Tamaño del árbol mínimo en el refinamiento de un simplex regular sobre arquitecturas MIC

J.M.G. Salmerón[1] [2], J.J. Moreno[1], L.G. Casado[1] y E.M.T. Hendrix[3]

Resumen— **En Optimización Global, mediante Ramificación y Acotación, es habitual usar la bisección por lado mayor como método de refinamiento cuando el espacio de búsqueda es un símplice. En problemas donde la dimensión es mayor que 3 pueden existir diversos lados mayores y la elección de uno u otro afecta al tamaño del árbol binario que se genera en el refinamiento. Estamos interesados en conocer el tamaño del menor árbol. Debido a que la dificultad del problema aumenta con la dimensión y con la disminución del tamaño de los símplices finales, se hace necesario el desarrollo de algoritmos paralelos que resuelvan el problema en un tiempo razonable. En este trabajo nos centraremos en un algoritmo paralelo que ha permitido solucionar instancias del problema que no han sido resueltas por otros algoritmos debido a sus requerimientos de memoria. En este trabajo se estudia y compara la ejecución de este algoritmo en dos arquitecturas paralelas diferentes: CPU Intel Xeon y tarjeta aceleradora Intel Xeon Phi.**

Palabras clave— **Simplex, árbol binario, manycore, multicore, PThreads, Intel Xeon Phi, MIC**

I. Introducción

EL problema de obtener el árbol binario de tamaño mínimo en el refinamiento mediante la bisección por lado mayor de un simplex regular es un problema de optimización combinatoria, debido a la existencia de múltiples opciones de selección del lado a dividir cuando la dimensión es mayor de 3 [1].

Estudios previos muestran soluciones paralelas a este problema usando las librerías Pthreads y TBB (Intel Threading Building Blocks) [2]. La principal conclusión de esos estudios es que la versión paralela debe imitar en lo posible la búsqueda en profundidad realizada por la versión secuencial, con el fin de reducir el consumo de memoria del algoritmo paralelo. El robo de tareas de TBB, que se realiza en niveles más cercanos a la raíz del árbol hace que el algoritmo se quede sin memoria, ya que abre muchas ramas del árbol a la vez. Aquí estudiaremos el comportamiento de la versión IPTH basada en Pthreads en la Xeon Phi, que dispone de una limitada cantidad de memoria. Se ha descartado el uso de la versión TBB debido a sus mayores requerimientos de memoria [2].

Se pretende comparar el rendimiento, consumo de memoria y de energía del algoritmo IPTH en la Xeon Phi y en un equipo con procesador Intel Xeon tradicional. El principal objetivo es determinar si el uso de la Xeon Phi para la resolución de este tipo de problemas, que son altamente irregulares y en general con una alta demanda de memoria es una alternativa competitiva a una arquitectura más tradicional. Una de las principales ventajas de la Xeon Phi frente a una GPU o GPGPU es que para portar el código C/C++ y hacer uso de la tarjeta Xeon Phi basta con compilarlo para su uso en la arquitectura MIC (*Many Integrated Core*) de la Phi.

En la sección II se describe el algoritmo de Optimización Global (OG) que hace uso de técnicas de Ramificación y Acotación para encontrar el valor mínimo de una función objetivo en el espacio de búsqueda definido por un simplex regular. Una de las acciones a realizar es la de división o refinamiento del símplice seleccionado y no rechazado, mediante la bisección del lado mayor. En la sección III se presenta el algoritmo secuencial que calcula el menor tamaño de un árbol que se generaría si no hubiera eliminación de nodos en el algoritmo de OG. Su versión paralela se describe en la sección IV. La sección V muestra la comparación de las ejecuciones de este algoritmo en la Intel Xeon Phi y en un equipo multicore. Finalmente se muestran las principales conclusiones en la sección VI.

