Una plantilla genérica para el patrón de paralelismo de divide-y-vencerás en sistemas multinúcleo

Carlos H. González
Dept. de Electrónica e Sistemas
Universidade da Coruña
cgonzalezv@udc.es

Basilio B. Fraguela
Dept. de Electrónica e Sistemas
Universidade da Coruña
basilio.fraguela@udc.es

Resumen

El patrón divide-y-vencerás organiza de forma natural el paralelismo en estructuras de datos y problemas que se expresan de forma recursiva. Existen herramientas como las librerías Threading Building Blocks (TBB) de Intel, que permiten paralelizar cierto tipo de problemas creando una capa de abstracción sobre una implementación de este patrón. En este artículo proponemos una plantilla que mejora la programabilidad de estos problemas en sistemas multinúcleo y la comparamos en términos de programabilidad y rendimiento con la TBB.

1. Introducción

La estrategia divide-y-vencerás aparece en numerosos problemas secuenciales [1]. Se aplica cuando la solución de un problema puede ser encontrada dividiéndolo en subproblemas más pequeños, resolubles separadamente, y uniendo los resultados parciales en una solución global para el problema completo. Esta estrategia puede a menudo aplicarse recursivamente a los subproblemas hasta alcanzar un caso base o indivisible. Este caso se resuelve entonces directamente. La independencia en la resolución de los subproblemas definidos en cada nivel de recursión lleva a la concurrencia, dando lugar al patrón de paralelismo de divide-y-vencerás.

El aprovechamiento de los modernos sistemas multinúcleo precisa de herramientas que faciliten la programación paralela. Los enfoques explorados hasta ahora con este objetivo son la creación de nuevos lenguajes, de directivas de compilador o librerías. Las librerías de esqueletos algorítmicos no precisan soporte del compilador, y al construírse sobre patrones de diseño paralelo, proporcionan una especificación clara al flujo de ejecución, paralelismo, sincronización y comunicaciones de las estrategias típicas para la resolución paralela de problemas. Divide-y-vencerás ha sido de hecho identificado como uno de los esqueletos básicos de la programación paralela [4]. Un ejemplo reciente de librería ampliamente adoptada que proporciona esqueletos paralelos para sistemas multinúcleo son los Threading Building Blocks (TBB) de Intel [9]. Trabajan en base a la descomposición recursiva de problemas y del robo de tareas.

El resto de este artículo se organiza de la siguiente manera. La siguiente sección presenta las TBB, explicando algunas de sus características y plantillas. Nuestra propuesta para expresar el patrón divide-y-vencerás sobre las TBB se presenta en la sección 3. En la sección 4 se evalúa nuestra propuesta. El trabajo relacionado se expondrá en la sección 5, seguido de nuestras conclusiones en la sección 6.

2. La librería Intel TBB

Intel Threading Building Blocks (TBB) [9] es una librería en C++ desarrollada por Intel para la programación de aplicaciones multihilo. Proporciona desde operaciones atómicas y mutexes a contenedores especialmente diseñados para operaciones paralelas. Sin embargo, el mecanismo principal que emplea para expre-

sar paralelismo es el uso de plantillas de algoritmos que proporcionan algoritmos paralelos genéricos. Las plantillas algorítmicas más importantes proporcionadas por las TBB son `parallel_for` y `parallel_reduce`, que expresan computaciones independientes elemento a elemento de un problema y una reducción paralela, respectivamente. Estas plantillas tienen dos parámetros obligatorios. El primero es un *rango* o espacio de iteración que define los límites de un bucle a paralelizar. El segundo, que llamaremos *cuerpo*, define la computación a realizar sobre el rango. Para modelar el concepto Rango, que representa un conjunto de valores divisible recursivamente, la clase debe proporcionar un constructor de copia, un método `empty` que indique si un rango está vacío, un método `is_divisible` para informar si el rango puede ser particionado en dos subrangos y por último un constructor especial de división. Las plantillas de las TBB usan estos métodos para particionar recursivamente el rango inicial de entrada en subrangos más pequeños que se procesen en paralelo. La descomposición recursiva se complementa con un planificador mediante robo de tareas, encargado de balancear la carga entre los distintos hilos.

