Algoritmos paralelos de memoria compartida que determinan el menor tamaño de un árbol binario al refinar un simplex regular

G. Aparicio[1], J.M.G. Salmerón[1], L.G. Casado[1], R. Asenjo[2], I. García[2] y E.M.T. Hendrix[2]

Resumen— **En el ámbito de la optimización global basada en técnicas de ramificación y acotación, cuando el espacio de búsqueda es un n-símplex regular es habitual utilizar como regla de división la bisección por el lado mayor, debido a que garantiza la convergencia del algoritmo. Cuando la dimensión del n-símplex es mayor de 2 existen varios lados mayores que pueden utilizarse para realizar la bisección. La elección del lado mayor influye en el tamaño del árbol binario completo que se genera. Una selección eficiente del lado mayor puede reducir el coste computacional de los algoritmos de ramificación y acotación mencionados. En este estudio estamos interesados en conocer el tamaño del árbol o árboles mínimo(s). Para obtener una solución de las instancias más complejas del problema en un tiempo razonable es necesario el desarrollo de algoritmos paralelos. La complejidad del problema es debida a la necesidad de analizar todas y cada una de las posibles combinaciones de selecciones de los distintos lados mayores en el refinamiento. Aquí se comparan la eficiencia de distintas propuestas de algoritmos paralelos para sistemas de memoria compartida.**

Palabras clave— **Símplice, multicore, árbol binario, Pthreads, C++ threads, TBB.**

I. Introducción

LOS algoritmos de Ramificación y Acotación (del inglés Branch-and-Bound, (BnB) se aplican a la resolución de problemas de optimización global mediante la realización de una búsqueda exhaustiva del mínimo de una función objetivo en un dominio dado. En este trabajo estamos interesados en problemas donde el espacio de búsqueda está definido por un n-símplex regular, definido en un espacio $(n+1)$-dimensional [1].

Los algoritmos de BnB están caracterizados por las reglas de Acotación, Selección, División, Rechazo y Terminación. Se pretende estudiar las características del árbol de búsqueda generado cuando solo se tienen en cuenta las reglas de División y Terminación, es decir, ningún nodo del árbol es eliminado. Se usará como regla de división la bisección del lado mayor (BLM) [2], [3] y como regla de terminación la longitud del lado mayor de un subproblema o nodo del árbol.

Se ha encontrado una secuencia de lados mayores a dividir en el refinamiento de un 3-simplex regular que genera el árbol de tamaño mínimo [4]. El tener precomputada al secuencia de lados mayores a dividir permite la reducción del tiempo de cómputo de algoritmos de Ramificación y Acotación sobre este espacio de búsqueda. Para saber si una determinada secuencia genera un árbol de tamaño mínimo es necesario conocer dicho tamaño. En el cálculo del tamaño del árbol mínimo es necesario analizar los distintos arboles que se pueden obtener dependiendo del lado seleccionado a dividir en cada sub-símplex. La combinación de las distintas elecciones de lados mayores a dividir generará distintos árboles de búsqueda. Por lo tanto, nos encontramos ante un problema combinatorio con una alta carga computacional que puede ser acelerado utilizando técnicas de computación paralela.

El objetivo de este estudio es analizar la eficiencia de varios métodos de paralelización en sistemas de memoria compartida que resuelven este problema. En concreto, se estudian algoritmos paralelos que utilizan Posix Threads (Pthreads), Intel Threading Building Blocks (TBB) y Threads de C++ v11.

En la sección II se describe el un algoritmo básico de Ramificación y Acotación sobre un dominio definido por un simplex regular. En la sección III se describe la bisección por el lado mayor. En la sección IV se presenta el algoritmo secuencial que calcula el menor tamaño de un árbol. Las distintas versiones paralelas se muestran en la sección V. La sección VI muestra una comparativa de la eficiencia de las versiones estudiadas y finalmente se concluye en la sección VII.