II. Algoritmo de Optimización Global basado en Ramificación y Acotación

La resolución de un problema de Optimización Global consiste en encontrar el valor mínimo o máximo de una función objetivo $f : \mathbb{R}^d \to \mathbb{R}$. Cuando la única restricción se basa en límites en el espacio de búsqueda, estos problemas se denominan sin restricciones. Para problemas de minimización, el problema de Optimización Global puede escribirse como

$$f^* = f(x^*) = \min_{x \in S_1} f(x), \tag{1}$$

donde S_1 es un n-símplice regular con $n = d - 1$, que define el espacio de búsqueda. Este espacio de búsqueda aparece, por ejemplo, en problemas de diseño de productos a partir de mezcla de materiales base, donde existen restricciones en el diseño. El conjunto de posibles mezclas es el símplice unidad, donde x_i representa la fracción del material base i en la mezcla x, véase [3–5]. Otra aplicación de este refinamiento es la de determinar si una matriz es copositiva [6–8].

Por simplicidad y sin pérdida de generalidad, en este estudio se va a hacer uso de un símplice regular con longitud de lado igual a 1, que se define como:

$$S_1 = \left\{ x \in \mathbb{R}^{n+1} \ : \ \sum_{j=1}^{n+1} x_j = \frac{\sqrt{2}}{2};\ x_j \geq 0 \right\} \tag{2}$$

[1]Grupo Supercomputación-Algoritmos, Departamento de Informática, Universidad de Almería (ceiA3), e-mail: josemanuel@ual.es, jrm069@inlumine.ual.es, leo@ual.es

[2]Becario del programa FPU del Ministerio de Educación, Cultura y Deporte (FPU14/00634)

[3]Departamento de Arquitectura de Computadores, Universidad de Málaga, España, e-mail: Eligius.Hendrix@wur.nl

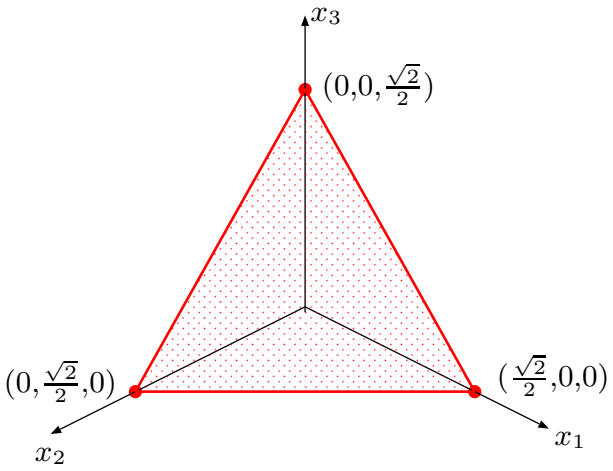

Fig. 1

Un 2-símplice regular con lados de longitud unidad.

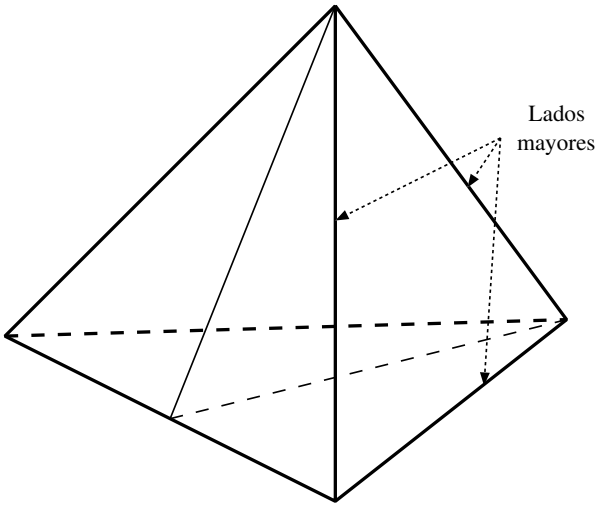

Fig. 2

Primera bisección del lado mayor en un 3-símplice.

La figura 1 muestra un 2-símplice regular con lados de tamaño unidad.