La clase que proporciona el cuerpo o computación sobre el rango tiene requisitos diferentes dependiendo de la plantilla. De esta forma, `parallel_for` sólo necesita que la clase tenga un constructor de copia y que sobreescriba el método `operator()` recibiendo como parámetro un objeto de la clase rango empleada. La computación paralela se realiza en este método. `parallel_reduce` requiere adicionalmente un constructor de división y un método `join`. El constructor de división se usa para crear copias del objeto cuerpo para los distintos hilos que participen en la reducción paralela. El método `join` tiene como entrada un cuerpo `rhs` que contiene la redución de un subrango justo a la derecha (esto es, el siguiente) del que ha reducido el objeto actual. La operación debe ser asociativa, aunque puede no ser conmutativa.

Por último, las plantillas de las TBB permiten un tercer parámetro opcional llamado *particionador*. Este objeto indica la política a seguir para generar nuevas tareas paralelas. Por defecto se utiliza `simple_partitioner`, que divide los rangos recursivamente creando nuevas tareas según se haya definido en la clase rango. En este caso el programador tiene el control total de la generación de tareas paralelas. El `auto_partitioner` deja a la librería TBB decidir cuándo los rangos deben ser divididos para balancear la carga de una forma automática. Es posible que los rangos no se dividan incluso cuando son divisibles, ya que su división no se necesite para balancear la carga.

3. Una plantilla de algoritmo para problemas divide-y-vencerás

La programación de un algoritmo del tipo divide-y-vencerás puede resultar compleja empleando las plantillas estándar de las TBB. A continuación proponemos una plantilla que encapsule los elementos básicos de este tipo de algoritmos. Los componentes básicos de un algoritmo divide-y-vencerás son la identificación del caso base, su resolución, la partición en subproblemas de un problema no base, y la combinación de los resultados de los subproblemas. Por ello, debemos intentar proporcionar una forma de expresar estos problemas utilizando sólo un método para cada uno de estos componentes. Para aumentar la flexibilidad, la partición de un problema no base puede dividirse en dos subtareas: calcular el número de hijos, que puede no ser fijo, y construir estos hijos.

Las subtareas identificadas en la implementación de un algoritmo divide-y-vencerás puede agruparse en dos conuntos, dando lugar a dos clases. La decisión de si un problema es el caso base de la recursión, el cálculo del número de subproblemas de problemas no base y la división de un problema sólo dependen de los datos del problema de entrada. Estos métodos se refieren y manipulan la estructura del problema de entrada. Conforman por tanto un objeto con un rol similar al rango en las plantillas estándar de las TBB. Llamaremos a este objeto *info* porque porporciona información sobre la estructura del problema. Por otro la-

```
template<typename T, int N>
struct Info : Arity<N> {
  bool is_base(const T& t) const; //
        es t el caso base de la recursión?
  int num_children(const T& t) const; //
        número de subproblemas de t
  T child(int i, const T& t) const; //
        obtener el subproblema i de t
};

template<typename T, typename S>
struct Body : EmptyBody<T, S> {
  void pre(T& t); // preprocesar t antes de dividir
  S base(T& t); // resolver caso base
  S post(T& t, S *r); // combinar soluciones hijas
};
```

Figura 1: Plantillas con pseudo-firmas para los objetos info y cuerpo usados por `parallel_recursion`

do, al contrario de los objetos rango, nosotros elegimos no encapsular los datos del problema dentro del objeto info. Esto reduce la carga del programador, evitando la necesidad de escribir un constructor para este objeto en la mayoría de los problemas.

