente por un compilador [1]. Se han propuesto otros lenguajes de programación paralela de un nivel más abstracto, que restringen los posibles mecanismos de sincronización que puede explotar el programador (ver p.e. [2]). Estos modelos son más fáciles de entender y programar. Además, se pueden desarrollar herramientas y técnicas que ayudan a tomar las decisiones de mapeo en el momento de la compilación.

Los modelos de paralelismo anidado presentan un punto medio entre expresividad, complejidad y facilidad de programación [2]. La estructura de sincronización se basa en el posible anidamiento de estructuras paralelas independientes. Presentan un modelo semántico simple y bien definido [3], modelos analíticos de cálculo de costes [4], [5] y técnicas de planificación eficientes [6]. Ejemplos de modelos de programación paralela basados en paralelismo anidado incluyen BSP [7], skeleton based (e.g. SCL [8], Frame [9]), SPC [4], Cilk [10] y CUDA [11].

En [12] presentamos SPC-XML, un framework de programación paralela-anidada, que puede extenderse con plug-ins para incluir nuevas técnicas de mapeo y planificación. El interfaz de este framework utiliza un lenguaje intermedio de representación de programas paralelos basado en XML. En sus últimas versiones el lenguaje se denomina Xspcl. Está construido como un lenguaje de coordinación [13]. Por tanto las trozos de código secuencial se especifican en un lenguaje de programación secuencial clásico, como C, y se provee de un lenguaje nuevo para especificar la composición de esos trozos. Xpscl se basa en un álgebra de procesos que permite componer las tareas secuenciales de forma secuencial o paralela, de forma anidada y/o recursiva.

En este trabajo proponemos un nuevo lenguaje C-SPC y la correspondiente herramienta de front-end para el framework SPC-XML. El lenguaje tiene las mismas funcionalidades que Xpscl, pero es más próximo al programador de un lenguaje secuencial tradicional. Este lenguaje se presentará como un lenguaje de coordinación, una extensión de C para la especificación de la sincronización. El lenguaje utilizará los mecanismos abstractos de SPC-XML para evitar que el programador tenga que especificar detalles de bajo nivel sobre particiones de datos, planificación, balance de carga, etc. Por tanto, permitirá al programador cambiar las distribuciones de datos, o los mecanismos de implementación de una forma simple y transparente. De esta forma se podrán obtener programas portables y fácilmente adaptables a partir de especificaciones algorítmicas paralelas.

En la sección 2 presentaremos algunos conceptos básicos de SPC-XML que se reflejan en el nuevo lenguaje. En la sección 3 describiremos el lenguaje y los recursos que este ofrece al programador, utilizando ejemplos ilustrativos. En la sección 4 mostraremos como el lenguaje permite incrementar la portabilidad del código, generando automáticamente implementaciones eficientes con un esfuerzo mínimo. En la sección 5 presentamos las conclusiones.

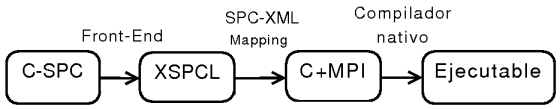
II. Conceptos básicos de SPC-XML

SPC-XML es un framework para programación paralela anidada [12]. Los programas se especifican en algún lenguaje que se traduce a través de un front-end a una representación interna en XML. Este len-

[1]Dpto. de Informática, Univ. de Valladolid, e-mail: javsaez29@gmail.com, {arturo,diego}@infor.uva.es

guaje se denomina Xspcl. A partir de estas especificaciones, es posible detectar y explotar gran cantidad de información estructural sobre el código. Para ello se utilizan herramientas de transformación de XML (Xslt 2.0). Estas herramientas permiten disparar módulos (plug-ins) que se han registrado especificando los detalles estructurales del código o de la máquina objetivo para los que son más adecuados. El resultado después de aplicar los módulos de reestructuración y mapeo es un programa en lenguaje C, con llamadas a librerías MPI donde los detalles de partición de datos, comunicación, planificación han sido introducidos por el sistema de transformación. Los programas se apoyan en una librería propia del framework que incluye un pequeño run-time system extensible, con herramientas de manejo de variables, arrays, mecanismos de balance de carga dinámicos, etc. El proceso de compilación de un programa se muestra en la figura 1.