II. Optimización Global y Algoritmos BnB

La resolución de un problema de Optimización Global consiste en encontrar el valor mínimo o máximo de una función objetivo f, satisfaciendo ciertas restricciones en el espacio de búsqueda. Para problemas de minimización, el problema de Optimización Global puede escribirse como

$$f^* = f(x^*) = \min_{x \in S} f(x). \quad (1)$$

Cuando se requiere una búsqueda exhaustiva del mínimo global, se suelen utilizar algoritmos BnB. Un algoritmo BnB realiza una búsqueda mediante la descomposición sucesiva del problema inicial en subproblemas de menor tamaño hasta alcanzar la(s) solución(es) final(es) o una aproximación a ella(s). Esta aproximación está determinada por la precisión

[1]Grupo de Investigación TIC146: Supercomputación–Algoritmos, Universidad de Almería (ceiA3), España, e-mail: guillermoaparicio@ual.es, josemanuel@ual.es, leo@ual.es

[2]Departamento de Arquitectura de Computadores, Universidad de Málaga, España, e-mail: rasenjo@uma.es, igarciaf@uma.es, Eligius.Hendrix@wur.nl

requerida para la solución. Esta descomposición genera un árbol de búsqueda que se poda cuando se garantiza que un subproblema no contiene una solución global. La poda o rechazo de un subproblema está normalmente basada en cálculos de cotas de la función objetivo para ese subproblema.

Algoritmo 1 Ramificación y Acotación

Require: S: simplex inicial, ϵ: precisión
1: $\Lambda := \{S_1 = S\}$ *Conjunto de trabajo*
2: $\Omega := \{\}$ *Conjunto final*
3: $ns := 1$ *Número de símplices*
4: **while** $\Lambda \neq \emptyset$ **do**
5: Selecciona S_i de Λ *Regla de selección*
6: Evalúa S_i *Regla de acotación*
7: **if** S_i no puede ser eliminado **then** *Regla de rechazo*
8: **if** $w(S_i) \leq \epsilon$ **then** *Regla de terminación*
9: Almacena S_i in Ω
10: **else**
11: $\{S_{2i}, S_{2i+1}\} :=$ BLM(S_i) *Regla de división*
12: Almacena S_{2i}, S_{2i+1} en Λ
13: $ns := ns + 2$
14: **return** Ω y $f(x^*)$

El esquema básico de un algoritmos de BnB se muestra en el algoritmo 1. El comportamiento, y por tanto la eficiencia del algoritmo dependerá de como se establezcan las reglas de Selección, Acotación, Rechazo, División y Terminación. En este estudio no estudiaremos las reglas de Acotación, Rechazo ni de Selección ya que solo estamos interesados en encontrar la regla de División basada en la bisección del lado mayor que genere un árbol de menor tamaño. Por lo tanto, no existe poda y el árbol se generaría completamente. Su tamaño no solo dependerá del lado mayor elegido para ser dividido, sino también de la regla de terminación usada. En nuestro caso, un simplex con un tamaño de su lado mayor, o una anchura $w(S)$, menor que un umbral ϵ no será nuevamente dividido.

III. Particionamiento de un n-simplex regular

En algunos problemas de Optimización Global es habitual el uso de un espacio de búsqueda definido por un n-simplex regular. Por simplicidad y sin pérdida de generalidad, en este estudio se va a hacer uso de un simplex regular con longitud de lado igual a 1, que se define como:

$$S = \left\{ x \in \mathbb{R}^{n+1} \ : \ \sum_{j=1}^{n+1} x_j = \frac{\sqrt{2}}{2};\ x_j \geq 0 \right\}. \quad (2)$$

La figura 1 muestra un 2-simplex regular con lados de tamaño unidad. La selección de un lado mayor en un 2-simplex no es necesaria ya que o el lado mayor es único o el subsimplex a dividir es regular. A partir de $n = 3$ sí existen varias opciones para elegir el lado mayor a dividir, como se muestra en la figura 2.

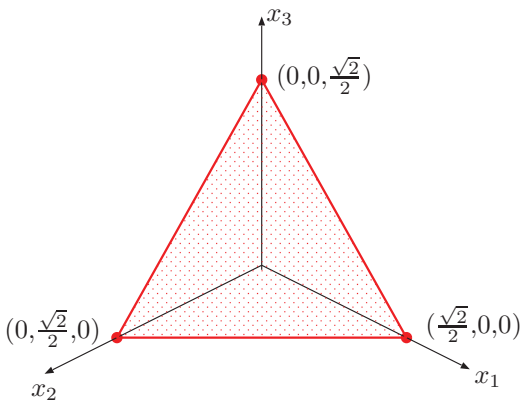

Fig. 1. Un 2-simplex regular con lados de longitud unidad.