El procesado del caso base y la combinación de las soluciones a los subproblemas de un problema dado son responsabilidad de un segundo objeto análogo al cuerpo de las plantillas de las TBB, así que lo llamaremos también cuerpo. Muchos algoritmos divide-y-vencerás procesan un problema de entrada de tipo `T` para conseguir una solución de tipo `S`. Por ello el cuerpo debe soportar tipos de datos diferentes para ambos conceptos, aunque por supuesto pueden ser el mismo. En algunos casos es útil realizar un procesamiento en los datos de entrada antes de comprobar su divisibilidad y el correspondiente cálculo del caso base o recursión. En consecuencia el cuerpo de nuestra plantilla necesita un método `pre`, que puede estar vacío, que se aplica al problema de entrada antes de realizar ninguna comprobación. En lo que respecta al método que combina las soluciones de los subproblemas, que llamaremos `post`, sus entradas serán un objeto de tipo `T` que define el problema en un punto de la recursión, y un puntero a un vector con las soluciones de sus subproblemas, para que se pueda soportar fácilmente un número variable de hijos.

```
 1 template<typename T, typename S,
 2           typename I, typename B,
 3           typename P>
 4 S parallel_recursion(T& t, I& i, B& b, P& p) {
 5   b.pre(t);
 6   if(i.is_base(t)) return b.base(t);
 7   else {
 8     int n = i.num_children(t);
 9     S result[n];
10     if(p.do_parallel(i, t))
11       parallel_for(int j = 0; j < n ; j++)
12         result[j] = parallel_recursion(i.child(j, t),
                 i, b, p)
13     else
14       for(int j = 0; j < n ; j++)
15         result[j] = parallel_recursion(i.child(j, t),
                 i, b, p);
16     }
17     return b.post(t, result);
18   }
19 }
```

Figura 2: Pseudocódigo de la plantilla `parallel_recursion`

La Figura 1 muestra las plantillas que describen los objetos info y cuerpo necesarios para la plantilla que proponemos. La clase info debe heredar de la clase `Arity` $<$ `N` $>$, donde `N` es bien el número de hijos de cada subproblema no base, cuando su valor es una constante, o el identificador `UNKNOWN` si no hay un número fijo de subproblemas en las particiones. Esta clase proporciona un método automático `num_children` si `N` es una constante. Para el cuerpo, puede derivarse opcionalmente de la clase `EmptyBody` $<$ `T, S` $>$, que proporciona vacíos los métodos que necesita el cuerpo. Por tanto, heredar de él evita tener que escribir métodos innecesarios.

En la Figura 2 vemos el pseudocódigo de la plantilla que proponemos, llamada `parallel_recursion` por similitud con las presentes en las TBB. Sus parámetros son la representación del problema de entrada, el objeto info, el objeto cuerpo, y un parámetro opcional que define la política para crear nuevas tareas paralelas. Al contrario de las plantillas estándar de las TBB, la nuestra devuelve un valor que tiene tipo `S`, el tipo de la solución. Una especialización de la plantilla permite un tipo de retorno `void`.

En el pseudocódigo vemos que `Info :: is_base` no es el opuesto exacto

```
1 struct FibInfo : public Arity<2> {
2   bool is_base(const int i) const {
3     return i <= 1;
4   }
5
6   int child(const int i, const int c) const {
7     return c − i − 1;
8   }
9 };
```

```
11 struct Fib: public EmptyBody<int, int> {
12   int base(int i) { return i; }
13
14   int post(int i, int* r) { return r[0] + r[1]; }
15 };
16 ...
17 int result = parallel_recursion<int> (n, FibInfo(), Fib(),
           auto_partitioner());
```

Figura 3: Cálculo del n-ésimo número de Fibonacci con `parallel_recursion`

del método `is_divisible` de los rangos TBB. Las TBB utilizan `is_divisible` para expresar la divisibilidad del rango, pero también para comprobar si es más eficiente dividir el rango y procesar los subrangos en paralelo que procesar el rango secuencialmente. Incluso si el usuario escribe `is_divisible` de forma que devuelva cierto para todos los casos no base, la librería puede ignorarlo y dejar de particionar incluso cuando indique divisibilidad si se está empleando `auto_partitioner`. Por estas razones el `operator()` de un cuerpo estándar debe ser capaz de procesar rangos que representen tanto casos base como no base. Sin embargo el método `Body :: base` de la Figura 1, procesa el problema si y solo si `Info :: is_base` es cierto, como se ve en la línea 4 de la Figura.