Fig. 1. Fases de compilación



A. Shapes

SPC-XML utiliza variables array de forma generalizada. Definimos *Shape* como la signatura de un array: el número de dimensiones y el numero de elementos en cada dimensión.

B. Topologías virtuales automáticas

El sistema de mapeo se basa en los siguientes elementos, descritos gráficamente en la figura 2. El programador no trabaja nunca con la topología física o el número de procesadores de la máquina. El lenguaje permite definir topologías virtuales en función de un shape y una *Función de topología*. Estas funciones son módulos del framework que asignan internamente los procesadores físicos a la forma del shape. El framework actual sólo contiene módulos para mapear un número genérico de procesadores en forma de array. En el futuro, el framework podría ser ampliado para incluir shapes que definan grafos o estructuras más genéricas y módulos de mapeo más complejos.

C. Procesos lógicos y Funciones de layout

Cuando el programa necesita ejecutar secciones de código en paralelo, se utiliza una primitiva que define *Procesos lógicos*. Estos pueden definirse enumerándolos (más apropiado para paralelismo de tareas) o como el shape de un array (más apropiado para paralelismo de datos). A cada proceso lógico se le asigna una tarea. La asignación de datos (de uno o más arrays) a cada proceso lógico se puede controlar utilizando una *Función de layout*. El layout asigna los índices de los procesos lógicos a los procesos de la topología virtual. Las funciones de layout son identificadores asociados a módulos del framework. Estos módulos pueden implementar mecanismos de reparto regular en compilación, basados en fórmulas (como asignación por bloques en una o más dimensiones, stride, etc.) También pueden estar asociados a mecanismos dinámicos de balance de carga, que se corresponden con módulos del run-time.

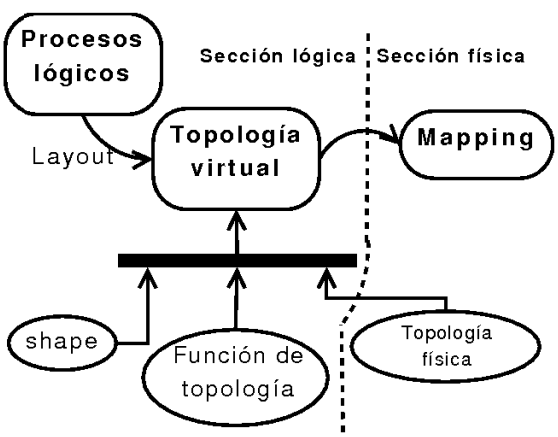
Por tanto, el programador sólo utiliza identificadores de funciones de layout y topología para especificar el paralelismo. El sistema se encarga de generar las fórmulas de reparto, balancear la carga, las comunicaciones necesarias, etc.

D. Dependencias y optimizaciones

La semántica del modelo propuesto implica que los procesos paralelos son independientes unos de otros. El sistema provee de mecanismos semi-automáticos para evitar que al asignar diferentes procesos-lógicos al mismo procesador físico, la ejecución del primero pueda afectar a variables de entrada del segundo. Para ello, y para poder calcular las comunicaciones necesarias, el sistema debe poder determinar las dependencias de datos entre las secciones de código secuencial. El lenguaje fuerza al programador a que éstas se especifiquen como funciones, con un interfaz que defina explícitamente si los datos son de entrada, salida o entrada/salida.

Gracias a la información de dependencias el sistema también realiza reestructuración de código y optimizaciones diversas, incluyendo eliminación de barreras. Esto permite generar códigos finales que utilizan las estructuras de sincronización del algoritmo original, evitando posibles restricciones de sincronización asociadas al paralelismo anidado (barreras).