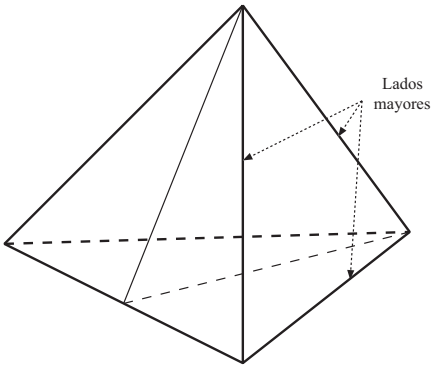

Fig. 2. Primera bisección del lado mayor en un 3-simplex.

IV. Algoritmo secuencial para determinar el tamaño de uno de los menores árboles binarios

El algoritmo secuencial debe chequear todas las opciones de división de lado mayor, para tener en cuenta solo aquellas que generan un árbol binario de menor tamaño. Esto se puede hacer mediante un algoritmo recursivo. Los parámetros iniciales del algoritmo 2 serían el n-simplex regular inicial, uno de sus lados (todos son iguales) y la precisión de refinamiento requerida ϵ.

Algoritmo 2 TamañoÁrbol (S, L, ϵ)

Require: S: simplex, L: lado mayor, ϵ: precisión
1: **if** $w(S) \leq \epsilon$ **then**
2: **return** 1
3: $\{S_1, S_2\}$=DivideSímplice(S,L)
4: **for** cada lado mayor L_i de S_1 **do**
5: $r1_i :=$ TamañoÁrbol(S_1,L_i,ϵ)
6: **for** cada lado mayor L_i de S_2 **do**
7: $r2_i :=$ TamañoÁrbol(S_2,L_i,ϵ)
8: $r_1 :=$ mín$_i\{r1_i\}$
9: $r_2 :=$ mín$_i\{r2_i\}$
10: **return** $1 + r_1 + r_2$

El algoritmo 2 realiza una búsqueda en profundidad, lo que reduce los requerimientos de memoria.

La figura 3 muestra la ejecución del algoritmo 2 para un 2-simplex y $\epsilon = 0{,}5$. La figura 3 permite resaltar algunos aspectos que, para simplificar, no se han incluido en el algoritmo 2:

- Si los símplices hermanos S_1 y S_2 son simétricos,

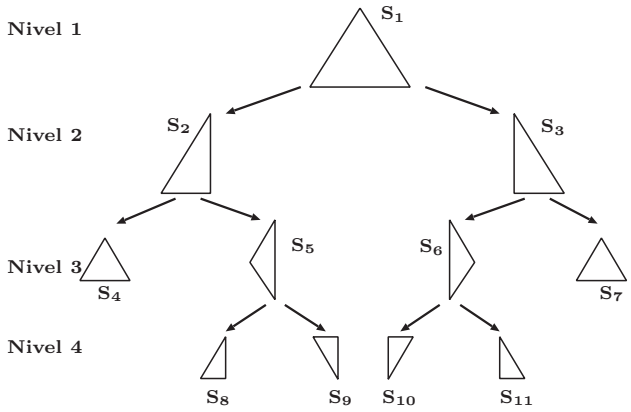

Fig. 3. Árbol binario para $\epsilon = 0{,}5$.

ambos generan subárboles de igual tamaño. Por lo tanto solo es necesario procesar uno de ellos. En ese caso se devuelve como resultado el doble del tamaño del subárbol procesado. Por ejemplo, los símplices S_2 y S_3 o S_8 y S_9 de la figura 3 son hermanos simétricos.

- Si el simplex es regular, solo hay que procesar un lado, ya que todos los lados son de igual tamaño, produciendo además hermanos simétricos. Por ejemplo, los símplices S_1, S_4 y S_7 de la figura 3 son regulares.

En el algoritmo 2, sería conveniente poder determinar antes de su procesado parejas de hermanos resultado de la división de distintos lados mayores de un simplex que son parejas simétricas. De forma anñaloga a simplices simétricos, solo sería necesario procesar una de las parejas. Esto se abordará en trabajos futuros.

V. Versiones paralelas

El paralelismo es fácilmente aplicable en la construcción de árboles ya que las ramas del árbol pueden visitarse en paralelo. Una de las dificultades que pueden presentar los árboles, y que ocurre en los árboles de este estudio, es que pueden existir ramas con diferentes tamaños, tal como muestra la figura 3. Se ha observado experimentalmente que la diferencia de niveles entre ramas aumenta conforme lo hace n. Por lo tanto es necesario el uso de un balanceo de la carga que permita obtener una buena eficiencia en los algoritmos paralelos que recorren este tipo de árboles.