La decisión sobre si el procesado de los subproblemas hijo se hace secuencialmente o en paralelo es responsabilidad del particionador (líneas 8 a 14 en la Figura 2). El comportamiento de los particionadores es como sigue. El `simple_partitioner` genera una subtarea paralela para cada hijo generado en cada nivel de la recursión, siendo el particionador por defecto. El `auto_partitioner` funciona de forma algo diferente al de las plantillas estándar. En ellas, el particionador puede detener la división del rango, incluso si `is_divisible` devolvió cierto, para balancear óptimamente la carga. En `parallel_recursion` este particionador también intenta balancear la carga automáticamente. Sin embargo, no detiene la recursión en la subdivisión del problema. De esta forma el problema se divide en múltiples subproblemas siempre que `Info :: is_base` es falso. Finalmente hemos añadido un nuevo particionador llamado `custom_partitioner` que toma su decisión sobre la paralelización basandose en un método opcional `Info :: do_parallel(const T t)` que devuelve cierto si los hijos de `t` deberían procesarse en paralelo.

3.1. Ejemplos de uso

La Figura 3 muestra el código para calcular el n-ésimo número de Fibonacci usando `parallel_recursion`, el cual sólo necesita 13 líneas de código. Este código tiene la virtud de que no sólo paraleliza el cálculo, incluso hace innecesaria la función secuencial `fib`, gracias a la capacidad de `parallel_recursion` de expresar completamente problemas que se resuelven recursivamente.

Como segundo ejemplo, el problema del viajante implementado en el benchmark Olden *tsp*, se paraleliza con `parallel_recursion` en la Figura 4. La garantía de que el método `post` que combina las soluciones obtenidas en cada nivel de la recursión se aplique a las soluciones de los problemas hijos generados por un problema dado y de que su problema padre es también una entrada al método simplifica extraordinariamente la implementación, dejándola en 21 líneas de código. Una implementación con `parallel_reduce` necesitaría del almacenamiento y gestión de información extra que permitiera garantizar la corrección de la operación, pues la reducción de tsp no es asociativa.

4. Evaluación

Ahora compararemos la implementación de varios algoritmos divide-y-vencerás usando

```
1 struct TSPInfo: public Arity<2> {
2   static int sz_;
3
4   TSPInfo(int sz ) {
5     sz_ = sz;
6   }
7
8   bool is_base(const Tree t) const {
9     return (t->sz <= sz_);
10  }
11
12  Tree child(int i, const Tree t) const {
13    return (i == 0) ? t->left : t->right;
14  }
15 };
```

```
17 struct TSPBody : public EmptyBody<Tree, Tree> {
18   Tree base(Tree t) {
19     return conquer(t);
20   }
21
22   Tree post(Tree t, Tree* results) {
23     return merge(results[0], results[1], t);
24   }
25 };
26 ...
27 parallel_recursion<Tree>(root, TSPInfo(sz), TSPBody(),
         auto_patitioner());
```

Figura 4: Paralelización de tsp de Olden con `parallel_recursion`

`parallel_recursion`, las plantillas propuestas por las TBB y OpenMP [2], en términos de programabilidad y rendimiento. No se puede comparar directamente OpenMP con las plantillas, ya que también depende del compilador. Se ha incluido en este estudio como una línea base que muestre la mínima sobrecarga necesaria para paralelizar aplicaciones, ya que la inserción de directivas de compilación en un programa normalmente necesita menos reestructuración que la definición de clases que se usa con los esqueletos orientados a objetos. De esta forma la comparación de las versiones con los esqueletos estándar de TBB y de `parallel_recursion` con respecto a OpenMP ayuda a medir mejor el esfuerzo de paralelización que ambos tipos de esqueletos requiere.