Fig. 2. Mapeo de tareas a procesadores en SPC-XML



III. Definición del lenguaje C-SPC

C-SPC es una extensión del lenguaje C, cuyo objetivo es crear un lenguaje para expresar algoritmos paralelos de forma explícita. Pretende además, ser intuitivo para programadores acostumbrados a lenguajes clásicos imperativos. Esta extensión del lenguaje consiste en una serie de estructuras de programación o primitivas con las que abarcaremos toda la funcionalidad del lenguaje intermedio Xspcl. Para más

detalle acerca de la explicación de cada sentencia o estructura ver [14].

A. Declaración de funciones

Las funciones encapsulan un trozo de código secuencial. Independiente desde el punto de vista de la ejecución paralela. Con un interfaz claro de entrada y salida que permita derivar posibles dependencias de datos cuando se componen con otras funciones.

Fig. 3. Ejemplo de función C-SPC

```
void pivoting(inout double vector[], out int
 pivotPos){
  int lo = 0;
  int hi = outVcard(vector,0)-1;;
  double pivot = outV1(vector,0);
  while( lo < hi ) {
/*SEARCH BACKWARD FOR LESSER THAN PIVOT*/
    while ((outV1(vector,hi) >= pivot)
             && (lo < hi)) { hi--; }
    if (lo != hi){
     outV1(vector,lo) = outV1(vector,hi);
     lo++;
    }
/*SEARCH FORWARD FOR GREATER THAN PIVOT*/
    while (outV1(vector,lo) <= pivot
             && lo < hi){ lo++; }
    if (lo != hi){
     outV1(vector,hi) = outV1(vector,lo);
     hi--;
    }
  }
  outV1(vector,lo) = pivot;
  outV(pivotPos) = lo;
}
```

Como muestra el ejemplo de la figura 3, la declaración de una función en C-SPC difiere de C en dos aspectos:

- Declaración de parámetros: Como hemos comentado, el Framework necesita conocer el ámbito de actuación de cada parámetro para detectar las dependencias de datos, por lo que hay que indicárselo mediante las palabras reservadas *in*, *out* o *inout*. Éstas apuntan a que es un parámetro de entrada, salida o entrada y salida respectivamente. En esta primera versión del front-end, el retorno de valores se realizará utilizando exclusivamente las interfaces *out* e *inout*.
- Referencia a los parámetros desde el código nativo: Utilizamos unas macros para acceder a los parámetros desde el código nativo. Las variables en SPC-XML, excepto los tipos simples de datos, se declaran como tipo array. Estas variables, no son arrays comunes del lenguaje C, sino estructuras de datos que contienen tanto los datos del array como información necesaria para generar particiones, mantener copias y calcular comunicaciones. Este solapamiento de estructuras es necesario para que el Framework SPC-XML pueda realizar de forma correcta la paralelización.

  Por tanto, de cara al programador, es necesario un interfaz que permita el acceso a los arrays de forma transparente. Para ello utilizaremos las siguientes macros:

  · inVcard(<identificador>, <número>)
  outVcard(<identificador>, <número>)
  para acceder al tamaño de la cardinalidad de la dimensión <número>, del array <identificador>.
  · inVx(<identificador>, <lista-índices>)
  outVx(<identificador>, <lista-índices>)
  siendo x un número entero que determina el número de índices que van a aparecer en la lista. Con esta macro accedemos a las posiciones del <identificador> que indiquen los índices, la posición de éstos en la lista determinan la dimensión a la que se refieren.

  Para ambos casos, con el prefijo *in* nos referiremos a parámetros de entrada. Con el prefijo *out* a parámetros de salida mientras que para el caso de parámetros declarados como de entrada y salida (inout) utilizaremos el prefijo *in* cuando se realicen lecturas y *out* en el caso de escrituras.

B. Declaración de procesos

En esta sección presentamos las estructuras de programación y primitivas propuestas en nuestro lenguaje de coordinación. La función principal de los procesos es componer en secuencia o en paralelo los bloques básicos, es decir, las funciones.