Los algoritmos de Ramificación y Acotación (RyA) se aplican a la resolución de problemas de Optimización Global cuando se requiere una búsqueda exhaustiva del mínimo global en el espacio de búsqueda. Un algoritmo de RyA realiza una búsqueda mediante el refinamiento sucesivo del problema inicial en sub-problemas de menor tamaño hasta alcanzar la(s) solución(es) final(es) o una aproximación a ella(s). Esta aproximación está determinada por la precisión requerida en la solución. El refinamiento genera un árbol de búsqueda que se poda cuando se garantiza que un sub-problema no contiene una solución global. La poda o rechazo de un sub-problema está normalmente basada en cálculos de cotas de la función objetivo para ese sub-problema. Los algoritmos de RyA están caracterizados por las reglas de Acotación, Selección, División, Rechazo y Terminación. Se pretende encontrar el tamaño del menor 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) [9, 10] y como regla de terminación la longitud del lado mayor de un sub-problema o simplex. La división del lado mayor es un método popular de refinamiento iterativo en el método de los elementos finitos, ya que es muy simple y puede ser aplicado en espacios de grandes dimensiones [11]. La selección de un lado mayor en un 2-símplice no es necesaria ya que, o el lado mayor es único, o el símplice a dividir es regular. A partir de dimensión 4 existen varias opciones para elegir el lado mayor, como se muestra en la figura 2.

El esquema básico de un algoritmos de RyA 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. Como hemos mencionado, 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. La regla de terminación no procesa más un simplex S cuando $w(S) \leq \epsilon \cdot w(S_1)$, donde $w(S)$ es el tamaño del simplex que se define como la longitud de su lado mayor.

Algorithm 1 Ramificación y Acotación

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

III. Algoritmo secuencial para el cálculo del tamaño del árbol binario mínimo

El algoritmo secuencial debe comprobar todas las opciones de división de lado mayor, y tener en cuenta solo aquellas que generan un árbol binario de menor tamaño. Esta acción se puede realizar mediante recursión, como muestra el algoritmo 2, el cual explora todas las opciones de división realizando una búsqueda en profundidad. Se realiza un exploración en profundidad porque presenta unos requerimientos de memoria menores que otros tipos de búsqueda. Los parámetros iniciales del algoritmo 2 son el n-simplex regular inicial S_1, uno de sus lados L (todos

son iguales) y la precisión requerida ϵ.

Algorithm 2 TamañoÁrbolMínimo(S, L, ϵ)

Require: S: simplex, L: lado mayor, ϵ: precisión
1: **if** $w(S) \leq \epsilon$ **then**
2: **return** 1
3: $\{S_l, S_r\}$:= DivideSímplice(S, L)
4: **for** cada lado mayor L_i de S_l **do**
5: rl_i := TamañoÁrbolMínimo(S_l, L_i, ϵ)
6: **for** cada lado mayor L_i de S_r **do**
7: rr_i := TamañoÁrbolMínimo(S_r, L_i, ϵ)
8: $r_l := \min_i\{rl_i\}$
9: $r_r := \min_i\{rr_i\}$
10: **return** $1 + r_l + r_r$

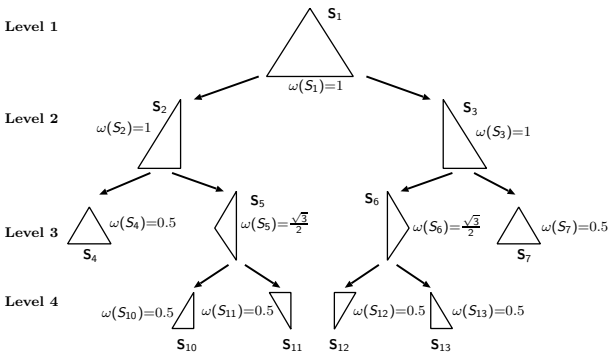

Fig. 3
Árbol binario para $\epsilon = 0{,}5$.

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

- Si los símplices hermanos S_2 y S_3 son simétricos, ambos generan sub-árboles 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 sub-árbol 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 la existencia de parejas simétricas de hermanos, pero esto es computacionalmente más costoso que determinar si dos símplices hermanos son simétricos y se abordará en trabajos futuros.