Los algoritmos usados para la evaluación son el cálculo recursivo del n-ésimo número de Fibonacci (fib), la mezcla de dos secuencias ordenadas de enteros en una secuencia de salida ordenada (merge), la ordenación de un vector de enteros usando quicksort (qsort), el cálculo del número de solución al problema de las N Reinas (nqueens) y cuatro códigos basados en árboles tomados de los benchmarks Olden [10]. El primer benchmark Olden es treeadd, que suma los valores en los nodos de un árbol binario. La ordenación de un árbol binario balanceado (bisort), una simulación de un sistema de salud jerárquico (health), y el problema del viajante (tsp) completan la lista.

El Cuadro 1 describe los tamaños de los problemas, el número de subproblemas en los que se puede dividir cada problema (aridad) y si la combinación o reducción de los resultados de los subproblemas es asociativa o no, o si no es necesaria. Podemos ver que todos los algoritmos excepto nqueens y health se expresan naturalmente dividiendo cada problema en dos, lo que se adapta bien a las plantillas de las TBB. nqueens prueba todas las posibles posiciones de las reinas en la fila i del tablero que no entren en conflicto con las reinas ya colocadas en las $i-1$ filas superiores. Cada posible posición da lugar a un problema hijo que procede a examinar las localizaciones en la siguiente fila. De esta forma el número de problemas hijo varía de 0 a N, el tamaño del tablero, dependiendo del número de fila y de las posiciones en las que ya se han colocado reinas. En health, cada nodo del árbol tiene cuatro subnodos que pueden ser procesados en paralelo, así que este es el número natural de subproblemas. Los subnodos se colocan en un vector. Esto beneficia a las plantillas de las TBB, ya que posibilita el uso como rango de un `blocked_range`, que es una clase incluida en las TBB que define un espacio de iteración unidimensional, ideal para paralelizar operaciones en vectores.

4.1. Programabilidad

El impacto del uso de un método de que intente facilitar la programación no es fácil de medir. En esta sección se emplean tres métricas cuantitativas para este propósito: las SLOC (líneas de código fuente, excluyendo comen-

Nombre	Descripción	Aridad	Asoc
fib	cálculo recursivo del 43º número de Fibonacci	2	Sí
merge	mezcla de dos secuencias ordenadas de 100 millones de enteros	2	-
qsort	quicksort de 10 millones de enteros	2	-
nqueens	conteo de soluciones para N Reinas en un tablero 14 × 14	var	Sí
treeadd	suma de valores en un árbol binario de 24 niveles	2	Sí
bisort	ordenar un árbol binario balanceado de 22 niveles	2	No
health	2000 pasos de simulación en un árbol cuaternario de 6 niveles	4	No
tsp	problema del viajante en un árbol binario de 23 niveles	2	No

Cuadro 1: Benchmarks usados

tarios y líneas vacías), el esfuerzo de programación [7], y el número ciclomático [8]. Las SLOC dependen más del estilo de programación del usuario que las otras dos métricas. El esfuerzo de programación es una función del número de operandos únicos, operadores únicos, el total de operandos y el total de operadores que se encuentran en un programa. Los operandos se corresponden con las constantes y los identificadores, mientras que los símbolos o combinaciones de símbolos que afectan al valor o ordenación de los operandos constituyen los operadores. Según [7] la métrica de esfuerzo de programación calculada a partir de estos valores es aproximadamente porporcional al esfuerzo de programación requerido para implementar un algoritmo. Finalmente, el número ciclomático [8] es $V = P + 1$, donde P es el número de puntos de decisión o predicados en un programa. Cuanto más pequeño sea V, menos complejo es el programa.