En la figura 4 se muestra la declaración de un proceso con una implementación paralela de la ordenación rápida. Todo proceso en C-SPC comienza con la palabra reservada *xspcl* para diferenciarlo de las funciones de código secuencial. A continuación se indica el identificador del proceso y la declaración de parámetros, utilizando el mismo formato que el explicado para las funciones, sección III-A. Tras la sección

Fig. 4. Ejemplo de proceso C-SPC

```
xspcl void par_quick_sort(inout double
           vector[])
 workload=inVcard(vector,0) *
        log((double)inVcard(vector,0));
 local{
   int pivotPos;
 }
{
 if(xspclV(vsize)>1){
   pivoting($vector,$pivotPos);
   if(xspclV($pivotPos)==0){
     par_quick_sort($vector[1:]);
   }
   else{
     if(xspclV($pivotPos)==
         xspclVcard($vector,0)-1){
       par_quick_sort($vector[:{-2}]);
     }
     else{
       parallel(;loadBalance;flat){
         parblock{
           par_quick_sort($vector[$(pivotPos)\-1:]);
         }
         parblock{
           par_quick_sort($vector[$(pivotPos)+1:]);
         }
       }
     }/*else*/
   }/*else*/
 }/*if*/
}
```

de declaración puede aparecer una cláusula *workload* que provee al sistema de compilación de información para realizar el balanceo de carga del proceso que se está implementando. Como aproximación, se puede dar el valor de la complejidad temporal del algoritmo. Aunque se recomienda especificar la complejidad exacta y no la asintótica, para que los resultados de la distribución de la carga de trabajo sean más correctos.

Indicar el valor de la carga de trabajo permite a la función layout realizar un balanceo de carga. Si el valor se conoce en tiempo de compilación se pueden utilizar estas funciones en tiempo de compilación. En cambio, si el valor de la carga de trabajo depende de algún parámetro y no es, por tanto, conocido hasta el momento de la ejecución, el balanceo de carga deberá ser dinámico.

Si fuera necesario declarar variables locales, se utilizará un bloque específico para ello. El bloque se identifica con la palabra reservada *local* y se encierra entre llaves. La declaración de variables locales sigue el mismo patrón que en el lenguaje C. Los tipos soportados son los tipos primitivos y los arrays.

Dentro de este bloque también se pueden declarar nuevos nombres para topologías virtuales. La cláusula *topology* es una herramienta que proporciona el Framework y que nos permiten asociar un identificador con una topología virtual. Estos identificadores se podrán utilizar dentro del proceso en la declaración de una sección paralela. La sintaxis de esta cláusula es: `topology(<name>;<shape>;<function>);`

name es un identificador con el que nombramos la topología definida. *Shape* se corresponde con una lista de corchetes que define el número de dimensiones y *function* con el identificador de una función de topología reconocido por el framework.

El bloque principal de un proceso se corresponde con la sección de código. Esta sección contendrá las primitivas y estructuras de programación con las que se coordinará la composición.

B.1 Composición secuencial

Al igual que en los lenguajes tradicionales, las instrucciones de código que aparecen una detrás de otra están compuestas de forma secuencial.

B.2 Composición paralela

. `parallel(<shape>;<layout>;<topology>){ }`

Con la primitiva parallel iniciamos una sección paralela con ciertas características, dadas por los parámetros shape, layout y topology, pudiendo ser cualquiera de ellos vacíos, ya que existen valores por defecto. El parámetro shape se escribe como una lista de corchetes con los tamaños de cada dimensión (pueden ser referencias a tamaños de otras estructuras de datos). Esto define la estructura de tareas lógicas en forma de array multidimensional. Si los tamaños son vacíos, la distribución se hará de forma equitativa entre las distintas dimensiones. El parámetro shape es útil para especificaciones basadas en paralelismo de datos, donde se crea una tarea lógica por cada elemento de una estructura. Cuando este parámetro no aparece (como en el ejemplo del algoritmo de ordenación), se crean tantas tareas como primitivas parblock hay dentro del parallel. Esto es más propio de algoritmos que utilizan paralelismo de tareas.