El algoritmo 3 es una extensión simple del algoritmo 2 donde se crea una nueva hebra para procesar un subárbol. El número de hebras no puede superar el umbral MaxHebras que se define como parámetro de entrada. Cada hebra ejecuta el algoritmo secuencial con la posibilidad de crear una nueva hebra para procesar el trabajo pendiente. Una hebra que termina su trabajo, devuelve el resultado y muere, decrementando antes la variable MaxHebras, lo que permitirá la generación de otra hebra. En principio la primera hebra que acceda a la variable compartida NHebras y verifique que es menor que MaxHebras podrá generar una nueva hebra. De esta forma, el balanceo de la carga es inherente a la creación dinámica de hebras.

Algoritmo 3 TamañoÁrbolParalelo (S, L, ϵ)

Require: S: simplex, L: lado mayor, ϵ: precisión, MaxHebras: máximo número de hebras.

```
1:  if w(S) ≤ ε then
2:     return 1
3:  {S1, S2}=DivideSímplice(S,L)
4:  for cada lado mayor Li de S1 do
5:     if NHebras < MaxHebras then
6:        r1i := CreaHebra(TamañoÁrbol(S1,Li,ε))
7:     else
8:        r1i := TamañoÁrbol(S1,Li,ε)
9:  for cada lado mayor Li de S2 do
10:    if NHebras < MaxHebras then
11:       r2i := CreaHebra(TamañoÁrbol(S2,Li,ε))
12:    else
13:       r2i := TamañoÁrbol(S2,Li,ε)
14: Esperar resultados de las hebras creadas
15: r1 := mín_i{r1_i}
16: r2 := mín_i{r2_i}
17: return 1 + r1 + r2
```

Tomando como base el algoritmo anterior, se va ha estudiar el uso de diferentes librerías para el desarrollo de algoritmos paralelos en sistemas multicore que resuelvan el problema de calcular el menor tamaño de los posibles árboles binarios de búsqueda.

A. Posix Threads (Pthreads)

POSIX son las siglas correspondientes a Portable Operating System Interface for Unix. Es un estándar orientado a facilitar la creación de aplicaciones portables y de confianza bajo entornos Unix. La mayoría de las versiones populares de Unix como Linux o Mac OS X cumplen este estándar en gran medida. La biblioteca para el manejo de hebras en POSIX es Pthread. En Pthreads la creación, sincronización y destrucción de las hebras es explícita y controlada por el programador. Esto tiene sus ventajas e inconvenientes. Por un lado, permite un mayor control al programador, pero por otro lado es el programador el que debe garantizar la correcta utilización de la librería de una forma eficiente.

B. Threads de C++ version 11

Una de las mejoras que incluye la versión 11 de C++ (C++11) es el soporte nativo de aplicaciones multihebradas. Para ello incluye la clase tipo hebra(std::thread). Su uso en la práctica es muy similar al de Pthreads. Sin embargo, hay que destacar las siguientes diferencias:

- Pthreads se encuentra disponible en multitud de plataformas siendo un sistema muy portable. C++11 no esta disponible para algunas plataformas.
- Pthreads es una librería de C y no fue diseñada teniendo en cuenta aspectos fundamentales de C++11 como la vida de un objeto o las excepciones.
- La función pthread_cancel, que cancela una hebra en Pthreads, no tiene su equivalente en C++11.
- Pthreads permite el control del tamaño de la pila asociada a las hebras, mientras que C++11

no ofrece esta funcionalidad.
- C++11 proporciona la clase thread como una abstracción para una hebra en ejecución.
- C++11 tiene varias clases y templates para realizar exclusión mutua, utilizar variables de condición y bloqueos.
- C++11 proporciona un sofisticado conjunto de funciones y templates para crear objetos y funciones anónimas (expresiones lambda) en la clase thread.

C. Intel Threading Building Blocks

Threading Building Blocks (TBB) es una librería desarrollada en C++ por Intel para ayudar en la programación paralela en procesadores multicore. La librería consiste en algoritmos y estructuras de datos que facilitan al programador la programación con hebras. Por ejemplo, la ejecución de un programa TBB crea, sincroniza y destruye grafos de dependencias entre tareas, que se ejecutan respecto a las dependencias en el grafo, lo que puede ser completamente transparente al programador.