La Figura 5 muestra el sobrecoste relativo de líneas de código, esfuerzo de programación y número ciclomático de cada versión de los benchmarks, una empleando la plantilla adecuada de TBB (TBB) y otra con `parallel_recursion` (pr), comparándolas con una versión base implementada con OpenMP. Los valores positivos predominan en el gráfico indicando que, como se esperaba, OpenMP tiene la menor sobrecarga de programación, al menos contado con SLOC o esfuerzo de programación, que es la razón por la que ha sido elegido como línea base. Aún así, `parallel_recursion` es el ganador global en lo que respecta al número ciclomático. Esto se debe a que muchas de las condicionales y bucles (que también implican una condición para detectar su finalización) que se encuentran en los algoritmos divide-y-vencerás están subsumidos en el esqueleto `parallel_recursion`, mientras que otras alternativas los dejan expuestos en el código del programador. `parallel_recursion` necesita menos líneas de código, esfuerzo y condicionales que las plantillas de las TBB en todos los códigos excepto merge y qsort, donde se da la situación contraria. Según el indicador de esfuerzo de programación, los programas paralelizados con las TBB necesitan un 64.6 % más de esfuerzo que OpenMP, mientras que los basados en `parallel_recursion` necesitan de media un 33.3 % más de esfuerzo que OpenMP. Esto supone una reducción de casi un 50 % en términos relativos. La situación es la opuesta para merge y qsort, en los que el incremento medio de esfuerzo sobre OpenMP es del 13.4 % para los códigos que usan `parallel_for` y 30.1 % para los que usan `parallel_recursion`. Estos son los únicos benchmarks en los que no es necesario combinar el resultado de la solución de los problemas: sólo necesitan dividir un problema en subproblemas que se puedan resolver en paralelo. También son los dos benchmarks basados en arrays, donde el concepto de Rango alrededor del que se diseñaron las plantillas de TBB se adapta mejor. Por tanto, cuando estas condiciones se cumplan puede ser preferible probar el uso de los esqueletos estándar de las TBB.

La mejora en la programabilidad no se hace en general a costa de una pérdida de rendimiento. La Figura 6 muestra los cocientes de los tiempos de ejecución de la versión con plantillas TBB y de la versión con

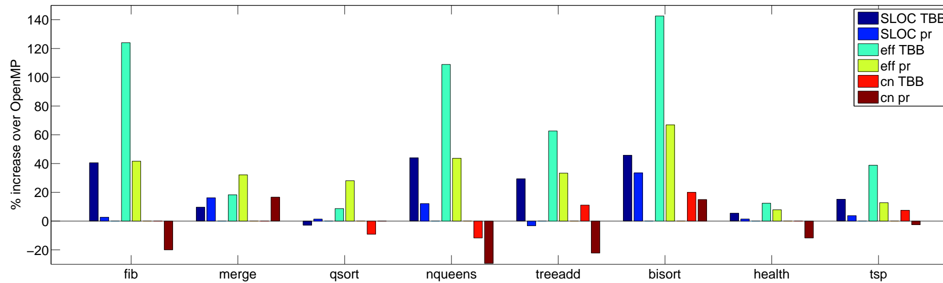

Figura 5: Estadísticas de productividad respecto a la versión base con OpenMP

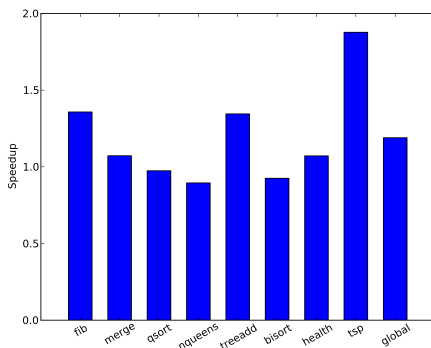

Figura 6: Diferencias de rendimiento entre las plantillas TBB y `parallel_recursion`