IV. Algoritmo paralelo para el cálculo del tamaño del árbol binario mínimo

Este algoritmo 2 permite ser paralelizado fácilmente debido a que diferentes ramas del árbol general (pueden existir más de dos ramas en un nodo) pueden ser procesadas de forma paralela. Uno de los inconvenientes que presentan este tipo de problemas es la irregularidad del árbol ya que normalmente cada rama o sub-árbol tendrá un tamaño distinto que no es conocido a priori, por lo tanto requerirá de un balanceo dinámico de la carga entre hebras. Además, los aspectos a tener en cuanta que se presentaron en la sección anterior aumentan aún más la irregularidad de los datos.

El algoritmo 3 muestra la primera aproximación del algoritmo 2 en paralelo. En él se crea una nueva hebra siempre que exista trabajo pendiente hasta alcanzar el número máximo permitido, MaxHebras. Cada hebra ejecuta el algoritmo con el sub-símplice que se le haya asignado y creará nuevas hebras siempre que le sea posible. En caso contrario, ejecutará el algoritmo secuencial. Las hebras que terminen de procesar el sub-árbol asignado, devuelven el tamaño mínimo del sub-árbol como resultado y terminarán su ejecución, decrementando la variable NHebras para permitir que otras hebras puedan crear nuevas hebras que ejecuten el trabajo pendiente. Las variables NHebras y MaxHebras serán de ámbito global a todas las hebras. Durante la ejecución, la primera hebra que acceda a la variable NHebras<MaxHebras será la que creará una hebra nueva. De esta forma, el balanceo dinámico de la carga es inherente a la creación dinámica de las hebras.

Algorithm 3 TamÁrbolMínParalelo (S, L, ϵ)

Require: S: simplex, L: lado mayor, ϵ: precisión.
1: **if** $w(S) \leq \epsilon$ **then**
2: **return** 1
3: $\{S_l, S_2\}$:= DivideSímplice(S, L)
4: **for** cada lado mayor L_i de S_l **do**
5: **if** NHebras < MaxHebras **then**
6: rl_i := CreaHebra(TamÁrbolMínParalelo(S_l, L_i, ϵ))
7: **else**
8: rl_i := TamÁrbolMínParalelo(S_l, L_i, ϵ)
9: **for** cada lado mayor L_i de S_r **do**
10: **if** NHebras < MaxHebras **then**
11: rr_i := CreaHebra(TamÁrbolMínParalelo(S_r, L_i, ϵ))
12: **else**
13: rr_i := TamÁrbolMínParalelo(S_r, L_i, ϵ)
14: Esperar resultados de las hebras creadas
15: $r_l := \min_i\{rl_i\}$
16: $r_r := \min_i\{rr_i\}$
17: **return** $1 + r_l + r_r$

El problema presenta unos bajos requerimientos computacionales por símplice, pero unos altos requerimientos de memoria para almacenar el árbol de búsqueda, debido a que se exploran varios sub-árboles en paralelo. Por ello, es conveniente emplear un *Cut-off* que limite el paralelismo hasta una cierta profundidad del árbol, a partir de la cual no se permite la creación de hebras.

La principal limitación al paralelismo del algoritmo 3 es que una hebra, para procesar el tamaño mínimo de un sub-árbol, debe esperar el resultado de la(s) hebra(s) que ha creado para su procesamiento (véase la línea 14). Para resolver este problema se realizó una implementación paralela llamada IPTH que en lugar de ser recursiva es iterativa [2]. En IPTH una hebra generada para calcular el tamaño mínimo de un sub-arbol, almacena ese tamaño en el nodo correspondiente, explora otros sub-árboles pendientes en el nodo actual, termina el nodo ac-

tual si todos sub-árboles han sido calculados, o busca trabajo en ramas superiores del árbol si el trabajo pendiente ya está asignado a otras hebras. De esta forma las hebras no están constantemente creándose y destruyéndose sino que una vez creadas continúan trabajando mientras encuentren trabajo por hacer y solo se destruyen cuando, durante la subida por el árbol en busca de trabajo, llegan al nodo raíz del árbol sin haber encontrado trabajo pendiente, aunque este puede existir en ramas que están siendo procesadas. Las hebras que alcancen la raíz del árbol, al morir permiten la creación de nuevas hebras que ayuden a las que están procesando niveles inferiores del árbol general.