Con layout indicamos mediante un identificador la función que distribuirá las tareas lógicas. Algunas funciones implican distribución estática; otras dinámica, en donde se estudia el valor del workload de cada tarea lógica para generar un balance de carga.

El último parámetro de parallel corresponde a la topología virtual. Se define mediante el identificador de la función de topología deseada. También puede contener el nombre de una topología definida mediante la primitiva topology en el bloque local del proceso. En el ejemplo se utiliza la topología *flat*. Es la más simple. Asigna los procesadores considerando que son un pool de procesadores iguales interconectados. Cómo se pone en correspondencia eso con la topología física real es labor del framework. En C-SPC el programador sólo trabaja con la topología virtuales más adecuada para el algoritmo paralelo.

Las primitivas que pueden incluirse dentro del bloque parallel son uno o más *parblock* y clausulas *reduce* opcionales.

. `parblock{ }`

Cada parblock contiene una especificación de tarea. Se pueden utilizar varios parblock para enumeran las tareas a realizar, o un sólo parblock, que se multiplica tantas veces como procesos lógicos se han definido. Dentro de este bloque se pueden incluir estructuras de control de flujo condicional y de repetición, invocaciones a funciones, procesos y primitivas parallel, lo que otorga al lenguaje la capacidad de permitir el paralelismo anidado.

. `reduce(<var>;<operador>)`

La primitiva reduce ofrece la posibilidad de, al terminar una sección paralela, reducir el valor de la variable especificada en el parámetro *var*, siguiendo un patrón o función concreta, dada en el parámetro *operador*.

B.3 Control de flujo

. Estructuras if-else y while

La sintaxis y el funcionamiento coinciden con los utilizados en cualquier lenguaje secuencial. Las estructuras que pueden anidarse dentro de este bloque coinciden con las admitidas en la clausula parblock.

. Estructura loop

La estructura loop es un bucle controlado por índice. El rango de valores por los que pasará el índice se define siguiendo la notación Fortran 95 *desde:hasta:paso*. Si no fuera necesario referenciar el índice dentro del bucle, puede omitirse el nombre del mismo.

B.4 Invocaciones a funciones y procesos

La sintaxis de las invocaciones sigue las reglas de C, con ligeras modificaciones:

. `nombre_de_funcion(<lista_de_parametros>);`

Los valores válidos de los parámetros pueden ser literales o referencias a variables o subdominios de variables SPC-XML. Los subdominios de un array se expresan con notación similar a Fortran95 (`<inicio>:<final>:<salto>`) en cada dimensión. El Framework SPC-XML ofrece la posibilidad de asignar un valor alternativo al parámetro cuando el valor referenciado no sea válido (fuera del rango del array). Para utilizar esta herramienta utilizaremos el siguiente formato:

- `<valor>:<valor>`
  Si el primer valor es una referencia a un subdominio de una variable array, se evalúan los límites del subdominio. Si alguno queda fuera del dominio original de la variable, se utiliza el segundo valor como parámetro. En el framework el segundo valor se denomina *ifout* ya que sólo se utiliza si el primer valor es una referencia fuera de los límites de la variable referenciada.

Cuando varias tareas lógicas, por limitación en el numero de procesadores, se secuencializan en el mismo procesador, pueden aparecer problemas de consistencia. Por ejemplo, varias tareas pueden tener parámetros de salida que alteran valores que se utilizan en entrada en otras tareas. Teóricamente las tareas deben ejecutarse independientemente, pero la secuencialización forzada puede romper este modelo. Para evitar estos problemas, el Framework permite proteger la consistencia del valor de un parámetro. Al aplicar la primitiva *protect* sobre un parámetro, indicamos que la tarea debe realizar una copia única del parámetro antes de que otras tareas puedan modificarlo. Podemos proteger el valor del parámetro, su valor alternativo o incluso ambos. Para ello debemos de hacerlo de la siguiente forma:

- `protect(<parámetro>):protect(<ifOut>)`
  Para mayor comodidad el front-end permite sustituir la primitiva protect por los símbolos "{" "}". Siguiendo este formato el último ejemplo quedaría de la siguiente manera:
- `{<parámetro>}:{<ifOut>}`

B.5 Otras herramientas

El lenguaje soporta primitivas *import* para poder utilizar las funciones y procesos de otros ficheros escritos en C-SPC. También dispone de variables reloj (XspclClock) y funciones de medición de tiempos. Además dispone de comentarios especiales que que se propagan a directivas de documentación en la representación intermedia Xpscl y por tanto al programa final en MPI.