`parallel_recursion`. Un valor por encima de 1 indica un mejor rendimiento de la versión `parallel_recursion`, mientras que un valor por debajo lo indicaría en favor de TBB. Estas mediciones se han realizado en un sistema HP rx7640 con 8 procesadores Itanium Montvale de doble núcleo, usando el compilador icpc v11.0, y nivel de optimización O3. Como vemos, de media la versión con `parallel_recursion` es un 19 % más rápida. Observamos que la mayor aportación a este valor proviene del benchmark tsp, que presenta un comportamiento excepcionalmente bueno en el sistema considerado. Aún si excluimos este valor, la media de mejora de rendimiento sería de un 9.1 %. Este aumento de rendimiento es posible gracias a que `parallel_recursion` se construye sobre el API de bajo nivel de las TBB, no sobre sus plantillas. Además, sigue políticas diferentes a la hora de gestionar la creación de tareas, y tiene requerimientos diferentes de sincronización y de estructuras de datos.

5. Trabajo relacionado

Aún cuando TBB sea probablemente la librería de operaciones esqueletales más extendida actualmente, parcialmente gracias al soporte ofrecido por Intel, no es de ningún modo la única. La librería eSkel [5] ofrece operaciones esqueletales paralelas para C sobre MPI. Es un API de más bajo nivel, con algunos detalles de implementación específicos de MPI.

Muesli [3] permite aprovechar las ventajas de la orientación a objetos en C++. También está enfocada a memoria distribuida, centrándose alrededor de contenedores distribuidos y clases esqueleto que definen topologías de proceso. Muesli necesita llamadas polimórficas en tiempo de ejecución, lo que genera sobrecargas potencialmente grandes.

Quaff [6] evita esto siguiendo la misma aproximación que las TBB: confiar en la metaprogramación que permite el mecanismo de plantillas de C++ para resolver el polimorfismo en tiempo de compilación. La característica más distintiva de Quaff es que lleva al programador a codificar el grafo de tareas de la aplicación con definiciones de tipos para producir código de paso de mensajes optimizado. Como resultado, aunque permite anidamiento de esque-

letos, este anidamiento debe definirse estáticamente Por ello las tareas no pueden generarse dinámicamente en niveles arbitrarios de la recursión y subdivisión del problema, como sí hacen las TBB y nuestra propuesta. para expresar algoritmos divide-y-vencerás, difiere sustancialmente de las plantillas estándar de las TBB y de nuestra propuesta.

6. Conclusiones

En este trabajo se ha presentado una nueva plantilla algorítmica, `parallel_recursion`, implementada sobre las TBB. Esta plantilla encapsula los elementos del patrón divide-y-vencerás, facilitando la implementación paralela de este tipo de algoritmos. Se ha mostrado que mejora la programabilidad, e incluso puede mejorar el rendimiento en estos problemas más de un 9 %. Como trabajo futuro nos planteamos la combinación de esta plantilla con otras adaptadas a otros patrones de paralelismo así como su aplicación a problemas más complejos.

Agradecimientos

Este trabajo ha sido financiado por la Xunta de Galicia bajo el proyecto INCITE08PXIB105161PR y el Ministerio de Ciencia e Innovación, cofinanciado por el Fondo Social Europeo, con cargo al proyecto con referencia TIN2007-67536-C03-02. Agradecemos al CESGA (Centro de Supercomputación de Galicia) por el uso de sus computadores.

Referencias

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[3] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli - A Comprehensive Overview. Technical Report Working Papers, ERCIS No. 7, University of Münster, 2009.

[4] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[5] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[6] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Jean-Thierry Lapresté. Quaff: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[7] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[8] McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[9] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. O'Reilly, 1 edition, July 2007.

[10] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.