TBB realiza un balanceo de carga entre hebras mediante un método denominado robo de tareas (thread-stealing). Mediante este método, una hebra, que no tenga tareas para ejecutar en su cola de tareas, robaría tareas de la colas de tareas de otra hebras. Este balanceo dinámico de la carga puede ser completamente transparente para el programador, lo que facilita el desarrollo de aplicaciones paralelas escalables.

La versión TBB del algoritmo 3, en lugar de crear una hebra nueva que evalúe el subárbol generado por la división de uno de los lados mayores (lineas 6 y 11), crea una nueva tarea por cada posible división de un simplex a evaluar. La tarea será ejecutada por la hebra que la creó o por otra, dependiendo de la necesidad de realizar un balanceo de la carga. El número de hebras lo determina el usuario y es fijo durante la ejecución del algoritmo paralelo.

VI. Resultados

Los algoritmos se han compilado con gcc/g++ (GNU compiler Collection) versión 4.8.1 con la opción -O3, usando las librerías de *Pthreads*, *TBB* y *Threads de C++11*. Los programas y se han ejecutado en un nodo de *BullX-UAL* con 16 cores y en la máquina multiprocesador *Aloe* con 32 cores. BullX consiste en dos procesadores Intel® Xeon® E5-2650 de 8 cores a 2,00 GHz y 64 GB de RAM. Aloe consiste en dos procesadores Intel® Xeon® E5-2698-V3 de 16 cores a 2,30 GHz y 256 GB de RAM.

Los experimentos se han realizado para dos instancias del problema:

- I1: 3-simplex con una precisión de $\epsilon = 0{,}01$,
- I2: 4-simplex con una precisión de $\epsilon = 0{,}125$.

No se han usado valores mayores de n ni menores de ϵ ya que los tiempos de ejecución de los algoritmos serían excesivos debido a la complejidad de optimización combinatoria que plantean.

Como muestra el algoritmo 2, el problema a resolver presenta un bajo coste computacional por subsimplice, pero un alto coste de gestión dinámica de memoria para el recorrido del árbol, cuando tenga un gran número de subsimplices a procesar. Por lo tanto el gestor de memoria dinámica juega un papel importante en el tiempo total de ejecución de los algoritmos. El compilador gcc usa su propia librería de gestión dinámica de memoria. TBB también usa su propia librería llamada tbbmaloc_proxy. Existe la posibilidad de usar otras librerías de gestión dinámica de memoria, como por ejemplo tcmalloc de google performance tools o *LLAlloc* de *Lockless Inc.*

TABLA I
Tiempo de ejecución en segundos del algoritmo secuencial con diferentes gestores de memoria en la máquina Bullx

Instancia	GCC	TCMalloc	TBBMalloc	LLAlloc
I1	1357.45s	1103.19s	1249.46s	698.26s
I2	269.62s	223.11s	249.97s	152.71s

La Tabla I muestra el tiempo de ejecución en la máquina BullX de la versión secuencial del algoritmo usando las distintas librerías de gestión dinámica de memoria. Puede observarse que el mejor gestor de memoria es LLAlloc. Los tiempos de la versión secuencial (también tomados en la máquina BullX) con LLAlloc han sido de 698.26s para I1 y de 152.71s para I2. Dichos tiempos han sido tomados como referencia para el cálculo del speedup. Se ha observado experimenalmente que TBB con LLAlloc obtiene mejores tiempos de ejecución que cuando usa su propio gestor de memoria TBBMalloc.

TABLA II
Tiempo de ejecución (T) y Speed-up (SP) para resolver I1 utilizando un número NT de hebras con Pthreads (PTH), TBB Y Threads de C++11 (TC11) en la máquina Bullx

NT	T(PTH)	SP(PTH)	T(TC11)	SP(TC11)	T(TBB)	SP(TBB)
1	699.90s	0.99	699.97s	0.99	783.20s	0.89
2	480.93s	1.45	480.31s	1.46	394.43s	1.77
4	256.88s	2.72	259.27s	2.70	193.32s	3.62
8	315.10s	2.22	272.66s	2.57	97.64s	7.16
16	294.97s	2.38	286.74s	2.44	48.90s	14.30
32	351.84s	1.98	342.58s	2.04	48.68s	14.37