IV. Implementaciones portables

El framework SPC-XML permite crear de forma automática programas en lenguaje C con llamadas a rutinas MPI a partir de especificaciones en C-SPC. Tanto los algoritmos de paralelismo de datos como de tareas tienen especificaciones simples en C-SPC, con estados globales bien definidos y semánticas claras. La cantidad de líneas de código relacionadas con los detalles de paralelismo, comunicación y sincronización se reduce drásticamente en C-SPC.

Ejemplo	C-SPC	MPI
Cellular-Automata (1D bands)	2	20
Cellular-Automata (2D blocks)	2	45
Quick-Sort (fixed partitions)	4	10
Quick-Sort (dynamic balance)	4	16

TABLA I
Número de líneas de código paralelo en C-SPC y en programas MPI equivalentes

Para comprobarlo hemos desarrollado varios programas de ejemplo siguiendo las pautas encontradas en libros de texto sobre el uso correcto de MPI (ver por ejemplo [15]). Hemos utilizado implementaciones no triviales, donde se explotan técnicas para evitar interbloqueos por saturación de buffer, optimizaciones de código, etc. A la vez hemos programado los mismos algoritmos en C-SPC y utilizado el framework para obtener programas MPI equivalentes.

En la siguiente discusión nos centraremos en dos ejemplos: uno orientado a paralelismo de datos (Cellular-Automata en 2D) y otro en paralelismo de tareas (QuickSort). Cada uno ha sido realizado con dos implementaciones diferentes. El Cellular-Automata ha sido implementado con partición de datos por bandas en una sola dimensión o por bloques en las dos dimensiones. El QuickSort se ha implementado con una partición de procesadores fija, mitad de procesadores para cada sub-vector, y con un balance de carga dinámica, asignando un numero de procesadores por sub-vector proporcional al coste asintótico de ordenar el sub-vector ($n \log n$).

El framework utiliza técnicas automáticas para detectar en el código situaciones de barreras eliminables, optimizaciones de la estructura de comunicación, etc. Los programas obtenidos automáticamente utilizan el mismo número y la misma estructura de comunicaciones que los desarrollados y optimizados a mano. En [12] se pueden consultar más detalles sobre las técnicas de implementación. En las versiones actuales el tiempo de ejecución es entre un 5 % y un 20 % mayor que en las versiones codificadas y optimizadas manualmente.

Sin embargo el esfuerzo de desarrollo es mucho menor. En la Tabla I se muestra la cantidad de líneas de código relacionadas con los detalles de paralelismo, comunicación y sincronización en cada versión de los ejemplos. No contamos las líneas relacionadas con la inicialización de MPI, la declaración de estructuras de datos y la comprobación de errores de las llamadas a las librerías. Sólo hemos contado las líneas relacionadas con: cálculos y comprobaciones asociadas a la partición de datos; creación de tipos compuestos en

Ejemplo	C-SPC	MPI
Cellular-Automata	1	30
Quick-Sort	1	8

TABLA II
Número de líneas de código alteradas para cambiar el layout de datos y la asignación de procesadores.

MPI; envíos y recepciones de datos. En C-SPC contamos las primitivas relacionadas con la creación de topologías y la composición paralela.

Como se puede ver en la tabla, el programa de cellular-automata, basado en paralelismo de datos, requiere en MPI una partición manual de datos para adaptarse explícitamente a la granularidad derivada del número de procesadores. El uso de diferentes particiones implica diferentes esfuerzos de desarrollo. Sin embargo en C-SPC, se puede expresar en una sola línea los datos necesarios para crear la topología virtual y la partición de datos. Todas las comunicaciones se generan automáticamente.