V. Resultados

El algoritmo IPTH [2] ha sido compilado con Intel *icc* empleando la opción *-mmic* para su ejecución en la tarjeta aceleradora Intel Xeon Phi. En este caso sobre el modelo 3120 que incorpora 57 cores a 1,10 GHz y 6 GB de memoria RAM integrada.

Las instancias ejecutadas para los experimentos han sido las siguientes:

$I3$: d=4, ϵ=0,025 y $Cut\text{-}off = \omega(S) \leq 2\epsilon \cdot \omega(S_1)$
$I4$: d=5, ϵ=0,125 y $Cut\text{-}off = \omega(S) \leq 2\epsilon \cdot \omega(S_1)$
$I5$: d=6, ϵ=0,25 y $Cut\text{-}off = \omega(S) \leq 2\epsilon \cdot \omega(S_1)$

Se han empleado estos valores de dimensión y precisión para obtener resultados en un tiempo razonable, debido a la complejidad combinatoria del problema a resolver.

La tabla I compara los tiempos secuenciales obtenidos en la Xeon Phi con los obtenidos en un nodo BullX [2]. Un nodo de BullX dispone de 2 Intel Xeon E5-2650 (Sandy Bridge) de 8 cores cada uno a 2 GHz con 64 GB de RAM. Los tiempos en la Xeon Phi son un orden de magnitud más grandes que los obtenidos en BullX.

TABLA I
Comparación de tiempos secuenciales en BullX y Phi.

Instancia	BullX	Phi
I3	23,19 s	267,21 s
I4	122,04 s	1.311,19 s
I5	10.193,70 s	100.207,99 s

La figura 4 muestra los resultados de la ejecución de la instancia I3 en la tarjeta Phi para un número diferente de MaxHebras. En la figura está representada la ganancia de velocidad (speedup, SP) y el consumo de memoria de cada ejecución (abscisa derecha en escala logarítmica). Para cada hebra se muestra la media de memoria consumida, la desviación estándar y los valores mínimos y máximos observados durante la ejecución. El consumo de memoria en cada instante de tiempo se ha obtenido mediante la salida del comando *ps* de Linux usando una modificación del script *memusg*[1] que realiza un muestreo cada 0,05 segundos.

[1] https://gist.github.com/netj/526585

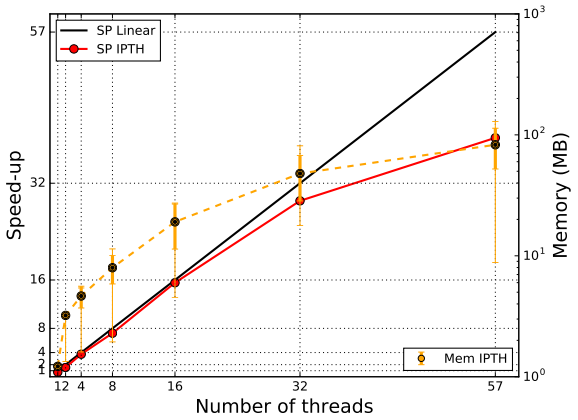

Fig. 4
I3 speedup (SP) y consumo de memoria (Mem) en Phi.
T_{sec} = 267,21 s

Las figuras 5 y 6 muestran el speedup y los datos de consumo de memoria, al igual que 4, para las instancias I4 y I5, respectivamente.

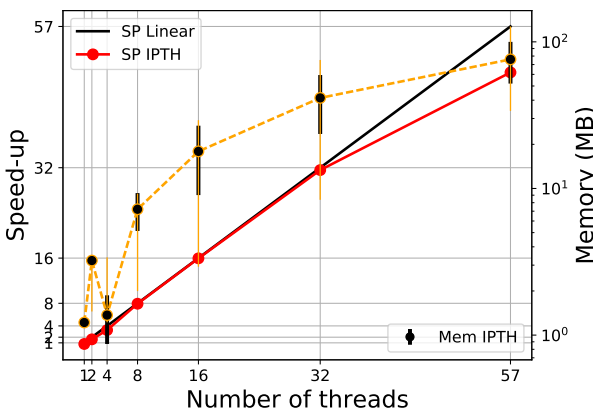

Fig. 5
I4 speedup (SP) y consumo de memoria (Mem) en Phi.
T_{sec} = 1.311,19 s

La ejecución del algoritmo usando un número mayor de hebras no aporta una ganancia en velocidad sustancial respecto al uso de Maxhebras=57.