Las tablas II y III muestran los tiempos de ejecución y la ganancia en velocidad para distintos números de hebras en los algoritmos paralelos estudiados en la máquina BullX. La ganancia en velocidad se muestra también en las figuras 4 y 5. La librería TBB presenta los mejores resultados, con una ganancia en velocidad cercana a la lineal hasta 16 hebras, que es el número de cores del sistema en el caso de la máquina BullX. Esto es debido a la buena gestión de las tareas en las colas de la hebras, lo que permite evitar tiempos ociosos en los procesadores. Las versiones basadas en Pthreads y C++11 presentan una

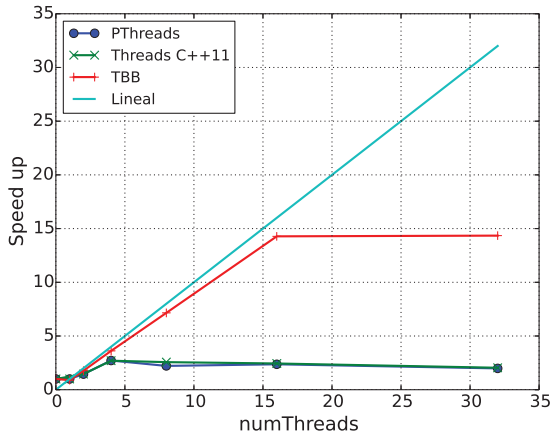

Fig. 4. Ganancia en velocidad para I1 en la máquina Bullx

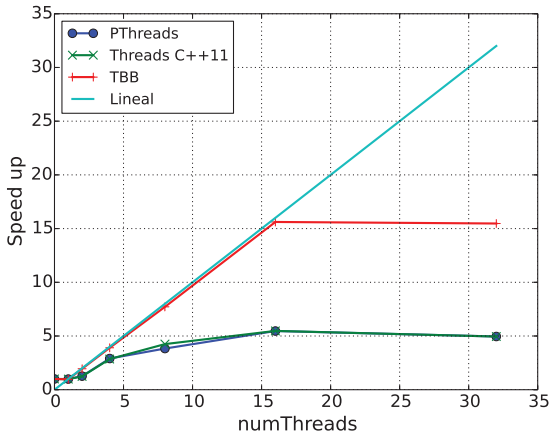

Fig. 5. Ganancia en velocidad para I2 en la máquina Bullx

TABLA III
TIEMPO DE EJECUCIÓN (T) Y SPEED-UP (SP) PARA RESOLVER I2 UTILIZANDO UN NÚMERO NT DE HEBRAS CON PTHREADS (PTH), TBB Y THREADS DE C++11 (TC11) EN LA MÁQUINA BULLX

NT	T(PTH)	SP(PTH)	T(TC11)	SP(TC11)	T(TBB)	SP(TBB)
1	154.23s	0.99	154.42s	0.99	158.08s	0.97
2	120.81s	1.26	123.07s	1.24	78.40s	1.95
4	52.59s	2.90	53.52s	2.85	39.12s	3.93
8	39.85s	3.83	35.97s	4.25	19.80s	7.76
16	27.95s	5.46	27.89s	5.48	9.78s	15.71
32	30.82s	4.95	30.85s	4.95	9.87s	15.57

baja ganancia en velocidad. Esto es debido a que el modelo de hebras dinámico presentado en el algoritmo 3 tiene el inconveniente de que las hebras deben esperar la terminación de las hebras que han creado (ver línea 14).

En las figuras 6 y 7 se observan los resultados de ejecutar las instancias del problema I1 e I2 en la máquina Aloe. Se corroboran los resultados obtenidos en la máquina BullX. La versión TBB es escalable y obtiene speedups cercanos al lineal. Sin embargo las ejecuciones utilizando Pthreads o Threads de C++ versión 11 tienen problemas de escalabilidad.

Aun así, estamos interesados en mejorar la versiones basadas en la generación dinámica de hebras ya que esto permitirá a la aplicación paralela adaptarse a los recursos disponibles en el sistema en tiempo de ejecución [5].