En el caso del QuickSort, al estar orientado a paralelismo de tareas, la implementación en MPI es más directa. Sin embargo, aún son necesarios varios cálculos para determinar los trozos de vector que debe utilizar cada subclusters de procesadores según avanza la recursividad y las comunicaciones adecuadas para recuperar los resultados. En el caso de C-SPC, de nuevo se expresan todos estos detalles en una sola línea. Las otras tres son dos líneas con la palabra clave parblock y una declaración de workload.

En la Tabla II se muestra el número de líneas de código que es necesario alterar para cambiar la distribución de datos en cada ejemplo de los propuestos. En el caso de C-SPC, sólo es necesario cambiar el nombre de la función de layout o de generación de topología virtual. El framework genera automáticamente todos los detalles. En el caso de MPI, son necesarios varios cambios en los cálculos de distribución de datos y pueden aparecer nuevas estructuras de comunicación.

V. Conclusión

En este documento hemos presentado el lenguaje C-SPC. Un lenguaje de programación de paralelismo anidado construido como una extensión simple de C. Presentamos una herramienta de front-end que permite traducir los programas C-SPC a Xspcl, el lenguaje de representación intermedio que utiliza el framework de mapeo e implementación automáticos SPC-XML. Estas herramientas permiten generar automáticamente programas MPI a partir de código C-SPC. El lenguaje C-SPC permite explotar todas las características que proporciona SPC-XML con una sintaxis más amigable para programadores habituados a C, utilizando especificaciones algorítmicas paralelas de alto nivel y sin entrar en detalles sobre particiones de datos o comunicación. Presentamos ejemplos y una discusión sobre la cantidad de código necesaria para desarrollar programas en C-SPC o directamente en MPI. Los resultados indican que C-SPC permite generar automáticamente programas eficientes con un mínimo esfuerzo de programación y depuración. Alterar los programas para utilizar diferentes distribuciones de datos o topologías virtuales es extremadamente sencillo, ya que estas están imbuidas en funciones o plug-ins del framework. El trabajo futuro incluirá la actualización del lenguaje conforme se actualiza el framework SPC-XML y la creación de un componente para el entorno de desarrollo Eclipse, que provea de un marco de trabajo amigable para el programador.

Agradecimientos

Este trabajo ha sido parcialmente financiado a través del Ministerio de Educación y Ciencia (TIN2007- 62302), del Ministerio de Industria (FIT-350101- 2006-46 y FIT-350101-2007-27) y de la Junta de Castilla y León. Parte de este trabajo ha sido realizado bajo el proyecto HPC-EUROPA (RII3-CT-2003-506079).

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] D.B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, no. 2, pp. 123–169, Jun 1998.

[3] K. Lodaya and P. Weil, "A kleene iteration for parallelism," in *Proc. FST & TCS'98*, V. Arvind and R. Ramanujam, Eds. 1998, vol. 1530 of *LNCS*, pp. 355–366, Springer-Verlag.

[4] 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.

[5] R.A. Sahner and K.S. Trivedi, "Performance and reliability analysis using directed acyclic graphs," *IEEE Trans. on Software Eng.*, vol. 13, no. 10, pp. 1105–1114, Oct 1987.

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

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

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

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

[10] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. of 5th PPoPP*. ACM, 1995, pp. 207–216.

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

[12] A. González-Escribano, A.J.C. van Gemund, and V. Cardeñoso-Payo, "SPC-XML: A structured representation for nested-parallel programming languages," in *Euro-Par 2005, Parallel Processing*, P.D. Medeiros J.C. Cunha, Ed. ACM, IEEE, 2005, vol. 3648 of *LNCS*, pp. 782–792, Springer-Verlag.

[13] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, Feb 1992.

[14] A. González-Escribano, "Xspcl (v0.7) An intermmediate-programming language representation for nested-parallel environments," Tech. Rep., Universidad de Valladolid, 2008.

[15] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI - Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 2nd edition, 1999.