Las figuras 7, 8 y 9 muestran la ganancia de velocidad y consumo de memoria en un nodo BullX. La ganancia de velocidad en ambas plataformas es buena.

La figura 10 muestra la salida del script *memusg* para la ejecución en la Phi con MaxHebras=57. Debido a que al principio hay muchas hebras trabajando en distintas ramas del árbol general, el uso de memoria es alto y según avanza la ejecución las ramas evaluadas son eliminadas de memoria por lo que la tendencia general del consumo de memoria es a la baja. La tendencia en el consumo de memoria es similar para el resto de instancias ejecutadas en la Phi.

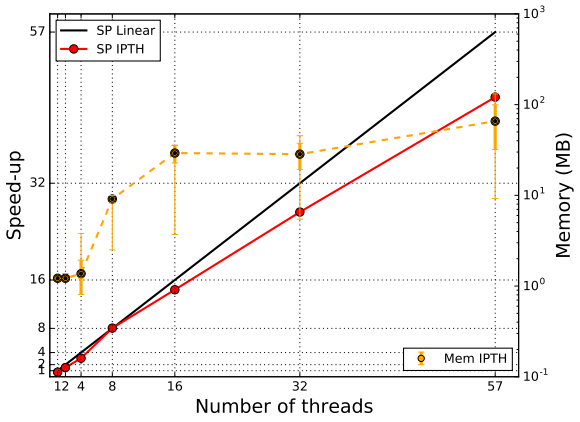

Fig. 6
I5 speedup (SP) y consumo de memoria (Mem) en Phi. T_{sec} = 100.207,99 s

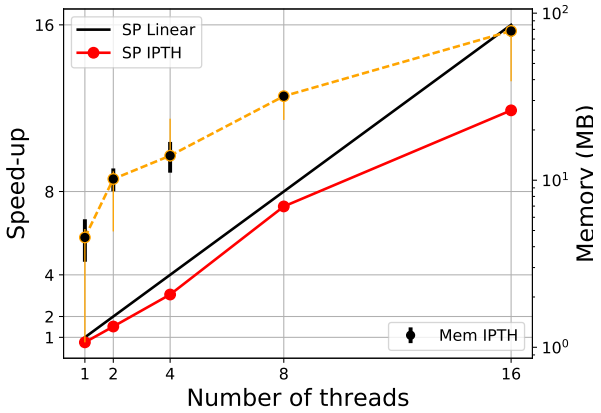

Fig. 8
I4 speedup (SP) y consumo de memoria (Mem) en BullX. T_{sec} = 122,04 s

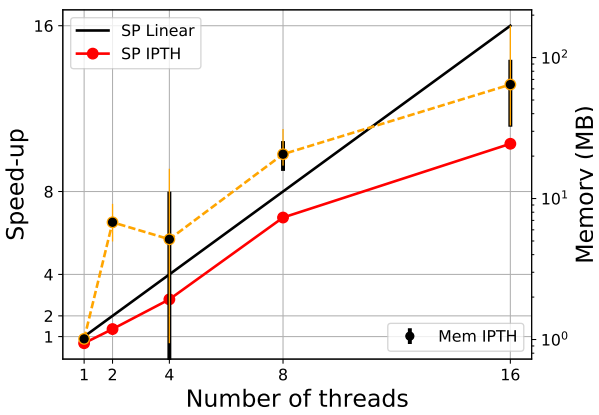

Fig. 7
I3 speedup (SP) y consumo de memoria (Mem) en BullX. T_{sec} = 23,19 s

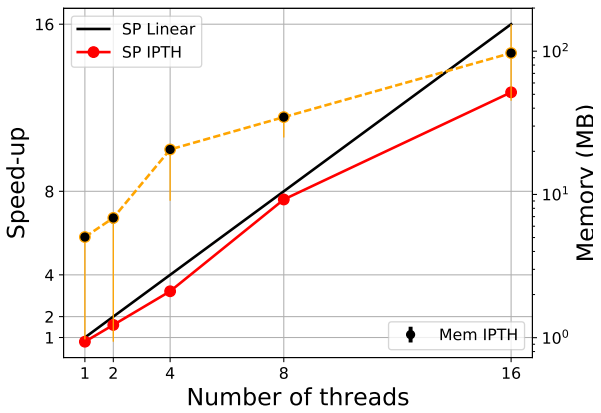

Fig. 9
I5 speedup (SP) y consumo de memoria (Mem) en BullX. T_{sec} = 10.193,7 s

La figura 11 muestra el uso de la memoria del nodo de BullX en la ejecución con MaxHebras=16. Las diferencias en la tendencia del consumo de memoria que muestran las figuras 10 y 11, están debidas principalmente al distinto número de MaxHebras utilizado. Un número menor de hebras expande menos el árbol general de búsqueda en las etapas iniciales del algoritmo.