VII. CONCLUSIONES Y TRABAJO FUTURO

Los algoritmos paralelos para determinar el menor tamaño de un árbol binario generado en el refinamiento de un simplex regular mediante la bisección de un lado mayor en sistemas multicore presentan un bajo coste computacional por subsímplice, para un 3-simplex y un 4-simplex, pero un alto coste en la gestión dinámica de memoria cuando el árbol a procesar es grande. Por lo tanto, el gestor de memoria usado juega un papel importante en el rendimiento final del algoritmo. Se han comparado los gestores de memoria de GNU gcc, Intel TBB, Google y Lockless

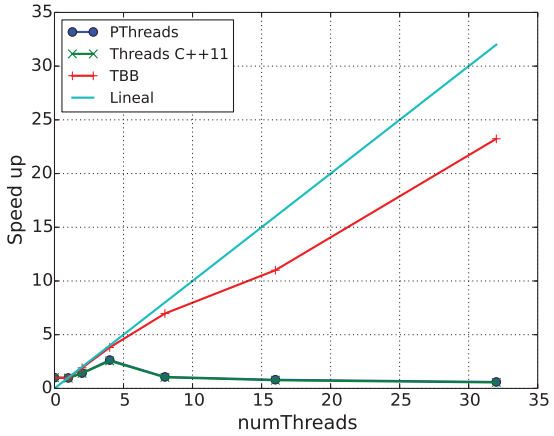

Fig. 6. Ganancia en velocidad para I1 en aloe

Inc., resultando este último el más eficiente.

En cuanto a las librerías de desarrollo de aplicaciones multihebradas, todas han mejorado con el uso del gestor de memoria LLAlloc. TBB presenta la mejor eficiencia, cercana a la lineal, gracias a la gestión de las colas de tareas asociadas a las hebras en ejecución. Debido a su facilidad de uso resulta la mejor opción.

El modelo paralelo basado en la generación dinámica de hebras con Pthread y C++11 presenta una baja ganancia en velocidad debido a que unas hebras deben esperar a la terminación de otras. Estamos interesados en mejorar este modelo ya que permitirá a la aplicación multihebrada adaptar su nivel de paralelismo a los recursos disponibles en el sistema en tiempo de ejecución.

AGRADECIMIENTOS

Este trabajo ha sido subvencionado por el Ministerio de Ciencia e Innovación (TIN2008-01117 y TIN2012-37483) y la Junta de Andalucía (P011-TIC7176) y financiado en parte por el Fondo Europeo de Desarrollo Regional (ERDF).

REFERENCIAS

[1] L.G. Casado, I. García, B.G. Tóth, and E.M.T. Hendrix, "On determining the cover of a simplex by spheres cente-

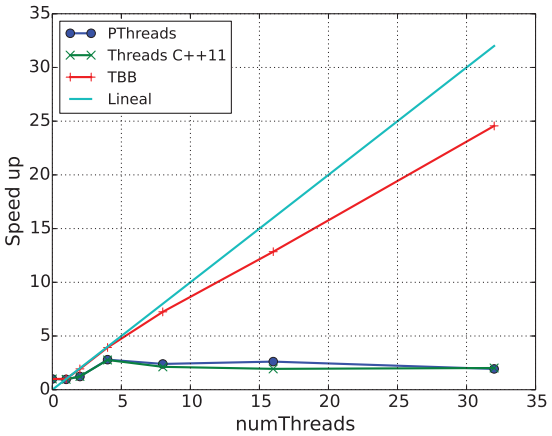

Fig. 7. Ganancia en velocidad para I2 en aloe

red at its vertices," *Journal of Global Optimization*, vol. 50, no. 4, pp. 645–655, 2011, DOI:10.1007/s10898-010-9524-x.

[2] Andrew Adler, "On the Bisection Method for Triangles," *Mathematics of Computation*, vol. 40, no. 162, pp. 571–574, 1983, DOI:10.1090/S0025-5718-1983-0689473-5.

[3] R. Horst, "On generalized bisection of n-simplices," *Mathematics of Computation*, vol. 66, no. 218, pp. 691–698, 1997, DOI:10.1090/S0025-5718-97-00809-0.

[4] Guillermo Aparicio, Leocadio G. Casado, Boglárka G-Tóth, Eligius M. T. Hendrix, and Inmaculada García, "On the minimim number of simplex shapes in longest edge bisection refinement of a regular n-simplex," *Informatica*, vol. 26, no. 1, pp. 17–32, 2015, DOI:10.15388/Informatica.2015.36.

[5] J.F. Sanjuan-Estrada, L.G. Casado, and I. García, "Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures," *Journal of Supercomputing*, vol. 58, pp. 376–384, 2011, DOI: 10.1007/s11227-011-0594-4.