La tabla II muestra una comparación de la energía necesaria para resolver la misma instancia en la tarjeta Xeon Phi 3120 y en una y dos CPU(s) Xeon E5-2650 de un nodo BullX. En el caso de la Phi se usaran todos los procesadores disponibles. El consumo energético se ha obtenido mediante *perf*[2] en los dominios *package* (Cores) y *dram* (Memoria) utilizando la interfaz Intel RAPL (*Running Average Power Limit*) para el procesador Intel Xeon E5-2650 de Bull y con el comando *micsmc* para la Xeon Phi. El consumo de la Xeon Phi es siempre superior al de un nodo de BullX independientemente de del número de cores usados. El tiempo de ejecución en la Phi con MaxHebras=57 es mejor que en el nodo BullX cuando MaxHebras<6.

VI. Conclusiones

Se ha podido ejecutar en una Xeon Phi, que dispone de una capacidad de memoria RAM limitada, un algoritmo con una estructura de datos altamente irregular y que se diseñó para paliar los altos requerimientos de memoria que suelen demandar los algoritmos de Ramificación y Acotación. La comparación con un nodo con dos CPUs Xeon E5-2650 muestra que la Phi usando todos sus cores es competitiva en tiempo de ejecución cuando se usan menos de 6 cores en una de las CPUs del nodo y el consumo energético es mucho mayor en la Xeon Phi 3120. Aunque la Phi es más barata que un nodo de las características presentadas, la relación entre rendimiento, consumo y precio no hacen a la Phi 3120 una buena opción de compra para resolver este

[2] https://perf.wiki.kernel.org

TABLA II

Rendimiento y consumo: Tarjeta Xeon Phi contra procesador Xeon

	Phi 57 hebras		BullX 8 hebras		BullX 16 hebras	
Instancia	Tiempo	Energía	Tiempo	Energía	Tiempo	Energía
I3	6,76 s	963,82 J	3,40 s	326,33 J	2,19 s	239,71 J
I4	26,83 s	3.838,32 J	16,33 s	1.538,53 J	10,12 s	1.182,66 J
I5	2.167,10 s	319.406,92 J	1.228,62 s	124.051,30 J	800,27 s	94.572,92 J

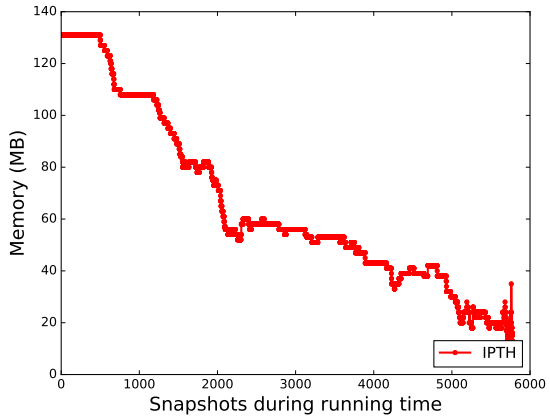

Fig. 10

Consumo de memoria de I5 con 57 hebras en Phi

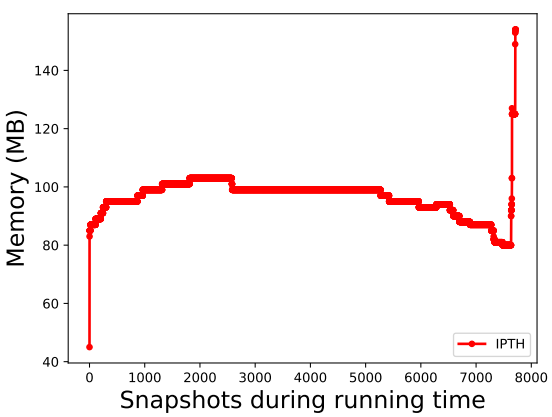

Fig. 11

Consumo de memoria de I5 con 16 hebras en BullX

tipo de problemas. Se pretende repetir estos estudios sobre una Xeon Phi de última generación como la 7290 con 72 cores y 16GB de RAM.

Agradecimientos

Este trabajo ha sido subvencionado por el Ministerio de Ciencia e Innovación (TIN2015-66680) y financiado en parte por el Fondo Europeo de Desarrollo Regional (ERDF).

Referencias

[1] J.M.G. Salmerón, G. Aparicio, L.G. Casado, I. García, E.M.T. Hendrix, and B.G. Tóth, "Generating a smallest binary tree by proper selection of the longest edges to bisect in a unit simplex refinement," *Journal of Combinatorial Optimization*, pp. 1–14, 2017.

[2] G. Aparicio, J.M.G. Salmerón, L. G. Casado, R. Asenjo, and E.M.T. Hendrix, "Parallel algorithms for computing the smallest binary tree size in unit simplex refinement," *Journal of Parallel and Distributed Computing*, 2017, IN PRESS.

[3] L.G. Casado, I. García, B.G. Tóth, and E.M.T. Hendrix, "On determining the cover of a simplex by spheres centered at its vertices," *Journal of Global Optimization*, vol. 50, no. 4, pp. 645–655, 2011, DOI:10.1007/s10898-010-9524-x.

[4] E.M.T. Hendrix, L. G. Casado, and I. GarcÃa, "The semi-continuous quadratic mixture design problem: Description and branch-and-bound approach," *European Journal of Operational Research*, vol. 191, no. 3, pp. 803–815, 2008, DOI:10.1016/j.ejor.2007.01.056.

[5] J. Ashayeri, A.G.M. van Eijs, and P. Nederstigt, "Blending modelling in a process manufacturing: A case study," *European Journal of Operational Research*, vol. 72, no. 3, pp. 460 – 468, 1994, DOI:10.1016/0377-2217(94)90416-2.

[6] Immanuel M. Bomze, Mirjam Dür, Etienne de Klerk, Cornelis Roos, Arie J. Quist, and Tamás Terlaky, "On copositive programming and standard quadratic optimization problems," *Journal of Global Optimization*, vol. 18, no. 4, pp. 301–320, 2000.

[7] Stefan Bundfuss and Mirjam DÃ¼r, "Algorithmic copositivity detection by simplicial partition," *Linear Algebra and its Applications*, vol. 428, no. 7, pp. 1511 – 1523, 2008.

[8] Julius Zilinskas and Mirjam Dur, "Depth-first simplicial partition for copositivity detection, with an application to maxclique," *Optimization methods & software*, vol. 26, no. 3, pp. 499–510, 2011, Relation: http://www.rug.nl/fmns-research/bernoulli/index Rights: University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

[9] 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.

[10] 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.

[11] Antti Hannukainen, Sergey Korotov, and Michal Křížek, "On numerical regularity of the face-to-face longest-edge bisection algorithm for tetrahedral partitions," *Science of Computer Programming*, vol. 90, pp. 34–41, 2014, DOI:10.1016/j.scico.2013.05.002.